COMP20007 Design of Algorithms, Semester 1, 2018

Assignment 1: Multi-word Queries

Task 3: Analysis of Algorithms

Overview

Both Task 1 and Task 1 used an inverted file index for a multi-word query to identify top-matching documents in a collection. However, the tasks differed significantly in their approach to organise, calculate and identify the top-matching documents. Task 1 utilised an array-based accumulation approach, while Task 2 utilised a Priority Queue-based merge approach. This report will analyse the effect that these approaches will have on the algorithms' utilisation of time resources, and the impact of the size and organisation of input parameters.

Asymptotic Time Complexity

Task 1

Task 1 identified the top k results through an array-based approach. The algorithm initialises an empty array of document scores and iterate over all the document and totalling the scores for each document. Once the total for each document has been found, a heap of n_results elements is created and the score array is iterated over, comparing the score of each document with the smallest score in the heap. If a higher document score is found, the minimum element is replaced and the heap is rebuilt. The results are then sorted and printed to the console.

```
let n be the number of documents, m be the number of terms and k be the number of  results
initialise_array():
            for 0 to n:                                                                          O(n)
                        score_arr[i] = 0.0                                                       O(1)
                                                                                    Total:       O(n)
sum_doc_scores():
            for 0…m:                                                                             O(m)
                        for 0…n:                                                                 O(n)
                                    score_arr[n] += score                                        O(1)
                                                                                    Total:    O(n * m)
get_top_k_results():
            for o..k:                                                                            O(k)
                        heap_insert( A[k] )                                                      O(1)
            for k…n:                                                                             O(n)
                        if score_arr[k] > heap_min():                                            O(1)
                                    heap_insert( A[k] )                                          O(log(k))
                                                                                Total:       O(n + klog(k) * log(n))
                                                        where k*log(n) is the expected number of update steps in total
print_top_k():
            merge_sort(heap)                                                                     O(klog(k))
            print_heap()                                                                         O(k)
                                                                                    Total:    O(klog(k) + k)
Therefore Algorithm is O(n * m)
```

Task 2

Task 2 identifies the top k elements through a heap-based approach. The array initialises and empty heap, storing lists of documents sorted on ascending document id, and a key of the id of the first element in the array (i.e. the smallest id). Once all document lists have been added to the heap, a heap of n_result documents is created, with the key being the score of the document. While there are still non-empty lists in the document heap, the algorithm iterates through the list, only accessing the first document. As they are sorted in ascending order, this is the document with the smallest id in the list. Therefore, one pass over all the document lists is guaranteed to access all instances of the same document that are stored in the document lists. The total score for this document is therefore calculated as the document lists are passed over. This total is compared with the minimum score stored in the top k score heap, and replaces the minimum element of the heap if it is greater. After each list is accessed and the score totalled, the first document in the list is removed and the document heap is rebuilt in order of ascending id of the first element in each list. The results are then sorted and printed to the console.

```
let n be the number of documents, m be the number of terms and k be the number of  results

intiialise_heap()
            return heap()                                                                        O(1)

get_doc_lists():
            for 0…m:
                        heap_insert(doc_list)                                                    O(log(m))
                                                                                Total: O(m + m * log(m)^2)

get_top_k_results():
            while(list in doclist):
                        Document doc
                        Document current_doc = heap_min(doc_heap)                                O(1)
                        if id of current_doc equal to id of doc:                        <= m − 1 in m iterations
```

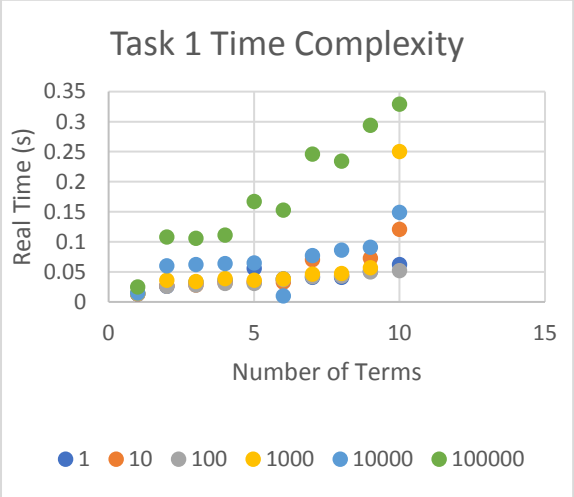
```
                        doc_score += score                                          O(1)
                    else                                                    >= 1 in m iterations
                        if doc_score > heap_min():                            k*log(n) iterations
                            heap_insert(doc)                                        O(log(k))
                        doc = current_doc
                    Remove doc from doc_list                                        O(log(1))
                    rearrange_doc_heap()                                            O(log(m))
                                                          Total: O(m * ( k * log(n) * log(k)) + log(m)))

print_top_k():
        merge_sort(heap)                                                            O(klog(k))
        print_heap()                                                                O(k)
                                                                            Total:  O(klog(k) + k)

Therefore algorithm is O(m * k * log(n) * log(k))
```

Input Parameters

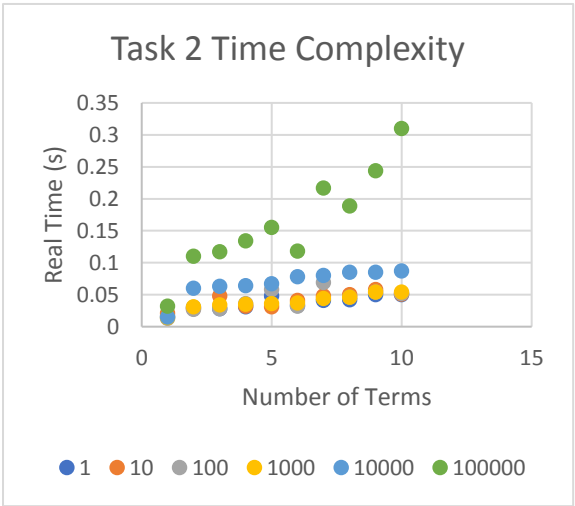
Task 1



The algorithmic complexity of Task 1 is in the order of $O(m * n)$, and therefore use of time resources of increases primarily as a function of the number and length of document lists that are input as parameters. As seen in the Time Complexity graph, both inputs can be seen to be contributing factors to increases in time complexity, as the time taken to iterate over the n documents in the m document lists dominates the run time of the algorithm.

The worst case scenario is one in which m and n are equal, in which the time complexity of the algorithm will decay to $O(n^2)$, in which the time taken will increase exponentially. As k increases, the time taken to sort the heap will increase and, as k is constrained by the value of n, the worst case scenario has a time complexity of $O(n * log(n))$. However, as the best, average and worst case scenario of heap sort are equal and use consistent memory, the complexity will still be in the order of $O(n * log(n))$.

Task 2



The algorithmic complexity of Task 2 is in the order of $O(m * k * log(n) * log(k))$, and so the increases in time complexity are primarily dependent on the number and length of document lists, as well as the number of results requested. This can be observed in the significant differentials between the trendlines for the number of terms requested, indicating that this is the dominating factor in the time complexity. This occurs due to the increase in time taken to iterate over the lists and update the heap of top k elements.

The worst case scenario is one on which k is equal to n, and n is equal to m, in which the time complexity will decay to $O((n * \log(n))^2)$. Similarly, the best, average and worst case scenario of sorting the heap will be in the order of $O(k * log(k))$.

Conclusion

In the general application of a text search algorithm in a large collection of documents, the largest input parameter is most likely to be the number of documents in the list. This would indicate that the preferred algorithmic approach will generally be the Priority Queue Merge approach used in Task 2. Therefore, for a large collection of documents and small number of terms entered, the Task 2 will be significantly more efficient. However, as the size of m and k increases, for m>n the time complexity of Task 2 will approach $O(m^2 * \log(n)^2)$, indicating that Task 1 may be more efficient ($O(m * n)$) in the worst case scenario. Moreover, the time complexity of the algorithm is not dependent on the ordering of the input parameters, as the lists of documents are assumed to be in ascending order, and the complexity of the heap sort algorithm ($O(k * \log(k))$) does not change with the input ordering of the list.