SCHOOL OF COMPUTING AND INFORMATION SYSTEMS

COMP20007 Design of Algorithms, Semester 1, 2018
Assignment 1: Multi-word Queries

Due: 12 noon, Monday 9 April

Overview

In this assignment you will implement a small part of a text search engine: using an existing *inverted file index* for a multi-word query to find the top-matching documents in a large document collection.

The assignment consists of 2 coding tasks and 1 written task. Each task is described in detail in the sections below.

Provided files

Before you attempt this assignment you will need to download the following code and data files from the LMS:

`main.c`	Entry point to program. **Do not change**.
`list.h, list.c`	Singly-linked list module. **Do not change**.
`index.h, index.c`	Inverted file index module. **Do not change**.
`query.h`	Prototypes for functions you must implement. **Do not change**.
`query.c`	**Define your functions here**, according to specification below.
`heap.h, heap.c`	Binary min-heap module. Empty: see week 4 lab task.
`Makefile`	To assist with compilation. Edit if you add any extra files.
`data/`	Directory containing many term-based inverted file index files.
`documents.txt`	List of all documents indexed.

At this point, you should be able to compile the supplied code (above) by running `make`. Once your assignment is completed, you will be able to execute it by running a command of the form

```
./a1 -t task -r num_results -d num_documents list of query terms
```

where

- `task` is either 1 or 2 representing which coding task to execute,
- `num_results` is the number of search results to find and display,
- `num_documents` is the maximum number of documents to consider (optional; the default is 'all documents', or 131,563), and
- `list of query terms` is a space-separated list of words making up a query for your program to 'search' for.

For example, the command `./a1 -t 1 -r 12 algorithms are fun` will run your solution to task 1 on the entire document collection, printing the top 12 results matching the combination of words 'algorithms', 'are', and 'fun'.

Data structures

Your solution will need to work with the data structures defined in `index.h` and `list.h`. Here's a brief overview (consult the header files for the finer details and a full list of available functions):

- An `Index` is used to represent a multi-term inverted file index. An `Index` has an array of string 'terms'. For each term, it also stores a `List` of `Document`s.
- Each `List` in the `Index` is a linked list of `Node`s, with each `Node` containing a pointer to a single `Document`. These lists can easily be traversed using an appropriate `while` or `for` loop.
- A `Document` contains an integer `id` and a floating-point `score`. The `id` is an ordinal integer identifying a particular document in the overall collection (see `documents.txt`, which lists all documents by their `id`, starting with document 0). The `score` is a non-negative real number calculated based on the prevalence of the corresponding term within this document.
- Documents occur in each of the index's lists in order of increasing `id`. Only documents with `score` greater than zero for a term are present in the list for that term.

Additionally, you will require an array-based binary min-heap module to correctly implement each of the coding tasks. It is up to you exactly how to design and implement this module. Your solution to the week 4 lab tasks will be helpful here. Note that sample lab solutions will be released in the weeks after each lab and that these may be used in your assignment (with proper attribution).

Furthermore, you may create any additional modules necessary to support your solution. If you add additional modules, it is your responsibility to correctly extend your makefile—you must ensure that your solution can be compiled after submission simply by running `make`.

Coding tasks

The first two tasks of the assignment require you to implement functions inside `query.c`.

Task 1: Array-based accumulation approach (3 marks)

Implement the function `print_array_results()` defined in `query.c`. This function has three input parameters: `index` (an `Index` pointer) containing data for the query to perform; `n_results` (an integer) describing the number of results to output; and `n_documents` (another integer) containing the total number of documents in the collection to be queried.

This function should find the top `n_results` matching document ids and their associated total scores. The total score for a document is the sum of its score from each term in the query. For this function, use the following method to find these top-scoring documents:

1. Initialise an array of `n_documents` floating-point numbers, all set to `0.0`.
2. Iterate through each document list in the index. For each document in the list, add the document score to the corresponding position in the array (using the document id as an array index).
3. Use the priority queue-based top-k selection algorithm to find the maximum `n_results` total scores in the resulting array, and their associated document ids.

The function should print these results as per the instructions in the 'Output format' section below.

Task 2: Priority queue-based multi-way merge approach (3 marks)

Implement the function `print_merge_results()` defined in `query.c`. This function has two input parameters: `index` (an `Index` pointer) containing data for the query to perform; and `n_results` (an integer) describing the number of results to output.

This function should also find the top `n_results` matching document ids and their associated total scores. For this function, use the following method to find these top-scoring documents:

1. Use a priority queue-based multi-way merge algorithm to iterate through the `n_terms` document lists concurrently:
   - Initialise a priority queue to order the document lists based on the id of their first documents.
   - Repeatedly retrieve a document from the document list at the front of the priority queue, and rearrange the priority queue so that this list is positioned according to the id of its next document (stepping through the list). Stop after processing all documents in all lists.
2. While iterating through the lists in this way, accumulate total scores for each document id. Use the priority queue-based top-k selection algorithm to find the maximum `n_results` total scores and associated document ids.

The function should print the results as per the instructions in the 'Output format' section below.

Output format

The output of both functions should follow the same format: the top-scoring `n_results` results with non-zero scores, printed one result per line to `stdout` (e.g. via the `printf()` function).

The results should be printed in descending order of total score (that is, with the highest total scoring document on the first line, and then the second, and so on). In the case of multiple documents with the same score, such documents may be printed in any order relative to each other. Each document should be on its own line. The line should be formatted with the document id printed first as a 6-digit decimal integer (padded on the left with spaces if necessary), followed by a single space, followed by the floating-point total score of the document to a precision of 6 digits after the decimal point. The following format string will be useful: `"%6d %.6f\n"`

There should never be more than `n_results` lines of output. If many documents have the same score as the document with the `n_results`$^{\text{th}}$-highest score, any subset of these documents may be printed such that there are exactly `n_results` lines of output. Moreover, there should be no additional information printed to `stdout` (if you must print additional information, you can use `stderr`). Finally, only documents with non-zero total scores should ever be shown. Therefore, for some queries, there may actually be fewer than `n_results` lines of output.

Example output

To help you validate the output format of your functions, we provide some samples of correct output for three basic queries. Download the sample files from the LMS. The files contain one example of correct output from a selection of commands, described in the following table.

These examples are intended to help you confirm that your output follows the formatting instructions. Note that for some of these inputs there may be **more than one correct output** due to different possible orderings of documents with the same score. These samples represent only one ordering. Note also that these samples represent only a small subset of the inputs your solution will be tested against after submission: matching output for these inputs doesn't guarantee a correct solution. You are expected to test your functions comprehensively to ensure that they behave correctly for all inputs.

Filename	Command
`helloworld-12.txt`	`./a1 -t 1 -r 12 hello world`
`algorithms-99.txt`	`./a1 -t 1 -r 99 algorithms are fun`
`catoutofbag-8.txt`	`./a1 -t 1 -r 8 the cat is out of the bag`

Since the expected output format for task 2 is identical to that of task 1, you can test task 2 by replacing `-t 1` with `-t 2` in each command.

Warning: These files contain Unix-style newlines. **They will not display properly in some text editors, including the default Windows text editor Notepad.**

Written task

The final task of the assignment requires you to write a report addressing the topics described below.

Task 3: Analysis of algorithms (4 marks)

First, analyse the different approaches from task 1 and task 2 in terms of their asymptotic time complexity. Then, explore the effect that the various input parameters (such as the number and length of document lists in the query and the number of results requested) have on the time taken by your program to produce its results. Support your investigation with experimental evidence gathered by timing your program with various input configurations. Include a plot to visualise your experimental results in your report. To conclude, give a precise characterisation of the circumstances where we should prefer one approach over the other.

Your report must be no more than two pages in length. You may use any document editor to create your report, but **you must export the document as a PDF for submission**. You should name the file `report.pdf`.

Submission

Via the LMS, submit a single archive file (e.g. `.zip` or `.tar.gz`) containing **all files required to compile your solution (including `Makefile`) plus your report (as a PDF).** When extracted from this archive on the School of Engineering student machines (a.k.a. `dimefox`), your submission should compile without any errors simply by running the `make` command.

Please note that when compiling your program we will use the original versions of the files marked '**Do not change**' in the Provided files list. Any changes you have made to these files will be lost. This may lead to compile errors. Do not modify these files.

Submissions will close automatically at the deadline. As per the Subject Guide, the late penalty is 20% of the available marks for this assignment for each day (or part thereof) overdue. Note that network and machine problems right before the deadline are not sufficient excuses for a late or missing submission. Please see the Subject Guide for more information on late submissions and applying for an extension.

Marking

The two coding tasks will be marked as follows:

- 2 of the available marks will be for the correctness of your program's output on a suite of test inputs. You will lose partial marks for minor discrepancies in output formatting, or minor mistakes in your results. You may score no marks if your program crashes on certain inputs, produces completely incorrect answers, or causes compile errors. We will compile and test your program on the School of Engineering student machines (a.k.a. `dimefox`).
- 1 of the available marks will be for the quality and readability of your code. You will lose part or all of this mark if your program is poorly designed (e.g. with lots of repetition or poor functional decomposition), if your program is difficult to follow (e.g. due to missing or unhelpful comments or unclear variable names), or if your program has memory leaks.

The written report will be marked as follows:

- 1 of the available marks will be for the clarity and accuracy of your analysis of the asymptotic time complexity of the two approaches. You will lose partial marks for minor inaccuracies, or misuse of terminology or notation.
- 2 of the available marks will be for the quality of your investigation into the observed performance of both approaches. You will lose marks if your investigation is not supported by experimental evidence, if your report does not include a visualisation of your results, or if your investigation does not fully address the topic.
- 1 of the available marks will be for the clarity and accuracy of your conclusion.
- Additional marks will be deducted if your report is too long (past the two-page limit) or is not a PDF document.

Academic honesty

All work is to be done on an individual basis. Any code sourced from third parties must be attributed. All submissions will be subject to automated similarity detection. Where academic misconduct is detected, all parties involved will be referred to the School of Engineering for handling under the University Discipline procedures. Please see the Subject Guide and the 'Academic Integrity' section of the LMS for more information.