COMP20007 Assignment 1: Multi-word Queries

Sample report

Introduction

The aim of this project is to implement two approaches to finding the k top-scoring documents based on an inverted file index that correspond to an input query of words. Here are the definitions for the variables used throughout this report:

- n : The total number of documents, `n_documents`, with a default value of 131563.
- w : The number of words in query. Words are labelled from word 1 to word w.
- d_i : The number of entries in word i. Furthermore, let $d = d_1 + d_2 + \cdots + d_w$.
- r : The number of (distinct) documents that are listed in the w document lists. Note that $r \leq \min\{d, n\}$.
- k : The (maximum) number of results to return, `n_results`. It is assumed that $k \leq r$.

Task 1 uses an array of size n that store the total scores of all n documents. The algorithm iterates through the w document lists and accumulates the score of each document. The maximum k total scores and their associated documents ids are then found by performing a priority queue-based top-k selection algorithm. Note that documents with a score of zero are skipped and will not be inserted into the priority queue.

Task 2 uses a priority queue D to heapify the document lists based on the id of the first entry. Another priority queue S is to contain the top k documents seen so far. Entries from the document lists are extracted in nondecreasing order of id, and D is updated after each extraction, where the priority key is the id of the next entry to be extracted. All scores for the same document id are then added together, and S is updated if necessary (according to the priority queue-based top-k selection algorithm again), where the priority key is the score of the document. For both tasks, results are returned in decreasing order of score.

Analysis

All priority queues in both tasks are implemented using min heaps. In task 1, there are k top-scoring documents to be found from n documents, hence the heap size is k, and there are at most r insertions/updates to the heap. As the height of the heap is $\log k$, the *worst case* time complexity of the top-k selection algorithm is $O(n+r\log k)$.

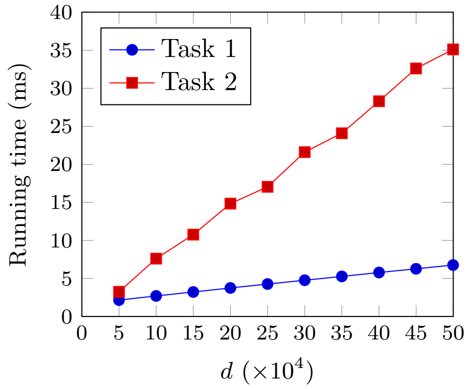
However, this worst case is highly improbable, instead it is more realistic to consider the *average case*. Note that only documents with nonzero scores can possibly be inserted to the priority queue, and there are r of them. The probability that the i^{th} document with nonzero score is in the top k-items out of the first i items is $\dfrac{\min(i,k)}{i}$. Therefore, the expected number of insertions/updates is $\displaystyle\sum_{i=1}^{r} \frac{\min(i,k)}{i} = k + \sum_{i=k+1}^{r} \frac{k}{i} \in O\left(k \log \frac{r}{k}\right)$. After taking account of reading the d entries from the document lists, the average time complexity of task 1 is $O(d + n + k \log \frac{r}{k} \log k)$. The fact that returning the results in decreasing order of score takes $O(k \log k)$ time is ignored as it is taken care of by the $O(k \log \frac{r}{k} \log k)$ term.

The average time complexity of task 2 differs in two ways. Firstly, whenever a document entry is extracted, D is updated in $O(\log w)$ time. Secondly, iteration occurs over r (instead of n) documents, but $r < d$, so this term is absorbed by the $O(d \log w)$ term. Therefore, the average time complexity of task 2 is $O(d \log w + k \log \frac{r}{k} \log k)$.

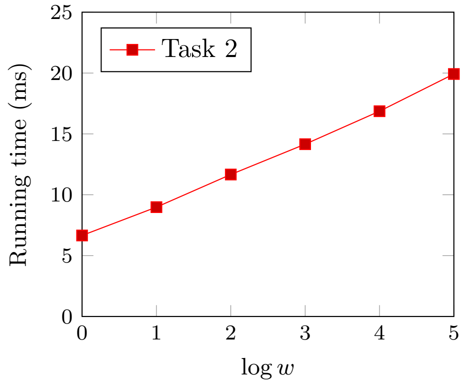
Verification

For most queries, it can be assumed that $d > n + k \log \frac{r}{k} \log k$ (unless the query does not contain common words, or k is very large). In this case, the average time complexities for Task 1 and Task 2 can be simplified to $O(d)$ and $O(d \log w)$ respectively. Graph 1 verifies the effect of d on the time complexities by plotting the running time of both tasks against large values of d (with $w = 5$). Graph 2 verifies the $\log w$ component of the task 2 time complexity by running several queries chosen to share similar values of d (between $121,300$ and $121,500$) for $w = 1, 2, 4, 8, 16, 32$. The average time for a query is obtained from 10^4 instances of the same query.

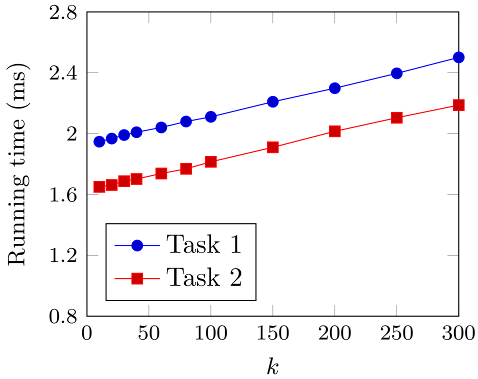
Graph 1: Effect of d on running time



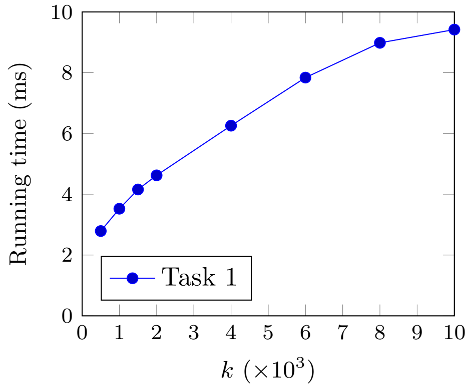
Graph 2: Effect of w on running time



Graph 3: Effect of k (smaller values) on running time



Graph 4: Effect of k (larger values) on running time



The plots in Graph 1 and Graph 2 follow a linear trend, although Task 2 in Graph 1 noticeably deviates from a straight line. This can be explained by the fact that the average number of operations in updating the priority queue D varies from query to query, hence the slight deviations. The faster growth in time for Task 2 is expected from the additional $\log w$ term, which is not present for Task 1. It is also noted that if both trends are extrapolated for $d = 0$, only Task 1 would have a nonzero y-intercept, which is adequately explained by the n term in its unsimplified time complexity, which verifies the presence/absence of the n term in Task 1/Task 2.

Graph 3 and Graph 4 consider the case where the query does not contain common words. Here, the value of d is relatively smaller ($d = 15732, r = 13270, w = 5$ for all data in Graph 3 and Graph 4), in order to investigate the effect of k. From both graphs, it is seen that Task 2 outperforms Task 1 in term of running time by a constant amount as the value of k varies, which is expected since $n > d \log w$.

However, the plots on Graph 3 appear to be linear in k. Theoretically, the effect of k on the running time should be $O(k \log \frac{r}{k} \log k)$, however it appears to be $O(k)$. A plot of the function $y = x\left(1 + \log \frac{r}{x}\right) \log x$ itself (added 1 to ensure nonzero y-value when $x = r$) also appears mostly linear, except for values of x that are close to r, where the growth slows down, which is confirmed by Graph 4. Therefore, this apparent discrepancy from the theoretical analysis is not too surprising.

Conclusion

The theoretical average time complexities of Task 1 and Task 2 have been found to be $O(d + n + k \log \frac{r}{k} \log k)$ and $O(d \log w + k \log \frac{r}{k} \log k)$ respectively. This has mostly been verified by experimental tests with varying input. The only interesting discrepancy is that $O(k \log \frac{r}{k} \log k)$ term for both tasks may be practically simplifed to $O(k)$ (as long as $k \ll r$). For most queries, which are likely to contain common words, d would be comparable to (if not larger than) n, in which case the average time complexities of Task 1 and Task 2 can be simplifed to $O(d)$ and $O(d \log w)$ respectively. Therefore, the approach of Task 1 is more preferrable than Task 2.

An exception to this would be the case where the query does not contain common words, and therefore $d + n$ may be larger than $d \log w$, which makes the approach of Task 2 more preferrable. In this case, the value of k would have a more pronounced effect on the time complexity for smaller values of d. Finally, note that if space is a concern, then Task 2 has the advantage of only using $O(k)$ space, whereas Task 1 would use $O(n)$ space.