COMP 1406Z

Fall 2023

Course Project Report

Teacher: Dave Mckenney

Student: Prince Karakura

Instructions for running the GUI:

1. Go into the "SearchApp" class.
2. Run the program from the main method within the class.
3. There will be two list views (titles and scores) that display the search results. Enter the search query in the textbox, select the radio boost button if desired and then press the search button.
4. The list views will update with the results while the query textbox and boost radio button will go back to their default settings.

List of functionalities successfully completed:

- GUI application using MVC paradigm.
- A class that implements the ProjectTester interface fully
- A website class that implements the SearchResult interface
- Three utility classes: one for loading the idf, linkmap, and websitedata object streams; one for keeping track of static crawl data; and one for multiplying matrices and calculating Euclidian distance.

Outline for the project's classes and interfaces:

Crawler

The main responsibility of the Crawler class is to visit each link when the public **crawl()** method is called and to gather information along the way. It does this through a bunch of smaller private methods that are called within the **crawl()** method. As it does this, the Crawler class makes instances of the LinkMappingData class, the IdfData class and the WebsiteData class. As these instances are made, the crawler's main job is to correctly utilize

each of their own methods to gather information and then tell them to store themselves within the crawler's directory as object streams. Another task that the Crawler class does as well is clear the crawler directory using the public **clearDirectory()** method.

IdfData

The main responsibility of the IdfData class is to calculate the idf value of each word that appeared among the crawled links and save that value to a string to double HashMap instance attribute using the **calculateIdf()** method. Then, the object can be saved with its filename and HashMap attribute to an object output stream within the crawler directory for use when needed through its **storeIdf()** method.

LinkMappingData

The main responsibility of the LinkMappingData class is to update the link to filename HashMap instance attribute through the **updateLinkMap()** method and then use the **storeLinkMap()** method to store the object as a stream at the end of the full crawl.

WebsiteData

The main responsibility of the WebsiteData class is to store all of the required information from each website page within its attributes for each class instance. It accomplishes this through the public **storePageData()** method call that subsequently calls its multiple private methods within the specified order to first extract the preliminary data with the private **extractData()** method. Afterwards, the **storePageData()** method will store the incomplete data using the public **storeWebsite()** method. Then, the website's complete data is filled in after the crawl by the Crawler class that uses its **get…()** methods, its **storePageRankValue()**

method, its **addIncomingLink()** method, its **setScore()** method, and its **storeTfIdf()** method. The class also specifies how to compare two instances of itself in its **compareTo()** method.

DataTool

The main responsibility of the DataTool class is to keep track of the overall data generated from each link visit and make it accessible to each of the data classes, hence why the methods are static. The **getNumWebsitesVisited()** method allows each individual WebsiteData instance to name itself after which it increments the website counter with the **addNumWebsitesVisited()** method. The get method is also used by the LinkMappingData class to update the website to filename HashMap. This class also allows IdfData to calculate the idf values based on the overall crawl data from all links visited using the **getWordWebsiteFrequency()** and the **addWordWebsiteFrequency()** method.

LoadingTools

The main responsibility of the LoadingTools class is to load the stored data whenever one of its methods is called. These methods are static as they aren't specifically different for any one instance of the class but are instead reliant on different file path parameter inputs. The methods provided in this class are **loadLinkMap()**, **loadIdfData()**, and **loadWebsiteData()** respectively for each object class. Additionally, there is a **loadAllWebsites()** method that is used to create a static list of all crawled data for SearchQuery instances to use.

MatrixTools

The main responsibility of the MatrixTools class is to calculate the result of a matrix multiplication using the static **multMatrix()** method and to calculate the Euclidean distance

using the static **euclideanDistance()** method. Both methods are static as they don't rely on any one specific instance of the class.

ProjectTesterImp

The main responsibility of the ProjectTesterImp class is to provide access to the other classes' functionality in an abstract format. The class uses its **initialize()** and **crawl()** method to perform a crawl from a seed URL and store the data into the crawler-storage directory. This creates many instances of the other classes along the way, but it's abstracted from us. The class also has multiple **get…()** methods for accessing the attributes of a specific website instance that is stored using the methods within the **LoadingTools** class. Finally, the class has a **search()** method that makes an instance of the SearchQuery class and returns a sorted list of WebsiteData objects type cast to the SearchResult interface according to the method's parameters.

SearchQuery

The main responsibility of the SearchQuery class is to provide a list of objects implementing the **SearchResult** interface that is sorted by the most relevant to the query based on the parameters provided within its **searchFor()** method. This method calls many subsequent private methods and interacts with many of the other classes while computing the resultant output. It also contains the IdfData object and a list of all the crawled objects as static variables. This allows it to be efficient in its run time when the **searchFor()** method is called.

SeachAppView1

The main responsibility of the SearchAppView1 class is to specify the javafx layout when an instance of the class is created as well as provide a way for its attributes to be accessed by its

get…() methods and a way for the layout to display new information through its **update()** method.

SearchApp

The main responsibility of the SearchApp class is to create a javafx Stage with the **SearchAppView1** layout being functional. It does this with its **main()** method that calls the javafx **start()** method. This class contains a model attribute of the SearchQuery class to conduct any needed computation. Within the **start()** method, the SearchApp class specifies how to handle user interaction when the search button is pressed using the model and when to update the list views, textbox, and radio button.

Serializable

This interface is implemented by the **WebsiteData, IdfData,** and **LinkMappingData** classes to signal that they can be turned into Object Output Streams for storage.

SearchResult

This interface is implemented by the **WebsiteData** class to ensure it has the required methods to be typecasted into a **SearchResult** object by the **SearchQuery** class when it returns the list of ranked search objects.

SearchView

This interface is implemented by the **SearchAppView1** class to ensure it has the required methods needed from a view class by the controller class, **SearchApp**, to function properly.

Comparable<WebsiteData>

This interface is implemented by the **WebsiteData** class to override the **compareTo()** method and specify that the comparison of two **WebsiteData** instances should be based on their titles' lexicographical ordering.

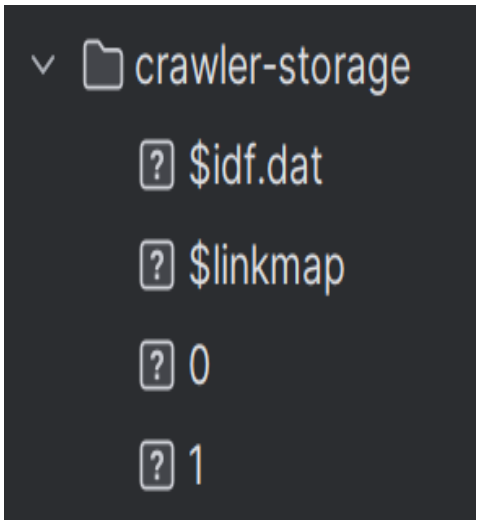
EventHandler

This interface is imported by the **SearchApp** class to override the **handle()** method when the search button is pressed and specify its own behavior.

ProjectTester

This interface is implemented by the **ProjectTesterImp** class to ensure that the test files method calls are identical in name and functionality.

Data Storage Format:



<= All data is saved within the "crawler-storage" directory

<= The complete IdfData object is saved as "$idf.dat" and is accessible in O(1) time.

<= The complete LinkMappingData object is saved as "$linkmap" and is accessible in O(1) time.

<= All visited links are saved as a WebsiteData object with a number as a filename, which is stored in the URL to filename Hashmap within the LinkMappingData object.

This allows us to access the stored Object file of any specific URL through O(1) time.

Discussion on OOP design:

I applied the OOP principles in many areas of my project, which allowed my code quality to be much better than it was for the python implementation of the course project.

Abstraction was applied in many areas of my code such as the Crawler to Website Data relationship. The WebsiteData class was solely responsible for methods relating to its own attributes and storing itself, therefore keeping it fully disconnected from the Crawler class. Whereas the Crawler made specific instances of the WebsiteData class and utilized its methods in an abstract format, completely unaware of the computation behind the methods. Abstraction was even more present when implementing the methods in the ProjectTesterImp class as I simply went under the assumption that the method calls to my other classes worked as intended/described. Thereby increasing code readability, decreasing code duplication, and creating better code modularity.

Encapsulation was applied in every area of my code as all of the classes' attributes were made private to ensure future modifications wouldn't screw up the internal details of the class. Attributes like the outgoingLinks ArrayList and the wordTfIdfMap HashMap lists within the WebsiteData class were made unmodifiable except through proper method calls; only providing get methods for both. Even an attribute like the incomingLinks ArrayList that has an add…() method was properly encapsulated so it would only be modified by explicit and intentional use of the method. Many methods were also encapsulated to eliminate the risk of improper use. For example, the SearchQuery class only provides one public method that subsequently uses private methods in an explicit order of operation to dissuade the risk of the wrong order of method calls. Encapsulation therefore helped in hiding methods and attributes from external classes to make the code much more robust and more easily extendable.

Inheritance was applied to my SearchApp and SearchAppView1 classes as they extended from Application and Pane, respectively. This inheritance made my code more compact as it allowed me to reuse the methods defined in those super classes. Furthermore, inheritance was everywhere in my project as I implemented 6 different interfaces. Each one of these implementations allowed me to be flexible in my own specific code while ensuring that the methods required or overridden would still be functional in the bigger scope of the project. This

use of inheritance in my code allowed me to create code that's more flexible, reusable, and extendable.

Polymorphism was applied to my project mainly in the Crawler class and the SearchApp class. The Crawler class contains polymorphism in its multitude of loops that load and store the page data during and after the crawl. For example, the storing of the incoming links requires it to use the WebsiteData methods during runtime to use a page's outgoing links and populate those links' data with itself. This process is done using polymorphism as the Crawler uses the getOutgoingLinks() method to then tell each link to update its own incoming link with the current link and then to store itself. This demonstrates polymorphism as each instance will update and store itself differently, but the way it does can't be determined until runtime. The SearchApp class also uses polymorphism as the list views won't know exactly how to be updated when the search button is pressed until during runtime when it has the required data to do so. The use of polymorphism in this manner allowed my project to be more generalized and extensible in the class content as the addition of another type of data would work with the SearchApp controller class if it implements the SearchResult interface.

Overall, the use of these OOP principles instead of alternative approaches benefited me in allowing my program to be more robust and flexible, my code to be more reusable and organized, and my classes to be more modular and extensible.

Things that improved run time efficiency:

- Making methods static when they aren't instance specific. Such as the attributes in the SearchQuery class which greatly reduce the runtime when made static instead of loading the same information for each individual instance of the class.
- Using recursion to clear the storage directory.
- Crawling through each link once.
- Storing all of the requestable data within the WebsiteData and IdfData attributes.
- Storing the link to filename HashMap in an object so accessing data is O(1) time.
- Using selection sort for the list of SearchResult object so the sorting can stop at the number of results requested.

Note: In the SearchQuery class, all of the stored WebsiteData objects are statically loaded to calculate their score with any given parameters to the searchFor() method and to be sorted in the SearchResult list. However, this may cause memory problems since too many stored websites could cause Ram to run out while loading them all. Loading all of these objects and not storing them is beneficial for our run time however as the websites all need to be accessed anyways no matter the specific instance.