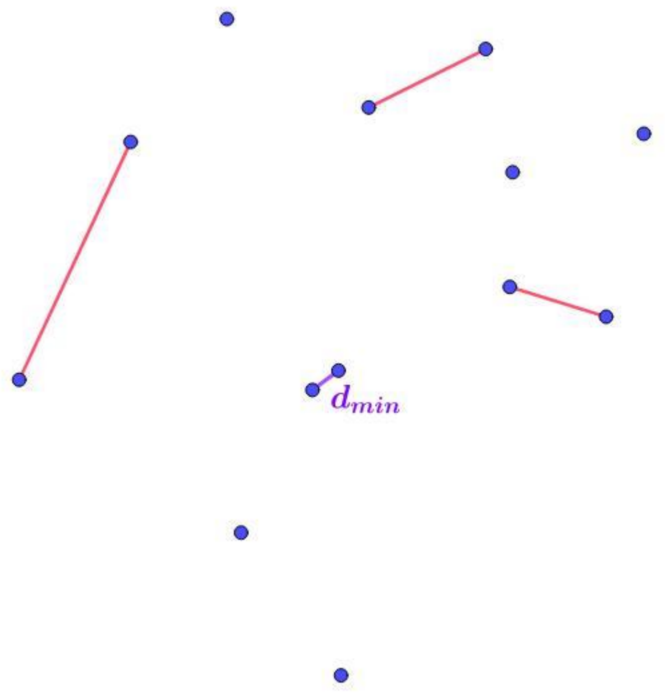
Algorithms and data structures

Project report



Bourseau Julien

April 2022

Table of contents

1 Introduction

1.1 Contextualization

We consider a cloud of S points in a plane, it is about finding the couple of S points such that the distance between its points is minimal. This founding problem in algorithmic geometry has many applications, such as the prevention of collision between two nearby objects (air or sea space).
Since point clouds are large for these applications, it is necessary to find an algorithm of satisfactory complexity.

1.2 Purpose of the report

We will detail the different algorithms built during the project. More precisely, their design features, their performance on different point clouds and their algorithmic complexity will be presented.

2 Study of algorithms

2.1 Naive algorithm

2.1.1 Guiding idea

The first intuition is to study all possible combinations of points.
For each pair of points, calculate the distance and see if it is less than the previously calculated minimum. If so, we update this minimum by this distance.

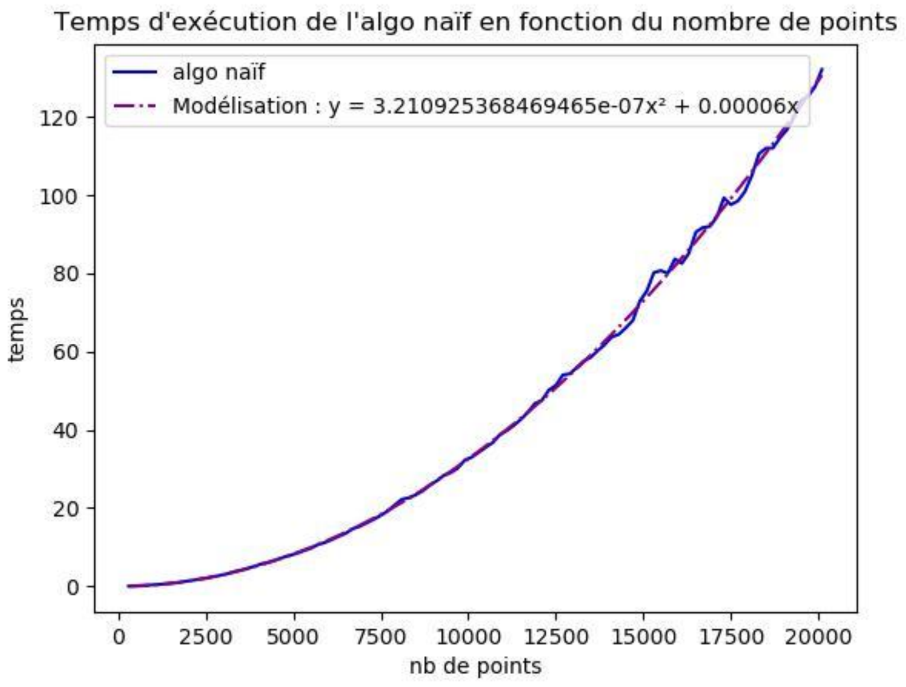
2.1.2 Complexity

It is obvious that this algorithm is not effective. If there are n points, then there will be n2 possible combinations to study. More formally: The initialization of the variables as well as the formatting of the result to be returned is in O(1).
We do n2 loop turns, each turn being in O(1) (the distance.to function is in O(1)).
The complexity of the naive algorithm is therefore well in O(n2). The use of a

such an algorithm seems extremely expensive for a large number of points.

2.1.3 Performance

We trace the performance of the naive algorithm for clouds of increasingly large points, as well as a modelling of degree 2.



The curve confirms a complexity in O(2). Thus, for a cloud of just 20,000 points, it takes the algorithm more than two minutes to run, which is very inefficient. Moreover, there does not seem to be any obvious improvement.
It is therefore necessary to find a more efficient algorithm.

2.2 Divide and conquer

2.2.1 Implementation

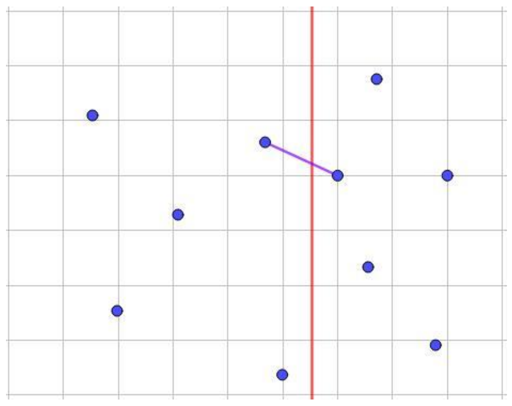
In order to be more efficient, we will break down the initial problem into several sub-problems, solve the latter and combine the sub-problem solutions to find the final solution.

0. Initialization From our table of initial points, we create two tables tabx and taby sorted by abscissas and ascending ordinates.

1. Divide If tabx and taby contain at least 4 points (two pairs of points), we cut the cloud of points into two equal parts, and retry the 4 tables then obtained. This recursively cuts the tabx and taby until the stop condition n <= 3.

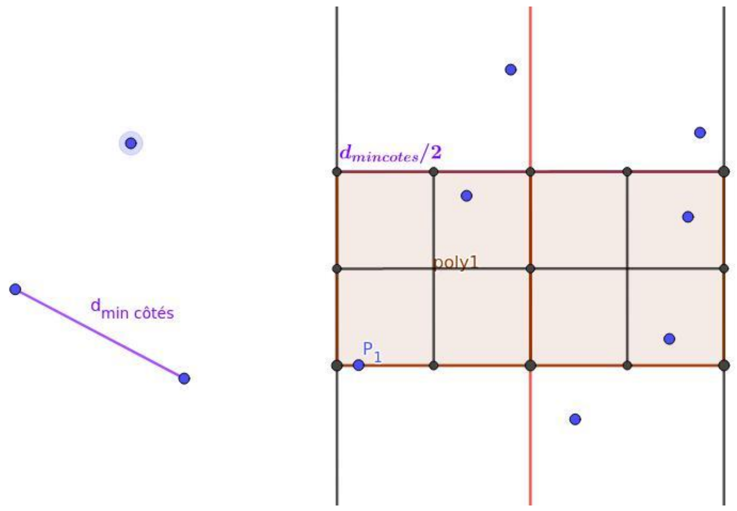
2. Rule When n <= 3, the problem is solved using the naive algorithm, which is inexpensive because the number of points is low.

3. Combine We then compare all the distances found to keep that minimal, that one will note dmincts. We must not forget the pairs of points being in two separate tables but close to the border, as shown in the example below.



As one already has a possible candidate for the minimum distance, dmin odds, it is enough to look at the pairs of inter-table points being less than dmin odds distance. For this, we study the vertical width band

2 dmin odds centered at the border.
We use the function in band that will return the points in this band, then we calculate the smallest distance between these points thanks to the naive algo-rithmus, which we will note dminbande.
The final solution is then min(dmin odds, dmin band.
However, the minimum search step in the band can be improved.
First of all, you should not consider each pair twice.
In addition, we know that the minimum distance between two points on the same side of the border is dmin odds, so for any point P1, if there is a point P2 at a distance less than dmin odds P1, then it is in the rectangle passing through P1 opposite:



The rectangle is divided into 8 square sides dmin odds/2. One cannot have more than one point per square, because otherwise the distance between these neighbors would be less than dmin odds, which is absurd. So we can just look at the points that follow P1.

2.2.2 Complexity calculation

The initialization step is a sort by the sort function, which has an average complexity of O(nlog(n)).
For the dividing step, we go through half of tabx to make the two new tables, so in O(n/2), and we divide taby in two in O(n/2) also. So this step will be done in O(n).

The stage to reign is in O(1) because there are only 3 points.
For the combination step there are no more than 7 distance calculations for each point, so no more than 7n distance calculations. This step is done in O(n) operations.

We divide a problem of size n into 2 sub-problems of size n/2, and the cost of managing outside recursive calls is, as we have just seen, in O(n). We therefore have the formula of recurrence, noting C the complexity:

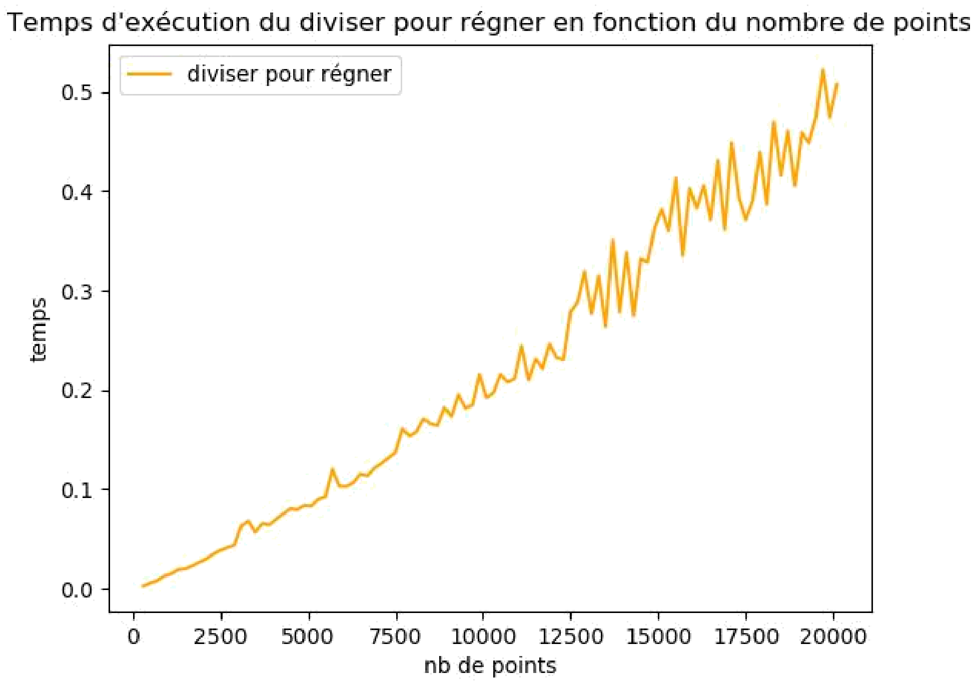
$$C(n) = 2 \qquad C(n/2) + O(n)$$

So according to the master theorem, the complexity of the algorithm is O(nlog(n)). Less formally, this means that there are O(n) operations by recursive calls, and the order of O(log(n)) recursive calls.

2.2.3 Performance

We suspect that this algorithm is much more effective than the naive algorithm given its complexity:

The curve grows slightly faster than the linear function, so we have the desired behavior. The program runs in less than half a second, while the so-called naive algorithm took 20 seconds.



3 Conclusion

Two algorithms have been studied in this report, both responding to the given problem, and although the first seems useless in view of its slowness, it nevertheless allows to verify the results of the second.
The naive algorithm cannot pass all test sets because of its complexity. The divide and conquer algorithm passes tests 0 and 1, it could, in terms of performance, pass test 2 without timeout, but it seems that an exception is not supported, making the result false.