Analysis of Algorithms

BLG 335E

Project 2 Report

SELİN YILMAZ

yilmazsel21@itu.edu.tr

Faculty of Computer and Informatics Engineering

Department of Computer Engineering

Date of submission: 25.11.2024

1. Implementation

1.1. Sort the Collection by Age Using Counting Sort

Counting Sort is a non comparison-based sorting algorithm. It becomes particularly efficient when the range of input values is smaller than the number of elements to be sorted because it takes linear time complexity. For each distinct element in the input array, Counting Sort counts the frequency and uses that information to place the elements into correct positions. Counting sort is stable which means it preserves the relative order of elements (e.g. we have two 3 in our vector. Counting sort puts the first 3 before the second one in the sorted vector.).

As a pull-down, the counting sort needs additional memory for the counting array. So it is less efficient for larger numbers.

When implementing the code (ascending part), I followed these steps:

1. **Find the range:** The range needs to be from 0 to max number in the array. So I used a `getMax()` function to find the maximum age.
2. **Count frequencies:** I found how many times a value is used in the given array.
3. **Cumulative addition:** Every element in the count array has the addition of less than or equal to itself in it (cumulative sum array).
4. **Create the sorted array:** Starting from the end of the *items* vector, place the elements into their correct position.

The pseudocode I followed when doing the implementation is:

Algorithm 1 Counting Sort - Ascending

```
for $i \leftarrow 0$ to $k-1$ do
    $C[i] \leftarrow 0$
end for
for $j \leftarrow 0$ to $n-1$ do
    $C[A[j]] \leftarrow C[A[j]] + 1$
end for
for $i \leftarrow 1$ to $k$ do
    $C[i] \leftarrow C[i] + C[i-1]$
end for
for $j \leftarrow n-1$ down to $0$ do
    $B[C[A[j]] - 1] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
end for
```

This was for ascending order. For the descending order, I followed:

Algorithm 2 Counting Sort - Descending

```
for $i \leftarrow 0$ to $k-1$ do
    $C[i] \leftarrow 0$
end for
for $j \leftarrow 0$ to $n-1$ do
    $C[A[j]] \leftarrow C[A[j]] + 1$
end for
for $i \leftarrow k-1$ to $0$ do
    $C[i] \leftarrow C[i] + C[i+1]$
end for
for $j \leftarrow n-1$ down to $0$ do
    $B[C[A[j]] - 1] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
end for
```

To observe how much performance changes, I tested the sorting on different sizes of datasets. Also, I sorted each of them both ascending and descending order. The results can be seen in Table 1.1.

File Name	Order	Execution Time (s)
items_l.csv	Ascending	0.00239829
items_l.csv	Descending	0.00713725
items_m.csv	Ascending	0.000796362
items_m.csv	Descending	0.00218137
items_s.csv	Ascending	0.000454535
items_s.csv	Descending	0.000815163

Table 1.1: Counting Sort Execution Times

It can be derived that, the dataset size impacts the sorting time. Also, the ordering type slightly changes the passed time. The reason there is not a sharp difference is the complexity of counting sort being $O(n+k)$. *n* stands for the number of elements in the input array, and *k* stands for the number of elements in the counting array.

1.2. Calculate Rarity Scores Using Age Windows (with a Probability Twist)

To find the *rarity score*, the following formula is given to us.

$$\text{R} = (1 - \text{P}) * (1 + \text{age} / \text{age}_{max})$$

ageMax here indicates the biggest age in the dataset and can be found using *getMax()* function. *P* stands for probability, and it is calculated using the following formula:

$$P = \begin{cases} \frac{\text{countSimilar}}{\text{countTotal}} & \text{if countTotal} > 0 \\ 0 & \text{otherwise} \end{cases}$$

countTotal is the number of elements that are in the given range. The range is found by adding and subtracting the *nearAgeWindow* from age.

countSimilar is the number of elements that have the same origin and type within the range.

Pseudocode I followed during the implementation is below.

Algorithm 3 Calculate Rarity Score (A, w)

```
for $i \leftarrow 0$ to $n - 1$ do
  $countTotal \leftarrow 0$
  $countSimilar \leftarrow 0$
  $minAge \leftarrow A[i] - w$
  $maxAge \leftarrow A[i] + w$
  $probability \leftarrow 0.0$
  for $j \leftarrow 0$ to $n - 1$ do
    if $A[j] \geq minAge$ and $A[j] \leq maxAge$ then
      $countTotal \leftarrow countTotal + 1$
      if $A[j].type = A[i].type$ and $A[j].origin = A[i].origin$ then
        $countSimilar \leftarrow countSimilar + 1$
      end if
    end if
  end for
  if $countTotal \neq 0$ then
    $probability \leftarrow countSimilar/countTotal$
  end if
  $ageMax \leftarrow getMax(A)$
  $A[i].rarityScore \leftarrow (1 - probability) \cdot (1 + A[i].age/ageMax)$
end for
```

To observe how much performance changes, I tested the calculation on different sizes of datasets. These datasets are the sorted datasets from the previous part. I calculated each of them both ascending and descending order. The results can be seen in Table 1.2.

File Name	Order	Execution Time (s)
items_l.csv	Ascending	29.7758
items_l.csv	Descending	23.6631
items_m.csv	Ascending	5.01343
items_m.csv	Descending	3.41248
items_s.csv	Ascending	1.36317
items_s.csv	Descending	0.85581

Table 1.2: Rarity Score Calculation Times

It can be observed that there is a dramatic performance change between large and medium datasets. This occurred because the time complexity of *calculateRarityScore()* function is $O(n^2)$.

By changing some parameters we can affect the rarity score.

1. Changing the near-age window we can change rarity. A larger window means increasing the number of comparable artifacts which results in the reduction in rarity. In contrast, a smaller window potentially increases the rarity because it becomes more unique.
2. Older artifacts (bigger ages) contribute more to the rarity score.
3. If countSimilar = countTotal, P becomes 1 and it makes R=0. Which means the artifact is not rare.

1.3. Sort by Rarity Using Heap Sort

Heap sort is a comparison-based sorting algorithm. It divides its inputs into a sorted and an unsorted region. Then, it iteratively decreases the unsorted side by getting the largest element from it to put it in the sorted side. It works as an in-place algorithm so it does not require extra space. However, it is not stable.

It consists of two functions: *heapify()* and *heapSortByRarity()*. Pseudocode for *heapify* when descending is false is as follows. When descending is true, only

Algorithm 4 Heapify(A, i) - Ascending

```
$l \leftarrow 2 * i + 1$
$r \leftarrow 2 * i + 2$
$largest \leftarrow i$
if $l < A.heap-size$ and $A[l] > A[largest]$ then
    $largest \leftarrow l$
end if
if $r < A.heap-size$ and $A[r] > A[largest]$ then
    $largest \leftarrow r$
end if
if $largest \neq i$ then
    exchange $A[i]$ with $A[largest]$
    Heapify$(A, largest)$
end if
```

comparisons in the first and second if statements will change to <.

The pseudocode for *heapSortByRarity* when descending is false is as follows.

To observe how much performance changes, I tested the sorting on different sizes of datasets. These datasets are sorted by age, and their rarity scores are calculated

Algorithm 5 Heap Sort By Rarity(A) - Ascending

```
for $i \leftarrow A.heap-size/2-1$ downto $0$ do
    $Heapify(A, i)$
end for
for $i \leftarrow A.heap-size-1$ downto $1$ do
    exchange $A[i]$ with $A[0]$
    $A.heap-size \leftarrow A.heap-size-1$
    $Heapify(A, 0)$
end for
```

File Name	Order	Execution Time (s)
items_la.csv	Ascending	0.0462018
items_ld.csv	Descending	0.0299665
items_ma.csv	Ascending	0.0107372
items_md.csv	Descending	0.00959945
items_sa.csv	Ascending	0.0044823
items_sd.csv	Descending	0.00507584

Table 1.3: Heap Sort Execution Times

beforehand. The execution times of heap sorting by their rarity scores can be seen from Table 1.3.

It can be seen that there is a rather small performance change between dataset sizes because of the heap sort's complexity. *heapify()* function takes $O(n)$ times, and building the heap takes n-1 times $O(\log n)$ which is equal to $O(\log n)$. $O(n)$ + $O(n \log n)$ = $O(n \log n)$.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 4 edition, 2022.