CS 310
Assignment 321

Cameron Moberg

March 23, 2017

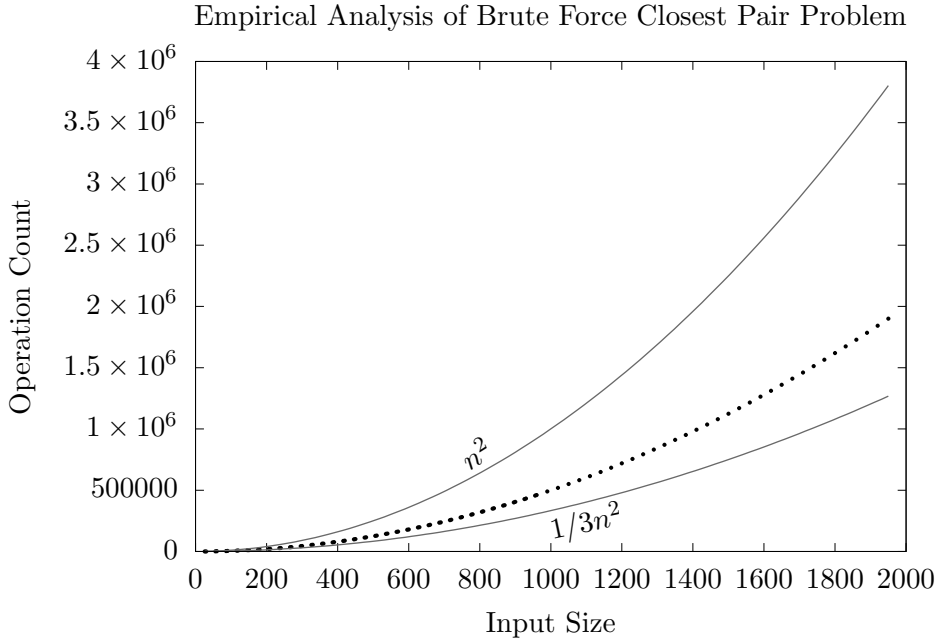
There are several ways to solve the closest-pair problem; the two done in this analysis are the brute force method and the divide and conquer method. The purpose of this study is to perform an empirical analysis of the two methods.

For both methods, the value for n is taken to be the number of points in the plane.

1 Brute Force

To analyze the brute force algorithm, we chose calculating the distance on line 67 as the basic operation because this happens most frequently, is the most expensive, and is the philosophical heart of this algorithm.

An empirical analysis of running the algorithm for multiple values of n produces the results shown below. Standard function $f(n) = n^2$ with constant multipliers, has been added to illustrate the analysis.



An examination of the code itself explains the empirical results when we observe that there is a doubly-nested for-loop that iterates over the input size. Therefore, since the brute force algorithm is deterministic we conclude that the algorithm is described by

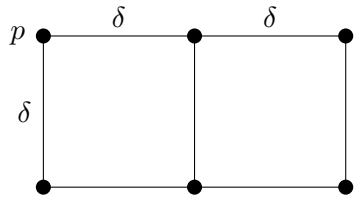
$$T(n) \in \Theta(n^2)$$

This analysis agrees with our author, however, in this case the elementary operation is calculating the squared distance, rather than the square root, which is only calculated once instead of every iteration. This only speeds up the algorithm by a constant factor, not impacting the efficiency class [Lev12].

2 Divide and Conquer

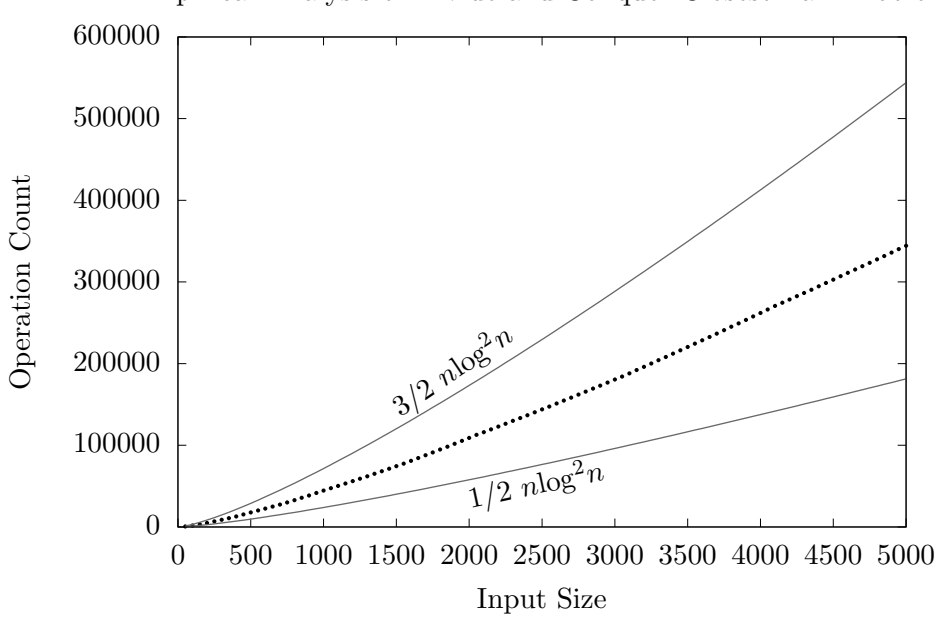
To analyze the divide and conquer algorithm, we chose the sort method on lines 122–124 as the basic operation since it is by far the most expensive operation compared to all other possible choices. After research into the `std::sort` function that C++11 uses it has been determined that the sorting algorithm used is of the runtime complexity $O(n \log n)$[11]. Since we cannot (easily) edit the `std::sort` function, we must assume that the basic operations after each sort is increased by an $n \log n$ amount of operations, where n is the number of elements sorted.

There is also the chance that all the points are included in the strip between the left and right sets of points. This makes it seem like we must use the brute force method on them which would make the local work n^2. However, due to the geometry of the closest-pair problem, we only need to check the 6 closest pairs. This is because of the constraint that for any point, p, we are only interested in the other points that are at most δ distance away, where δ is the calculated distance from earlier calculations. Shown below is a geometric example of this concept.



Notice how at most there are 5 other points in the 2δ by δ rectangle. Showing we only need to check at most the next 6 pairs, meaning at most $6n$ comparisons are needed to check all pairs in the strip [Lev12].

An empirical analysis of running the algorithm for multiple values of n produces the results shown below. The standard function $f(n) = n \log^2 n$ with constant multipliers, has been added to illustrate the analysis.



An examination of the code itself explains the empirical results when we observe that this algorithm is recursive, lending us the ability to use the Master Theorem, which is of the form

$$T(n) = aT(\frac{n}{b}) + f(n) \text{ where } a \geq 1, b > 1, \text{ and } d \geq 0$$

The divide and conquer algorithm has two recursive calls where each recursive call operates on half of the input. As previously stated, the local work ($f(n)$) for this algorithm is `std::sort`, which is $(n \log n)$. Additionally, there is no official documentation stating the best case runtime of C++11's sorting algorithm, so our analysis must use Big-O instead of Big-Θ. Since $f(n) \in O(n \log n)$ we cannot use the Master Theorem traditionally, and must use a special case where

if it is true for some constant, $k \geq 0$, such that:

$$f(n) = \Theta(n^c \log^k n) \text{ where } c = \log_b a$$

then:

$$T(n) = \Theta(n^c \log^{k+1} n)$$

Thus, after applying this case of the Master Theorem with $c = \log_2 2 = 1$ and $k = 1$ (since $f(n) = n \log n = n^{\log_2 2} \log^1 n$) we arrive at $T(n) = O(n \log^2 n)$ [GT01]. Seeing that the divide and conquer algorithm always runs to completion and the best-case scenario is not offered we conclude, from the Master Theorem, that the algorithm is described by

$$T(n) \in O(n \log^2 n)$$

We can infer, however, the possibilities of what the Big-Ω for `std::sort` could be. The efficiency of this algorithm is determined by the local work: `std::sort` ($f(n)$ from now on). By observation, we see that $f(n) \in \Omega(n \log n)$, or $f(n) \in \Omega(n)$. Notice that $f(n) \notin \Omega(1)$ because sorting algorithms must process the entire input to ensure it is sorted. Also notice that $f(n)$ cannot be of a worse complexity than $\Omega(n \log n)$ because it has already been determined that $f(n) \in O(n \log n)$. So from the two possible scenarios shown above there are two possible Big-Ω runtimes for $T(n)$. Consider the following cases.

Case 1: $f(n) \in \Omega(n)$
In this case we can use the Master's Theorem traditionally with $a = 2, b = 2, d = 1$ showing the entire divide and conquer algorithm would be of the form

$$T(n) \in O(n \log^2 n)$$

$$T(n) \in \Omega(n \log n)$$

Case 2: $f(n) \in \Omega(n \log n)$
In this case both the Big-O and Big-Ω will be the same efficiency, tightening the bound to Big-Θ. After using the special case of the Master Theorem, it can be shown that algorithm would be of the form

$$T(n) \in \Theta(n \log^2 n)$$

Unfortunately, neither scenario agrees with our author's analysis who determined the algorithm was

$$T(n) \in \Theta(n \log n)$$

This is because the author states that, "The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions"[Lev12]. However, the author does not account for the fact that the points must be sorted by Y coordinate every recursive call. As stated above sorting is $O(n \log n)$ which will dominate the linear work. Since the local work has changed, the author's calculated $f(n) = n$ should now be $f(n) = n \log n$, which changes the runtime efficiency of this algorithm.

References

[GT01] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. 1st ed. Wiley, 2001, pp. 268–270. ISBN: 9780471383659.

[11] "Working Draft, Standard for Programming Language C++". In: N3225 (2011). `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2011/n3242.pdf`, p. 911.

[Lev12] Anany Levitin. *Introduction to The Design and Analysis of Algorithms*. 3rd ed. Pearson, 2012. ISBN: 9780132316811.