Analysis of Algorithms

BLG 335E

Project 2 Report

Hilal Kartal

kartalh21@itu.edu.tr

Faculty of Computer and Informatics Engineering

Department of Computer Engineering

Date of submission: 20.11.2024

1. Implementation

1.1. Sort the Collection by Age Using Counting Sort

1.1.1. Theoretical Background

- Counting Sort is an non-comparison based sorting algorithm. It counts the amount of time an element shows up and uses these values the place them into their respective spaces.
- Time Complexity: As the algorithm goes over two arrays, the time needed is the time used to go over these arrays.
  - Input array size: N
  - Index array size: M
  - Complexity: $O(N + M)$

1.1.2. Technical Details

Counting Sort Code for Ascending:

```
    std::vector<Item> output_array(items.size());
    if(ascending == true){
        if(attribute == "age"){

            int index_array_size = getMax(items, "age");

            std::vector<int> index_array(index_array_size + 1, 0);

            for(Item item: items){
                index_array[item.age]++;
            }

            for(int i = 0; i < index_array.size() - 2; i++){
                index_array[i+1] = index_array[i] +
                   index_array[i+1];
            }

            for(int j = index_array.size() - 1; j > 0; j--){
                index_array[j] = index_array[j - 1];
            }
            index_array[0] = 0;

```

```
22          for(Item item : items){
23              int index = index_array[item.age];
24              output_array[index] = item;
25              index_array[item.age]++;
26          }
27      }
28  }
29  ...
```

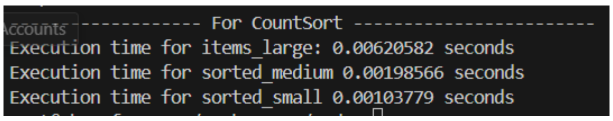
- Line 1: Defined aa output array of same size with input array.
- Lines 2 and 3: Checking the conditions for $ascending$ and $attribute$
- Line 5: Define the size of the index array with the max element of the input array.
- Line 7: Define the vector for index array with all elements initiliazed to 0.
- Lines 9, 10 and 11: For each element in input array increment their value by 1 in the index array.
- Lines 13, 14 and 15: Sum two consecituve indexes together.
- Lines 17, 18, 19 and 20: Shift the indexes to right by 1 to get the starting indexes.
- Lines 22, 23, 24 and 25: Go over input array once more and place each value to their respective place in output array by checking it from index array.

Counting Sort Code for Descending:

```
1   std::vector<Item> output_array(items.size());
2   }else if(ascending == false){
3       if(attribute == "age"){
4
5           int index_array_size = getMax(items, "age");
6
7           std::vector<int> index_array(index_array_size + 1, 0);
8
9           for(Item item: items){
10              index_array[item.age]++;
11          }
12
13          for(int i = 0; i < index_array.size() - 2; i++){
14              index_array[i+1] = index_array[i] +
                   index_array[i+1];
15          }
16
17            for(int j = index_array.size() - 1; j > 0; j--){
18                index_array[j] = index_array[j - 1];
19            }
20            index_array[0] = 0;
21
22            for(Item item : items){
23                int index = index_array[item.age];
24                output_array[(output_array.size() - 1) - index] =
                      item;
25                index_array[item.age]++;
26            }
27        }
28    ...
```

- Line 1: Defined an output array of same size with input array.
- Lines 2 and 3: Checking the conditions for $ascending$ and $attribute$
- Line 5: Define the size of the index array with the max element of the input array.
- Line 7: Define the vector for index array with all elements initiliazed to 0.
- Lines 9, 10 and 11: For each element in input array increment their value by 1 in the index array.
- Lines 13, 14 and 15: Sum two consecituve indexes together.
- Lines 17, 18, 19 and 20: Shift the indexes to right by 1 to get the starting indexes.
- Lines 22, 23, 24 and 25: Go over input array once more and place each value to their respective place in output array by checking it from index array.

1.1.3. Benchmarking Sorting Algorithms



```
------------ For CountSort -----------------------
Execution time for items_large: 0.00620582 seconds
Execution time for sorted_medium 0.00198566 seconds
Execution time for sorted_small 0.00103779 seconds
```

Figure 1.1: Times for Counting Sort

As the dataset gets smaller, time required for the algorithm to run also gets smaller.

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

1.2.1. Theoretical Background

To calculate the rarity scores for each item, we need to count the similar and total items in the same age window, and we also need to calculate the probablity.

- Rarity is calculated by: $R = (1 - P)\left(1 + \frac{\text{age}}{\text{agemax}}\right)$
- Probablity is calculated by: $P = \begin{cases} \frac{\text{countSimilar}}{\text{countTotal}} & \text{if countTotal} > 0, \\ 0 & \text{otherwise.} \end{cases}$

While in the given case the array comes as sorted, we should consider the cases it may not come as sorted so the code goes over all of the items in the array while look,ng for similar and total items.

1.2.2. Technical Details

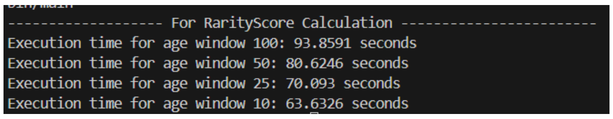
Rarity Score Calculation Code:

```
    if (items.empty() == true){
        return;
    }
    for(Item& item : items){
        int min_age = item.age - ageWindow;
        int max_age = item.age + ageWindow;
        int countSimilar = 0;
        int countTotal = 0;

        for(Item subitem : items){
            if(subitem.age >= min_age && subitem.age <= max_age){
                countTotal++;
                if(subitem.type == item.type && subitem.origin ==
                   item.origin){
                     countSimilar++;
                }
            }
        }

        double probablity = 0.0;
        if(countTotal > 0){
            probablity = static_cast<double> (countSimilar) /
               static_cast<double> (countTotal);
        }
```

```
23          item.rarityScore = (double) (1.0 - probablity) * (1.0 +
               (static_cast<double>
               (item.age)/static_cast<double>(getMax(items, "age"))));
24      }
```

- Line 1, 2 and 3: Checking if the array is empty.
- Line 4: Entering a foreach loop to calculate and assign the $rarityscores$ of each item.
- Lines 5 and 6: Initializing the minimum and maximum ages from the $ageWindow$.
- Lines 7 and 8: Initializing the similar and total counts.
- Line 10: Entering the foreach loop to go over each item and check if they are in the age range or they are similar.
- Lines 11 and 12: Checks if the item is in the age range. If it is increments the total count.
- Lines 13 and 14: Checks if the item is a similar item. If it is increments the similar count.
- Line 19: Initializes the probablity.
- Lines 20 and 21: Checks the condition for probablity and calculates accordingly.
- Line 23: Calculates and assigns the rarity score for the item.

1.2.3. Defining & Analyzing Different Rarity Score Calculations



```
bin/main
------------------ For RarityScore Calculation -----------------------
Execution time for age window 100: 93.8591 seconds
Execution time for age window 50: 80.6246 seconds
Execution time for age window 25: 70.093 seconds
Execution time for age window 10: 63.6326 seconds
```

Figure 1.2: Times for Different Age Windows for Rarity Score

As the age window gets smaller time required to calculate it also gets smaller.

1.3. Sort by Rarity Using Heap Sort

1.3.1. Theoretical Background

- HeapSort uses complete binary trees to store and compare values.
- Values are either stored descendingly(max heap) or ascendingly(min heap).

- After getting the values $heapified$, heapsort takes the root, adds it to the end of the array and heapifies the reamining array recursively.
- Time Complexity: First we create a heap using $heapify$, then we delete each element one by one.
  - Heapify: $n * log(n) \rightarrow$ (n insertion and log(n) adjustments)
  - Deletion of the elements: $n * log(n) \rightarrow$ (n deletion and log(n) adjustments)
  - Complexity: $2 * n * log(n) \rightarrow O(n * log(n))$

1.3.2. Technical Details

Heapify Code for Descending :

```
1      if(descending == false){
2          int left_child = 2*i + 1;
3          int right_child = 2*i + 2;
4          int biggest = i;
5          if(left_child < n && items[left_child].rarityScore >
              items[biggest].rarityScore){
6              biggest = left_child;
7          }
8
9          if(right_child < n && items[right_child].rarityScore >
              items[biggest].rarityScore){
10             biggest = right_child;
11         }
12
13         if (biggest != i) {
14         std::swap(items[i], items[biggest]);
15         heapify(items, n, biggest, descending);
16         }
17     }else if(descending == true)
18     ...
```

- Line 1: Checking the conditions for $descending$
- Lines 2, 3 and 4: Initializing the left and right childs indexes and initializing the biggest element as parent.
- Line 5, 6 and 7: Comparing left child with the parent to figure out who is bigger.
- Line 9, 10 and 11: Comparing right child with the current biggest to figure out who is bigger.

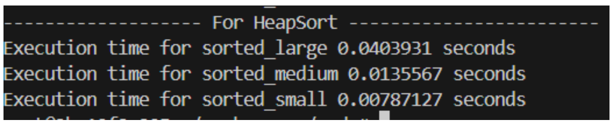
- Lines 13 and 14: Swapping the parent with the current biggest if the biggest changed from the parent.
- Line 15: Going back to heapify from the changed side recursively.

HeapSort Code:

```
1    for (int i = items.size() / 2 - 1; i >= 0; i--) {
2        heapify(items, items.size(), i, descending);
3    }
4
5    for (int i = items.size() - 1; i > 0; i--) {
6        std::swap(items[0], items[i]);
7        heapify(items, i, 0, descending);
8    }
9
10   return items;
```

- Line 1, 2 and 3: Heapifying the array. It starts from middle and comes to the left side as it heapifies the right side.
- Line 5, 6 and 7: Adding the last element at the end and heapfing the remaining array.

1.3.3. Benchmarking Sorting Algorithms



```
------------------ For HeapSort -----------------------
Execution time for sorted_large 0.0403931 seconds
Execution time for sorted_medium 0.0135567 seconds
Execution time for sorted_small 0.00787127 seconds
```

Figure 1.3: Times for Heap Sort

As the dataset gets smaller, time required for the algorithm to run also gets smaller.

1.4. Analysis of Results

- When sorting the values by age, we preferred to use counting sort as it is more effective with repeating values even if it requires additional space.
- When sorting the values by rarity scores, we preferred to use heap sort as the values are different from each other by such small amounts.
- The most time consuming part is the calculation of rarity scores as it goes over the list N^2 times.