COMP20007 Design of Algorithms – Assignment 1

Theoretical Analysis

- Let $N > 0$ be the maximum number of documents in the index, `n_documents`.
- Let $n = rN,\ r \in (0,1]$ be total number of documents that appear in the document lists for the query and that are processed by the algorithm, such that the length of each document list is bounded above by n.
- Let $k > 0$ be the number of results requested, `n_results`.
- Let $m > 0$ be the number of terms in the search query, `index->num_terms`.
- Assume $n \gg k$ and $n \gg m$, and by definition $N \geq n$.

The tables below show the number of operations expected for each function in query.c. We can add these together to analyse the overall time complexity of the algorithm.
(This assumes expected case runtime for the top k selection algorithm. However, using worst case – inserting every document with a non-zero score, taking $n \log k$ operations – would impact both algorithms equally, as they perform the same number of insertions in the same order, so this is immaterial to the analysis.)

Task 1:

initialise_doc_scores	$O(N)$
array_scores	$O(nm)$
array_top_k (including top_k_insert)	$O(N + k \log n \log k)$
print_results	$O(k \log k)$
print_array_results	$O(2N + nm + k \log n \log k + k \log k)$
Task 1 Complexity	$O(N + nm)$

Task 2:

initialise_doc_list_queue	$O(m)$
initialise_doc_list_tracker	$O(m)$
merge_top_k, including • get_doc_score • top_k_insert Total	$O(n)$ $\times\, O(m \log m)$ $+\, O(n + k \log n \log k)$ $= O(nm \log m + n + k \log n \log k)$
print_results	$O(k \log k)$
print_merge_results	$O(2m + n + nm \log m + k \log n \log k + k \log k)$
Task 2 Complexity	$O(n + nm \log m)$

From this, we can observe:

- In task 1, the fact that we iterate over N documents rather than n means that task 2 may be faster for lower values of the ratio r – that is, shorter document lists.
- In task 2, the use of a priority queue in processing the document lists means that finding the score for each document takes $O(nm \log m)$ instead of $O(nm)$ operations. As $\log_2 2 = 1$, We might expect task 2 to be faster for a 1-term query, but slower for queries of 3 or more terms.
- Since $n = rN$, and k has no impact on the complexity of either algorithm, changes to N or k would impact both tasks equally.

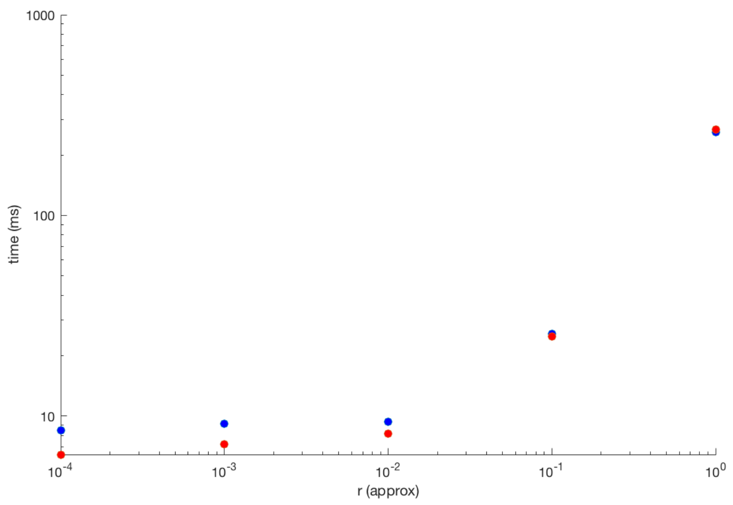
Experimental Results

Each experimental result for runtime was obtained by taking the average of 10 "best of 3" tests. However, there is still some natural variation for each execution, and since the difference between the runtime of the two tasks is often smaller than this variation, these results should be taken as a broad approximation only.

As per the above analysis, we will focus on variations in r; the proportion of documents that are processed by the query, and m; the number of terms in the query. Consistent with the theory above, changes in k (n_results) made virtually no difference to execution time, and will be skipped in this discussion.
In the plots below, task 1's runtime is marked in blue, and task 2's runtime is marked in red.

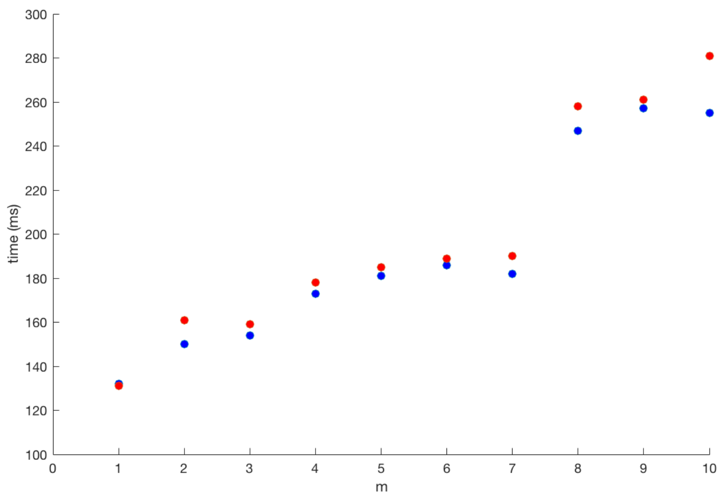
For $\boldsymbol{r}$: We approximate r in the sample data with five different queries, all 3 terms long, in the regions:

- $r \in (0,0.0001]$ – "omniscient omnipotent cryptocurrency"
- $r \in (0.0001,0.001]$ - "concurrent moron emoji"
- $r \in (0.001,0.01]$ - "turing chess algorithms"
- $r \in (0.01,0.1]$ – "solar system computer"
- $r \in (0.1, 1]$ – "is a day"



As expected, task 2 performs slightly better for words with short document lists, since it does not need to iterate over the documents in which none of the query terms appear. As $r \to 1, n \to N$, and thus we see essentially the same runtime with long document lists.

For $\boldsymbol{m}$: We test $m = 1, 2, \dots, 10$. To partially control for the effect of r, we use common words, and add one term to the end of the string each time we increase m. The final query (at $m = 10$) is: "a c program can do simple algorithms on your computer".



For shorter queries, we do see roughly the same performance of both algorithms, but as the number of terms grows, task 1 starts to gain a slight lead.

In conclusion: the array accumulator approach is preferable for multi-word queries, provided those queries include at least one common term – as you might expect, for example, from complete phrases and sentences. The multi-way merge approach is preferable for short queries – particularly single-word queries, or queries consisting entirely of very rare, context-specific terms.