CS 232 Introduction to C and Unix

Project 3: Building a Web Search Engine

Crawler is due Sunday, April 9 by 11:59pm

Complete project is due Thursday, April 27 by 11:59pm

This is a substantial project. Start as soon as it is released. If you get stuck, feel free to come to office hours. I'm here to help you, but I can't help if you don't ask.

This is NOT a group project. You should use your group's code from the previous project, but all the modifications and additions to the code should be done yourself.

READ THE ENTIRE PROJECT DESCRIPTION CAREFULLY BEFORE BEGINNING. THEN READ IT AGAIN! I know there is a lot of information here, but it is here to help you through this project.

Project goals:

This project will test your ability to use the C standard library functions, including file I/O and some mathematical functions. In addition, you will gain experience designing and writing C programs with multiple files, as well as modifying existing code to fit a new purpose. This project will also strengthen your understanding of memory management and data structures in C. Finally, this project should be fun and give you a sense of accomplishment when you finish! Not everyone can say that they have built a web search engine!

Project description:

The program you write will involve two phases:

- Indexing several web pages
  In this phase, you will use a web crawler to find several web pages. For each page, you will gather the word counts for that page in a trie. Note that you will need a separate trie for each web page, so you can keep their counts separate.
- Answering web queries from the gathered statistics
  In this phase, you will query the user for search terms, and then check the web page statistics to find the pages that best match the query.

For this project, **it is critical that you understand the algorithms and think about what data structures you will need before you begin coding**.

The indexing phase

The web pages you index will be found by a web crawler. There is a limit on the total number of pages you will index, which is detailed later.

To avoid indexing only one piece of the web, the web crawler will be restarted from different pages. That is, it will start at a page, go for some maximum number of hops, then get a new

start page and go for some maximum number of hops, get a new start page, and so on. The start pages and numbers of hops will be listed in a file that your program should open and read. The name of that file (the "URL file") is the first command-line argument to your program.

The file format is:
startURL1 maxNumHops1
startURL2 maxNumHops2
...

The number of start URLs can vary. You can assume that no URL contains a space. If a start URL is supposed to go for a maximum of 10 hops, then you will index up to 11 pages (because you should index the start URL itself).

The web crawler should crawl according to the following pseudocode, which is a different crawling algorithm than we used in our prior crawling project:

Let MAX_N be the maximum number of pages that your program should index (given on the command line).
Let n be the number of pages that the program has indexed.

```
while there is another start URL in the file and n < MAX_N:
      url ← the next start URL from the file
      hopsLimit ← the max number of hops from the new start URL
      hopNum ← 0
      while true:
            if url is not already crawled(/indexed):
                  crawl(/index) the url
                  n++
            end if
            hopNum++
            if hopNum <= hopsLimit and n < MAX_N:
                  new_url ← one hop from url (according to getLink)
                  if new_url is not valid (i.e., url was a dead end):
                        break;
                  end if
            else:
                  break;
            end if
      end while
end while
```

You get to choose how much of the code from the web crawler project you want to use in this project (in addition to the getLink function). Regardless of your choice, the main function from the web crawler project will not be an appropriate main function for this project, but you can look at it for ideas and/or modify it to form a different function in this project.

Each URL the web crawler encounters should be indexed (assuming that it has not been indexed already). To index a web page, you will need to store:

- its URL
- the frequency of each distinct term that occurs in the page. You can think of a term as a word. Its frequency is just the count of how many times it occurs in the page.
- the total number of terms in the page. This is the sum of all the frequencies for all the terms on the page. For instance, "hi hi" would have a total of 2 terms even though there is only one distinct term.

To index the page and store the frequency of each term, you should use the function you wrote in your previous project that takes a URL and returns a trie of term frequencies. You will need to modify that function to also tell you the total number of terms in the page. The suggested way to do this is to add an int* argument to the function, and have the function set the int at that address to be the total number of terms. For example,

```
struct myTrie* indexPage(const char *url, int *pTotalNumTerms){
      ...
      *pTotalNumTerms = sumOfTermCounts;
      ...
}

otherFn(...){
      int totalNumTerms;
      struct myTrie* result = indexPage(url, &totalNumTerms);
      ...
}
```

You will notice that this is the same way you get multiple values back from scanf: pass in addresses, so the function can set the values appropriately.

Output from Indexing Phase

As your program crawls and indexes the web pages, it should print some information to standard output: print "Indexing...", then print each web page you are indexing, and the **terms (one on each line, preceded by a tab)** that you find in that page. You should be able to use the same formatting from the indexing portion of the previous project. If a duplicate page occurs, do not print its name or its terms again. Here is some example output (not from actual web pages):

```
Indexing...
http://www.cs.ipfw.edu
      computer
      science
      computer
      science
```

```
      department
      ...
http://www.etcs.ipfw.edu/
      engineering
      indiana
      ...
http://www.cs.ipfw.edu/undergraduate/bscs.php
      ...
http://www.its.ipfw.edu/resources/facilities/tlabs/default.shtml
      ...
http://www.its.ipfw.edu/resources/accounts
      ...
```

Answering web queries

Once your program has indexed all of the web pages it is supposed to index, it is time to answer the user's web queries. Your program should have a loop where it prompts the user to enter a query (on standard input), then returns the resulting links, then prompts the user for another query, and so on. When the user enters an empty query, the program should exit. A query should be a list of whitespace-separated, lower-case terms (e.g., "ipfw computer science"). Your program must check that the query actually conforms to these standards; if not, the program should print an error message to stderr, then proceed to prompt the user for another query.

Assuming that the query was not empty and was well-formed, the program will need to print out a list of links as the result of the web search. To get the list of web pages that match the query, your program will assign each page a score, using the formula described below. The 3 pages with the highest scores should be printed as the result of the search (along with their scores). If less than 3 pages exist with valid, non-zero scores, then omit the unneeded listings in the output.

Output from Query Phase

When running a correct program, you should see the crawling/indexing output (described above), then something like the following example. Boldface indicates user input while normal font indicates output to the terminal (via stdout or stderr). To earn points for your work, you must use the exact formatting and messages shown here. (The numbers and URLs here are not necessarily valid, just the formatting.)

```
Enter a web query: computer science ipfw
Query is "computer science ipfw".
IDF scores are
IDF(computer): 0.0043174
IDF(science): 0.0018258
IDF(ipfw): 0.0027458
```

```
Web pages:
1. http://www.cs.ipfw.edu (score: 0.3918)
2. http://www.etcs.ipfw.edu/ (score: 0.2892)
3. http://www.cs.ipfw.edu/undergraduate/bscs.php (score: 0.2785)

Enter a web query: IPFW
Please enter a query containing only lower-case letters.

Enter a web query: ipfw computers
Query is "ipfw computers".
IDF scores are
IDF(ipfw): 0.0027458
IDF(computers): 0.0028493
Web pages:
1. http://www.cs.ipfw.edu (score: 0.3503)
2. http://www.its.ipfw.edu/resources/facilities/tlabs/default.shtml
(score: 0.3418)

Enter a web query: indiana-purdue
Please enter a query containing only lower-case letters.

Enter a web query:
Exiting the program
```

Computing the score for a single web page

This section describes the formula for computing the score of a web page for a given query. Each query is a list of terms. The overall score for a query and a document ***d*** is the sum of the scores for each term in the query with respect to the document ***d***.

Example: Let ***S(d,t)*** be the score for a term ***t*** with respect to the document ***d***. Then for a query "ipfw computers", the overall score of http://www.ipfw.edu is
S(http://www.ipfw.edu, ipfw) + ***S***(http://www.ipfw.edu, computers)

The score function we will use for ***S(d,t)*** is called the TF-IDF score (term frequency-inverse document frequency). It is the product of two other scores:

- ***TF(d,t)***: the term frequency of a term ***t*** and document ***d*** is the number of times that ***t*** occurs in ***d*** divided by the number of times any word occurs in ***d***. In other words, it is the fraction of words in ***d*** that are equal to ***t***. Efficient computation of the TF score is the reason to store the sum of all word counts for each document when you index it.
- ***IDF(t)***: the inverse document frequency is the natural logarithm of [(1.0 + number of indexed documents) / (1.0 + number of indexed documents that contain ***t*** at least once)]. The C standard library provides the "log" function to compute the natural logarithm.

Example: For a web page ***d***, its overall score for the query "computer science" is
S(***d***, computer) + ***S***(***d***, science)
= ***TF***(***d***, computer) * ***IDF***(computer) + ***TF***(***d***, science) * ***IDF***(science)

The reasoning behind the TF-IDF scoring function is explained on Wikipedia (http://en.wikipedia.org/wiki/Tf%E2%80%93idf). Note that we are using a slightly different function than they use on that page, so get your formula from this description, not another source.

In order to compute this score, you should write a function to query the trie for how often a word occurred.

Details, Hints, and Tips

- As always, break down the entire problem into pieces, thinking about what steps the program will need to take. **It is often beneficial to test the pieces as you complete them, rather than writing them all and then testing the whole program**. For example, one of the first pieces to check would be the part of the program that reads in the URL file. You can check this by just reading in the URLs and numbers of hops and printing them back to the console. Testing your program piece-by-piece will help you fix bugs in each part, so that debugging isn't such a daunting task. This is one reason we wrote the indexing part as a separate project.
- Your program should take three command-line arguments:
  - The first is the name of the file containing the URLs to index (described above).
  - The second is the maximum number of web pages to index. This is not the number of start URLs in the file, but rather the maximum number of web pages to index (including the start URLs and every page to which the crawler hops, excluding duplicates that are not re-indexed).
  - The third is a long int that should be passed to srand before beginning the web crawling. That should be the only call to srand in your program. This will seed the random number generator so that results can be duplicated. Use atol to convert a string to a long.
- Your code should be logically organized across multiple files. The main function should be in a file webSearch.c that contains only the main function (and #include's as needed). The other functions and structures should be organized into at least two .c files, each with a corresponding header file.
- Your can re-use functions from Project 1 and Project 2, such as the getLink and getText functions, as well as other functions you developed.
- You should use your trie data structure from the previous project, modifying it as needed.
- You should create a Makefile (note the capital ‘M’) that has a target "webSearch" that compiles and links your code, creating an executable “webSearch”.
- To include the "log" function from the math library, you will need to add a "-lm" at the end of the typical compile command: gcc -g obj1.o obj2.o webSearch.c -o webSearch -lm
- TA and I will be collecting the Makefile and all of the .c and .h files from your project directory (i.e., “project3” directory) (but not subdirectories) at Bitbucket. Make sure that all of your code is in those files.
- You should ensure that your program does not have memory leaks by using the heap checker.

- In Blackboard, you can download a tar file “project3.tar”. Upload this file to your C9 python workspace under the subdirectory “project3” that is under “project” directory. The tar file can be unzipped by the following command:

  $ tar -xvf project3.tar

  The tar file contains the following files:
  - test_all: script for testing the correctness of the final program
  - test_valgrind: script for checking memory leakage
  - webSearchSol/webSearch: an executable program for demonstrating the expected behavior of the final program
  - webSearchSol/urlFiles: a list of sample URL files
  - webSearchSol/searchFiles: a list of files containing searched words

  If your program is correct, the resulting “results_all.txt” should be an empty file. Moreover, the file “valgrind_results.txt” contains the information about whether your program does not have the memory leakage and does not have any error. To run the scripts “test_all” and “test_valgrind”, as well as the sample program “webSearch”, please make sure to use “chmod” command to change the executable property of files.

Checking Your Program

Since there are many different URL and query possibilities, a thorough testing of your program will require *many* test cases. I have made a *few* test cases for you (please refer to the script “test_all”), but you should supplement those with your own testing, using as many different types of test cases as you can imagine. (However, you can assume that the URL file will follow the format described above.) For any given test case, you can compare your program’s output to the correct output given by the correct searching program webSearchSol/webSearch. I suggest that you use redirection to capture the output of your program and the correct program to two files, which you can then compare using the “diff” command. Again, please refer to the script “test_all”.

Submission

Before pushing the files (all .c and .h files, as well as Makefile) to your Bitbucket repository, please use Git to run

```
$ git pull origin master
```

TA may have updated the score file to your Bitbucket repository. Running the above command can get the latest score file and keep your cloud9 project directory in sync with your Bitbucket repository.

Remember that when you push the code for this project to your Bitbucket repository, please operate under the “project” directory, instead of “project3” subdirectory.

If you have any questions, please let the instructor know.

Grading rubric:

TAKE NOTE! The grading for this project will be based upon the number of my grading test cases where a section of your program's output matches mine exactly. This is different from previous projects, where you could get partial credit for code that is almost correct. For example, if you are missing an "i++" in a previous project, you could still get some credit for that part of the code. However, in this project, that omission would (likely) cause all the test cases to fail, resulting in 0 points for that part of the project.

The main reason for this grading rubric is that, from the perspective of software development, partially correct code can be worse than no code at all, since it can hide bugs! Thus, completely correct code should be your objective any time you write software. At this point in the course, you have enough experience with C that you should be able to write code that is completely correct.

You will earn points for having correct output for the following pieces of the project: crawling (50pts), indexing (40pts), parsing the query (65pts, verified by checking the words in your "IDF (foo):" lines), IDF scores and ***S(d,t)*** (50pts), and your sorted web page results (50pts). Some of these pieces depend upon previous pieces, so **make sure you get the first pieces working correctly before moving on**! In particular, YOUR CRAWLING MUST BE WORKING CORRECTLY IN ORDER TO GET CREDIT FOR ANY OTHER PARTS OF THE PROJECT. Among other testing, you should verify your crawling on the train1.txt, train2.txt, test0.txt, test1.txt, test2.txt and other URL files, as these will be a critical part of my grading tests.

Make sure that your program compiles! Programs that do not compile will not earn any points.

Grading deadlines:

You will notice that there are two different due dates for this project. This is to encourage you to start working on the project immediately and to ensure that your web crawling and indexing are working correctly.

First due date (200 of 400 points):

By the first due date, you must have the web crawling and indexing working correctly. This includes having a working Makefile in place to create a webSearch program that takes the 3

command-line arguments listed in the description. Moreover, it must print out the URLs of each page and index them. For example, your program might print out something like

```
Indexing...
http://www.cs.ipfw.edu
      computer
      science
      ...
http://www.etcs.ipfw.edu/
```

- Makefile and file structure – 50 pts
- Implementation (if the code is compiled) – 115 pts
    - Command-line arguments – 25 pts
    - Crawling – 50 pts
    - Indexing – 40 pts
- No memory leaks (if the code is well implemented) – 25 pts
- Coding style (if the most of the above are implemented) – 10 pts

Second due date (200 of 400 points): For this due date, your project should be complete.

- Implementation (if the code is compiled) – 165 pts
    - Parsing the query – 65 pts
    - Calculating IDF of each word – 25pts
    - Calculating S(d,t) for each page – 25pts
    - Finding and printing and the top three pages – 50pts
- No memory leaks (if the code is well implemented) – 25pts
- Coding style (if the most of the above are implemented) – 10 pts