Fuzzy Sorting of Intervals

With Comparison To Bubble Sort and Quicksort

Abstract

There are many different algorithms that can be used to sort data containing intervals such as dates or timespans. In this report I tackle the Fuzzy Sorting of Intervals problem put forward by Cormen, et. al. (2009), by implementing and analyzing a fuzzy sorting algorithm that will focus on sorting intervals of number ranges and comparing it to implementations of quick sort and bubble sort. All algorithms performed differently and showed how different algorithms have their own strengths and weaknesses when tackling this specific computational problem. I found that depending on the amount of resources available, the amount of iterations possible and the accuracy needed each algorithm could be a viable solution. For large amounts of data and low amount of computational resources, fuzzy sorting came out as the true winner as its required resources were low compared to the two other algorithms while delivering a similar result.

Introduction

For this project I decided to investigate the fuzzy sorting of intervals problem, described in 'Introduction to Algorithms' by Cormen, et al. (2009). For comparison purposes I implemented the fuzzy sorting algorithm alongside the bubble sort and quick sort algorithm. All three algorithms were implemented in the C# programming language, which is a higher level language meaning that it has active garbage collection and while memory management is possible it was not used.

Problem

The fuzzy sorting of intervals problem put forward by Cormen, et. al. (2009) can be summarized as: We wish to fuzzy sort a set of closed intervals **(*n*)** with the form of **[*a_i*, *b_i*]** where ***a_i*** is less than or equal to ***b_i***. With the fuzzy sorting we wish to create a permutation of the intervals that for ***j = ...n*** there exist ***c_j*** which satisfies ***c_1 <= c_2 … <= c_n***.
The algorithm must be randomized for the fuzzy-sorting of the intervals and should have a general structure that quicksorts the leftend-points (***a_i***) but also takes into consideration the overlapping intervals to improve the running time. The more overlapping intervals that exist the easier it should become for the algorithm to sort the intervals.
It should be implemented such that the algorithm's expected runtime is equal to ***O(n log n)*** but also have expected runtime of ***O(n)*** where the intervals overlap. The performance should naturally improve as the overlap increases. (Cormen, et. al.)

Theory

In the following section I will be going through the theory behind the three sorting algorithms used for this project.

Quicksort

Quicksort is a divide and conquer algorithm that takes a given entry in the provided array and uses said entry as a pivot for sorting the other entries in the array. Quicksort is also commonly referred to as 'quick partitioning sort', due to the nature of the algorithm. The partitioning is always done around the pivot. (Cormen, et. al.)
Due to the nature of this algorithm its expected worst-case running time is defined as **O(*n* lg *n*)**.

Fuzzy Sort

The fuzzy sorting algorithm is based off of the quick sorting algorithm described in the previous section, with the key difference being that it randomly selects a left endpoint for the pivot and then quick sort the left endpoints. The fuzzy sorting algorithm also does away with some of the sorting as it does not have to sort overlapping intervals; i.e. if we have intervals $[\mathbf{a}_i, \mathbf{b}_i]$ and $[\mathbf{a}_j, \mathbf{b}_j]$ then we just choose the intersection of the two such that $\mathbf{c}_i <= \mathbf{c}_j$ or $\mathbf{c}_j <= \mathbf{c}_i$.
Depending on the presence of overlapping intervals, the fuzzy sorting algorithm can have two potential running times. The worst-case expected running time, if no overlapping intervals are present, is **O(*n* lg *n*)**. Should there be overlapping intervals the expected worst-case running time would instead be **O(n)**. (Cormen, et. al.)

Bubble Sort

Bubble sort is a type of comparison sorting algorithm, given its name due to how it takes the higher value entries in a given array and "bubble" them to the top of the list. An example of this is if we have a list of numbers, e.g. (5 1 4 2 8). Then in the first pass of the algorithm 5 will move up until it hits the 8, leaving the list as (1 4 2 5 8). The second pass will then grab 4 and move it up until it hits the 5, leaving us with the result of (1 2 4 5 8).
Since the bubble sorting algorithm works in this way, the worst-case running time of the algorithm is defined as **O($\mathbf{N^2}$)**.

Implementation

The overall implementation of these sorting algorithms was made in a C# CLI application, meaning that the algorithms can be run using commands through the command prompt. The application comes with six different runtime options; generating data, clearing memory, running each individual sorting algorithm, and the option to run all algorithms at the same time using threading.

Data Used

The data used for this implementation is randomly generated by a static class within the application. When the application is running and the "Generate Data" option is selected the following code will be run:

```
GenerateRandomIntervals(amount, intervalRange)
```

```
    result = A;

    for index = 0 to index < amount
        start = Random number between 0 and intervalRange
        end = Random number between start value and intervalRange + 1
        entry = new Interval(start, end, index)
        Add entry to result

    return result
```

The intervals themselves are made as a data container class which holds the start, end, and number values. The number value itself is only used for evaluation purposes and does not interact with any of the sorting algorithms.

```
Interval
    start = Start of interval
    end = End of interval
    number = Original order number of the interval
```

Base Sorter Class

All three sorting algorithms all derive from the same base sorting class. The reason for this is that they all share the data swapping method and the quick sort and fuzzy sorting algorithms also share the method for finding the intersection with a random pivot.

```
Swap(A, $a_i$, $b_i$)
    temp = intervals[$b_i$]
    intervals[$b_i$] = intervals[$a_i$]
    intervals[$a_i$] = temp
```

The swapping method, as depicted above, takes in the array of intervals alongside the index value of two intervals within the array. It then exchanges the position of the interval found at $\boldsymbol{a_i}$ with $\boldsymbol{b_i}$ and vice versa.

```
FindIntersectionWithRandomPivot(A, a, b)
    size = b - a + 1;
    R = random integer number between 0 and size
    P = A[R]

    Swap(A, R, b)
    intersection = A(P.a, P.b)

    for i = a while i <= b - 1
```

```
        if intervals[i].b compared to intersection.a > 0 or
            Intervals[i].a compared to intersection.b < 0 then continue

        if intervals[i].a compared to intersection.a > 0 then
            intersection.a = intervals[i].a
        if intervals[i].b compared to intersection.b < 0 then
            intersection.b = intervals[i].b

    return intersection;
```

The method used for finding the intersection used in the fuzzy and quick sort algorithms takes in the array of intervals alongside a starting and end point. These points are then used to generate an interval object with random start and end points. This is done based on the random pivot point of R, which is a number between 0 and the delta between the end and start point plus one.

Fuzzy Sort

The fuzzy sorting implemented in this project has three unique methods besides those shared in the base sorting class. The first one being the main sorting method.

```
FuzzySort(A, p, r)
   if (p >= r) return;
   (a, b) = FindIntersectionWithRandomPivot(A, p, r)
   t = PartitionRight(A, (a, b), p, r)
   q = PartitionLeftMiddle(A, (a, b), t, p, r)

   FuzzySort(intervals, p, q - 1)
   FuzzySort(intervals, t + 1, r)
```

The main sorting method is a recursive method that calls itself after each partition iteration. For each iteration elements are partitioned either left or right of the randomized intersection. The first recursive iteration is focused on the elements that have been partitioned into the left/middle section in the current iteration. Once this is done, a recursive call will then be made to the elements partitioned in the right column to sort those elements.

```
PartitionRight(A, (a, b), p, r)
   if (A == null || (a, b) == null || p > r || r < 0) return -1
   index = p - 1

   for i = p, while i <= r - 1, i++
       current = A[i]
       if current.a compared to (a, b).a > 0 continue
       index += 1
```

```
        Swap(A, index, i)

    Swap(A, index + 1, r)
    return index + 1
```

The PartitionRight method takes in the array of entries, the randomized intersection and a start and end point. It then loops through the array of entries and compares each element to the intersection. Should the starting point of a given entry be higher than that of the intersection it is then swapped.

```
PartitionLeftMiddle(A, (a, b), r, p, q)
    index = p - 1;
    if (A == null || (a, b) == null || r == -1 || p > q|| q> 0) return -1

    for i = p, while i <= r - 1, i++
        current = A[i]
        if current.b compared to (a, b).b >= 0) continue
        index += 1
        Swap(A, index, i)

    Swap(A, index + 1, r)
    return index + 1
```

The same approach is also used in the PartitionLeftMiddle method, but instead focuses on the end value of both the given entry in the array and the intersection, as can be seen in the above pseudo-code.

Quicksort

As the fuzzy sorting algorithm in this project is based on the quicksort algorithm, they do share most of their logic.

```
Sort(A, p, r)
    if p >= r return
    (a, b) = FindIntersectionWithRandomPivot(A, p, r)
    t = Partition(A, (a, b), p, r)

    Sort(A, p, t - 1)
    Sort(A, t + 1, r)
```

The quicksort algorithm implemented in this project starts by finding the randomized intersection within the given array of entries. It then partitions the entries based on this intersection, and then recursively loops back over itself.

```
Partition(A,(a, b), p, r)
   index = p - 1;
   if (A == null || (a, b) == null || p > r || r < 0) return -1

   for i = p, while i < end, i++
       current = A[i]
       if current.a compared to (a, b).a >= 0 continue
       index += 1
       Swap(A, index, i)

   Swap(A, index + 1, r)
   return index + 1
```

While the fuzzy sorting partitions the entries both left and right, the quicksort implemented in this project only partitions the entries to one side, meaning that it only checks for the start of the interval and not the end. It is the same partitioning logic used by the PartitionRight method of the fuzzy implementation.

Bubble Sort

The bubble sorting algorithm implemented in this project only consists of one method which consists of a nested for-loop.

```
Sort(A)
   N = A.n - 1;

   for i = N, while i > 0, i--
       swapped = false
       for j = 0, while j < i, j++
           if A[j].a <= A[j + 1].a continue
           Swap(A, j, j + 1)
           swapped = true

       if !swapped break
```

The sorting method takes the amount of entries found in the given array and starts at the end of that number. It then loops backwards through that while performing a nested for-loop on the entries within the entry range of i, which gets smaller and smaller for each iteration of the first for-loop. The method basically takes the entries and sorts them one step to the right for each iteration until there are no more entries that are being swapped to the right.

Performance & Comparisons

In order to determine the performance and effectiveness of the three algorithms 7 test cases were run. The cases were set up as follows and yielded the given results:

	Fuzzy Performance		Quicksort Performance		Bubble Performance	
Data Size	Time (ms)	Memory (bytes)	Time (ms)	Memory (bytes)	Time (ms)	Memory (bytes)
200 Entries - Range 100	1.2977ms	848 bytes	0.7301ms	5040 bytes	0.9074ms	624 bytes
200 Entries - Range 400	0.8709ms	980 bytes	1.4121ms	4848 bytes	1.0066ms	624 bytes
800 Entries - Range 100	1.5898ms	720 bytes	1.0497ms	23024 bytes	7.7377ms	624 bytes
800 Entries - Range 400	1.2954ms	720 bytes	1.8163ms	18928 bytes	7.4337ms	624 bytes
2000 Entries - Range 200	0.889ms	816 bytes	2.2803ms	58224 bytes	44.5899ms	624 bytes
2000 Entries - Range 1000	1.8297ms	784 bytes	2.3509ms	45424 bytes	44.3876ms	624 bytes
5000 Entries - Range 300	4.0576ms	2832 bytes	34.4658ms	1504524 bytes	289.08ms	624 bytes

Fuzzy Sorting Performance

Through testing with the 7 different data sizes, the fuzzy sorting algorithm was the one that seemed to perform the best in terms of both memory usage and execution time. However, since all but one test were done with only one iteration, due to the nature of fuzzy sort, the middle section was not sorted correctly due to high amounts of overlapping intervals.
From the data it can be seen that in the first 6 test cases, where only one iteration was run, the speed of the algorithm would be significantly shorter when the algorithm had to sort entries with shorter interval ranges. However, it still performed at amazing speeds even in the final test run.

Quick Sort Performance

The quick sort algorithm used in this project was the most memory consuming out of the three. As the amount of entries grew, the more memory the algorithm needed to actively distinguish between the different entries.
In terms of speed, the quick sort algorithm did well on the first 6 test runs, the ones where only one iteration was run. However, in the seventh test where a total of 10 iterations were ran, the speed of the algorithm greatly diminished.

Bubble Sort Performance

The bubble sort algorithm started out with fast performance time on the first two test cases run, as it only had to process 200 entries. The variance of time range did not seem to affect the processing speed in any of the first 6 test cases, as it can be seen that only a change of 0.1ms seems to happen.
As for the memory usage of the algorithm, it would seem that in this implementation of the algorithm no additional objects were created while sorting. The algorithm therefore only used the memory that was already in use by the original list of entries.

Comparison

When comparing all of the data collected through the seven test cases, it is clear that each algorithm had their own strong areas, and each their own weaknesses.
When looking at the results of the sorting (see results folder in the Git repository), it can be seen that on the first 6 test cases that the fuzzy sorting algorithm did not, as expected, sort the overlapping intervals. However, when multiple iterations were run on the same data, the fuzzy algorithm would sort all intervals into the correct order while still keeping both the duration and the memory usage to a minimum.
Bubble and quick sort did manage to produce the desired result, but they either used more memory or a longer time to produce said result than the fuzzy sort algorithm.

Conclusion

Throughout this project I tried to implement a version of the fuzzy sorting algorithm that could help solve the problem stated in the beginning of this paper. Through my use of the bubble sort and quick sort algorithms, I came to understand the problem further and how the performance of the aforementioned algorithms would falter when used on a larger scale than the tests run in this experiment.
From my analysis of the performance metrics I came to the understanding that my implementation of the fuzzy sorting algorithm achieved its goal in replicating the one initially described in the problem. Furthermore, I came to understand that the computational problem of sorting intervals can be solved using multiple different approaches however all have their own drawbacks and benefits. I can therefore conclude that the method used for solving the computational problem of sorting intervals should be chosen based on the context of the use case.