CS214 Spring 2021
Project II

David Menendez

Due: Friday, April 16, 2021, at 11:00 PM (ET)

This is a group project. You may work alone, or with a partner. Your partner may be in a different section.

If you are working with a partner, both partners must declare the partnership on or before Wednesday, April 7, using the form at `https://forms.gle/GscYB3dsTTBUMLFu6` or by e-mailing your TA.

1 Summary

Write a program that analyzes a set of files and reports their similarity using the method described below. Your program will use two or more threads to read directory entries and analyze the word frequencies of text files, then compute the Jensen-Shannon distance between the word frequencies of pairs of files.

2 User interface

Your program will be given the names of one or more files and directories, and up to four optional arguments specifying program parameters. The file names and directories are used to determine the set of files to analyze. File names given as arguments are added directly to the set of files. For each directory, your program will add any files whose names have a particular suffix to the set, and recursively traverse any subdirectories.

For each (unordered) pair of files in the analysis set, your program will output the Jensen-Shannon distance between their word frequencies.

For example:

```
$ ls bar
baz.txt quux spam/
$ ls bar/spam
eggs.txt bacon.c
$ ./compare foo bar
0.03000 foo bar/baz.txt
0.12000 foo bar/spam/eggs.txt
0.80000 bar/baz.txt bar/spam/eggs.txt
```

Table 1: Parameters specified by options

Parameter	Option	Argument Type	Default
Directory threads	`-d`N	Positive integer	1
File threads	`-f`N	Positive integer	1
Analysis threads	`-a`N	Positive integer	1
File name suffix	`-s`S	String	`".txt"`

In this example, the arguments are `foo` (a file) and `bar` (a directory). One file in `bar` matches the default suffix (`".txt"`) and is added to the file set. The subdirectory `bar/spam` is also traversed, and its files that end with the suffix are added. This results in the file set `foo`, `bar/baz.txt`, and `bar/spam/eggs.txt`. (Note that `foo` does not need to end with the suffix, because it was explicitly given by the user.)

For each pair of files in the file set, the program outputs the JSD and the (path) names of the files being compared. Comparisons are printed in decreasing order of combined word count (that is, the total number of words in both files). Note that pairs are unordered, so the output will include comparisons of `foo` and `bar/baz.txt` or `bar/baz.txt` and `foo`, but not both.

2.1 Regular arguments

Your program will take one or more regular arguments, which must be the name (or a path to) a file or directory.

Each file name is added to the set of files to examine.

Each directory name is added to the set of directories to traverse. For each directory in the set, your program will read through its entries. Any file whose name ends with the file name suffix is added to the set of files to examine. Any directory is added to the set of directories to traverse. Entries that are not directories and that do not end with the specified suffix are ignored.

Exception Any entry in a directory whose name begins with a period is ignored.

2.2 Optional arguments

Your program takes up to four optional arguments, which are used to set program parameters. For simplicity, we will assume any argument that begins with a dash (`-`) specifies an option. The first character following the dash indicates which parameter is being specified, and the remainder of the argument gives the value for that parameter. Table 1 gives the syntax for the four options, and the default values for the parameters of the options are omitted.

For simplicity, you may require all optional arguments to occur before the regular arguments in the argument list, or allow them to be intermixed. You may require that an optional argument occur at most once, or allow later arguments to override earlier ones.

You must allow options to be given in any order.

Your program must detect any invalid optional arguments and halt with an error message. Possible errors are an invalid option (e.g., `-x`), or an invalid or missing argument (e.g., `-f0` or `-d`).

Note The file name suffix may be an empty string, which would be indicated by the option `-s`.

Table 2: Word frequency distribution example

Text: `I can't understand thieves' cant.`

Word	Occurrences	Frequency
cant	2	0.4
i	1	0.2
thieves	1	0.2
understand	1	0.2

3 Operation

Your program will compute the word frequency distribution for each file in a set of files. For each pair of files in this set, it will compute the Jensen-Shannon distance (JSD) between the distributions for those files

3.1 Word frequency distribution (WFD)

The *frequency* of a word is the number of times the word appears divided by the total number of words. For example, in the text "spam eggs bacon spam", the frequency of "spam" is 0.5, and the frequencies of "eggs" and "bacon" are both 0.25.

The *word frequency distribution* (WFD) for a file is a list of every word that occurs at least once in the file, along with its frequency. Ignoring rounding errors, the frequencies for all the words will add up to 1.

Tokenizing To compute the WFD for a file, your program must read the file and determine what words it contains. For this project, we define a word to be a sequence of word characters, where word characters include letters, numbers, and the dash (hyphen). Words are separated by whitespace characters. Other characters, such as punctuation, are ignored.

To compute the WFD, keep a list of every word found in the file and its count. Each time a word is encountered, look for it in the list and increment its count or add it with count 1. Once every word has been read, divide the count for each word by the total number of words.

Words are case-insensitive. The simplest way to handle this is to convert words to upper- or lowercase while reading.

Table 2 shows the WFD for a short example file. Note that "I" has been converted to lowercase and that "can't" and "cant" are treated as the same word (because the apostrophe is ignored).

Data structure You will need to maintain a list of mappings between words and counts (later frequencies). You are free to choose any data structure, but you may find a linked list to be a good balance of efficiency and simplicity. It is recommended that you choose a structure that allows you to iterate through the words in lexicographic order.

Your program must not assume a maximum word length. Instead, word storage will be dynamically allocated.

(a) F_1

hi	0.5
there	0.5

(b) F_2

hi	0.5
out	0.25
there	0.25

(c) $\overline{F}$

hi	0.5
out	0.125
there	0.375

$$
\begin{aligned}
KLD(F_1||\overline{F}) &= 0.5 \cdot \log_2\left(\frac{0.5}{0.5}\right) + 0.5 \cdot \log_2\left(\frac{0.5}{0.375}\right) \\
&\approx 0.5 \cdot 0 + 0.5 \cdot 0.415 \\
&\approx 0.2075 \\
KLD(F_2||\overline{F}) &= 0.5 \cdot \log_2\left(\frac{0.5}{0.5}\right) + 0.25 \cdot \log_2\left(\frac{0.25}{0.125}\right) + 0.25 \cdot \log_2\left(\frac{0.25}{0.375}\right) \\
&\approx 0.5 \cdot 0 + 0.25 \cdot 1 + 0.25 \cdot -0.585 \\
&\approx 0.1038 \\
JSD(F_1||F_2) &\approx \sqrt{\tfrac{1}{2}0.2075 + \tfrac{1}{2}0.1038} \\
&\approx 0.3945
\end{aligned}
$$

Figure 1: Computing the JSD for two files

3.2 Jensen-Shannon distance (JSD)

The Jensen-Shannon distance (JSD) is a symmetric measure of the similarity of two discrete distributions. We will write $F_i(x)$ to mean the frequency of word x in file i. Note that $F_i(x) = 0$ if x does not appear in file i.

We define the *mean frequency* for a word to be the average of its frequencies in the two files.

$$\overline{F}(x) = \tfrac{1}{2}(F_1(x) + F_2(x)) \tag{1}$$

Next we compute the Kullbeck-Leibler Divergence (KLD) between each file and the mean.

$$KLD(F_i||\overline{F}) = \sum_x F_i(x) \cdot \log_2\left(\frac{F_i(x)}{\overline{F}(x)}\right) \tag{2}$$

This is computed by adding the frequencies of each word in the file scaled by the base-2 logarithm of the frequency divided by the mean frequency for that word. The result will be a value between 0 and 1.

Finally, the JSD is the square root of the mean KLD between the files and the mean distribution.

$$JSD(F_1||F_2) = \sqrt{\tfrac{1}{2}KLD(F_1||\overline{F}) + \tfrac{1}{2}KLD(F_2||\overline{F})} \tag{3}$$

Figure 1 shows the steps to compute the JSD for two files containing "hi there hi there" and "hi hi out there", respectively. Figures 1a and 1b give the WFD for the files, and fig. 1c shows their mean WFD.

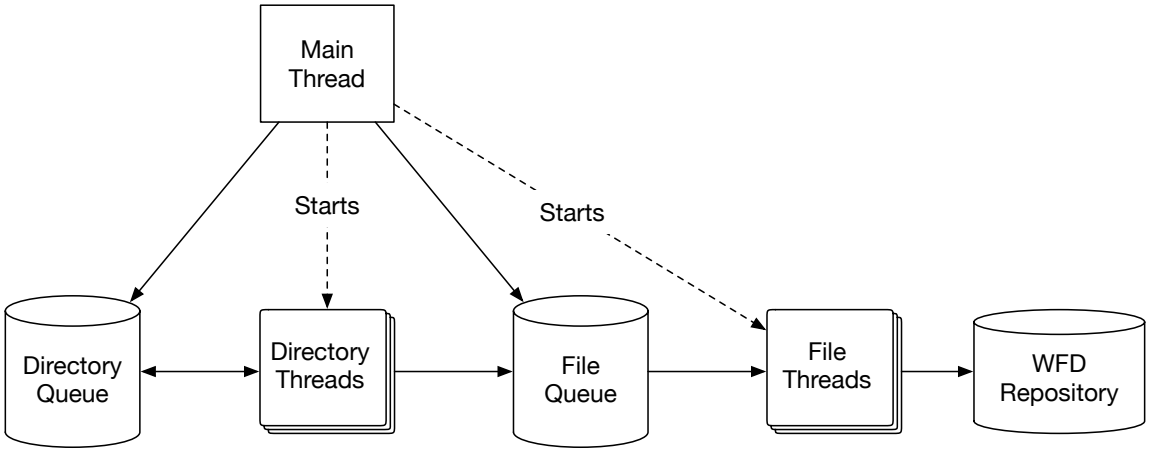

Figure 2: Organization of the collection phase

Implementation If you have chosen a WFD structure that allows you to iterate through the word lists in alphabetical order, it is simple to compute the JSD using simultaneous iteration for both lists. Keep a running total of the KLD for both files. If you encounter a word appearing in both lists, compute the mean frequency and then add the scaled frequencies to both running totals. If word appears in one list but not the other, treat its frequency as 0 in the file where it does not appear. Once all words have been considered, compute the JSD.

4 Program organization

Your program will operate in two phases. The collection phase recursively traverses directories and reads files to find the WFD (see section 3.1) for each requested file. The analysis phase computes the JSD (see section 3.2) for each pair of requested files.

Both phases are multithreaded and use synchronized data structures to perform their tasks.

4.1 Collection phase

The collection phase uses two groups of threads to find the WFD for each file in a set of requested files. It uses two synchronized queues to keep track of the directories and that need to be examined.

Figure 2 shows the organization of the collection phase. The components of the collection phase are:

Directory queue Contains a list of directories that have been seen but not yet traversed. This will be a synchronized queue (or stack) that must be unbounded.

File queue Contains a list of files that have been seen, but not yet examined. This will be a synchronized queue (or stack) that may be bounded or unbounded.

WFD repository Once a file has been examined, its name, count of tokens, and WFD is added to this structure. This may be any synchronized data structure. See section 3.1 for details of how to represent the WFD.

Main thread The main thread creates the queues and repository. For each regular argument, it determines whether it is a file or directory and adds it to the appropriate queue. Based on the program parameters (see section 2.2), it starts the requested number of file and directory threads. Once all the file threads have completed, it proceeds to the analysis phase.

Directory thread Each directory thread is a loop that repeatedly dequeues a directory name from the directory queue and reads through its directory listing. Each directory entry is added to the directory queue. Each non-directory entry that ends with the file name suffix is added to the file queue.

A directory thread will finish when the queue is empty and all other directory threads have finished or are waiting to dequeue.

File threads Each file thread is a loop that repeatedly dequeues a file name from the file queue, tokenizes the file and computes its WFD, and adds the data for that file to the WFD repository.

A file thread will finish once the queue is empty and all directory threads have finished.

Note that the working directory is shared by all threads, so the directory threads will not be able to use `chdir()`. Instead, they will need to concatenate the directory name and the file or subdirectory name to get a path. Assume that all file and directory names given as arguments are relative to the working directory (that is, you do not need to modify or examine them).

We will discuss synchronized queues in class, but you will need to customize them somewhat to work here. When the directory queue is empty, you will need to keep track of the number of threads waiting to dequeue. Once the last thread tries to dequeue, you can be certain that no new directories will be found. At this point, you can set a flag indicating that traversal has ended and wake up the directory threads so that they can terminate. Similarly, file threads should terminate if the file queue is empty and traversal has ended (meaning no more files will be added).

Exercise Why is it safe for the file queue to be bounded, but not the directory queue?

Error conditions If any file or directory cannot be opened, report an error and continue processing. It is sufficient to call `perror()` with the name of the file or directory to report the error. You may also exit with status `EXIT_FAILURE` when the program completes.

For unexpected errors, such as failure to create threads or lock and unlock mutexes, your program may report an error and terminate immediately.

4.2 Analysis phase

The analysis phase computes the JSD (see section 3.2) and combined word count for each pair of files listed in the WFD repository (see section 4.1). It divides this work among one or more analysis threads, as indicated by the analysis threads option (see section 2.2).

Each analysis thread is given a portion of the file pairs and computes the JSD for each pair. Once all comparisons have been made, the main thread will sort the list of comparisons in descending order of combined word count and then print the results.

Data structure For each comparison, we will need the names of the files being compared, the combined word count, and the JSD. If there are n files, there will be $\frac{1}{2}n(n+1)$ comparisons, so it is feasible to create an array of structs, have each thread write to non-overlapping portions of the array, and finally sort the array using `qsort()`.

Division of labor The simplest way to divide the work among the analysis threads is to give each thread a roughly-equal portion of the comparison set to compute. If there are 30 files and 5 analysis threads, then each thread will compute 93 of the 465 comparisons. (Naturally, we must be able ot handle cases where the number of comparisons does not divide evenly.)

While this method divides up the number of tasks (roughly) equally, it assumes that all comparisons will take equally long to compute. Another method to divide the work is to keep track of which comparison needs to be computed next and have each analysis thread take the next comparison needed. This will reduce the possibility that some threads will finish early while other threads have significant amounts of work left to do.

Error conditions If the collection phase found fewer than two files, report an error and exit.

Any numeric errors occurring in this phase (e.g., word frequencies outside the range [0,1]) indicate problems in the collection phase. You may choose to check these using `assert()`.

5 Advice

Don't panic! While this assignment has many components, the components themselves are relatively simple. If you begin before we have gone over synchronized queues in class, you can still design your data structures and begin work on the analysis phase.

While use of the C file IO functions (e.g., `fopen()`) is permitted for this assignment, you will not find them any easier or more convenient than using `open()` and `read()`. Using `getline()` will just add complexity to your program for no real benefit.

Recall that `printf()` and `fprintf()` are synchronized, meaning they can be safely used in different threads. During debugging, you may find it helpful to print log messages describing what actions your threads are doing. Use the `DEBUG` macro trick to easily enable or disable these messages as needed.

Always check for errors. Functions like `malloc()` or `pthread_mutex_lock()` never fail under normal circumstances, which means you definitely want to be informed if they do fail—it may indicate a logic error in your program! If writing the error checks becomes tedious, define a macro to write them for you. How best to handle error conditions is a design choice, but the usual thing to do when something "impossible" happens is to report an error and terminate the program.

That being said, checking `printf()` for an error result is probably overkill.

6 Submission

Submit a Tar archive containing your source code, makefile, and a README. The README must contain:

- Your name and NetID[1]
- Your partner's name and NetID, if you worked with a partner.
- A brief description of your testing strategy. How did you determine that your program is correct? What sorts of files and scenarios did you check?

Do not include executables, object files, or testing data. Before submitting, be sure to confirm that your archive contains the correct files!

For safety, both partners should create and submit an archive.

7 Grading

Your program will be scored out of 100 points: 50 points for each part. Your grade will be based on test cases and on manual examination. Testing will be performed on iLab machines, so be certain that your code will compile and execute on the iLab!

Your code should be free of memory errors and undefined behavior. We may compile your code using AddressSanitizer, UBSan, Valgrind, or other analysis tools. You will lose points if these tools report memory errors, space leaks, or undefined behavior. You are advised to take advantage of AddressSanitizer and UBSan when compiling your program for testing.

Late submissions This project may be handed in up to 72 hours past the due date. Late submissions will be penalized according to the following scale:

Up to	Penalty
30 minutes	5%
2 hours	10%
24 hours	25%
72 hours	50%

Plan to complete your assignment early! The late submission grace period is provided for emergency use and for last-minute salvage. Build slack into your schedule so that unexpected delays or difficulties do not result in a late submission.

[1] If you prefer not to share your NetID with your partner and/or Google, contact your TA and request a unique identifier to use for this project.