Department of Computing and Information Systems

COMP10002 Foundations of Algorithms
Semester 2, 2016
Assignment 2

Learning Outcomes

In this project you will demonstrate your understanding of dynamic memory and linked data structures, and extend your skills in terms of program design, testing, and debugging. You will also learn about inverted indexes, and the basic principles of web search algorithms.

Indexed Document Retrieval

The idea of an *inverted index* was mentioned briefly in class. To build an index for some input text, the words are isolated, together with their document numbers (in our case, line numbers in the input file), and arranged so that, for every word, a list of the documents that contain that word is constructed. In terms of notation, if t is an indexed *term*, then f_t is the number of documents in the collection that contain that term at least once; and for any given document d, the value $f_{d,t}$ is the number of times that t appears in d. For example, consider the text:

```
line one has one word twice
line two has words once only
line three follows lines one and two, but not four
line four is like the other lines, not like line five
line five has word one and word two and word three
six is the littlest one
```

If each line of that input file is taken to be a "document", then the first few lines of a simple document-level inverted index for it would be

```
and 2 3 1 5 2
but 1 3 1
five 2 4 1 5 1
follows 1 3 1
four 2 3 1 4 1
has 3 1 1 2 1 5 1
```

where the first integer following each word is the f_t value for that term t, with exactly that many $\langle d, f_{d,t}\rangle$ pairs after it all on the same input line. For example, term t = "and" appears in $f_t = 2$ documents, document $d = 3$ (with $f_{d,t} = 1$, that is, one occurrence), and in document $d = 5$ (with $f_{d,t} = 2$ occurrences). The full index file for the original six lines is available on the LMS, together with larger examples. Make sure that you understand the structure, and what the values represent.

Stage 1 – Reading the Index (marks up to 8/15)

Write a program that reads an index file with this format, specified as the first (and only) argument on the command-line, and builds (using `realloc()` and `malloc()`) a data structure to store that index information. The only assumption you may make, purely for the purposes of reading the input strings, is that each term in the index will be at most 999 characters long. Apart from a single buffer of that size, all stored strings and lists of $\langle d, f_{d,t}\rangle$ pairs should be held in dynamic arrays of the correct length for the data they contain (or within a factor of two of that minimum length). As evidence of the operation of this stage of your program, it should report the number of terms in the index that was read, the total number of $\langle d, f_{d,t} \rangle$ pairs in the index, and up to ten of the pairs associated with the first two and the last two terms in the index, using *exactly* the output format that is shown here and in the LMS examples. Note that the terms are labeled from 1:

```
mac: ./ass2-soln test0-ind.txt
Stage 1 Output
index has 23 terms and 43 (d,fdt) pairs
term 1 is "and":
    3,1; 5,2
term 2 is "but":
    3,1
term 22 is "word":
    1,1; 5,3
term 23 is "words":
    2,1
```

You may assume throughout that all input files you will be provided with will be "correct", according to the description given above – there won't be any tricksy-wicksy deviations. That is, all strings will be lower-case alpha only; all elements will be separated by a single space; all lines will have the correct number of values; all lines (including the last one) will be ended by a single (DOS-style) CR-LF combination (see the discussion of this in the Assignment 1 FAQ page); etc. Note that the terms are provided in the index file in strict dictionary order, and that the d values in the set of pairs associated with each term are also are strictly increasing; you will need to make these two facts in Stage 2 and Stage 3. Your Stage 1 program must open and read from a named file (and *not* from `stdin` yet, see Stage 2 for that input) so you will need to be familiar with Chapter 11 of the textbook. Output should be written to `stdout` (but you may also write error messages to `stderr` if you wish).

Stage 2 – Queries and Term Lookup (marks up to 12/15)

Extend your program so that it reads "queries" from `stdin` and looks up their corresponding term numbers in the index. Terms that do not appear in the index should be noted accordingly. For example (not showing the Stage 1 output again):

```
mac: ./ass2-soln test0-ind.txt
four and five                     <<this line was typed by the user>>
Stage 2 Output
    "four" is term 5
    "and" is term 1
    "five" is term 3
line seven                        <<this line was typed by the user>>
Stage 2 Output
    "line" is term 9
    "seven" is not indexed
```

The eventual expectation in this stage is that you will use binary search to do this lookup; but it might be prudent to make use of linear search in the first instance, while you are debugging this stage and working on Stage 3, and then switch to binary search when you are sure of yourself, and ready to extend again. You may assume that each input line will contain at most 999 characters, and at most 20 different query terms. Lines should be processed interactively, so that the output for each line is generated before the next line is typed, with query terms limited to alphabetic characters, and case-folded to lowercase before the lookup is undertaken, to match the index. You do not need to retain each query line once it has been processed by Stage 2 and then Stage 3.

Stage 3 – Ranked Querying (marks up to 15/15)

Ok, now for the fun part. Suppose that the collection has N documents, and suppose that (as before) term t appears in f_t of them, and appears in document d a total of $f_{d,t}$ times. To determine the documents d (here, lines) that are the "strongest" matches for each query q, the following calculation is carried out on a per-document basis:

$$\mathit{score}(d, q) = \sum_{t \in q} \left(\frac{(1 + k) f_{d,t}}{L_d + f_{d,t}} \cdot \log_2 \frac{N + 0.5}{f_t} \right)$$

and

$$L_d = k \left((1 - b) + b \frac{|d|}{\mathrm{avg}_d\{|d|\}} \right) ,$$

where $|d| = \sum_{t \in d} f_{d,t}$ is the total number of words in document d, $\mathrm{avg}_d\{|d|\}$ is the average document length over the N documents (computed as the arithmetic mean), and $k = 1.2$ and $b = 0.75$ are constants. Broadly speaking, there are three key relationships being balanced in this equation: rare words with low f_t values are regarded as being more important than common words; words that appear multiple times in any given document have more weight than words that only appear a few times; and long documents are discounted relative to shorter ones. This computation is known as the "BM25" similarity mechanism.[1] A wide range of other such scoring mechanisms have been proposed and used in web search systems. The ones actually used by Google and similar are, of course, closely-guarded commercial secrets, but still embed these same three principles. For web documents, commercial search providers also make use of features such as page title text; information from relative font-sizes being used; the anchor text that points to each page; language cues; the link relationships between pages; query click-through rates from other users; and so on.

Add further functionality to your program so that as each query is read from `stdin`, it is processed against the index, and the top-3 matching documents (by score, with ties on score resolved according to increasing document number) identified and reported. For example (with the interleaved Stage 2 outputs suppressed, see the LMS for full examples of the output required):

```
mac: ./ass2-soln test0-ind.txt
four and five                     <<this line was typed by the user>>
Stage 3 Output
    document   5: score  3.619
    document   3: score  3.115
    document   4: score  2.978
line seven                        <<this line was typed by the user>>
Stage 3 Output
    document   4: score  0.474
    document   1: score  0.425
    document   2: score  0.425
```

For this process to be efficient, you need to combine a number of different sources of information, bringing components in as they are required. The document lengths $|d|$ should be stored as part of your index data structure, and can be computed once the full index has been read. A value for N can similarly be determined from the input data and stored as a component of the index structure you build. But that still leaves the task of processing the term lists to generate document scores, and determining the top few (where "few" is three in this project, but is 10 for Google and Bing, for example) in an

[1] Actually, I have simplified it to avoid some slightly complicated behavior that occurs for very common terms. The "true" BM25 method has $(N - f_t + 0.5)/(f_t + 0.5)$ in the second factor instead of $(N + 0.5)/f_t$, and also takes the frequency of the term in the query, $f_{q,t}$, into account.

appropriate manner. Be sure to start simple, and check your computations at every step as you build towards the required output.

As an overall outline of the process to be implemented, to compute the required *score*(d, q) values you should: (a) create and set to zero an array of N document score *accumulators*; (b) iterate over the index entries for the (indexed) terms that appear in the query, and for each such term, iterate over its $\langle d, f_{d,t}\rangle$ pairs, computing contributions to document scores one term at a time and adding them to the corresponding accumulator; and then (c) partially sort the array of accumulators (including their corresponding document numbers) to bring the biggest ones to the front. Note you don't have to fully sort the score array in order to generate the required output (that is, there is an algorithmic efficiency question here for you to consider).

My program to implement all three stages is a little under 500 lines long, including detailed comments and debugging output, about 100 lines longer than the solution for Assignment 1. *Start early!* If you want more of a challenge, try writing (but please don't submit) the indexing program that generated the index files. And, of course, a real system would generate a *query-biased caption*, or *snippet*, for each answer document (rather than just the document number), you might like to think about how that process gets done.

The boring stuff...

This project is worth 15% of your final mark. A rubric explaining the marking expectations will be provided on the FAQ page.

You need to submit your program for assessment; detailed instructions on how to do that will be posted on the LMS once submissions are opened. Submission will *not* be done via the LMS; instead you will need to log in to a Unix server and submit your files to a software system known as `submit`. You can (and should) use `submit` **both early and often** – to get used to the way it works, and also to check that your program compiles correctly on our test system, which has some different characteristics to the lab machines. *Failure to follow this simple advice is highly likely to result in tears.* Only the last submission that you make before the deadline will be marked.

You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** "lend" your "Uni backup" memory stick to others for any reason at all; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" when they ask for a copy of, or to see, your program, pointing out that your "**no**", and their acceptance of that decision, is the only thing that will preserve your friendship. *A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode. Students whose programs are so identified will be referred to the Student Center for possible disciplinary action without further warning. This message **is** the warning.* See `https://academichonesty.unimelb.edu.au` for more information.

Deadline: Programs not submitted by **10:00am on Monday 17 October** will lose penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. If you attend a GP or other health care professional as a result of illness, be sure to take a Health Professional Report form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops in to something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it in connection with any non-Special Consideration assignment extension requests.

Marks and a sample solution will be available on the LMS before Tuesday 1 November.

And remember, algorithms are fun!