Information Retrieval System

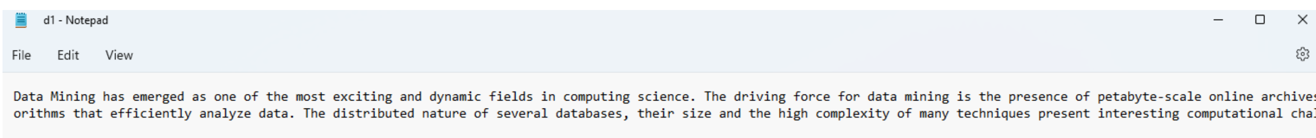
CSI 2290 Final Project

Aaron Kenward, Cameron Vogeli, Eric Daulo, Weiye Shi, Carlos Buenaventura, John Musa
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: akenward@oakland.edu, cmvogeli@oakland.edu, edaulo@oakland.edu , wshi@oakland.edu
caherrerabuenav@oakland.edu, johnmusa@oakland.edu

This project was designed to test students' abilities to read inputs from a file, parse the data, censor unwanted characters, and then tokenize them into a counted list file. This was to require skill in all disciplines of the course so far. After the dust had cleared with completing our code, the product was a complete tokenized file with words listed alphabetically and their instances counted. During this process, important functions like qsort and strcmp were discovered and relied upon to make quick work of the task. We concluded from this project that these kinds of algorithms are difficult and the people who write them every day in industry are extremely talented. The project was a mental, intellectual, and social exercise for all of us.

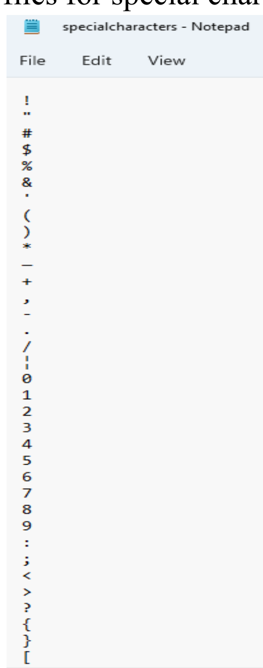
Files with Text

As an example of the text files we were working with, this is the way d1 appears:


d1 - Notepad
File Edit View

Data Mining has emerged as one of the most exciting and dynamic fields in computing science. The driving force for data mining is the presence of petabyte-scale online archives
orithms that efficiently analyze data. The distributed nature of several databases, their size and the high complexity of many techniques present interesting computational chal

This text file is full of characters, text or not, and we have to separate and count these. There is also files for special characters and stop words, like this:


specialcharacters - Notepad
File Edit View

!
"
#
$
%
&
'
(
)
*
_
+
,
-
.
/
¦
0
1
2
3
4
5
6
7
8
9
:
;
<
>
?
{
}
[

Reading and Writing Data to Files

In the main function, we create a FILE* for each input file and use the fopen function to open the input files in read mode. We also tested if it was possible to open each file. If one of these files could not be opened, we write a "couldn't open file" statement to stderr and exit the program with a 1. If they can be opened, we call our init function and pass each FILE* along with the path of the output file. At the end of our init function, we use fclose to close the FILE* passed to this function.The input and output files can be found under their own respective directories, all of which are under the assets folder.

To print each list to a file, we call the printToFile function called in init and pass the output file name and path as a const char*. In printToFile, we create a new FILE* and use fopen to open the output file in write mode. If the file cannot be opened we let the user know using fprintf. Otherwise, we iterate through the list of words and print each string to the file, separated by a new line.

When we first began the project, we didn't create the FILE* in the main function. We passed the file names as const char* to each function in the project. From there, each function had to create a new FILE* that would be closed at the end of the block. For efficiency and do-it-once principles, we opted to instead create the FILE*'s once and keep them open until we are finished with them.

Structures

It was decided to use two main structure types for this project: Str and Sqlist. Str was used for storing the words themselves and Sqlist was used for tokenizing the words and storing list properties. Str possesses two pieces of data: a character array to store the word and a double value to store the number of times it appears. Sqlist is a bit more abstract and firstly stores the corresponding Str structure previously defined. It also contains value for the length which defines the size of the list for the words.

Initialize List Structure

In order to utilize the list for counting and sorting the words, we first must initialize the values for this so it can run properly. This begins by creating a new structure of Sqlist type and passing it as a parameter to the initSqlist function. The other parameter is an integer value for the maxsize. For this function, the first step is to dynamically allocate memory for the list we will use. If the allocation is successful and passes our verification, then the listsize field of the new structure L is initialized to the maxsize value and the length field is set to 0.

Printing Words into the List

After initSqlist is called, we then have to read the data from the input files into the list. To do this, we have a function called getWords. The function starts by creating character arrays (strings): one for storing the text from a file [called string], one as a buffer for storing the string before stop words and special characters are removed, and one for storing the stop words. The function runs through a loop using fgets until a null condition is reached (end of file).

In the loop, it first uses strcpy() to copy the contents of the buffer into string. This string is then delimited into words by using strtok() and setting spaces as a delimiter. A character-type pointer is then set equal to this delimited string. After this, a while loop is used to read through every line of the stopwords.txt file and it uses strcmp to see if any entries in the list are stop words. If they are not, then they get committed to L to be parsed and tokenized, which we will discuss later.

Parsing and Removing Special Characters

A function called parseWords was used for this task. This function works by copying the data previously obtained through getWords into a buffer array. We then use a function called strtok which allows you to add delimiters as parameters, and we add all the special characters to that parameter. A loop is then used to remove the special characters systematically. It then adds the delimited word to the data field in L where it originally read from, and then iterates to the next data field to restart the cycle.

Alphabetizing the List

Before we alphabetize the words in our list, we first needed to make the words lowercase. We achieved this by calling the makeLowercase function. In that function, we traverse through the entire list of words. In each word element, we iterate through its characters one by one, checking if the current character is uppercase. If it is, we first cast the character as an integer and add 32 to its value. Then, we cast it back into a char and assign it to the current character.

Now that all the words are lowercase, we alphabetize the list of words using qsort. We do so by passing to it the list of words, the length of the array, the size of the list of the words in bytes and the compare function. In the compare function, we use strcmp to compare two words and return a value back to qsort.

Finding Frequency and Weight

To determine frequency and weight of each word, we call getFrequency after we have gotten the words from the file, put them into an array, made each of them lowercase and alphabetized them. In the getFrequency function, we first find the frequency of each element in the list. We do this by creating two for loops and strcmp. Each indexed element is compared to the next element in the list using strcmp. If strcmp returns a 0, we write the total number of times each element occurs after this element. When the inner loop ends, we check if the number of times each element occurs is the greatest we found so far. If so, it becomes the new maximum frequency in the list.

After we are done iterating through the entire list, we go through it once more. We again use two loops, but this time if the current word is equal to the next word, we record the current word's frequency and skip to the next different word in the list. Before we do, we use fprintf to print the current word, its frequency and its weight.

The current code is the first edition of a function that calculates frequency and weight. Since the frequency is best found when the array is already sorted, this is one of the last functions we wrote.

Conclusions

After the conclusion of our work, many observations and areas for improvement could be discerned. We noticed that we had to find some external documentation to complete our work. This was a new process for some but the result benefited our work.

There is also the topic of room for improvement. When one runs the program, they can see 2 instances of special characters that slipped past our algorithm, so there can be refinement there. Some things could have also been made into bespoke functions instead of being called multiple times, which would've made the code more fail-proof. All in all, the code ran well and with a satisfactory result.

Compiling and Running

To compile the project, navigate to the src/ folder located in the project's root directory. In that folder, you should see a file called main.c. Compile this file using the normal means, naming the output file anything you wish. To run the file, execute the newly created binary. You should get an output similar to the following:

```
~/De/csi2290-finalproject/src on root !4 ?1  gcc -o main main.c && ./main  ✔ at 17:28:57

============================================================================================
Current file: ../assets/outputs/Tokenizedd1.txt
============================================================================================
Word: $10               Frequency: 1.000000             Weight: 0.062500
Word: a                 Frequency: 2.000000             Weight: 0.125000
Word: ad                Frequency: 1.000000             Weight: 0.062500
Word: algorithms                Frequency: 1.000000             Weight: 0.062500
Word: analysis          Frequency: 1.000000             Weight: 0.062500
Word: analyze           Frequency: 1.000000             Weight: 0.062500
Word: and               Frequency: 6.000000             Weight: 0.375000
Word: archives          Frequency: 1.000000             Weight: 0.062500
Word: around            Frequency: 1.000000             Weight: 0.062500
Word: artificial                Frequency: 1.000000             Weight: 0.062500
Word: as                Frequency: 1.000000             Weight: 0.062500
Word: assimilate                Frequency: 1.000000             Weight: 0.062500
Word: at                Frequency: 1.000000             Weight: 0.062500
Word: availability              Frequency: 1.000000             Weight: 0.062500
Word: be                Frequency: 1.000000             Weight: 0.062500
Word: been              Frequency: 3.000000             Weight: 0.187500
Word: billion           Frequency: 1.000000             Weight: 0.062500
Word: bits              Frequency: 1.000000             Weight: 0.062500
Word: commercial                Frequency: 1.000000             Weight: 0.062500
Word: complexity                Frequency: 1.000000             Weight: 0.062500
Word: computational             Frequency: 1.000000             Weight: 0.062500
Word: computing                 Frequency: 1.000000             Weight: 0.062500
Word: concept           Frequency: 1.000000             Weight: 0.062500
Word: consequently              Frequency: 1.000000             Weight: 0.062500
Word: contain           Frequency: 1.000000             Weight: 0.062500
Word: data              Frequency: 9.000000             Weight: 0.562500
Word: database          Frequency: 1.000000             Weight: 0.062500
Word: databases                 Frequency: 2.000000             Weight: 0.125000
Word: detect            Frequency: 1.000000             Weight: 0.062500
Word: distributed               Frequency: 1.000000             Weight: 0.062500
Word: driving           Frequency: 1.000000             Weight: 0.062500
Word: dynamic           Frequency: 1.000000             Weight: 0.062500
Word: efficientl                Frequency: 1.000000             Weight: 0.062500
Word: emerged           Frequency: 1.000000             Weight: 0.062500
Word: enterprises               Frequency: 1.000000             Weight: 0.062500
Word: excess            Frequency: 1.000000             Weight: 0.062500
Word: exciting          Frequency: 1.000000             Weight: 0.062500
Word: expected          Frequency: 1.000000             Weight: 0.062500
Word: family            Frequency: 1.000000             Weight: 0.062500
Word: few               Frequency: 1.000000             Weight: 0.062500
Word: field             Frequency: 1.000000             Weight: 0.062500
Word: fields            Frequency: 1.000000             Weight: 0.062500
Word: for               Frequency: 3.000000             Weight: 0.187500
Word: force             Frequency: 1.000000             Weight: 0.062500
Word: form              Frequency: 1.000000             Weight: 0.062500
Word: has               Frequency: 1.000000             Weight: 0.062500
Word: have              Frequency: 3.000000             Weight: 0.187500
Word: hidden            Frequency: 1.000000             Weight: 0.062500
Word: high              Frequency: 1.000000             Weight: 0.062500
Word: hoc               Frequency: 1.000000             Weight: 0.062500
Word: in                Frequency: 5.000000             Weight: 0.312500
Word: information               Frequency: 1.000000             Weight: 0.062500
Word: intelligence              Frequency: 1.000000             Weight: 0.062500
Word: interesting               Frequency: 2.000000             Weight: 0.125000
Word: intersection              Frequency: 1.000000             Weight: 0.062500
Word: is                Frequency: 2.000000             Weight: 0.125000
```

Scrolling down, you should see each file's words along with their frequency and weights. Each input file is clearly marked as shown above. The output file of each input text file is located under

/assets/outputs/. Opening up “Tokenizedd1.txt” after executing the program should produce the following:

```


$10
a
a
ad
algorithms
analysis
analyze
and
and
and
and
and
and
archives
around
artificial
as
assimilate
```

The other output files can be viewed in a similar manner.