Analysis Report

Aekus Singh Trehan

No partner

Carleton University

COMP 1406 Z

Dave McKenney

Tuesday, Dec 13th, 2022

Crawler.java

Functionality:

The crawler.java goes through all connected sites from a given string URL and stores information about each site in Page objects. All functionality for the crawler is complete - more depth about functionality within the explanation for each function. There are 7 functions within this file all with their analysis below. Note that there are only 2 public functions because the only functions that should be able to get called from outside should be the fresh_crawl function to initialize the crawler and the crawl function to crawl new data. The rest of the functions are just helper functions to organize each step of the crawling process. There are additionally 7 main variables, totalPages, queue, incoming_links, allWords, read, pagesID, and pages, all of which are private as they do not need to be accessed from anywhere outside the crawler. The totalPages variable is an int value that keeps track of how many websites the crawler has read through for calculations for idf and pagerank values. The queue is a java queue full of URLs that need to be read through, when a link is found on a website, it will check if it hasn't been added to the queue or if it's been read previously and then add it to the queue. incoming_links is a hashmap from strings to ArrayList of strings which keeps track of every site's incoming links. At the end of the crawling process, the incoming_links hashmap will be read through and stored on each respective Page. The allWords dictionary keeps track of every word and how many sites contain each word. This is used for the calculation of the idf after all sites have been crawled. read is a HashSet of strings, representing all sites already crawled. It is a HashSet as there shouldn't be any duplicates and it allows for searching in O(1) time. pagesID is a hashmap from integers to Page objects representing a mapping from the page number to the object with that ID. Another

mapping is the pages hashmap which is a hashmap of strings to Page objects. The Page objects in both pages and pagesID are the same, however, this is a reverse mapping from pagesID as it maps the page object from the URL of the page.

public Boolean crawl(String seed):

The crawl function is the 'main' function of the crawler. This function will take in a string URL (must be an absolute URL, not a relative URL), get all the information about each URL from the webdev read_url function and then send it to the write_info function to be stored for later queries. It will keep repeating for all connected URLs and after going through all connected URLs, it calls on the write_final_info function for all the final calculations to occur before returning a boolean of whether the crawl was successful or not. This is important for the GUI model as the crawler view needs to know whether the crawl happened or not. The time complexity for this function on its own is O(N), where N is the total connected URLs from the initial website.

public void fresh_crawl():

The fresh crawl function is for initializing data before a new crawl. It deletes past crawl data as well as resets the idf hashmap within the Page class. It is a public function as it should be able to be called by any class to reset the data and prepare for a new crawl. The time complexity for this is O(1) because it doesn't do any computation itself. It calls on the private delete function within the crawler to delete the data and the static Page resetIdf function to reset the idf keys (words) and values.

private void delete(File path):

The delete function (as mentioned above) is where the files are deleted. It takes in a File object representing either a file or directory to delete. This function with recursively delete all files and directories within the given File and then delete that file. The time complexity is O(N) where N is the number of sites previously crawled. This makes sense because the file structure (detailed below) has a file for each page that was crawled plus one for idf. This means that this delete function will run N times + 2, one for the idf file and once to delete the directory itself. Note that this is assuming that the File taken in is the data directory as this function is only ever called by either itself or the fresh_crawl function which always sends in the data directory.

private void write_info(String content):

The write info function is for organizing all the info in the 'content' which is a string representing info from a URL. It will store information into Page objects about titles, word tf's, and outgoing links as well as save information in variables for later calculations of incoming links and tfidf and add new URLs to the queue for the crawl function to go through. To make it easier to understand I will separate the time complexity of the write_info function into multiple parts. Firstly for getting and storing the title of the URL the time complexity is O(N) where N is the size of the URL information passed in. This is because the .indexOf() built-in java function will have a time complexity of O(N*M) where N is the length of the string and M is the size of the substring but since we know that the substring is always a length of 7 it just becomes a constant leaving the time complexity to be O(N). The next part is to get all the information from inside the p tags from the site content. The time complexity would be O(NM) where N is the number of p tags there are and M is the number of words in each p tag. This is because the loop will keep going for each p tag (N) and then go through each word in each p tag (NM). It will also

write all the files for each word in that URL but that ends up being less than O(NM) by default since it's only going through each unique word. The next part is the incoming, outgoing and queue links. The queue has the highest time complexity out of all three as they all go through all of the site info looking for links and then when all the links are found, the queue portion will check if each link is in read and queue which is a total time complexity of O(N) where N is the length of the URL content.

private void write_final_info():

The write final info function is for doing all the calculations and saving all information that can only be done after all the connected URLs are crawled. This includes saving incoming links for every URL, saving idf values for each unique word found in the crawl, tfidf values for each word in each file, and getting pagerank values for each URL, before proceeding to write each Page object. The time complexity for this function is $O(MN^2)$. This is from calculating the pagerank vector. The matrix is multiplied by a matrix of the same length adding up to 1 until the distance between the two matrices is less than 0.0001. This is calling on the mult_matrix function whose time complexity is explained later as (N^2), where N is the number of URLs, and repeating that for the number of times it takes till the distance between them is less than 0.0001 (M). The other portions of the write final info function also have their time complexity but none are at the same power as this one, therefore making this one the overall time complexity. The space complexity for the main matrix is described as follows. The matrix is an array of an array of doubles that all have N values in them, N being the number of URLs crawled making the matrix list have a space complexity of $O(N^2)$.

private Double[][] mult_matrix(Double[][] a, Double[][] b):

This function takes in two lists in a list (matrices), a and b, and then goes through all values in A's rows and B's columns, multiplies them and then adds them to the total. The time complexity of this function is $O(N^2M)$ where N is the length of a[0] and M is the length of b.

private Double euclidean_dist(Double[][] a, Double[][] b):

The euclidean distance function takes in two matrices that are 2 dimensional (i.e. two single-row matrices), a and b, and gets the euclidean distance between them. Note that this is very similar to calculating the distance between two points, except it's for higher dimensions. The time complexity of this function is O(N) where N is the length of a[0] (the length of the matrix).

ProjectTesterImp.java

Functionality:

The ProjectTesterImp.java file is the implementation of the ProjectTester interface. This class is for automated testing as well as being the model for the GUI - as it is designed using the MVC paradigm. All functionality for the file is complete. The main functionality is how it is used in automated tests. The ProjectTester interface includes all the necessary methods that it would need to be able to successfully run all the tests and since it implements the interface, it must have all the methods. There are extra functions for the functionality of searching and running the implementations functions correctly but those are all private methods that are only used locally, such as the getTitle and compareTo functions. Additionally, it has the necessary functions to be the model for the GUI. The extra functions are guiCrawl, initializeSearch, and

isInitialized. The guiCrawl is the same method as the crawl method except it will return whether the crawl was successful or not with a boolean true or false. The initializeSearch and isInitialized methods are for the GUI when a crawl is not made from the GUI but search queries are to be made. The tester has a private boolean called initialized which is for checking to see if the dictionary from URLs to Page objects has been made. Because, as explained more in depth below, the pages dictionary contains all the information from the crawl, when you create a new tester object the pages dictionary doesn't go through the crawled data to initialize those values instead waiting for a crawl to happen first. If the crawl has already happened, this is a way to check if the dictionary has been initialized and if it hasn't, it will then initialize with the initializeSearch method.

Page.java

Functionality:

Page.java is a class to create objects to represent a website. It contains methods for setting all the attributes that need to be saved for all sites in the crawl process as well as getters for getting all the attributes in O(1) time when queried. Another thing it has is the static idf hashmap. If we think about idf values for a crawl, they are the same for the entire crawl. This means that if we create a static variable, the variable will be the same for each object. For the static variable, there have to be static methods as well. These include getters and setters as well as a resetIdf method to reset the previous crawl's idf values from the dictionary to make space for the new idf values.

Result.java

Functionality:

The Result.java class extends the SearchResult interface. The purpose of this class is to have all the data that needs to be displayed on the GUI as well as the result type for the search function. The functionality is quite simple as it only has score and title values with getters. You may ask why are setters not needed, and that is because this class is only used in the search function within the project tester. This means that instead of setting each variable separately, the way to create an instance of a Result is by sending in those values together, making for more simple code. The additional method that is included is the toString method. This is because, for each search result in the GUI, whatever is in the toString method, that's what will be displayed for each result. This means that because it's only the title and score, I can format the two in whatever way I think is best.

GUI Instructions

To run the GUI, there are a couple of things that you need to do. Note that these steps are for running the GUI in IntelliJ, if you would like to use another IDE, please search for how to set up JavaFX on that specific IDE. Firstly, you must have JavaFX installed on the device you plan on using to run the GUI. Then, if you go from File → Project Structure → Libraries, you must add the lib folder of your JavaFX downloaded folder. Once you apply that, the errors should go away and you should try running the ProjectApp file. This will then give you another JavaFX-related error, however, if you now go to Run → Edit Configurations…, you should now have a runtime configuration there that you can edit. If not, no big deal, you can add one. Click

the ‘+’, and select Application. Then under Main Class put ProjectApp, add your JavaFX SDK download and then click on Modify options and select the Add VM Options. Then copy and paste the following into the VM options but be sure to edit the path while keeping the quotes to the path to the lib folder on your machine: --module-path "pathtofile"--add-modules javafx.controls,javafx.fxml. Once you’ve done that, you should be able to run the GUI by running the ProjectApp file again. The GUI will open with a crawler screen. There, you will have the option to crawl a website whenever by entering the link of the website and then press crawl. The crawler will let you know when it is done crawling. Then to query the data that has been crawled, you need to go to the search section by clicking the To Search button. Once there, the first search will take half a second longer to initialize the database but after that results will be instant. You enter your query in the Search Query text box and then click on the Boost box if you would like to add boost to the calculations of the score and results will appear once you click the Search button. You can go back to perform another crawl whenever by clicking the To Crawl button.

The GUI has been implemented using the MVC paradigm. This is a way of developing user interfaces that separate the program logic into three components. The model, view and controller. The user sees the view and then interacts with the controller. Such interaction usually results in the model being modified in some way. Then these model changes are reflected in the view of the user interface and the user often gets visual feedback that the model has changed. Let's think of this project. The views are the SearchView and the CrawlView files. The CrawlView can be updated by just updating the label to tell the user if the crawl was successful or not so there isn’t any update method, however in the SearchView, the whole list has to be changed anytime the user enters a new query. To do this effectively, I have added an update

method which takes the output of the model - the ProjectTesterImp - and displays the list of results. So the input will go from the controller, the ProjectApp, to get that the button has been pressed as well as to get the words in the query and whether Boost is on or not and then, this will go to the model to do the computation, which is the ProjectTesterImp, to be sent to the view, which is the SearchView, to update what the user can see.

Overall Design

File Structure:

When thinking of the file structure I was thinking about every calculation I could do in the crawler so that the efficiency of the search would be as fast as possible. The original way that the files were stored was using FileInputStreams and FileInputStreams to save the data in the same way as the 1405 project where each page had its folder with files for everything. This was more efficient than the 1405 project as the files were saved in binary instead of text, taking up more space but still quite inefficient. This is because there is a mapping object that is called which will give the location of the information that is needed, then, that information is received and parsed in whatever way is necessary before being returned. The files are organized this way so that information is organized and that all the calculations can be done fairly efficiently. There is a more efficient way, however. By making a Page class and having instances of the Page class made for each crawled site, you can store all the necessary information for each site (such as tf, tfidf, title, URL, etc.) during the crawl and then at the end of the crawl, instead of writing each attribute separately, you can just write the whole object to the file using java's built-in ObjectOutputStream. This makes it so that all the information for each site is stored with much less space, as it is still in a binary file, and then each site's information is easier to read and store.

Now instead of having a mapping for going from the site URL to the number where it's stored, all of the files can be read before querying the data to get a hashmap of URLs to page objects so that when a query is made, that page's data can be found in O(1) time and the results will be returned in O(1) time as well. The downside to this approach is that it takes up significantly more space that the other because there is no space being used other than what's being returned but now, all the information from the files is being stored in one giant hashmap of objects.

Overall Code Decisions:

I would talk about how I maximized the efficiency of the project by using more efficient abstract data structures and what the thought process was for each one based on time complexity for basic things but just to generalize so that I don't make it all repetitive. I used hashmaps for variables where I was adding and then getting values frequently as these happen at O(1) time for hashmaps, and then when I needed to just have values in a list of some sort when I didn't know the size I would use an ArrayList as they will adapt to the size of the number of objects that are placed in them and then regular arrays when I knew what the length of the array was going to be. Overall I think I used the most efficient data types for most things as there was a lot of thought that went behind each choice but there may be some that are questionable.

The first major code decision was how I opted out of making the matmult its own file this time. I decided to do this because the two functions that would be included are only used from within the crawler. This means that if I make these private methods in the crawler class, then the user doesn't need to know about how the computation in the crawler is happening and these functions are encapsulated from the other classes. The way in which the crawler is executing the crawl function is abstracted making it better than having its own class for those functions which are only used and needed in one spot.

The second thing I decided on was to not include a Search class and a Mapping class and to instead have a Page class. Originally, I had both Mapping and Search classes, however, after changing the way I organized my files, I realized that they weren't needed. The original way I had it set up was that with the original file system, the ProjectTesterImp class would use the Search class which would use the Mapping class. The mapping class would find the location of the info that the user was asking for, which would be used by the Search class to get that information returned to the ProjectTesterImp to return that exact information to the test functions and/or the GUI. Before changing the file structure I realized that the Search class was redundant as it had the same functions as the project tester so the code from the search file to the tester file along with its private helper functions that are used in the computation. After switching the file structure, the search class was querying the page objects for the information which made it so that the amount of code in the search function went from like 15 for each function to be like 5 lines. This was from removing the try-catch for the creation of the FileInputStreams to read the information every time as well as querying the mapping to get the location to read from. Now, the search was initialized by reading all the Page files at once and keeping it in a dictionary for O(1) time queries and the code was just calling on the individual Page object to get the results. By removing the Search and Mapping classes, the time efficiency for all functions in the tester went down because instead of calling the search class to go through the pages dictionary, the tester class does it directly getting rid of the 'middle man'. Then, I realized that if I added an ID section to the page classes, then I would no longer need the mapping class to get the file number and/or URL, rather I could just map a hashmap from URL and/or id to the page object. The use of OOP here in the form of making Page objects and writing those directly to files saves a lot of time as instead of reading the information when it is queried, it saves the information to be

queried in O(1) time and also uses around the same amount of space as the mapping objects are no longer needed. Note, I have included the mapping class in the final submission for you to see, however, it is not necessary for any part of the project and can be removed without any errors being created.

The last major decision I made was to use the project tester as the model for the GUI. This to me made sense as the GUI needed to use both the crawler and the search abilities, so rather than creating both search (before it was removed from the design) and crawler objects within the controller, it made more sense to have the thing that connected the two. The alternative would be to have another class which called on the search and crawler classes directly, however, this is more inefficient as the tester already has the code to do this and through OOP, can be used in the same way rather than having redundant code. This, however, meant that there were additional functions that needed to be created so that the GUI could get all the information it needs. This decision made it so there isn't any duplicative code in the project and all code is reused as much as is needed.