Review: Verification of Closest Pair of Points Algorithms

Aditi Goyal
aditi.goyal@epfl.ch

Suhas Shankar
suhas.shankar@epfl.ch

Yatharth Goswami
yatharth.goswami@epfl.ch

December 31, 2022

1 Introduction

The paper [4] verifies the divide and conquer algorithm for finding the closest pair of points in a 2D plane using Isabelle/HOL [1] It verifies the time complexity bound of $\mathcal{O}(n \log n)$. This is one of the most fundamental problems in computational geometry. It appears as a step in many algorithms and has applications in real-world problems such as air-traffic control where we want to monitor the distance between any two planes to avoid collisions. The problem being applied in safety-critical applications demands verification of its implementation. Our reference paper was the first to verify the implementation of the divide and conquer algorithm 2.3, the basis of which comes from the book [3] . The paper also provides benchmarks against the best-proposed algorithm for the problem [2] by calculating the ratio of time taken by the best implementation and the Isabelle implementation, purely functional implementation and imperative implementation.

2 Preliminaries

2.1 Notations

For providing the function signatures we will use Scala 3 syntax.

- A point p is represented by a pair of integers. $p.\_1$ is the $x - coordinate$
- $d(p_1, p_2)$ calculates the distance between points p_1 and p_2
- $sorted\_y(ls)$ returns true if the list of points ls is sorted according to non-decreasing $y - coordinates$
- $sorted\_x(ls)$ returns true if the list of points ls is sorted according to non-decreasing $x - coordinates$
- $l.content$ represents the set of items in the list l

2.2 Delta sparsity

A set of points P is δ-sparse if the distance between all pairs of points in the set is $\geq \delta$. Formally,

$$sparse\,(\delta,\, P) = (\,\forall p_0 \in P\,.\,\forall p_1 \in P\,.\,p_0 \neq p_1 \rightarrow \delta \leq d(\,p_0,\, p_1))$$

Hence p_0 and p_1 are the closest pair of points in P when

$$p_0 \in P \wedge p_1 \in P \wedge p_0 \neq p_1 \wedge sparse\,(d(p_0, p_1),\ P)$$

2.3 Algorithm overview

The algorithm is based on divide and conquer paradigm. It starts by splitting the set of points into 2 sets by splitting them on the median $x - coordinate$ (say l). Then, it recursively solves the problem for the 2 halves. Let δ_l (δ_r) denote distance between closest pair of points in left(right) half. Now, we know that the closest pair of points will either lie on the left side or the right side or there will be 1 point from the left and other from the right side. We got the closest pair from the left and right half recursively, now what remains is what is called the *combine* step. Let δ denote the minimum of δ_l and δ_r. The algorithm then works by finding the closest pair of points in a strip of width 2δ centered around l to find a better candidate for the closest points (see figure below).

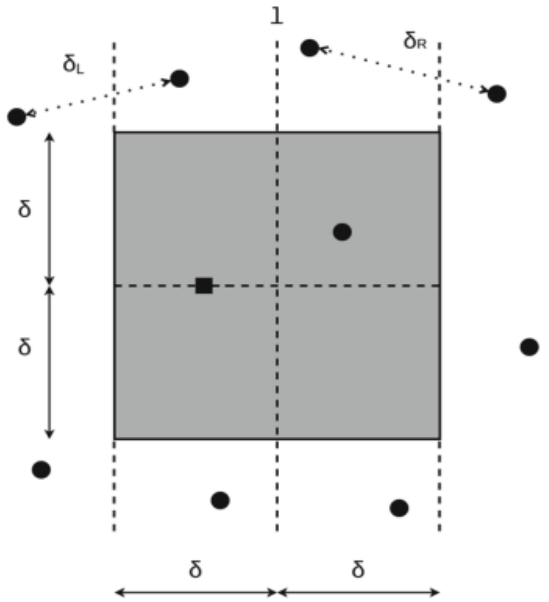

Figure 1: Combine step in the algorithm. Taken from [4]

2.4 Constraints and other details

For the purpose of verification, the authors have restricted themselves to solving it in the domain of integer point coordinates. Their implementation does not require the points to have distinct $x - coordinates$.

3 Implementation and Functional Correctness

We will describe the implementation of the divide and conquer algorithm that they have verified along with the specifications that hold in terms of lemmas and theorems.

3.1 The combine step

Let's talk about the most non-trivial aspect of the algorithm first. The combine step assumes that we have been given a pair of points with distance between them (say, δ) and a list of points (say, ps) which are sorted according to their $y - coordinates$. The list of points (ps) given are just the ones that are contained in the 2δ wide strip as described

in the algorithm overview. The combine step should return a pair of points having a distance $\delta' \leq \delta$ such that the list ps is δ' sparse. For this they start with the function $find\_closest\_pair$ with the following signature

$$find\_closest\_pair : (point, point, list[point]) \Rightarrow (point, point)$$

This function traverses the list from start to end and for each point, we only calculate it's closest neighbour in the rest of the list. This way, we essentially check all eligible candidates for closest pair. This function uses internally another function $find\_closest$ with the following signature

$$find\_closest : (point, real, list[point]) \Rightarrow point$$

This function takes in a point (x, y) and a list of points, a real number (say δ) and looks at points in list that have a $y - coordinate$ between y and $y + \delta$ and returns the point closest to (x, y) amongst these. If there are no points with y-coordinate in the given range, it returns the point in the list having the least y-coordinate greater than or equal to $y + \delta$. Though it looks like a function with linear time complexity, it is not! If we are given a sorted list of points, then only a constant number of points can have $y - coordinate$ between y and $y + \delta$. 3.3. Hence, this function runs in constant time complexity.

The *combine* function takes in as arguments, a pair of closest points in the left half, a pair of closest points in the right half and the original list sorted by $y - coordinates$. Then, it calculates δ as minimum of distance between closest pair of points in left half and distance between closest pair of points in right half. Then it calls *find_closest_pair* on the points lying in the strip centred at median $x - coordinate$ and having a width of 2δ.

$$combine : ((point, point), (point, point), int, point\_list) \Rightarrow (point, point)$$

The following lemmas verify the correctness of the combine step.

Lemma 3.1 *sorted_y(ps)* $\wedge\ (p_0, p_1) = find\_closest\_pair(c_0, c_1, ps)$
$\implies sparse(d(p_0, p_1), ps)$

Lemma 3.1 says that if ps is sorted according to increasing $y - coordinates$, the distance between any two points in ps is greater than or equal to the distance between the points returned by $find\_closest\_pair$ for any two points c_0, c_1

Lemma 3.2 $ps.contains(p_0) \wedge ps.contains(p_1) \wedge p_0 \neq p_1 \wedge d(p_0, p_1) < \delta \wedge$
$(\forall p \in P_L \,.\, p.\_1 \leq l) \wedge sparse(\delta, P_L) \wedge$
$(\forall p \in P_R \,.\, p.\_1 > l) \wedge sparse(\delta, P_R) \wedge$
$ps.content = (P_L + + P_R).content \wedge ps' = ps.filter((p) => d(p, (l, p.\_2)) < \delta)$
$\implies ps'.contains(p_0) \wedge ps'.contains(p_1)$

Lemma 3.2 says that given a partition P_L and P_R of the set of points in ps based on the x-coordinate being greater than or lesser than l, any two points p_0, p_1 that have $d(p_0, p_1) < \delta$ lie in the δ-strip described before.

Lemma 3.3 $sorted\_y(ps) \wedge ps.content = (P_L + + P_R).content \wedge$
$(\forall p \,.\, P_L.contains(p) \implies p.\_1 \leq l) \wedge sparse(d(p_{0L}, p_{1L}), P_L) \wedge$
$(\forall p \,.\, P_R.contains(p) \implies p.\_1 > l) \wedge sparse(d(p_{0R}, p_{1R}), P_R) \wedge$
$(p_0, p_1) = combine((p_{0L}, p_{1L}), (p_{0R}, p_{1R}), l, ps)$
$\implies sparse((d(p_0, p_1), ps))$

Lemma 3.3 states that given a partition P_L and P_R of the set of points in ps based on the x-coordinate being greater than or lesser than l, and points p_{0L}, p_{1L} such that p_{0L}, p_{1L} are the closest points in P_L and p_{0R}, p_{1R} are the closest points in P_R, the points p_0, p_1 returned by combine is the closest set of points in ps

3.2 Divide and conquer algorithm

The divide step of the algorithm divides the points based on their $x - coordinates$ into two equal halves and solves the problem recursively on these halves. Then, it combines using the step discussed above. We start by describing a function *closest_pair_rec* with the signature

closest_pair_rec: list[point] ⇒ (list[point], point, point)

It takes a list of points sorted by $x - coordinates$ and returns the input list sorted according to $y - coordinates$ and the closest pair of points from input list. It first calls itself recursively by splitting the original list into 2 halves based on the $x - coordiantes$. From the result of these calls, it merges the 2 halves (sorted by $y - coordinates$). It then calls the combine step with closest pairs from the left and right halves and the list of points sorted by $y - coordinates$. This function is called by another function - *closest_pair* which has the signature

closest_pair: list[point] ⇒(point, point)

This function takes as input a list of points and returns the closest pair of points from the list. The following theorem verifies states the correctness of *closest_pair_rec* formally.

Theorem 3.4 $1 < xs.length \wedge sorted\_x(xs) \wedge (ys, p_0, p_1) = closest\_pair\_rec(xs) \implies sparse(d(p_0, p_1), xs)$

Theorem 3.4 states that if we give a list sorted by $x - coordinates$ to the function *closest_pair_rec*, we get a pair of points with distance δ between them such that the list is δ sparse.
The following corollary and theorems prove that (p_0, p_1) are indeed the closest points.

Corollary 3.4.1 $1 < ps.length \wedge (p_0, p_1) = closest\_pair(ps) \implies sparse(d(p_0, p_1), ps)$

Theorem 3.5 $1 < ps.length \wedge (p_0, p_1) = closest\_pair(ps) \implies ps.contains(p_0) \wedge ps.contains(p_1)$

Theorem 3.6 $1 < ps.length \wedge ps.distinct \wedge (p_0, p_1) = closest\_pair(ps) \implies p_0 \neq p_1$

3.3 Time Complexity Verification

The paper proves the time complexity bound of $\mathcal{O}(n \log n)$ by claiming that the time taken for n points represented as $T(n)$ follows the recurrence relation:

$$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \mathcal{O}(n)$$

The first 2 terms on the right side come from recursive calls to the left and right halves. The last term comes from the combine step. Splitting into halves by $x - coordinates$, merging by y coordinates are trivially linear time operations. The interesting part of the proof is where running time of *find_closest_pair* is shown to be linear in size of input.

Since the function iterates over all points and finds closest point for each of the point, it essentially boils down to proving that the function *find_closest* has constant run time complexity. For a point $p = (x, y)$, the function *find_closest* takes a look at the points (which were already filtered to consist only of the 2δ strip around the center) having $y - coordinates$ from y to $y + \delta$. Also, the way we chose δ (as in combine step) ensures that the left half is δ sparse, as well as right half is δ sparse. So, during the call to $find\_closest$, we are looking at a $2\delta \times \delta$ rectangle, which when divided into middle gives 2 squares. Also, distance between any 2 points in the left square of size $\delta \times \delta$ is atleast δ so we can have atmost 4 points in the left square (basic human intuition). And similarly, for the right square. Thus, bounding the number of points in the rectangle by a constant. The authors translated this basic intution into rigorous geometric proofs for the Isabelle to understand. They used this analysis to verify another implementation of $find\_closest$ which just checks the next 7 points. They call this algorithm Basic-7 and say that it is inefficient since it checks 7 points in every case, which is not necessary each time.

3.4 Benchmarks

The authors have exported the Isabelle verified implementation of the above explained algorithm (say, Basic-δ) to OCaml. They also wrote an equivalent functional implementation in OCaml and one imperative implementation as well. This was done to compare the performance of the machine-generated code against similar implementations. The ratio of time taken of these implementations with the time taken by the best algorithm yet [2] are compared. It was shown that remarkably the exported code was just 2.28 times slower than the best algorithm. Also, Basic-7 was written imperatively and was found out to be 2.26 times slower than the imperative implementation of Basic-δ demonstrating the inefficiency.

4 Conclusion

The paper was the first to present formally verified implementations of the algorithm to find the closest pair of points in a plane without assuming the $x - coordinates$ to be distinct. There weren't many weak points in the paper. The paper explains and formulates all the theorems required to prove the correctness of the algorithm succinctly. This provides us with a road map to develop our verified implementation in Stainless. However, it failed to provide any proofs or intuition on how to prove the corresponding theorems. Additionally, we plan on providing benchmarks for verified implementations in Stainless and Scala, leaving aside the formal proofs for time complexity.

An extension of the paper could be to verify the implementation of the algorithm that solves the closest pair of points problem for greater than 2 dimensions. This would be a challenging but interesting task as its formalization and verification are not available in literature.

References

[1] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3540433767.

[2] Minghui Jiang and Joel Gillespie. "Engineering the Divide-and-Conquer Closest Pair Algorithm". In: *Journal of Computer Science and Technology* 22.4, 532 (2007), p. 532. URL: https://jcst.ict.ac.cn/EN/abstract/article_1388.shtml.

[3] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[4] Martin Rau and Tobias Nipkow. "Verification of Closest Pair of Points Algorithms". In: *Automated Reasoning*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. https://link.springer.com/content/pdf/10.1007/978-3-030-51054-1_20.pdf. Cham: Springer International Publishing, 2020, pp. 341–357. ISBN: 978-3-030-51054-1.