COMP 1406
Summative Project - OOP Search Engine

Hassan Ali, Kunwar Nir

I. GUI Instructions

Firstly, run the main function in the `Search.java` file to initiate the crawl. To run the GUI, navigate to the `App.java` file in the `src/` directory which is the controller. From there, make sure to configure **JavaFX** with accordance to the *IntelliJ* instructions outline in week 12's lessons. From there, running the `App.java` file will launch the GUI.

II. Functionality List

- Created a JavaFX GUI with accordance to the **MVC** paradigm
- GUI has ability to search with any given query
- GUI has ability to provide search results with or without the page rank boost
- Implement OOP concepts to create Web-Crawler
- Implemented tf, idf, and tf-idf calculations
- Implemented incoming and outgoing links
- Implemented pageRank
- Implemented search
- Handles html attributes
- Handles both absolute and relative urls
- Handles different page names
- Handles multiple paragraphs
- Can use tests built with ProjectTester in mind

III. File Outline

A. *Link.java*

This file is the class for the links. It provides the outline to create Link objects which are used throughout the program. The Link class has many class attributes which are the attributes that are held by all links. These attributes are:

- **url**: the String url of the link
- **outgoingLinks**: an *ArrayList* of Links that are the outgoing links for a given webpage
- **incomingLinks**: an *ArrayList* of Links that are the incoming links for a given webpage
- **countAll**: a <String, Integer> *HashMap* which holds a word as the key and the frequency of that word as the value
- **title**: a String of the title of the Link
- **pageRank**: a double value that stores the page rank of a link
- **wordCount**: an integer which holds the wordcount of a link
- **score**: a double value that holds the page score

This class, along with the constructor, has setter and getter methods for each of its attributes. For the attributes which use Abstract Data Types, there are methods that allow you to add one value to that attribute.

This class implements two interfaces, the **SearchResults.java** interface and the *Comparable* interface. With the implementation of the **SearchResults.java** interface, the class must include the `getTitle` and `getScore` methods. These methods are useful when data is being displayed in the GUI as the GUI makes use of these methods.

By implementation of the *Comparable* interface, the class implements the method `compareTo`. This method compares Links firstly based on score and then lexicographically. This is implemented to ensure that the Links are displayed in the correct order after the search is complete. We used this interface instead of sorting the links ourselves as the *TreeSet* abstract data type would be able to make use of this and it would be able to sort the links efficiently.

B. *SearchResults.java*

This interface is in place to ensure that all search results have `getTitle` and `getScore` methods. These methods are imperative when displaying the results of the search in the GUI.

C. *ImprovedQueue.java*

This file is the class for the ImprovedQueue. It allows for a queue data type to be created. It uses an *ArrayList* and a *HashMap* to store data. The *HashMap* allows for $O(1)$ lookup and the *ArrayList* allows for $O(1)$ indexing. It has the following methods:

- `addend`
- `removestart`
- `contains`
- `size`
- `toString`

D. *Crawler.java*

The Crawler class is the class that crawls through a seed link and stores the data that it finds. The file has two *HashMaps*; `data` and `IDFs`. The class has a `crawl` method which takes in a String value of the seed URL. There is also a parse method that reads the website of a given link. The parse method saves the necessary information in an instance of the **Link.java** class which includes the total word count, the individual word counts, the title, and the link (absolute and relative). The `crawl` method calls the parse method on each Link in the chain which originates from the seed.

There is also a `createFiles` method which goes through every link that has been crawled through and creates files based on the data associated with that link. It stores the page ranks, word counts, IDFs, etc in their corresponding files so that they can be accessed later on during the search. Reasoning explained here: **Data**.

There is a method named `createPageRanks`. This method uses the connections between the Links to create the page ranks for all of the web pages which were parsed during the crawl. We used *HashMaps* to reduce any lookup to $O(1)$ time and simplified all the matrix operations of creating the probability matrix into one block of two for loops.

The file also consists of other helper methods which are called within the other methods. These include:

- `deleteFolders`
- `getIdf`
- `getTf`
- `getTfIdf`
- `dotProduct`
- `log2`
- `euclideanDistance`

E. *Search.java*

The Search class acts as the model for the GUI as part of the MVC paradigm. The search class has a search method which reads through the files which are created by the crawl and saves and returns the top search results in an *ArrayList* of SearchResults.
There are also other helper methods in this class such as:

- `cosineSim`
- `euclideanNorm`

The `getPhraseVector` method returns the phrase vector of the query. Also, the `getSearchResults` is a getter method which returns the *ArrayList* of the search results.

F. *SearchData.java*

The SearchData class implements the ProjectTester interface. It has an attribute for searchResults which is a *List* of type **SearchResults.java**. The class also has the method `initialize`, this method clears all of the files present so that a new crawl can be initiated. The SearchData class has a `crawl` method which crawls through a seed. It also many getter methods which retrieve data from the files stored as result of a crawl. These getter methods include:

- `getOutgoingLinks`
- `getIncomingLinks`
- `getPageRank`
- `getTitle`
- `getIDF`
- `getTF`
- `getTFIDF`

The `search` method creates an object of the `Search.java` class and performs a search given a query and the page rank boost.

There is also a `readFile` method which is a helper method that reads a file at a given file path using Java **File I/O** concepts.

G. *SearchButtonPane.java*

This class is in place to create the search button pane for the GUI. This is so all of the buttons can be grouped together for purposes of encapsulation. Furthermore, this aids in the extensibility of the program since new buttons can be added to this class instead of the main **AppView.java** class.

This class extends the **JavaFX** Pane to create a new Pane and a new Button. It then sets the properties of the button and adds it to the pane. It also has a getter for the button so that it can be accessed by other files.

H. *AppView.java*

This file is the view for the GUI, in accordance to the **MVC** paradigm. It is a class that extends the JavaFX Pane. It creates many objects, including labels and textfields, and adds them to the pane. It also creates a **SearchButtonPane.java** object which it also adds to the pane. The classes also uses the data type *ListView* to store the search results. This is used to that the information can be displayed on the GUI.

The class has getter methods for all of its attributes (the textfields, labels, and buttons) so that they can be accessed by other classes and their properties can be read.

The class also has an `update()` method which takes in a model object of the class Search and a selectedIndex. It updates the *ListView* of the search results with the new search results present in the model.

I. *App.java*

This file is the controller for the GUI as part of the **MVC** paradigm. This file uses an instance of the **AppView.java** class to generate the GUI. It extends Application from **JavaFX**. This file creates a pane and adds an **AppView.java** instance to the pane. Since this class extends Application from **JavaFX**, it inherits the method `start` which allows the GUI to start and run.

The App.java file also handles all button clicks and then tells the view to update the displayed data.

IV. Overall Design

A. *OOP Principles*

Throughout our code, classes and object are used to streamline and organize the program.

The following is a UML Diagram displaying the class structure and the methods for each file pertaining to the GUI:

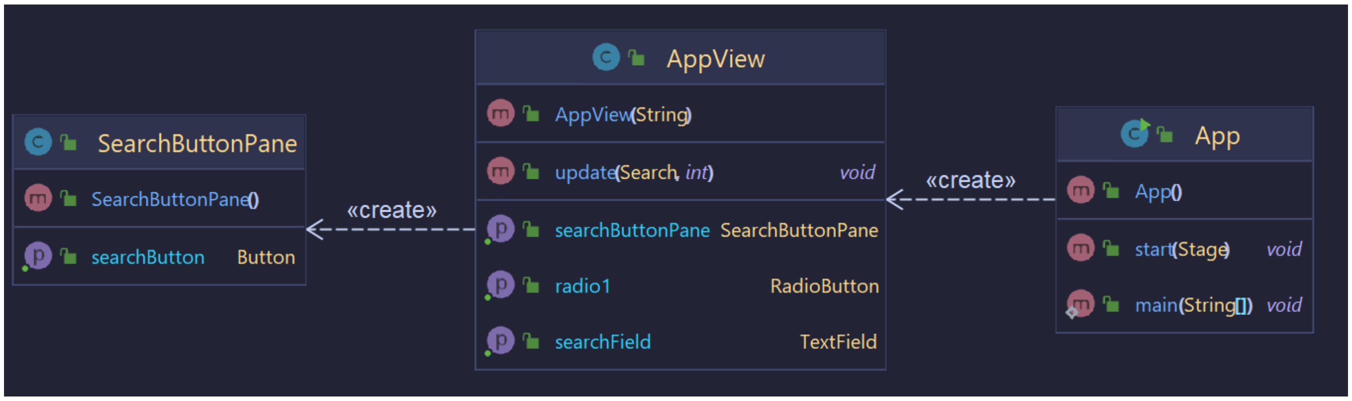

Fig. 1: GUI UML Diagram

The following is a UML Diagram displaying the class structure and the methods for each file pertaining to the model:

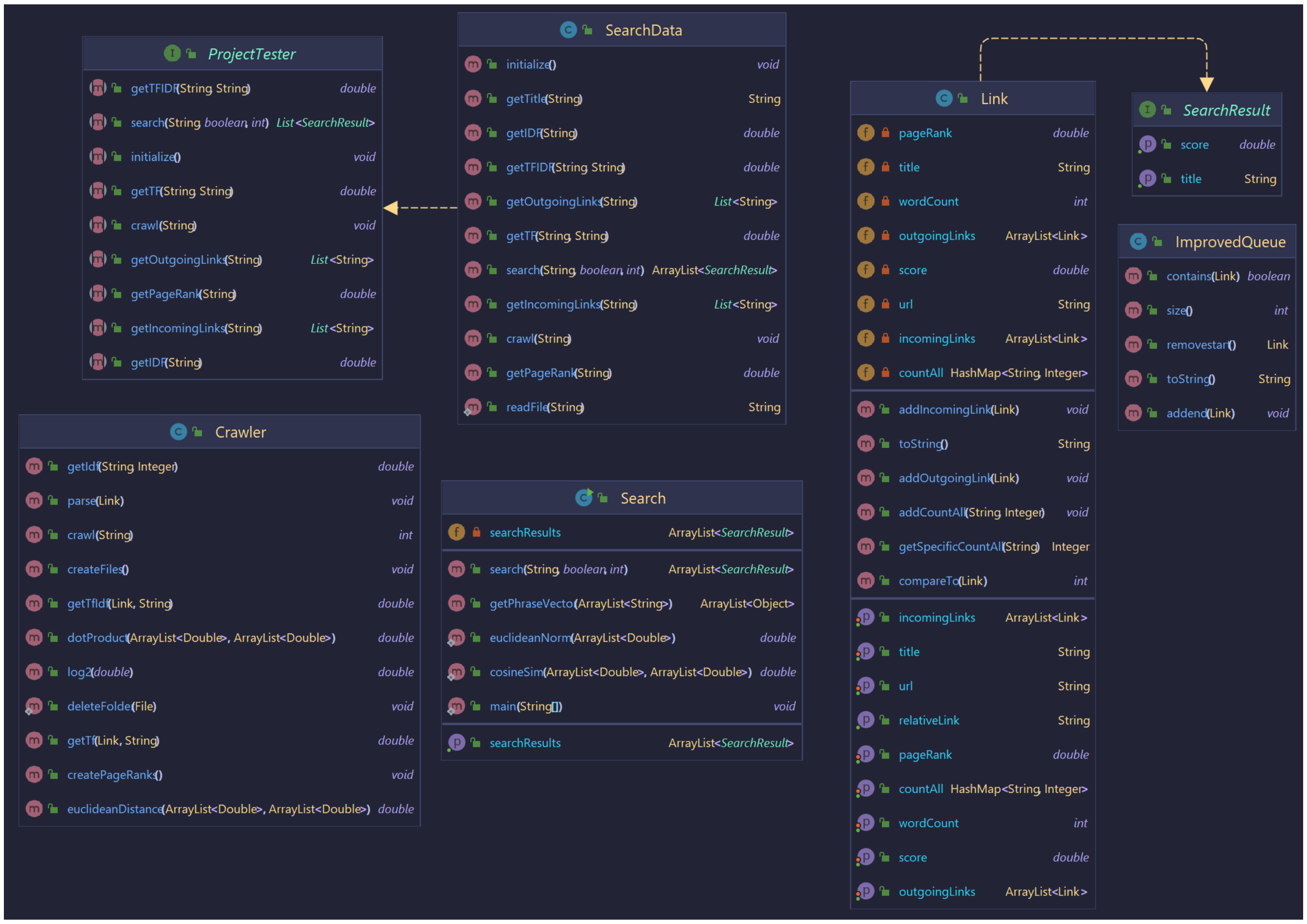
ProjectTester
getTFIDF(String String) double
search(String boolean, int) List<SearchResult>
initialize() void
getTF(String String) double
crawl(String) void
getOutgoingLinks(String) List<String>
getPageRank(String) double
getIncomingLinks(String) List<String>
getIDF(String) double
SearchData
initialize() void
getTitle(String) String
getIDF(String) double
getTFIDF(String String) double
getOutgoingLinks(String) List<String>
getTF(String, String) double
search(String, boolean, int) ArrayList<SearchResult>
getIncomingLinks(String) List<String>
crawl(String) void
getPageRank(String) double
readFile(String) String
Link
pageRank double
title String
wordCount int
outgoingLinks ArrayList<Link>
score double
url String
incomingLinks ArrayList<Link>
countAll HashMap<String, Integer>
addIncomingLink(Link) void
toString() String
addOutgoingLink(Link) void
addCountAll(String, Integer) void
getSpecificCountAll(String) Integer
compareTo(Link) int
incomingLinks ArrayList<Link>
title String
url String
relativeLink String
pageRank double
countAll HashMap<String, Integer>
wordCount int
score double
outgoingLinks ArrayList<Link>
SearchResult
score double
title String
ImprovedQueue
contains(Link) boolean
size() int
removestart() Link
toString() String
addend(Link) void
Crawler
getIdf(String Integer) double
parse(Link) void
crawl(String) int
createFiles() void
getTfIdf(Link, String) double
dotProduct(ArrayList<Double>, ArrayList<Double>) double
log2(double) double
deleteFolder(File) void
getTf(Link, String) double
createPageRanks() void
euclideanDistance(ArrayList<Double>, ArrayList<Double>) double
Search
searchResults ArrayList<SearchResult>
search(String, boolean, int) ArrayList<SearchResult>
getPhraseVector(ArrayList<String>) ArrayList<Object>
euclideanNorm(ArrayList<Double>) double
cosineSim(ArrayList<Double>, ArrayList<Double>) double
main(String[]) void
searchResults ArrayList<SearchResult>

Fig. 2: Model UML Diagram

This UML Diagram illustrates the use of **OOP**. There are two interfaces which are implemented. Furthermore, instances of classes are used as objects to organize the code. For example, in the **Crawler.java** class, the **Link.java** class is used throughout to create Link objects which are edited and accessed. We chose to do it this way so that the code would be easier to deal with. Furthermore, using the link example, when having a Link object, all of the necessary information pertaining to the link exists as attributes of the Link class; therefore when the files are creates, the information can be accessed from the getter methods of the Link class.

Abstract data types were used throughout the code in order to increase efficiency. We used *HashMaps* and *s* where they could be used in place of traditional data types such as Lists and Arrays. This was done to increase efficiency as retrieving data from a *HashMap* has a time-complexity of $O(1)$. This is due to the fact that *HashMaps* can retrieve data based on their key. This makes them well-suited for applications like ours that require fast access to data. *TreeSets* are also useful as they store data in ascending order. This means that they are easy and efficient to sort and search. We chose to use *TreeSets* for the score for the abovementioned reason.

We implemented principles of modularity in order to divide our program into smaller aspects (modules). This aided in reducing the complexity of the program. The modularity of the program is further enhanced through the use of encapsulation. By encapsulating our classes, they are each standalone modules that house all of their contained functionality independently. One example of this is the **Crawler.java** module. Since it is separate from the rest of the model, it can be edited without creating reference problems elsewhere in the code. Furthermore, in relation to the GUI, by creating it in accordance with the **MVC** paradigm it is inherently modular. This is helpful because if you had to edit how the GUI looks, you only need to edit and change the **AppView.java** class as that is the class that governs the view of the GUI. If you want to change the controller or actions of the button, you would only need to worry about the App.java file as that is the controller. Another advantage of making the different parts of the model modular is that multiple crawlers and searches can be used at once. If we were to expand this search engine, it would be important to crawl and search multiple times at once since multiple users would be using this search engine. Since the **Crawler.java** and the **Search.java** are classes that follow **OOP** principles, we would be able to create instances of them and use them simultaneously.

The code for our program is maintainable as it uses abstraction. Abstraction is used throughout to simplify elements. New aspects can be added without needing to fully understand the inner workings of existing classes which also falls into extensibility. For example, if a button were to be added to the view of the GUI, it could be done simply without having to overhaul code in any other classes. Furthermore, with the use of inheritance in our code, it reduces the repetition of code which promotes maintainability. Moreover, encapsulation also aids in maintainabilty by restricting access to the class attributes. This makes sure that only the relevant classes that have access to the attributes.

Extensibility of the code is enhanced through the use of polymorphism. One example of an instance where polymorphism is used in our code is when we cast Links as SearchResults. This aids in extensibility as new subclasses could later be added to the Link class (in case we want to specify different types of Links) and be used throughout the code without having to completely change the functionality of the **SearchResults.java** interface. Inheritance with View and the Controller is also useful as it ensure the subclasses follow the correct structure and have the relevant attributes and methods. This makes the program easier to edit and scale.

Our code is made robust through extensive use of try-catch **exception handling**. This ensure that only the correct functionality occurs. Since **File I/O** is a central aspect of the program, exception handling is imperative to ensure that the code is robust. Furthermore, encapsulation is used throughout the code to make it more robust. This is done by making class attributes private so that other classes cannot access and change them. Therefore, to edit and access them, they must use the provided getter and setters. These methods are equipped with a verifying system that ensures that the correct type of variable is being set, that there are no duplicates, it is within the range, etc. The robustness is greatly enhanced with the above implementation of encapsulation.

B. Data

The data is stored in the following format:

```
CourseProject1406/
├── Idfs/
│   ├── apple.txt
│   ├── banana.txt
│   ├── coconut.txt
│   └── ...
├── pages/
│   ├── http}{{people(scs(carleton(ca{˜davidmckenney{fruits2{N-0(html/
│   │   ├── countAll/
│   │   │   ├── apple.txt
│   │   │   ├── banana.txt
│   │   │   ├── coconut.txt
│   │   │   └── ...
│   │   ├── incominglinks.txt
│   │   ├── outgoinglinks.txt
│   │   ├── pageRank.txt
│   │   ├── title.txt
│   │   └── wordCount.txt
│   └── ...
└── src/
    └── course.project/
        ├── App.java
        ├── AppView.java
        ├── Crawler.java
        ├── ImprovedQueue.java
        ├── Link.java
        ├── ProjectTester.java
        ├── Search.java
        ├── SearchButtonPane.java
        ├── SearchData.java
        ├── SearchResult.java
        ├── TestingTools.java
        └── WebRequester.java
```

The file structure uses “fruits2” as an example.

We chose to have this file structure as it stores the information in a predictable and organized manner. It would also reduce the **RAM** overhead of holding a lot of objects in memory. Storing it in hard disk storage reduces the impact on memory. Storing the attributes separately in their own pages allows for $O(1)$ time lookups for everything besides the Outgoing and Incoming links. We also made IDFs as they are accessed often and are independent of the webpage and we realized it would beneficial to put them all in a directory of their own and organize them in **.txt** files by word.

The other information is specific to each webpage, therefore we provided each webpage with its own directory. Within that directory, all of the necessary information of the file is stored in **.txt** files.