INF421 - Convex Hull

Marco Paina, Jonas Wehrung-Montpezat

January 2023

1 Introduction

This project aims to solve the following problem: given a set of points in the plane, find its convex hull. The study is based on four different types of random datasets, where the convex hull shall be computed using two different algorithms.

2 The algorithms

2.1 Sweep

Algorithm 1 Sweep algorithm (upper hull)

```
sort(dataset)
$i \leftarrow 1$
hull = []
while i < size(dataset) do
  $c \leftarrow$ dataset[i]
  if hull.size() < 2 then
    hull.add(c)
    $i \leftarrow i + 1$
  else
    $a, b \leftarrow$ hull[-2], hull[-1]
    if Triangle (a,b,c) is clockwise then
      hull.add(c)
      $i \leftarrow i + 1$
    else
      hull.pop()
    end if
  end if
end while
return hull
```

Once the upper hull is computed, we apply a rotation of angle π so that the new set's upper hull corresponds to the original set's lower hull. After removing the redundancies, we get all points of the convex hull clockwise.

Let us take a look at the complexity of the algorithm. First, we sort the dataset in $O(n * log(n))$ time. Then, we enter the while loop. Each iteration of the loop happens in O(1). We just need to count how many times we enter the loop.

Let p be the number of points added and then removed from the hull. Then, we can see that $i + p$ increases by one in each iteration. Moreover, each point is added to the hull exactly once. Thus, with

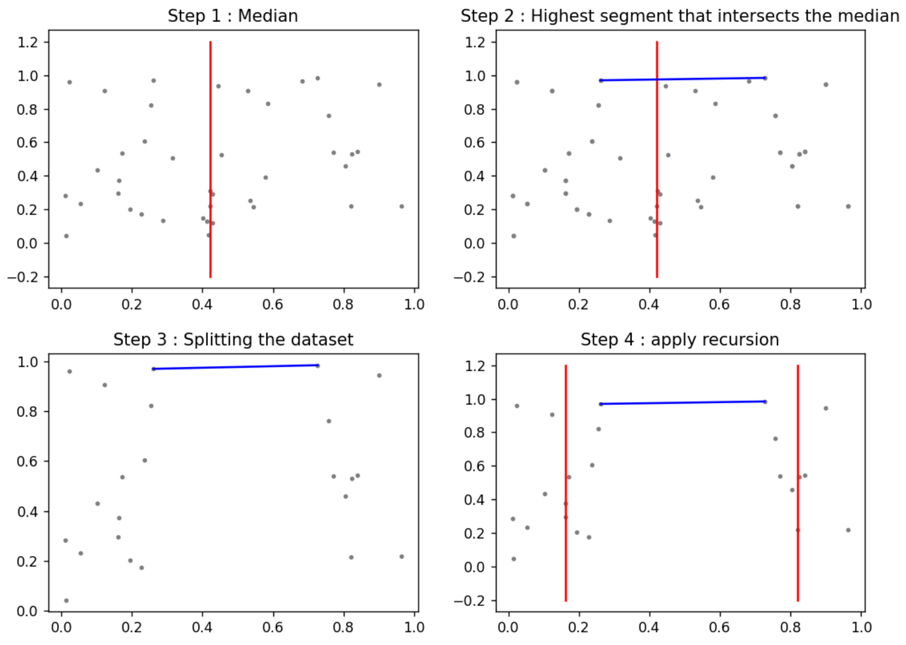
n the size of the dataset and h the size of the hull, at the end of the algorithm, $i + p = 2 * n - h$. This shows that, without sorting, the algorithm's complexity is $O(n)$.

This proves that the algorithm runs in $O(n * log(n))$.

2.2 Marriage Before Conquest

Even if the complexity of the previous algorithm was very good, we could do better without sorting the initial set. The idea here is to find the median of the x-coordinate of the points, split the dataset into two equal parts, compute their convex hulls and combine them. With algorithms being able to find the median in linear time, we can compute a linear-time algorithm. In fact, the complexity of the following algorithm (that computes only the upper hull, just like the previous algorithm) is $O(h * n)$.

Here is a step-by-step example of how the algorithm works:



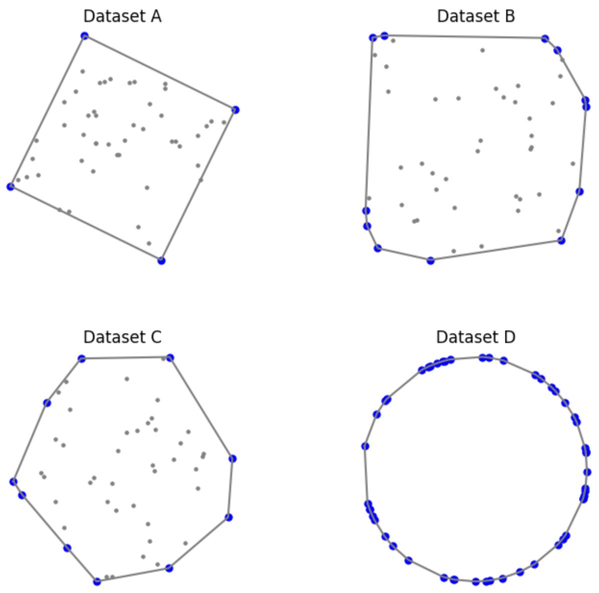
Algorithm 2 MBC algorithm (upper hull)

```
hull = []
if size(dataset) $\leq$ 2 then
    return dataset
end if
$x_m$ ← median(dataset)
segment ← highestSegment($x_m$, dataset)
hull.add(MBC(dataset.pointsLeftOf(segment.left)))
hull.add(MBC(dataset.pointsRightOf(segment.right)))
return hull
```

3 The datasets

There are four types of datasets: `A`, `B`, `C`, and `D`. They will all inherit from a `Dataset` class, which possesses the methods for the search for the convex hull, and will differ by how they are constructed.

Here are all four types of datasets with their convex hulls :



3.1 Dataset A

A Dataset `A` is a set of random points of $[0,1]^2$, to which we add the four corners of the square $(0,0), (0,1), (1,0), (1,1)$. We then shuffle all the points and rotate the dataset around the origin by a random angle.

The idea here is that we know how large the convex hull will be. It will contain exactly four points, the four corners. This will be important when comparing the two algorithms that we have implemented.

3.2 Dataset B

A Dataset `B` is a set of random points of $[0,1]^2$.

3.3 Dataset C

A Dataset `C` is a set of random points of the unitary disk.

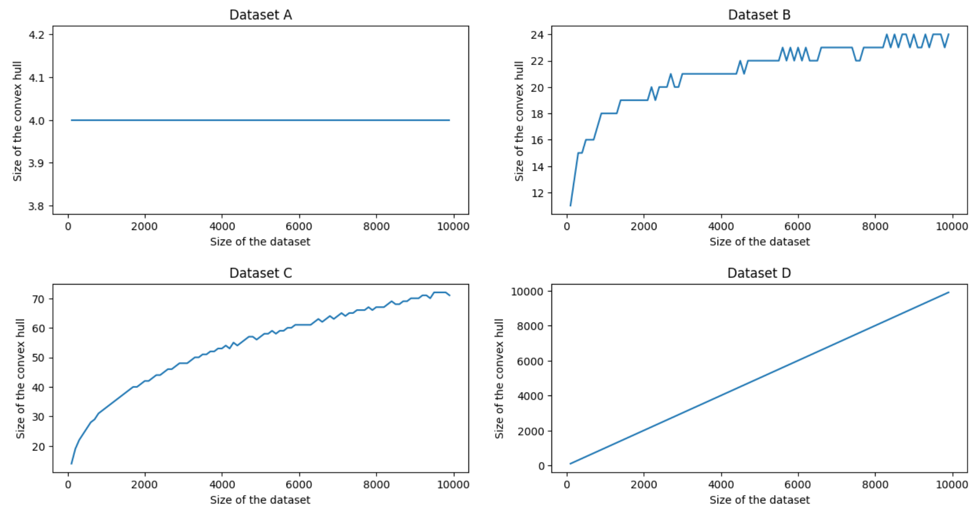
3.4 Dataset D

A Dataset `D` is a set of random points of the unitary circle.

The idea here is that we know how large the convex hull will be. It will contain all the points of the set. This will be important when comparing the two algorithms that we have implemented.

3.5 Properties of the datasets

To compare our algorithms, because `marriageBeforeConquest` is output-sensitive, we need to see how many points the convex hulls of the different types of datasets contain. And this is what we get :



There are actually theoretical results about the expected complexity of random convex hulls. For the dataset `C`, the number of points on the convex hull is $O(n^{\frac{1}{3}})$, and for the dataset `B`, the number of points on the convex hull is $O(log(n))$.

4 Comparison of the algorithms

4.1 Expected results

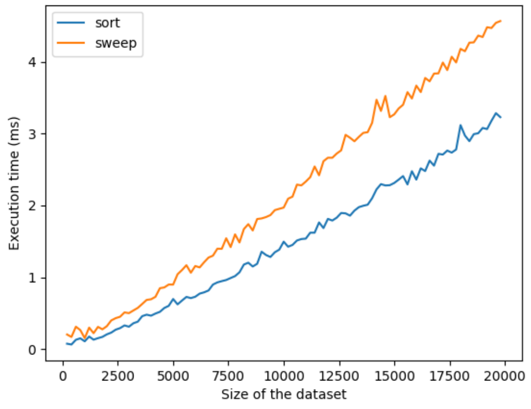
The idea of the `MarriageBeforeConquest` algorithm is that it is more efficient when the convex hull is small. For dataset `A`, the complexity is linear because the size of the convex hull is bounded. For dataset `B`, because the expected size of the convex hull is $O(log(n))$, the complexity of both algorithms should be the same. Still, `MarriageBeforeConquest` being much more complicated than simply sorting a set, the algorithm `Sweep` will be way faster. For `C`, the expected size of the hull is more significant than $O(log(n))$, so there is no point in using `MarriageBeforeConquest`. Finally, for `D`, the size of the hull is n, and the complexity of `MarriageBeforeConquest` is simply catastrophic.

The `MarriageBeforeConquest` algorithm is only efficient for dataset `A`. However, this algorithm is still very interesting because the datasets given to the algorithm may not always be truly random and may often have a bounded number of points on the convex hull. So let us see how good the two algorithms are!

4.2 Comparing Sort and Median

In the beginning, we were using Python for computing convex hulls because, despite being slower, Python allows plotting points very easily. The problem is the following. Let us say that Python is 30 times slower than Java or C. We want to compare `MarriageBeforeConquest`, written in Python, to a sorting algorithm already implemented in Python but actually written in C. This makes the sorting way faster than anything else and computing the median (also written in Python) was faster by sorting the set than using a linear algorithm.

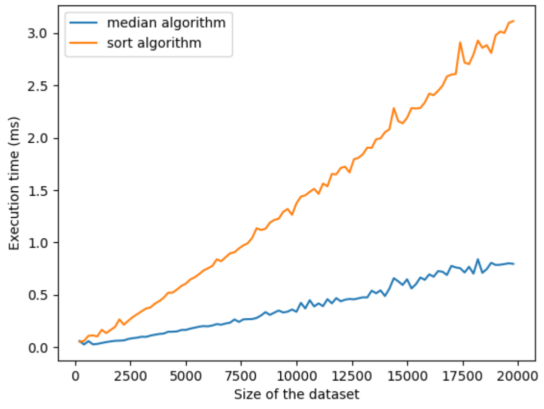
We needed either a slower sorting function or a faster language. So we decided to rewrite our code in Java (even if for plotting figures, we used Python by extracting the data from text files or using our former code). Let us see how significant the loss due to the sorting is by looking at how much time the sorting actually takes compared to the total sweeping of the dataset.



As we can see, almost 70% of the total execution time of the algorithm is taken by sorting (with Python, this ratio was 6%).

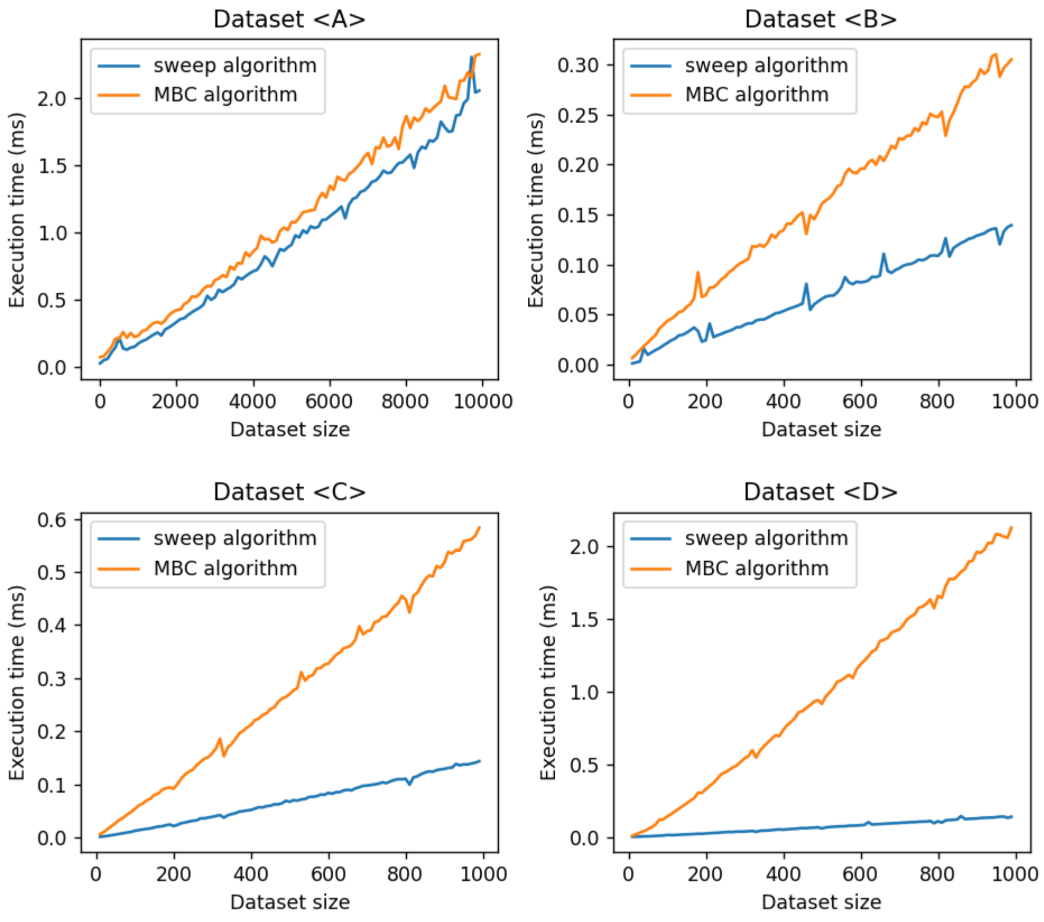
The tricky part will be the following: the `Median` algorithm must be significantly faster than the sorting algorithm for reasonable sets (it is difficult to get over 10 million points). This means that the time constant of the `Median` algorithm should be small enough in order to outperform the $log(n)$ of the sorting algorithm.

First, we implemented the `QuickSelect` algorithm with the `medianOfTheMedians` function. Still, as it turned out, this algorithm was slower than the sorting algorithm because of its bad constant. Using a random pivot appeared to be a better solution, and `Median` became significantly faster than `Sort`.



4.3 Comparing the algorithms

This is what we get:



As expected, there is no hope for an improvement for datasets B, C, and D. Still, we can see that `MarriageBeforeConquest` matches the performances of `Sweep` for datasets A. For bigger datasets, `MarriageBeforeConquest` may be significantly faster.

We tested it on a dataset of 40 000 000 points (beyond, the Java heap space is not large enough). `Sweep` is executed in 34s, `MarriageBeforeConquest` in 41s.

5 Conclusion

To conclude, the `Sweep` algorithm efficiently computes the convex hull of a dataset of points. The MBC algorithm succeeds in matching the performances of `Sweep` on datasets with a bounded convex hull (and may outperform it for bigger datasets than the ones we tested). However, for other cases, `Sweep` is much faster.