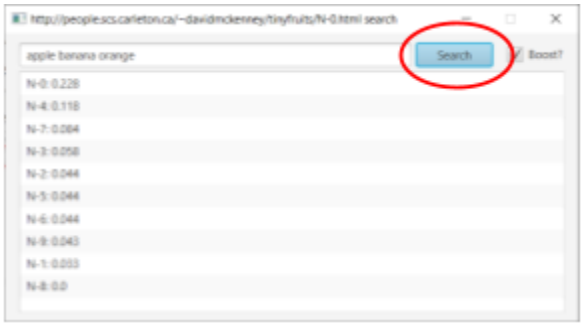
Part A: Running the GUI

To run the program, open the SearchController file in the IDE, and run the start method. It will then crawl the provided seed URL(a variable in the code file), then show the window. Depending on the given seed URL(currently set to http://people.scs.carleton.ca/~davidmckenney/tinyfruits/N-0.html), it may take a while to display the search window.

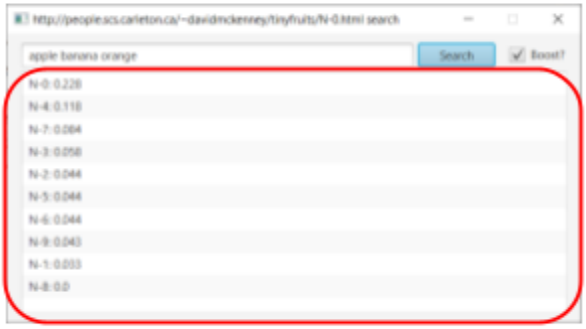
When the window is displayed, the user is able to type their query in the search box(dia.1) and can press the 'enter' key or the button labelled 'search' to search the crawled pages(dia.2). It will then display in the bottom box the top 10 results of the search(dia.3), the amount of which is adjustable in the SearchModel class.

There is also a checkbox labelled 'Boost?'(dia.4). When selected, the scores of each page will be multiplied by their page rank. The user must press the search button again, as the results will not automatically update. Similarly to the initialization process, the first time 'boost' is applied, the program may take a moment to calculate the page rank of each page.

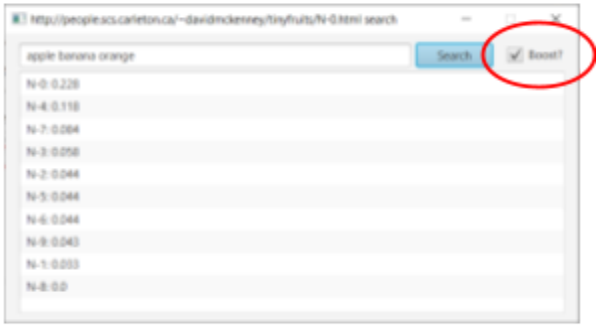
Dia. 1: The Search Box



Dia. 2: The Search Button



Dia. 3: The Search Results



Dia. 4: The Boost Button

Part B: Project Functionality

All required functionality should be included. Some variables are final, and may be changed before running for testing. When a file needed to search is deleted or is otherwise compromised in some way, a message will print to the console stating that an error has occurred, tagged with an ID value that can be used to search for the error location.

Note: due to the nature of the IntelliJ IDE, maven was forced upon the project, and you **may** need to import a package consisting of the file name. I have no idea what a package is, but I think it just helps locate the file. Also, you should follow the procedure of locating your JavaFX folder.

Part C: File Responsibilities and Use

Crawler

The 'Crawler' class stores the methods and constructors to create every file required from a seed URL and the URLs it links to, resetting the files each time. It parses each using methods, storing each piece of information in a 'Page' object. When it has finished crawling every URL once, it then creates the files needed with 'Page' objects and assigns them a unique ID, along with creating other miscellaneous files.

Page

A 'Page' object stores every piece of information needed; the words, title, and incoming and outgoing links. This object is used during the crawl. In addition, the crawler can set each 'Page' a unique pageID, giving an identifier for later searches that may be done in a future launch.

ProjectTester

The 'ProjectTester' interface is provided by the course(thanks Dave) and is used for the ProjectTesterImp.

ProjectTesterImp

The 'ProjectTesterImp' implements the 'ProjectTester' interface, and allows for interaction between the provided test resources and the program.

SearchModel

Following the MVC paradigm, 'SearchModel' can be seen as the backend of the program. It has all the methods and constructors to search for a query

and also is able to calculate page ranks. It accepts a query and returns the top 10 search results for that query.

SearchView

As with the MVC, 'SearchView' provides the GUI associated with the Model. It has the required aspects laid out by the requirements.

SearchController

Connecting the model and view, 'SearchController' gets input from the view and feeds it to the model, after which updates the view.

SearchResult

The 'SearchResult' interface is provided by the course(Dave) and is used for the 'SearchResultObject'.

SearchResultObject

Similar to the 'Page' object, 'SearchResultObject' is very similar, albeit simplified. It acts similarly to a 'Pair' but allows for extensibility in the future.

WebRequester

Provided by Dave, it takes an URL and converts it to a string.

Part D.1: How OOP is Used(and how it Improves)

OOP is a major design point in Java, and the crawler provided makes use of many aspects of JAVA to aid in making the results object-oriented.

At a glance, the best example of the usage of OOP is the separation of classes and objects. As an example, the crawler makes use of objects to crawl, and in itself is a class. Code is broken down into private methods that aid in parsing data. Variables are set so that they are hard to accidentally change, encapsulating them in the classes.

The crawl's main function, the 'crawl(seedURL)' method, makes all the files needed to perform future searches. In this crawl, methods used to parse data are used as well to read and extract information from web pages, accepting the contents of a page as input. Furthermore, a reset method is used to reset and the previous crawl easily. Splitting these tasks across methods makes the crawler far more extensible; Without knowing the inner workings, a

future developer is able to add additional parsing easily, add more files in case of more storage or more detailed information, or tweak one specific aspect of the crawl(seedURL) function.

To aid in this process, the 'Page' object stores all needed information, and later is accessed to make a file consisting of the information stored. Currently, a 'Page' stores titles, links, and words of a URL. However, it is also simple to add variables to store any other information in the future, such as the number of users that have visited the page, for example. A 'Page' object also stores a unique page ID number. In theory, accessing a page's information by URL name is possible, however having a pageID quantities the URL into a number, and makes it easier to identify which directory represents which page, making it more robust. An integer is much less likely to be mistaken than a string for the program.

Moving on from the crawler, the search classes take up what can be seen as the latter half of the program. After crawling the search function, taking a string as the query, the 'SearchModel' class searches through the files and finds the top 10(or however many are displayed) search results. These search results are represented by a 'SearchResultObject', storing a title and a score. It implements the given 'SearchResult' interface given by the course(thanks Dave). It can be noted that, for the specifications, a 'Pair<String, Double>' or a 'TreeMap' object can be used instead, nullifying the need to write the 'SearchResultObject' class. However, thinking about extensibility, it is the correct course of action to create this class. Another motivator to using a 'SearchResultObject' was to simplify the code for the 'ProjectTesterImp' class, whereby using polymorphism, a list of SearchResultObjects can be treated as the 'SearchResult' interface, and need no further coding.

As stated in Part B, the 'SearchModel' class follows the MVC paradigm, with 'SearchView' and 'SearchController' representing the other two components. By separating these classes, it becomes simpler to design the UI without affecting the model, and vice versa. This, like other design choices, aid in extensibility.

As a final note, some more general aspects will be explained. Each class encapsulates its components, making the program more robust. In addition, there was no implementation of many interfaces, save for a couple provided by the course(Dave). I have noted it is possible to have a 'Page' implement the 'SearchResult' interface, but have decided not to as it wouldn't provide much value nor make the most sense. The reset() methods found in the 'Page' and 'Crawler' classes are static, as it prevents any chance of a duplicate "resources" file, and allows the currentPageID to remain a static variable.

The same can also be said for the 'SearchController' seedURL and Crawl variables, however for a different purpose.

Part D.2: Runtime Efficiency Notes

Compared to the Comp1405 project, the runtime complexity has relatively stayed the same, aside from some small changes in method calls, the use of a GUI and the use of objects instead of dictionaries. They all have a maximum runtime complexity of O(1), and as such, the complexity between the two projects should have stayed the same. At the very most, I made the program more efficient by condensing multiple method calls into one.

It can be of note that in exchange for a more efficient program, I have made methods 'addMatrixScalar' and 'multMatrixScalar' access the adjacency matrix needed to calculate page rank individually. I made this decision for future extensibility, along with deciding it is worth the extra few seconds of computation during the method's first call.

Part D.3: How Data is Saved

At first, a "resource" file is created in the project folder, populated by n folders representing each page's information, and 3 other folders that store each URL's pageID, page rank, and the number of pages each word appears in. Of the n pages, they are each named with a number from 1 to n, corresponding to each page's ID number(found in URLToID.txt). This is very similar to my Comp 1405's method of storing, however puts all files in "resources" to visually make the file system neater, along with making it easy to simply delete the "resources" file to reset the crawl.

For the 3 “miscellaneous” files, data is stored in the following way in a more general form:

Information Name1
Information Number1
Information Name2
Information Number2
Etc.

For page rank and page id, the name is the page URL, and the number the pagerank or ID. The wordcount file has the word followed by its number of appearances.

For each page, each of its 4 aspects are stored in the same way, the contents of which depends on the file:

Information Name1
Information Name2
Information Name3
Etc.

For incoming and outgoing links, the name is the URL. For words, it’s the list of the page’s words. For the title, it is simply one line of the title. As stated before, these files are stored in a directory named after the page’s ID number.