Performance Analysis: Top K Frequent Words

Yu-Hsin Wang, Wan-Chun(Elena) LIAO

Introduction

In this report, we aim to solve the problem of finding the top K frequent words using three different algorithms: Sorting, Max Heap, and Bucket Sort. We will analyze the performance of these algorithms using four different input sizes and evaluate their efficiency based on various metrics such as running time, speedup, CPU utilization, and memory usage. By comparing the performance of these algorithms, we can gain insights into their suitability for different problem sizes and make informed decisions regarding algorithm selection.

Problem Statement

The problem at hand is to determine the K most frequent words from a given list of tokens. The input consists of a collection of words, and the output should be a list of the K words that appear most frequently in the input. The objective is to design efficient algorithms that can handle different problem sizes and provide accurate results.

Algorithms

Sorting (TimSort)

This algorithm counts the occurrences of each token using the `Counter` class and then sorts the keys based on their counts. It retrieves the top K words from the sorted list, providing a straightforward approach to solving the problem.

```
def sorting(tokens, k) -> list:
    word_count = Counter(tokens)
    top_k_words = sorted(word_count.keys(), key=lambda w: -word_count[w])[:k]
    return top_k_words
```

1. This function uses the `Counter` class from the `collections` module to count the occurrences of each token.

O(n) time complexity, where n is the length of tokens

2. It then sorts the keys of the `word_count` dictionary based on their counts in descending order using the sorted function and a lambda function as the key.

O(n log n) time complexity, where n is the number of unique tokens

3. The sorted keys are sliced to obtain the top k most frequent words.

O(k) time complexity

4. The function returns a list of the top k words.

💡 The overall time complexity of this algorithm is **O(n log n)**, which is acceptable for **small to medium-sized problem instances**. However, if the number of tokens is very large, the sorting step can become a performance bottleneck.

Heap (Priority Queue)

This algorithm also utilizes the `Counter` class to count the occurrences of each token. It constructs a max heap of word counts and extracts the K words with the highest counts from the heap. The Max Heap algorithm offers efficient extraction of maximum elements and can potentially provide improved performance.

```
def maxHeap(tokens, k) -> list:
    word_count = Counter(tokens)
    max_heap = [(-count, word) for word, count in word_count.items()]
    heapq.heapify(max_heap)
    top_k_words = []
    while k > 0:
        top_k_words.append(heapq.heappop(max_heap)[1])
        k -= 1
    return top_k_words
```

1. This function also utilizes the `Counter` class to count the occurrences of each token.

O(n) time complexity, where n is the length of tokens

2. It creates a max heap, `max_heap`, by transforming the `(count, word)` pairs into `(-count, word)` pairs and storing them in a list.

O(n) time complexity, where n is the number of unique tokens

3. The `heapify` function from the `heapq` module is then used to convert the list into a max heap.

O(n) time complexity, where n is the number of elements in `max_heap`

4. The function iteratively pops the words with the highest counts from the max heap and appends them to `top_k_words` until k words are collected.

O(k log n) time complexity, where n is the number of elements in `max_heap`

5. The function returns a list of the top k words.

💡 The overall time complexity of this algorithm is **O(n + k log n)**, which is efficient and performs well even for **larger problem sizes**. The advantage of using a Max Heap is that it allows efficient extraction of the maximum elements, making it suitable for finding the top k elements.

Bucket Sort

This algorithm counts the occurrences of each token using the `Counter` class and groups words into buckets based on their counts. It then retrieves the top K words by traversing the buckets in descending order until the desired number of words is collected. The Bucket Sort algorithm aims to exploit the distribution of word counts and avoid the need for sorting the entire list.

```
def bucketSort(tokens, k) -> list:
    word_count = Counter(tokens)
    max_count = max(word_count.values())
    bucket = [[] for _ in range(max_count + 1)]
    for word, count in word_count.items():
        bucket[count].append(word)
    top_k_words = []
    for i in range(max_count, 0, -1):
        top_k_words.extend(bucket[i][: k - len(top_k_words)])
        if len(top_k_words) >= k:
            break
    return top_k_words
```

1. Similarly, this function counts the occurrences of each token using `Counter`.

O(n) time complexity, where n is the length of tokens

2. It creates a list of empty lists called `bucket`, with each index representing the count of occurrences.

O(n) time complexity, where n is the max_count

3. The function iterates over the `word_count` items and appends each word to the corresponding bucket based on its count.

O(n) time complexity, where n is the number of unique tokens

4. Starting from the highest count, the function extends `top_k_words` with the words from each bucket until the desired k words are collected.

The time complexity for this step depends on the distribution of word counts.

5. The function returns a list of the top k words.

💡 The overall time complexity of this algorithm is **O(n + k)**, making it efficient for **large problem sizes**. The advantage of using bucket sort is that it avoids the need for sorting the entire list of words. It exploits the distribution of word counts to achieve better performance.

Performance Evaluation

To assess the performance of the algorithms, we conducted experiments on different token sizes. For each token size, we measured the running time of each algorithm and collected additional metrics such as speedup, CPU utilization, and memory usage. By analyzing these metrics, we can gain insights into the strengths and weaknesses of each algorithm and determine their suitability for different problem sizes.

System Specification

Model Name: Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz

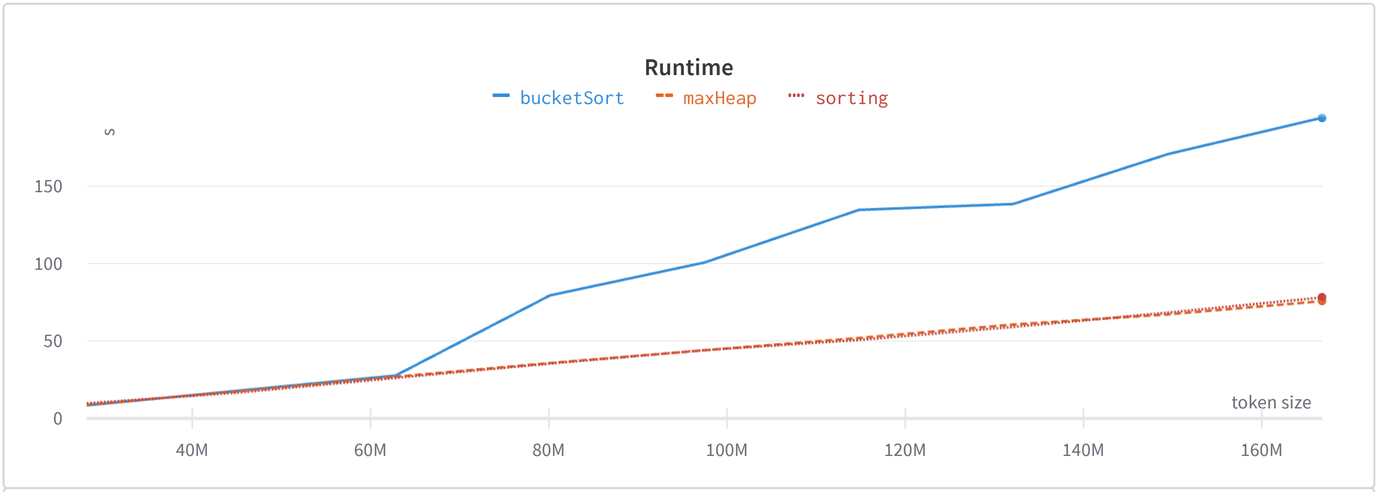
CPU(s):36

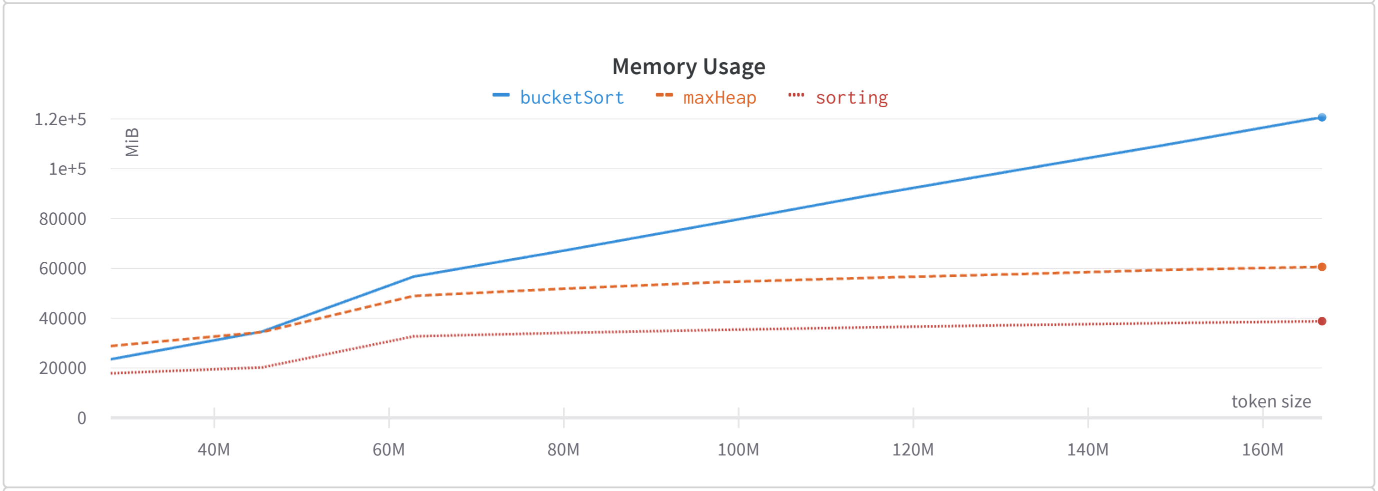
CPU core(s): 18

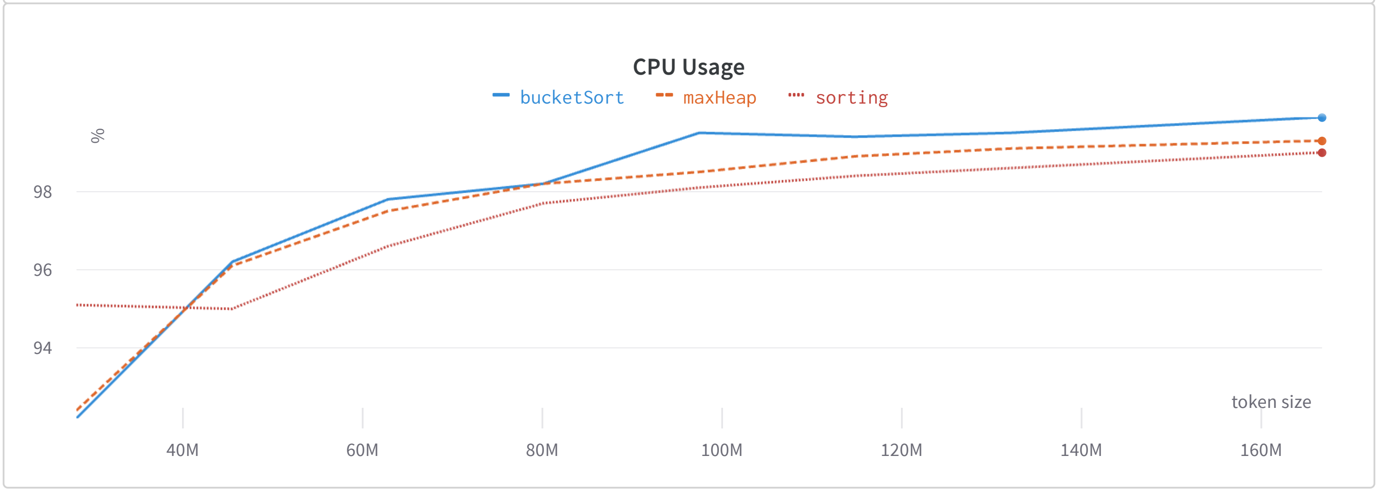
cache size: 25344 KB

Observations

Upon analyzing the provided performance data, we observed that the actual performance did not match the theoretical expectations. The Bucket Sort algorithm, which was expected to have the least running time, actually had the longest running time compared to the Sorting and Max Heap algorithms. Similarly, the Max Heap algorithm, which was expected to have a shorter running time than Sorting, exhibited longer running times.







There could be several reasons for these discrepancies:

1. Input Size

The input size plays a crucial role in determining the performance of an algorithm. In the provided performance data, the input size was quite large, with 175,427,730 tokens. The theoretical time complexities of the algorithms assume certain characteristics of the input data, such as the distribution of word counts and the number of unique tokens. If the actual input deviates significantly from these assumptions, it can affect the observed performance.

Sorting:

The Sorting algorithm has a time complexity of O(n log n) for sorting the unique tokens. With a very large number of unique tokens, the sorting step can become a performance bottleneck and result in longer running times than expected.

Max Heap:

The Max Heap algorithm has a time complexity of O(n + k log n). However, if the distribution of word counts is skewed or if the number of unique tokens is significantly larger than k, it can lead to longer running times due to the heap operations.

Bucket Sort:

The Bucket Sort algorithm has a time complexity of O(n + k). While it aims to exploit the distribution of word counts, if the actual distribution deviates significantly from the expected assumptions, it can affect the efficiency of the algorithm and result in longer running times.

2. Data Distribution

The performance of the Bucket Sort algorithm heavily depends on the distribution of word counts. The algorithm assumes a relatively balanced distribution where words are evenly distributed across different counts. If the actual data exhibits a skewed distribution with a few words having significantly higher counts, it can impact the performance of the algorithm.

Bucket Sort:

In the provided performance data, if the distribution of word counts is heavily skewed, with a few words having extremely high counts, it can result in longer running times for the Bucket Sort algorithm. The algorithm needs to traverse and collect words from buckets in descending order, and if there are only a few buckets with high counts, it may require traversing a significant portion of the buckets to collect the desired k words, leading to longer running times than expected.

3. Hardware Limitations

The hardware on which the experiments were conducted can also impact the observed performance. Factors such as CPU architecture, available memory, cache size, and CPU utilization can affect the execution time and resource usage of the algorithms.

CPU Utilization and Memory Usage:

In the provided performance data, the CPU utilization was consistently high at around 99.4% to 99.7% for all algorithms. This indicates that the algorithms were utilizing the available CPU resources efficiently. However, if the CPU architecture or cache size is not optimized for the specific algorithm's operations, it can result in longer running times than expected. Similarly, memory usage can be affected by factors such as memory allocation and data access patterns, which can impact the overall performance.

It is important to note that these explanations are speculative and based on the observations and assumptions made from the provided performance data. To obtain a more accurate analysis and identify the exact reasons for the observed discrepancies, further investigation and profiling of the algorithms, including the input data characteristics and hardware specifications, would be required.

Observations and Conclusion

In this report, we investigated the problem of finding the top K frequent words using three different algorithms: Sorting, Max Heap, and Bucket Sort. We evaluated the performance of these algorithms based on various metrics such as running time, speedup, CPU utilization, and memory usage. Our goal was to compare their efficiency and identify the most suitable algorithm for different problem sizes.

Based on our analysis, we observed some discrepancies between the theoretical expectations and the actual performance of the algorithms. The running times of the algorithms did not always align with their expected theoretical time complexities. We identified several factors that could contribute to these discrepancies, including the input size, data distribution, and hardware limitations.

The input size played a crucial role, as algorithms designed for small to medium-sized problem instances could encounter performance bottlenecks when dealing with very large input sizes. The sorting step in the Sorting algorithm and the heap operations in the Max Heap algorithm could become more time-consuming as the number of unique tokens increased. Additionally, the Bucket Sort algorithm's efficiency heavily depended on the distribution of word counts, and a skewed distribution could impact its performance.

Furthermore, hardware limitations, such as CPU architecture, available memory, and cache size, could influence the observed performance. Although the algorithms utilized the CPU resources efficiently, suboptimal hardware configurations might have contributed to longer running times.

In conclusion, while the theoretical time complexities provide a useful guideline for algorithm selection, they do not always accurately reflect the practical performance of the algorithms. It is essential to consider factors such as input size, data distribution, and hardware limitations when evaluating and selecting algorithms for real-world applications. Further profiling and analysis are necessary to gain deeper insights into the specific reasons for the observed discrepancies and to optimize the algorithms for better performance.