

Verification of Closest Pair of Points Algorithms

Martin Rau and Tobias Nipkow^[0000–0003–0730–515X]

Fakultät für Informatik, Technische Universität München

Abstract. We verify two related divide-and-conquer algorithms solving one of the fundamental problems in Computational Geometry, the *Closest Pair of Points* problem. Using the interactive theorem prover Isabelle/HOL, we prove functional correctness and the optimal running time of $\mathcal{O}(n \log n)$ of the algorithms. We generate executable code which is empirically competitive with handwritten reference implementations.

1 Introduction

The *Closest Pair of Points* or *Closest Pair* problem is one of the fundamental problems in Computational Geometry: Given a set P of $n \geq 2$ points in \mathbb{R}^d , find the closest pair of P , i.e. two points $p_0 \in P$ and $p_1 \in P$ ($p_0 \neq p_1$) such that the distance between p_0 and p_1 is less than or equal to the distance of any distinct pair of points of P .

Shamos and Hoey [25] are one of the first to mention this problem and introduce an algorithm based on *Voronoi* diagrams for the planar case, improving the running time of the best known algorithms at the time from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. They also prove that this running time is optimal for a deterministic computation model. One year later, in 1976, Bentley and Shamos [2] publish a, also optimal, divide-and-conquer algorithm to solve the Closest Pair problem that can be non-trivially extended to work in arbitrary dimensions. Since then the problem has been the focus of extensive research and a multitude of optimal algorithms have been published. Smid [24] provides a comprehensive overview over the available algorithms, including randomized approaches which improve the running time even further to $\mathcal{O}(n)$.

The main contribution of this paper is the first verification of two related functional implementations of the divide-and-conquer algorithm solving the Closest Pair problem for the two-dimensional Euclidean plane with the optimal running time of $\mathcal{O}(n \log n)$. We use the interactive theorem prover Isabelle/HOL [18,17] to prove functional correctness as well as the running time of the algorithms. In contrast to many publications and implementations we do not assume all points of P to have unique x -coordinates which causes some tricky complications. Empirical testing shows that our verified algorithms are competitive with handwritten reference implementations. Our formalizations are available online [23] in the Archive of Formal Proofs.

This paper is structured as follows: Section 2 familiarizes the reader with the algorithm by presenting a high-level description that covers both implementations. Section 3 presents the first implementation and its functional correctness

proof. Section 4 proves the running time of $\mathcal{O}(n \log n)$ of the implementation of the previous section. Section 5 describes our second implementation and illustrates how the proofs of Sections 3 and 4 need to be adjusted. We also give an overview over further implementation approaches. Section 6 describes final adjustments to obtain executable versions of the algorithms in target languages such as OCaml and SML and evaluates them against handwritten imperative and functional implementations. Section 7 concludes.

1.1 Related Verification Work

Computational geometry is a vast area but only a few algorithms and theorems seem to have been verified formally. We are aware of a number of verifications of convex hull algorithms [20,14,4] (and a similar algorithm for the intersection of zonotopes [12]) and algorithms for triangulation [7,3]. Geometric models based on maps and hypermaps [22,6] are frequently used.

Work on theorem proving in geometry (see [15] for an overview) is also related but considers fixed geometric constructions rather than algorithms.

1.2 Isabelle/HOL and Notation

The notation $t :: \tau$ means that term t has type τ . Basic types include *bool*, *nat*, *int* and *real*; type variables are written *'a*, *'b* etc; the function space arrow is \Rightarrow . Functions *fst* and *snd* return the first and second component of a pair.

We suppress numeric conversion functions, e.g. $real :: nat \Rightarrow real$, except where that would result in ambiguities for the reader.

Most type constructors are written postfix, e.g. *'a set* and *'a list*. Sets follow standard mathematical notation. Lists are constructed from the empty list `[]` via the infix cons-operator (`#`). Functions *hd* and *tl* return head and tail, function *set* converts a list into a set.

2 Closest Pair Algorithm

In this section we provide a high-level overview of the *Closest Pair* algorithm and give the reader a first intuition without delving into implementation details, functional correctness or running time proofs.

Let P denote a set of $n \geq 2$ points. If $n \leq 3$ we solve the problem naively using the brute force approach of examining all $\binom{n}{2}$ possible closest pair combinations. Otherwise we apply the divide-and-conquer tactic.

We divide P into two sets P_L and P_R along a vertical line l such that the sizes of P_L and P_R differ by at most 1 and the x -coordinate of all points $p_L \in P_L$ ($p_R \in P_R$) is $\leq l$ ($\geq l$).

We then conquer the left and right subproblems by applying the algorithm recursively, obtaining (p_{L0}, p_{L1}) and (p_{R0}, p_{R1}) , the respective closest pairs of P_L and P_R . Let δ_L and δ_R denote the distance of the left and right closest pairs and let $\delta = \min \delta_L \delta_R$. At this point the closest pair of P is either (p_{L0}, p_{L1}) ,

(p_{R0}, p_{R1}) or a pair of points $p_0 \in P_L$ and $p_1 \in P_R$ with a distance strictly less than δ . In the first two cases we have already found our closest pair.

Now we assume the third case and have reached the most interesting part of divide-and-conquer algorithms, the *combine* step. It is not hard to see that both points of the closest pair must be within a 2δ wide vertical strip centered around l . Let ps be a list of all points in P that are contained within this 2δ wide strip, sorted in ascending order by y -coordinate. We can find our closest pair by iterating over ps and computing for each point its closest neighbor. But in the worst case ps contains all points of P , and we might think our only option is to again check all $\binom{n}{2}$ point combinations. This is not the case. Let p denote an arbitrary point of ps , depicted as the square point in Figure 1. If p is one of

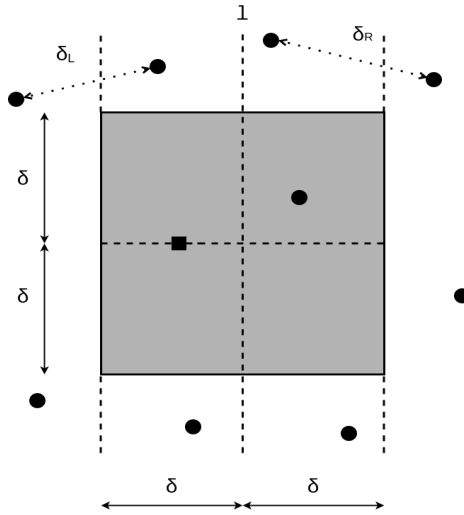


Fig. 1. The *combine* step

the points of the closest pair, then the distance to its closest neighbor is strictly less than δ and we only have to check all points $q \in ps$ that are contained within the 2δ wide horizontal strip centered around the y -coordinate of p .

In Section 4 we prove that, for each $p \in ps$, it suffices to check only a constant number of closest point candidates. This fact allows for an implementation of the *combine* step that runs in linear time and ultimately lets us achieve the familiar recurrence of $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \mathcal{O}(n)$, which results in the running time of $\mathcal{O}(n \log n)$.

We glossed over some implementation details to achieve this time complexity. In Section 3 we refine the algorithm and in Section 4 we prove the $\mathcal{O}(n \log n)$ running time.

3 Implementation and Functional Correctness Proof

We present the implementation of the divide-and-conquer algorithm and the corresponding correctness proofs using a bottom-up approach, starting with the *combine* step. The basis for both implementation and proof is the version presented by Cormen *et al.* [5]. But first we need to define the closest pair problem formally.

A point in the two-dimensional Euclidean plane is represented as a pair of (unbounded) integers¹. The library HOL-Analysis provides a generic distance function *dist* applicable to our point definition. The definition of this specific *dist* instance corresponds to the familiar Euclidean distance measure.

The closest pair problem can be stated formally as follows: A set of points P is δ -sparse iff δ is a lower bound for the distance of all distinct pairs of points of P .

$$\text{sparse } \delta \ P = (\forall p_0 \in P. \forall p_1 \in P. p_0 \neq p_1 \rightarrow \delta \leq \text{dist } p_0 \ p_1)$$

We can now state easily when two points p_0 and p_1 are a *closest pair* of P : $p_0 \in P, p_1 \in P, p_0 \neq p_1$ and (most importantly) $\text{sparse } (\text{dist } p_0 \ p_1) \ P$.

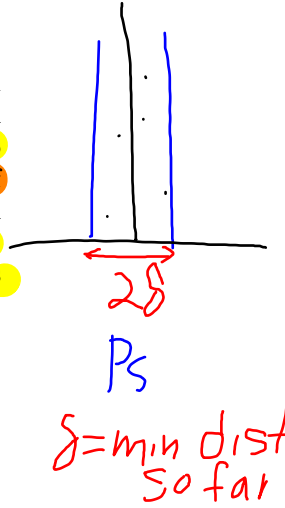
In the following we focus on outlining the proof of the sparsity property of our implementation, without going into the details. The additional set membership and distinctness properties of a closest pair can be proved relatively straightforwardly by adhering to a similar proof structure.

3.1 The Combine Step

The essence of the *combine* step deals with the following scenario: We are given an initial pair of points with a distance of δ and a list ps of points, sorted in ascending order by y -coordinate, that are contained in the 2δ wide vertical strip centered around l (see Figure 1). Our task is to efficiently compute a pair of points with a distance $\delta' \leq \delta$ such that ps is δ' -sparse. The recursive function *find_closest_pair* achieves this by iterating over ps , computing for each point its closest neighbor by calling the recursive function *find_closest* that considers only the points within the shaded square of Figure 1, and updating the current pair of closest points if the newly found pair is closer together. We omit the implementation of the trivial base cases.

```
find_closest_pair :: point × point ⇒ point list ⇒ point × point
find_closest_pair (c0, c1) (p0 # ps) =
  (let p1 = find_closest p0 (dist c0 c1) ps
   in if dist c0 c1 ≤ dist p0 p1 then find_closest_pair (c0, c1) ps
      else find_closest_pair (p0, p1) ps)
```

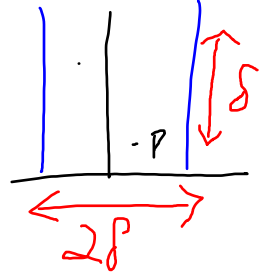
¹ We choose integers over reals because we cannot implement mathematical reals. See Section 6. Alternatively we could have chosen rationals.



```

find_closest :: point ⇒ real ⇒ point list ⇒ point
find_closest p δ (p0 # ps) =
  (if δ ≤ snd p0 - snd p then p0
   else let p1 = find_closest p (min δ (dist p p0)) ps
    in if dist p p0 ≤ dist p p1 then p0 else p1)

```



*

There are several noteworthy aspects of this implementation. The recursive search for the closest neighbor of a given point p of `find_closest` starts at the first point spatially above p , continues upwards and is stopped early at the first point whose vertical distance to p is equal to or exceeds the given δ . Thus the function considers, in contrast to Figure 1, only the upper half of the shaded square during this search. This is sufficient for the computation of a closest pair because for each possible point q preceding p in ps we already considered the pair (q, p) , if needed, and do not have to re-check (p, q) due to the commutative property of our closest pair definition. Note also that δ is updated, if possible, during the computation and consequently the search space for each point is limited to a $2\delta \times \delta'$ rectangle with $\delta' \leq \delta$. Lastly we intentionally do not minimize the number of distance computations. In Section 6 we make this optimization for the final executable code.

The following lemma captures the desired sparsity property of our implementation of the `combine` step so far. It is proved by induction on the computation. *

Lemma 1. $\text{sorted_snd } ps \wedge (p_0, p_1) = \text{find_closest_pair } (c_0, c_1) \text{ } ps$
 $\implies \text{sparse } (\text{dist } p_0 \text{ } p_1) (\text{set } ps)$

where `sorted_snd ps` means that ps is sorted in ascending order by y -coordinate.

We wrap up the `combine` step by limiting our search for the closest pair to only the points contained within the 2δ wide vertical strip and choosing as argument for the initial pair of points of `find_closest_pair` the closest pair of the two recursive invocations of our divide-and-conquer algorithm with the smaller distance δ .

$$\delta = \min(P_L, P_R)$$

```

combine :: point × point ⇒ point × point ⇒ int ⇒ point list ⇒ point × point
combine (p0L, p1L) (p0R, p1R) l ps =
  (let (c0, c1) =
      if dist p0L p1L < dist p0R p1R then (p0L, p1L) else (p0R, p1R)
    in find_closest_pair (c0, c1)
      (filter (λp. dist p (l, snd p) < dist c0 c1) ps))

```

Lemma 2 shows that if there exists a pair (p_0, p_1) of distinct points with a distance $< \delta$, then both its points are contained in the mentioned vertical strip, otherwise we have already found our closest pair (c_0, c_1) , and the pair returned by `find_closest_pair` is its initial argument.

Lemma 2. $p_0 \in \text{set } ps \wedge p_1 \in \text{set } ps \wedge p_0 \neq p_1 \wedge \text{dist } p_0 \text{ } p_1 < \delta \wedge$
 $(\forall p \in P_L. \text{fst } p \leq l) \wedge \text{sparse } \delta \text{ } P_L \wedge$
 $(\forall p \in P_R. l \leq \text{fst } p) \wedge \text{sparse } \delta \text{ } P_R \wedge$

×<
×>

$$\begin{aligned} \text{set } ps = P_L \cup P_R \wedge ps' &= \text{filter } (\lambda p. \text{dist } p(l, \text{snd } p) < \delta) ps \\ \Rightarrow p_0 \in \text{set } ps' \wedge p_1 \in \text{set } ps' \end{aligned}$$

We then can prove, additionally using Lemma 1, that *combine* computes indeed a pair of points (p_0, p_1) such that our given list of points ps is $(\text{dist } p_0 p_1)$ -sparse.

Lemma 3. $\text{sorted_snd } ps \wedge \text{set } ps = P_L \cup P_R \wedge$
 $(\forall p \in P_L. \text{fst } p \leq l) \wedge \text{sparse } (\text{dist } p_{0L} p_{1L}) P_L \wedge$
 $(\forall p \in P_R. l \leq \text{fst } p) \wedge \text{sparse } (\text{dist } p_{0R} p_{1R}) P_R \wedge$
 $(p_0, p_1) = \text{combine } (p_{0L}, p_{1L}) (p_{0R}, p_{1R}) l ps$
 $\Rightarrow \text{sparse } (\text{dist } p_0 p_1) (\text{set } ps)$

One can also show that p_0 and p_1 are in ps and distinct (and thus a closest pair of $\text{set } ps$), if $p_{0L} (p_{0R})$ and $p_{1L} (p_{1R})$ are in $P_L (P_R)$ and distinct.

3.2 The Divide & Conquer Algorithm

In Section 2 we glossed over some implementation detail which is necessary to achieve to running time of $\mathcal{O}(n \log n)$. In particular we need to partition the given list² of points ps along a vertical line l into two lists of nearly equal length during the divide step and obtain a list ys of the same points, sorted in ascending order by y -coordinate, for the *combine* step in *linear* time at each level of recursion.

Cormen *et al.* propose the following top-down approach: Their algorithm takes three arguments: the set of points P and lists xs and ys which contain the same set of points P but are respectively sorted by x and y -coordinate. The algorithm first splits xs at $\text{length } xs \text{ div } 2$ into two still sorted lists xs_L and xs_R and chooses l as either the x -coordinate of the last element of xs_L or the x -coordinate of the first element of xs_R . It then constructs the sets P_L and P_R respectively consisting of the points of xs_L and xs_R . For the recursive invocations it needs to obtain in addition lists ys_L and ys_R that are still sorted by y -coordinate and again respectively refer to the same points as xs_L and xs_R . It achieves this by iterating once through ys and checking for each point if it is contained in P_L or not, constructing ys_L and ys_R along the way.

But this approach requires an implementation of sets. In fact, if we want to achieve the overall worst case running time of $\mathcal{O}(n \log n)$ it requires an implementation of sets with linear time construction and constant time membership test, which is nontrivial, in particular in a functional setting. To avoid sets many publications and implementations either assume all points have unique x -coordinates or preprocess the points by applying for example a rotation such that the input fulfills this condition. For distinct x -coordinates one can then compute ys_L and ys_R by simply filtering ys depending on the x -coordinate of the points relative to l and eliminate the usage of sets entirely.

But there exists a third option which we have found only in Cormen *et al.* where it is merely hinted at in an exercise left to the reader. The approach

² Our implementation deals with concrete lists in contrast to the abstract sets used in Section 2.

is the following. Looking at the overall structure of the closest pair algorithm we recognize that it closely resembles the structure of a standard mergesort implementation and that we only need ys for the *combine* step after the two recursive invocations of the algorithm. Thus we can obtain ys by merging ‘along the way’ using a bottom-up approach. This is the actual code:

```

closest_pair_rec :: point list ⇒ point list × point × point
closest_pair_rec xs =
  (let n = length xs
   in if n ≤ 3 then (mergesort snd xs, closest_pair_bf xs)
   else let (xs_L, xs_R) = split_at (n div 2) xs;
            (ys_L, p0_L, p1_L) = closest_pair_rec xs_L;
            (ys_R, p0_R, p1_R) = closest_pair_rec xs_R;
            ys = merge snd ys_L ys_R
   in (ys, combine (p0_L, p1_L) (p0_R, p1_R) (fst (hd xs_R)) ys))

closest_pair :: point list ⇒ point × point
closest_pair ps =
  (let (ys, c0, c1) = closest_pair_rec (mergesort fst ps) in (c0, c1))

```

Function *closest_pair_rec* expects a list of points xs sorted by x -coordinate. The construction of xs_L , xs_R and l is analogous to Cormen *et al.* In the base case we then sort xs by y -coordinate and compute the closest pair using the brute-force approach (*closest_pair_bf*). The recursive call of the algorithm on xs_L returns in addition to the closest pair of xs_L the list ys_L , containing all points of xs_L but now sorted by y -coordinate. Analogously for xs_R and ys_R . Furthermore, we reuse function *merge* from our *mergesort* implementation, which we utilize to presort the points by x -coordinate, to obtain ys from ys_L and ys_R in linear time at each level of recursion.

Splitting of xs is performed by the function *split_at* via a simple linear pass over xs . Our implementation of *mergesort* sorts a list of points depending on a given projection function, *fst* for ‘by x -coordinate’ and *snd* for ‘by y -coordinate’.

Using Lemma 3, the functional correctness proofs of our mergesort implementation and several auxiliary lemmas proving that *closest_pair_rec* also sorts the points by y -coordinate, we arrive at the correctness proof of the desired sparsity property of the algorithm.

Theorem 1. $1 < \text{length } xs \wedge \text{sorted_fst } xs \wedge (ys, p_0, p_1) = \text{closest_pair_rec } xs$
 $\implies \text{sparse } (\text{dist } p_0 \ p_1) \ xs$

Corollary 1 together with Theorems 2 and 3 then show that the pair (p_0, p_1) is indeed a closest pair of ps .

Corollary 1. $1 < \text{length } ps \wedge (p_0, p_1) = \text{closest_pair } ps$
 $\implies \text{sparse } (\text{dist } p_0 \ p_1) \ ps$

Theorem 2. $1 < \text{length } ps \wedge (p_0, p_1) = \text{closest_pair } ps$
 $\implies p_0 \in \text{set } ps \wedge p_1 \in \text{set } ps$

Theorem 3. $1 < \text{length } ps \wedge \text{distinct } ps \wedge (p_0, p_1) = \text{closest_pair } ps$
 $\implies p_0 \neq p_1$

4 Time Complexity Proof

To formally verify the running time we follow the approach in [16]. For each function f we define a function $t.f$ that takes the same arguments as f but computes the number of function calls the computation of f needs, the ‘time’. Function $t.f$ follows the same recursion structure as f and can be seen as an abstract interpretation of f . To ensure the absence of errors we derive f and $t.f$ from a monadic function that computes both the value and the time but for simplicity of presentation we present only f and $t.f$. We also simplify matters a bit: we count only expensive operations where the running time increases with the size of the input; in particular we assume constant time arithmetic and ignore small additive constants. Due to reasons of space we only show one example of such a ‘timing’ function, $t.find_closest$, which is crucial to our time complexity proof.

```

t.find_closest :: point ⇒ real ⇒ point list ⇒ nat
t.find_closest p δ [] = 0
t.find_closest p δ [p0] = 1
t.find_closest p δ (p0 # ps) = 1 +
  (if δ ≤ snd p0 - snd p then 0
   else let p1 = find_closest p (min δ (dist p p0)) ps
        in t.find_closest p (min δ (dist p p0)) ps +
        (if dist p p0 ≤ dist p p1 then 0 else 0))

```

We set the time to execute *dist* computations to 0 since it is a combination of cheap operations. For the base cases of recursive functions we fix the computation time to be equivalent to the size of the input. This choice of constants is arbitrary and has no impact on the overall running time analysis but leads in general to ‘cleaner’ arithmetic bounds.

4.1 Time Analysis of the Combine Step

In Section 2 we claimed that the running time of the algorithm is captured by the recurrence $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$, where n is the length of the given list of points. This claim implies an at most linear overhead at each level of recursion. Splitting of the list xs , merging ys_L and ys_R and the filtering operation of the *combine* step run in linear time. But it is non-trivial that the function *find_closest_pair*, central to the *combine* step, also exhibits a linear time complexity. It is applied to an argument list of, in the worst case, length n , iterates once through the list and calls *find_closest* for each element. Consequently our proof obligation is the constant running time of *find_closest* or, considering our timing function, that there exists some constant c such that $t.find_closest p \delta ps \leq c$ holds in the context of the *combine* step.


Looking at the definition of *find_closest* we see that the function terminates as soon as it encounters the first point within the given list ps that does not fulfill the predicate $(\lambda q. \delta \leq \text{snd } q - \text{snd } p)$, the point p being an argument to

find_closest, or if *ps* is a list of length ≤ 1 . The corresponding timing function *t_find_closest* computes the number of recursive function calls, which is, in this case, synonymous with the number of examined points. For our time complexity proof it suffices to show the following bound on the result of *t_find_closest*. The proof is by induction on the computation of *t_find_closest*. The function *count f* is an abbreviation for *length* \circ *filter f*.

Lemma 4. $t_find_closest(p, \delta, ps) \leq 1 + count (\lambda q. snd\ q - snd\ p \leq \delta)\ ps$

Therefore we need to prove that the term *count* $(\lambda q. snd\ q - snd\ p \leq \delta)\ ps$ does not depend on the length of *ps*. Looking back at Figure 1, the square point representing *p*, we can assume that the list *p # ps* is distinct and sorted in ascending order by *y*-coordinate. From the precomputing effort of the *combine* step we know that its points are contained within the 2δ wide vertical strip centered around *l* and can be split into two sets P_L (P_R) consisting of all points which lie to the left (right) of or on the line *l*. Due to the two recursive invocations of the algorithm during the conquer step we can additionally assume that both P_L and P_R are δ -sparse, suggesting the following lemma which implies $t_find_closest\ p\ \delta\ ps \leq 8$ and thus the constant running time of *find_closest*.

Lemma 5. $distinct\ (p\ \# ps) \wedge sorted_snd\ (p\ \# ps) \wedge 0 \leq \delta \wedge$
 $(\forall q \in set\ (p\ \# ps). l - \delta < fst\ q \wedge fst\ q < l + \delta) \wedge$
 $set\ (p\ \# ps) = P_L \cup P_R \wedge$
 $(\forall q \in P_L. fst\ q \leq l) \wedge sparse\ \delta\ P_L \wedge$
 $(\forall q \in P_R. l \leq fst\ q) \wedge sparse\ \delta\ P_R$
 $\implies count\ (\lambda q. snd\ q - snd\ p \leq \delta)\ ps \leq 7$

 *Proof.* The library HOL-Analysis defines some basic geometric building blocks needed for the proof. A *closed box* describes points contained within rectangular shapes in Euclidean space. For our purposes the planar definition is sufficient.

$$cbox\ (x_0, y_0)\ (x_1, y_1) = \{(x, y) \mid x_0 \leq x \wedge x \leq x_1 \wedge y_0 \leq y \wedge y \leq y_1\}$$

The box is ‘closed’ since it includes points located on the border of the box. We then introduce some useful abbreviations:

- The rectangle *R* is the upper half of the shaded square of Figure 1:
 $R = cbox\ (l - \delta, snd\ p)\ (l + \delta, snd\ p + \delta)$
- The set R_{ps} consists of all points of *p # ps* that are encompassed by *R*:
 $R_{ps} = R \cap set\ (p\ \# ps)$
- The squares S_L and S_R denote the left and right halves of *R*:
 $S_L = cbox\ (l - \delta, snd\ p)\ (l, snd\ p + \delta)$
 $S_R = cbox\ (l, snd\ p)\ (l + \delta, snd\ p + \delta)$
- The set S_{PL} holds all points of P_L that are contained within the square S_L . The definition of S_{PR} is analogous:
 $S_{PL} = P_L \cap S_L, S_{PR} = P_R \cap S_R$

Let additionally ps_f abbreviate the term *filter* $(\lambda q. \text{snd } q - \text{snd } p \leq \delta)$ ps . First we prove $\text{length } (p \# ps_f) \leq |R_{ps}|$: Let q denote an arbitrary point of $p \# ps_f$. We know $\text{snd } p \leq \text{snd } q$ because the list $p \# ps$ and therefore $p \# ps_f$ is sorted in ascending order by y -coordinate and $\text{snd } q \leq \text{snd } p + \delta$ due to the filter predicate $(\lambda q. \text{snd } q - \text{snd } p \leq \delta)$. Using the additional facts $l - \delta \leq \text{fst } q$ and $\text{fst } q \leq l + \delta$ (derived from our assumption that all points of $p \# ps$ are contained within the 2δ strip) and the definitions of R_{ps} , R and $cbox$ we know $q \in R_{ps}$ and thus $\text{set } (p \# ps_f) \subseteq R_{ps}$. Since the list $p \# ps_f$ maintains the distinctness property of $p \# ps$ we additionally have $\text{length } (p \# ps_f) = |\text{set } (p \# ps_f)|$. Our intermediate goal immediately follows because $|\text{set } (p \# ps_f)| \leq |R_{ps}|$ holds for the finite set R_{ps} .

But how many points can there be in R_{ps} ? Lets first try to determine an upper bound for the number of points of S_{PL} . All its points are contained within the square S_L whose side length is δ . Moreover, since P_L is δ -sparse and $S_{PL} \subseteq P_L$, S_{PL} is also δ -sparse, or the distance between each distinct pair of points of S_{PL} is at least δ . Therefore the cardinality of S_{PL} is bounded by the number of points we can maximally fit into S_L , maintaining a pairwise minimum distance of δ . As the left-hand side of Figure 2 depicts, we can arrange at most four points adhering to these restrictions, and consequently have $|S_{PL}| \leq 4$. An analogous argument holds for the number of points of S_{PR} . Furthermore we know $R_{ps} = S_{PL} \cup S_{PR}$ due to our assumption $\text{set } (p \# ps) = P_L \cup P_R$ and the fact $R = S_L \cup S_R$ and can conclude $|R_{ps}| \leq 8$. Our original statement then follows from $\text{length } (p \# ps_f) \leq |R_{ps}|$. \square

X

✓

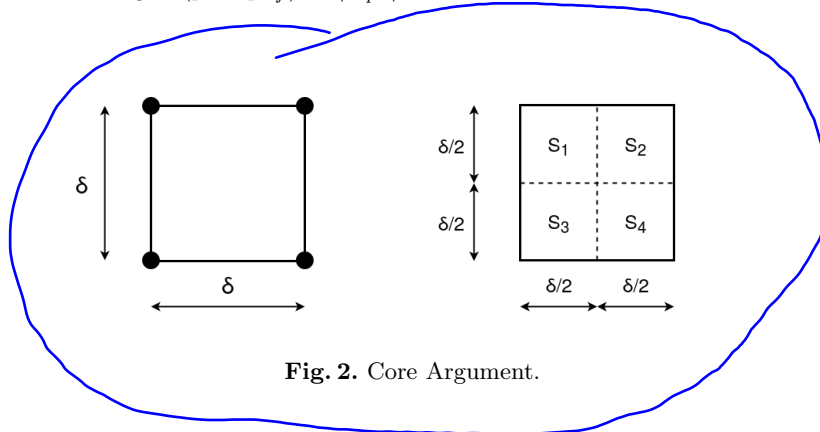


Fig. 2. Core Argument.

Note that the intermediate proof for the bound on $|R_{ps}|$ relies on basic human geometric intuition. Indeed Cormen *et al.* [5] and most of the proofs in the literature do. But for a formal proof we have to be rigorous. First we show two auxiliary lemmas: The maximum distance between two points in a square S with side length δ is less than or equal to $\sqrt{2}\delta$.

Lemma 6. $p_0 = (x, y) \wedge p_1 = (x + \delta, y + \delta) \wedge 0 \leq \delta \wedge$
 $(x_0, y_0) \in cbox \ p_0 \ p_1 \wedge (x_1, y_1) \in cbox \ p_0 \ p_1$
 $\implies \text{dist } (x_0, y_0) \ (x_1, y_1) \leq \sqrt{2} * \delta$

The proof is straightforward. Both points are contained within the square S , the difference between their x and y coordinates is hence bounded by δ and we finish the proof using the definition of the Euclidean distance. Below we employ the following variation of the *pigeonhole* principle:

Lemma 7. *finite* $B \wedge A \subseteq \bigcup B \wedge |B| < |A|$
 $\implies \exists x \in A. \exists y \in A. \exists S \in B. x \neq y \wedge x \in S \wedge y \in S$

Finally we replace human intuition with formal proof:

Lemma 8. $(\forall p \in P. p \in \text{cbox } (x, y) (x + \delta, y + \delta)) \wedge \text{sparse } \delta P \wedge 0 \leq \delta$
 $\implies |P| \leq 4$

Proof. Let S denote the square with a side length of δ and suppose, for the sake of contradiction, that $4 < |P|$. Then S can be split into the four congruent squares S_1, S_2, S_3, S_4 along the common point $(x + \delta/2, y + \delta/2)$ as depicted by the right-hand side of Figure 2. Since all points of P are contained within S and $S = \bigcup \{S_1, S_2, S_3, S_4\}$ we have $P \subseteq \bigcup \{S_1, S_2, S_3, S_4\}$. Using Lemma 7 and our assumption $4 < |P|$ we know there exists a square $S_i \in \{S_1, S_2, S_3, S_4\}$ and a pair of distinct points $p_0 \in S_i$ and $p_1 \in S_i$. Lemma 6 and the fact that all four sub-squares have the same side length $\delta / 2$ shows that the distance between p_0 and p_1 must be less than or equal to $\sqrt{2} / 2 * \delta$ and hence strictly less than δ . But we also know that δ is a lower bound for this distance because $p_0 \in P, p_1 \in P, p_0 \neq p_1$ and our premise that P is δ -sparse, a contradiction. \square

4.2 Time Analysis of the Divide & Conquer Algorithm

In summary, the time to evaluate *find_closest* $p \delta ps$ is constant in the context of the *combine* step and thus evaluating *find_closest_pair* $(p_0, p_1) ps$ as well as *combine* $(p_{0L}, p_{1L}) (p_{0R}, p_{1R}) l ps$ takes time linear in *length ps*.

Next we turn our attention to the timing of *closest_pair_rec* and derive (but do not show) the corresponding function *t_closest_pair_rec*. At this point we could prove a concrete bound on *t_closest_pair_rec*. But since we are dealing with a divide-and-conquer algorithm we should, in theory, be able to determine its running time using the ‘master theorem’ [5]. This is, in practice, also possible in Isabelle/HOL. Eberl [8] has formalized the Akra-Bazzi theorem [1], a generalization of the master theorem. Using this formalization we can derive the running time of our divide-and-conquer algorithm without a direct proof for *t_closest_pair_rec*. First we capture the essence of *t_closest_pair_rec* as a recurrence on natural numbers representing the length of the list argument of *(t.)closest_pair_rec*:

```
closest_pair_recurrence :: nat ⇒ real
closest_pair_recurrence n =
  (if n ≤ 3 then n + mergesort_recurrence n + n * n
   else 13 * n + closest_pair_recurrence [n / 2] +
    closest_pair_recurrence [n / 2])
```

The time complexity of this recurrence is proved completely automatically:

Lemma 9. $closest_pair_recurrence \in \Theta(\lambda n. n * \ln n)$

Next we need to connect this bound with our timing function. Lemma 10 below expresses a procedure for deriving complexity properties of the form

$$t \in O[m \text{ going_to at_top within } A](f \circ m)$$

where t is a timing function on the data domain, in our case lists. The function m is a measure on that data domain, r is a recurrence or any other function of type $nat \Rightarrow real$ and A is the set of valid inputs. The term ‘ $m \text{ going_to at_top within } A$ ’ should be read as ‘if the measured size of valid inputs is sufficiently large’ and utilizes Eberls formalization of Landau Notation [9] and the “filter” machinery of asymptotics in Isabelle/HOL [11]. For readability we omit stating the filter and m explicitly in the following and just state the conditions required of the input A . The measure m always corresponds to the *length* function.

Lemma 10. $(\forall x \in A. t\ x \leq (r \circ m)\ x) \wedge r \in O(f) \wedge (\forall x \in A. 0 \leq t\ x) \\ \implies t \in O[m \text{ going_to at_top within } A](f \circ m)$

Lemma 11. $distinct\ ps \wedge sorted_fst\ ps \\ \implies t_closest_pair_rec\ ps \leq (closest_pair_recurrence \circ length)\ ps$

Using Lemmas 9, 10 and 11 we arrive at Theorem 4, expressing our main claim: the running time of the divide-and-conquer algorithm.

Theorem 4. *For inputs that are distinct and sorted by x-coordinate:*
 $t_closest_pair_rec \in O(\lambda n. n * \ln n)$

Since the function *closest_pair* only presorts the given list of points using our mergesort implementation and then calls *closest_pair_rec* we obtain Corollary 2 and finish the time complexity proof.

Corollary 2. *For distinct inputs:* $t_closest_pair \in O(\lambda n. n * \ln n)$

5 Alternative Implementations

In the literature there exist various other algorithmic approaches to solve the closest pair problem. Most of them are closely related to our implementation of Section 3, deviating primarily in two aspects: the exact implementation of the *combine* step and the approach to sorting the points by y -coordinate we already discussed in Subsection 3.2. We present a short overview, concentrating on the *combine* step and the second implementation we verified.

5.1 A Second Verified Implementation

Although the algorithm described by Cormen *et al.* is the basis for our implementation of Section 3, we took the liberty to optimize it. During execution of `find_closest p δ ps` our algorithm searches for the closest neighbor of p within the rectangle R , the upper half of the shaded square S of Figure 1, and terminates the search if it examines points on or beyond the upper border of R . Cormen *et al.* originally follow a slightly different approach. They search for a closest neighbor of p by examining a constant number of points of ps , the first 7 to be exact. This is valid because there are at most 7 points within R , not counting p , and the 8th point of ps would again lie on or beyond the upper border of R . This slightly easier implementation comes at the cost of being less efficient in practice. Cormen *et al.* are always assuming the worst case by checking all 7 points following p . But it is unlikely that the algorithm needs to examine even close to 7 points, except for specifically constructed inputs. They furthermore state that the bound of 7 is an over-approximation and dare the reader to lower it to 5 as an exercise. We refrain from doing so since a bound of 7 suffices for the time complexity proof of our, inherently faster, implementation. At this point we should also mention that the specific optimization of Section 3 is not our idea but rather an algorithmic detail which is unfortunately rarely mentioned in the literature.

Nonetheless we can adapt the implementation of Section 3 and the proofs of Section 4 to verify the original implementation of Cormen *et al.* as follows: We replace each call of `find_closest p δ ps` by a call to `find_closest_bf p (take 7 ps)` where `find_closest_bf` iterates in brute-force fashion through its argument list to find the closest neighbor of p . To verify this implementation we then reuse most of the elementary lemmas and proof structure of Sections 3 and 4, only a slightly adapted version of Lemma 5 is necessary. Note that this lemma was previously *solely* required for the time complexity proof of the algorithm. Now it is already necessary during the functional correctness proof since we need to argue that examining only a constant number of points of ps is sufficient. The time analysis is overall greatly simplified: A call of the form `find_closest_bf p (take 7 ps)` runs in constant time and we again are able to reuse the remaining time analysis proof structure of Section 4. For the exact differences between both formalizations we encourage the reader to consult our entry in the Archive of Formal Proofs [23].

5.2 Related Work

Over the years a considerable amount of effort has been made to further optimize the *combine* step. Central to these improvements is the ‘complexity of computing distances’, abbreviated CCP in the following, a term introduced by Zhou *et al.* [26] which measures the number of Euclidean distances computed by a closest pair algorithm. The core idea being, since computing the Euclidean distance is more expensive than other primitive operations, it might be possible to improve overall algorithmic running time by reducing this complexity measure. In the same paper they introduce an optimized version of the closest pair

algorithm with a CCP of $2n \log n$, in contrast to $7n \log n$ which will be the worst case CCP of the algorithm of Section 3 after we minimize the number of distance computations in Section 6. They improve upon the algorithm presented by Preparata and Shamos [21] which achieves a CCP of $3n \log n$. Ge *et al.* [10] base their, quite sophisticated, algorithm on the version of Zhou *et al.* and prove an even lower CCP of $\frac{3}{2}n \log n$ for their implementation. The race for the lowest number of distance computations culminates so far with the work of Jiang and Gillespie [13] who present their algorithms ‘Basic-2’³ and ‘2-Pass’ with a respective CCP of $2n \log n$ and (for the first time linear) $\frac{7}{2}n$.

6 Executable Code

Before we explore how our algorithm stacks up against Basic-2 (which is surprisingly the fastest of the CCP minimizing algorithms according to Jiang and Gillespie) we have to make some final adjustments to generate executable code from our formalization.

In Section 3 we fixed the data representation of a point to be a pair of mathematical ints rather than mathematical reals. During code export Isabelle then maps, correctly and automatically, its abstract data type *int* to a suitable concrete implementation of (arbitrary-sized) integers; for our target language OCaml using the library ‘zarith’. For the data type *real* this is not possible since we cannot implement mathematical reals. We would instead have to resort to an approximation (e.g. floats) losing all proved guarantees in the process. But currently our algorithm still uses the standard Euclidean distance and hence mathematical reals due to the *sqr*t function. For the executable code we have to replace this distance measure by the squared Euclidean distance. To prove that we preserve the correctness of our implementation several small variations of the following lemma suffice:

$$\text{dist } p_0 \ p_1 \leq \text{dist } p_2 \ p_3 \longleftrightarrow (\text{dist } p_0 \ p_1)^2 \leq (\text{dist } p_2 \ p_3)^2$$

We apply two further code transformations. To minimize the number of distance computations we introduce auxiliary variables which capture and then replace repeated computations. For all of the shown functions that return a point or a pair of points this entails returning the corresponding computed distance as well. Furthermore we replace recursive auxiliary functions such as *filter* by corresponding tail-recursive implementations to allow the OCaml compiler to optimize the generated code and prevent stackoverflows. To make sure these transformations are correct we prove lemmas expressing the equivalence of old and new implementations for each function. Isabelles code export machinery can then apply these transformations automatically.

Now it is time to evaluate the performance of our verified code. Figure 3 depicts the running time ratios of several implementations of the algorithm of Section 3 (called Basic- δ) and Basic-7 (the original approach of Cormen *et al.*) over

³ Pereira and Lobo [19] later independently developed the same algorithm and additionally present extensive functional correctness proofs for all Minkowski distances.

Basic-2. Basis- δ is tested in three variations: the exported (purely functional) Isabelle code and equivalent handwritten functional and imperative implementations to gauge the overhead of the machine generated code. All algorithms are implemented in OCaml, use our bottom-up approach to sorting (imperative implementations sort in place) of Subsection 3.2 and for each input of uniformly distributed points 50 independent executions were performed. Remarkably the exported code is only about 2.28⁴ times slower than Basic-2 and furthermore most of the difference is caused by the inefficiencies inherent to machine generated code since its equivalent functional implementation is only 11% slower than Basic-2. Basic-7 is 2.26 times slower than the imperative Basic- δ which demonstrates the huge impact the small optimization of Subsection 5.1 can have in practice.

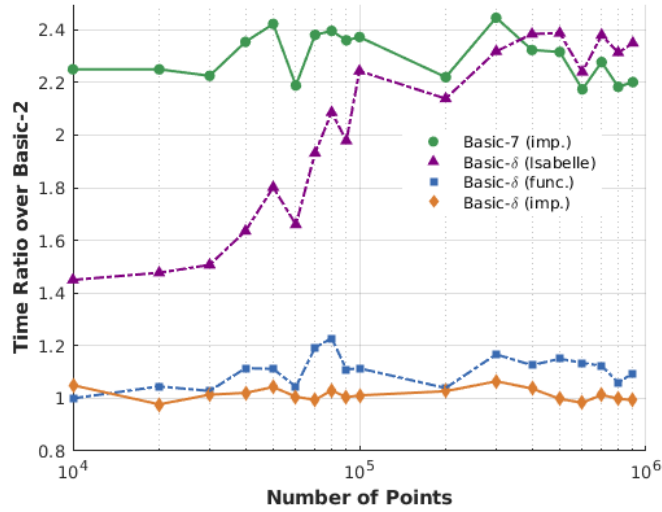


Fig. 3. Benchmarks.

7 Conclusion

We have presented the first verification (both functional correctness and running time) of two related closest pair of points algorithms in the plane, without assuming the x coordinates of all points to be distinct. The executable code generated from the formalization is competitive with existing reference implementations. A challenging and rewarding next step would be to formalize and

⁴ We measure differences between running times as the average over all data points weighted by the size of the input.

verify a closest pair of points algorithm in arbitrary dimensions. This case is treated rather sketchily in the literature.

Acknowledgements Research supported by DFG grants NI 491/16-1 and 18-1.

References

1. Akra, M., Bazzi, L.: On the solution of linear recurrence equations. *Computational Optimization and Applications* **10**(2), 195–210 (1998). <https://doi.org/10.1023/A:1018373005182>, <https://doi.org/10.1023/A:1018373005182>
2. Bentley, J.L., Shamos, M.I.: Divide-and-conquer in multidimensional space. In: *Proc. Eighth Annual ACM Symposium on Theory of Computing*. pp. 220–230. STOC '76, ACM (1976). <https://doi.org/10.1145/800113.803652>
3. Bertot, Y.: Formal verification of a geometry algorithm: A quest for abstract views and symmetry in Coq proofs. In: Fischer, B., Uustalu, T. (eds.) *Theoretical Aspects of Computing - ICTAC 2018*. LNCS, vol. 11187, pp. 3–10. Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_1
4. Brun, C., Dufourd, J., Magaud, N.: Designing and proving correct a convex hull algorithm with hypermaps in Coq. *Comput. Geom.* **45**(8), 436–457 (2012). <https://doi.org/10.1016/j.comgeo.2010.06.006>
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*, Third Edition. The MIT Press, 3rd edn. (2009)
6. Dufourd, J.: An intuitionistic proof of a discrete form of the Jordan Curve Theorem formalized in Coq with combinatorial hypermaps. *J. Autom. Reasoning* **43**(1), 19–51 (2009). <https://doi.org/10.1007/s10817-009-9117-x>
7. Dufourd, J., Bertot, Y.: Formal study of plane delaunay triangulation. In: Kaufmann, M., Paulson, L.C. (eds.) *Interactive Theorem Proving, ITP 2010*. LNCS, vol. 6172, pp. 211–226. Springer (2010). https://doi.org/10.1007/978-3-642-14052-5_16
8. Eberl, M.: Proving divide and conquer complexities in Isabelle/HOL. *Journal of Automated Reasoning* **58**(4), 483–508 (Apr 2017). <https://doi.org/10.1007/s10817-016-9378-0>
9. Eberl, M.: Verified real asymptotics in Isabelle/HOL. In: *Proceedings of the International Symposium on Symbolic and Algebraic Computation. ISSAC '19*, ACM, New York, NY, USA (2019). <https://doi.org/10.1145/3326229.3326240>
10. Ge, Q., Wang, H.T., Zhu, H.: An improved algorithm for finding the closest pair of points. *Journal of Computer Science and Technology* **21**(1), 27–31 (Jan 2006). <https://doi.org/10.1007/s11390-006-0027-7>, <https://doi.org/10.1007/s11390-006-0027-7>
11. Hölzl, J., Immler, F., Huffman, B.: Type classes and filters for mathematical analysis in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) *Interactive Theorem Proving*. pp. 279–294. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
12. Immler, F.: A verified algorithm for geometric zonotope/hyperplane intersection. In: *Certified Programs and Proofs*. pp. 129–136. CPP '15, ACM (2015). <https://doi.org/10.1145/2676724.2693164>
13. Jiang, M., Gillespie, J.: Engineering the divide-and-conquer closest pair algorithm. *Journal of Computer Science and Technology* **22**(4), 532–540 (2007)

14. Meikle, L.I., Fleuriot, J.D.: Mechanical theorem proving in computational geometry. In: Hong, H., Wang, D. (eds.) *Automated Deduction in Geometry*, ADG 2004. LNCS, vol. 3763, pp. 1–18. Springer (2004). https://doi.org/10.1007/11615798_1
15. Narboux, J., Janicic, P., Fleuriot, J.: Computer-assisted Theorem Proving in Synthetic Geometry. In: Sitharam, M., John, A.S., Sidman, J. (eds.) *Handbook of Geometric Constraint Systems Principles*. Discrete Mathematics and Its Applications, Chapman and Hall/CRC (2018)
16. Nipkow, T.: Verified root-balanced trees. In: Chang, B.Y.E. (ed.) *Asian Symposium on Programming Languages and Systems*, APLAS 2017. LNCS, vol. 10695, pp. 255–272. Springer (2017)
17. Nipkow, T., Klein, G.: *Concrete Semantics with Isabelle/HOL*. Springer (2014), <http://concrete-semantics.org>
18. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
19. Pereira, J.C., Lobo, F.G.: An optimized divide-and-conquer algorithm for the closest-pair problem in the planar case. *Journal of Computer Science and Technology* **27**(4), 891–896 (2012)
20. Pichardie, D., Bertot, Y.: Formalizing convex hull algorithms. In: Boulton, R.J., Jackson, P.B. (eds.) *Theorem Proving in Higher Order Logics*, TPHOLs 2001. LNCS, vol. 2152, pp. 346–361. Springer (2001). https://doi.org/10.1007/3-540-44755-5_24
21. Preparata, F.P., Shamos, M.I.: *Computational Geometry: An Introduction*. Springer-Verlag, Berlin, Heidelberg (1985)
22. Puitg, F., Dufourd, J.: Formalizing mathematics in higher-order logic: A case study in geometric modelling. *Theor. Comput. Sci.* **234**(1-2), 1–57 (2000). [https://doi.org/10.1016/S0304-3975\(98\)00228-X](https://doi.org/10.1016/S0304-3975(98)00228-X)
23. Rau, M., Nipkow, T.: Closest pair of points algorithms. *Archive of Formal Proofs* (Jan 2020), http://isa-afp.org/entries/Closest_Pair_Points.html, Formal proof development
24. Sack, J.R., Urrutia, J. (eds.): *Handbook of Computational Geometry*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands (2000)
25. Shamos, M.I., Hoey, D.: Closest-point problems. In: *16th Annual Symposium on Foundations of Computer Science* (sfcs 1975). pp. 151–162 (Oct 1975). <https://doi.org/10.1109/SFCS.1975.8>
26. Zhou, Y., Xiong, P., Zhu, H.: An improved algorithm about the closest pair of points on plane set. *Computer Research and Development* **35**(10), 957–960 (1998)