Formal Verification of a Closest Pair Algorithm

MIT 6.850 Final Project

Logan Weber
loganweb@mit.edu

Contents

1 Introduction

In the past few years, the Lean theorem prover and programming language has caught the attention of the math community. For a long time, proof assistants were thought of as unsatisfactory for formalizing undergraduate-level mathematics, let alone *research*-level mathematics. This image began to change when a challenge by Fields Medalist, Peter Scholze, was proposed: The Liquid Tensor Experiment. This challenge involved formalizing a foundational result in a new area of mathematics. The Lean community took up the challenge, and within 6 months, had formalized the most import aspects of the proof, exceeding Scholze's expectations.

The rise of Lean cannot be explained by any significant *technological* innovation—it was a *cultural* innovation. The most relevant comparison is to the Coq proof assistant, which has emphasized sound, type-theoretic principles since its conception. Lean instead focuses on directly accomodating the workflows of mathematicians, embracing language features that elude type-theoretic grounding (e.g., quotients) and that destroy computational properties (e.g., the law of excluded middle).

Because of Lean's focus on ergonomics, it has attracted a number of mathematicians, who have formalized a substantial body of undergraduate-level mathematics in a library known as Mathlib.

Earlier this term, Professor Indyk mentioned the difficulties a past student had when verifying one of the algorithms from lecture. In this project, our goal was to survey the state of formal verification as it pertains to geometric computing. The question we seek to answer is: how much of the proof burden does Mathlib alleviate in the process of formal verification?

Concretely, we set out to implement and formally verify the closest pair (with help) and helper finder algorithms. We did not make it to the helper finder algorithm, and our final achievements were [1]:

- An implementation for a restricted form of the closest pair (with help) problem.
- A partial verification of the algorithm, with some low-level lemmas assumed to be true.

In the remainder of this report, we cover background for the closest pair with help problem (Section 2), then we explain the process of verification (Section 3). Finally, in Section 4, we offer some concluding thoughts.

2 Background

In this section, we recap the closest pair with help problem and the algorithm presented in class that solves it (§ 2.1). Then, we cover simplifications and restrictions we imposed on the specification, to make it amenable to formal verification (§ 2.2). Finally, we collect all notation used in this report and motivate the nonstandard notation (§ 2.3).

2.1 Closest Pair With Help

Recall the problem specification for closest pair (with help) below.

Problem 1: Closest Pair With Help

Given a set of points $P \subseteq \mathbb{R}^d$, find the closest pair

$$(p^*, q^*) = \operatorname*{arg\,min}_{(p,q)\in P\times P, p\neq q} \|p - q\|_2,$$

knowing $\|p^* - q^*\|_2 \in (t, ct]$.

The algorithm we saw in Lecture 11 to solve this problem is as follows.

Algorithm 1 Closest Pair With Help

$g \leftarrow$ grid with side lengths $k = 1/\sqrt{d}$.
Insert each point p in the bucket in g with index $\lfloor p/k \rfloor$.
$xy \leftarrow$ none
for $p \in P$ **do**
  for $\boldsymbol{v} \in [-\lceil c+1 \rceil, \lceil c+1 \rceil]^d$ **do**
    $xy \leftarrow \arg\min\{xy, \arg\min_{q\in g[\lfloor p+\boldsymbol{v}/k \rfloor]}\{\|p-q\|\}\}$
  end for
end for
return xy

2.2 Simplified Closest Pair With Help

For the sake of both simplicity and computability, we impose the following restrictions on the specification in Section 2.1:

- **The points have integral components.** Lean does not include facilities for computing with real numbers, though, in principle, it could—with all results computed up to *finite* precision. Without such facilities, we are left with $\mathbb{Q}$ or $\mathbb{Z}$ as natural carrier types. We chose $\mathbb{Z}$, believing it would be simpler. In hindsight, using $\mathbb{Q}$ may have been just as easy and perhaps easier. Since $\mathbb{Q}$ forms a field, it has multiplicative inverses, which simplifies algebra.

[1] Code can be found at https://github.com/weberlo/verified-gridding.

- **The points are in the plane.** This assumption was for simplicity on the first pass through the verification. It is unclear to the author how much more difficult the verification would be in arbitrary dimensions.
- **The norm is the *squared* Euclidean norm.** We're working with integers and the outermost operation of $\|\cdot\|_2$ is a square root, which could return a value outside of $\mathbb{Z}$. Our fix is to consider the squared Euclidean norm $\|\cdot\|_2^2$.
- **The points are scaled so $t = 1$.** We assumed this property in class, but because we're working with integers, it's a stronger assumption; scaling down could lose information and cause us to return the wrong pair.

With these restrictions, the specification now looks like the following, with changes highlighted in blue.

Problem 2: Closest Pair With Help (Restricted)

Given a set of points $P \subseteq \mathbb{Z}^2$, find the closest pair

$$(p^*, q^*) = \underset{(p,q)\in P\times P, p\neq q}{\arg\min} \|p - q\|_2^2,$$

knowing $\|p^* - q^*\|_2^2 \in (1, c]$.

Trivial Grids. A notable corollary of these restrictions is that the mapping from points to their grid indices is the identity map. Recall, we wish to set the grid size k so the diameter of each cell is < 1, guaranteeing at most one point per cell. We assume the bottom and left cell edges are inclusive and the top and right edges are exclusive. Then, the furthest integer point from the bottom left $(0, 0)$ is at $(k - 1, k - 1)$.

$$\begin{aligned}\|(k-1, k-1)\|_2^2 &< 1\\ 2(k-1)^2 &< 1\end{aligned}$$

The only integral solution is 1, so each point's grid index is itself.

Different Search Bounds. Knowing the closest pair distance is at most c, our norm gives us bounds on how many cells away we must search. If $\|p^* - q^*\| \leq c$, then $(p_1^* - q_1^*)^2 + (p_2^* - q_2^*)^2 \leq c$, meaning $(p_1^* - q_1^*)^2 \leq c$ and $(p_2^* - q_2^*)^2 \leq c$. Then for each component a of p and b of q, we have

$$\begin{aligned}(a-b)^2 &\leq c\\ |a-b| &\leq \sqrt{c}^{\mathbb{N}}\end{aligned}$$

where $\sqrt{n}^{\mathbb{N}}$ is the largest number k such that $k^2 \leq n$. So if we search hypercubes of radius $\sqrt{c}^{\mathbb{N}} + 1$, we will certainly find the closest pair.

The Simplified Algorithm. With this simplified specification and the observations above, we verify the following modified algorithm (again, with changes highlighted in blue).

2.3 Notation

Here, we explain any notation in Table 2.3 that was not explained in the previous subsections and that will be useful in later sections.

Algorithm 2 Simplified Closest Pair With Help

$g \leftarrow$ grid with side lengths $k = 1$.
Insert each point p in the bucket in g with index p.
$xy \leftarrow \mathsf{none}$
for $p \in P$ **do**
  for $\boldsymbol{v} \in [-(\sqrt{c}^{\mathbb{N}} + 1), \sqrt{c}^{\mathbb{N}} + 1]^2$ **do**
    $xy \leftarrow \arg\min\{xy, \arg\min_{q \in g[p+\boldsymbol{v}]}\{\|p - q\|\}\}$
  end for
end for
return xy

Notation	Description
$P \subseteq \mathbb{Z}^2$	list of input points to the algorithm
$P' \subseteq P$	proposition that P' is a sublist of P
$[]$	the empty list
$p :: P'$	the list P' with p appended to the front
$p, q, r, s, x, y, z, w \in P$	points in P
$a, b \in \mathbb{Z}$	integers
$k, m, n \in \mathbb{N}$	natural numbers
$c \in \mathbb{N}^+$	the distance hint
$(P \times P)^?$	set of potentially null point pairs in P
$xy, zw, rs \in (P \times P)^?$	potentially null pairs of points in P
$\mathsf{some}(x, y) \in (P \times P)^?$	definitely non-null pairs of points in P
$\mathsf{none} \in (P \times P)^?$	definitely null pairs of points in P
$xy \leq zw$	proposition that the *nullable* pair xy is closer together than zw
$\mathrm{CP}_P(x, y)$	(x, y) is the closest pair in P
$B(p, c)$	set of points $\in \mathbb{Z}^2$ within distance c of the point p
$B_P(p, c)$	set of points $\in P$ within distance c of the point p (i.e., $B(p, c) \cap P$)
$\widetilde{\mathrm{CP}}_P(xy)$	xy is the closest pair in $\bigcup_{p \in P}(p, B_P(p, c))$, or none if there are no pairs in this set
$\mathsf{Grid} \coloneqq (\mathsf{grid} : \mathbb{Z}^2 \rightharpoonup [\mathbb{Z}^2]) \times (P : [\mathbb{Z}^2]) \times (c : \mathbb{N}^+)$	the type/fields of our grid structure
$G(P, c) : \mathsf{Grid}$	grid constructed from set of points P and distance hint c
$g : \mathsf{Grid}$	arbitrary grid structure
$g[p]$	points in cell at index p
$\|\cdot\|$	the *squared* Euclidean norm
$\sqrt{n}^{\mathbb{N}}$	the natural number square root
$\lfloor p \rfloor$	pointwise application of floor function to p

Figure 1: Summary of notation used throughout our report

Most of our notation is standard or self-explanatory, so we focus on motivating the outliers.

Potentially null point pairs. Because the algorithm searches for pairs within balls around points, we must account for the case where there are no such pairs. We handle this outcome by lifting the type of point pairs $(P \times P)$ to the type $(P \times P)^?$ of possibly null point pairs. This type has inhabitants of the form none, expressing a null pair, and $\mathsf{some}(x, y)$, expressing a non-null pair. When we have a term of type $(P \times P)^?$, and we don't yet know whether it's null, we write it as xy.

Ordering on potentially null point pairs. In the algorithm, we frequently compare a single ball's closest pair to the closest pair from a lower recursion level. Both of these results could be null, so we must case on whether zero, one, or both are null and choose the "closest" result accordingly.

Instead of littering our code with casework on nullable pairs, we impose an order $\mathit{xy} \leq \mathit{zw}$ on $(P \times P)^?$, with none representing a pair that is infinitely distant. Our order satisfies the following rules:

$$\begin{aligned} \mathsf{none} &\leq \mathsf{none} \\ \mathsf{none} &\not\leq \mathsf{some}(z, w) \\ \mathsf{some}(x, y) &\leq \mathsf{none} \\ (\mathsf{some}(x, y) \leq \mathsf{some}(z, w)) &\Leftrightarrow (\|x - y\| \leq \|z - w\|) \end{aligned}$$

Ball of points in P. Conceptually, our algorithm draws balls around points in P and finds other points in P lying within those balls. To express this concept in our proofs, we augment standard $B(p, c)$ notation to $B_P(p, c) \coloneqq B(p, c) \cap P$, which includes all points in P lying within $B(p, c)$.

Closest pair in union of balls. As we will see in Section 3.2.1, directly proving the (x, y) returned by our algorithm is the closest pair (i.e., $\mathrm{CP}_P(x, y)$) doesn't work. Instead, we define a predicate $\widetilde{\mathrm{CP}}_P(\mathit{xy})$, expressing that xy is the closest pair among all pairs in balls we draw around points in P. Then, we prove our algorithm satisfies this predicate.

Grid construction. We require a few lemmas expressing properties about the grid structure, so we notate it as a tuple $G(P, c)$ with fields

- $(\mathsf{grid} : \mathbb{Z}^2 \rightharpoonup [\mathbb{Z}^2])$: the hash table mapping grid indices to cells containing lists of points.
- $(P : [\mathbb{Z}^2])$: the set of points used to generate the grid. We need to carry this data around to express relations between the set of input points and the grid we create.
- $(c : \mathbb{N}^+)$: the distance hint. We include this for similar reasons to why we include P.

When we don't have access to the parameters a grid was constructed with, we notate it as g.

3 Verification

We omit any detailed explanation of our implementation, since that was not the difficult part. However, since it will be necessary to refer to the functions we prove properties of, we summarize our implementation in Table 3.

The goal of our verification is to prove the following theorem.

Theorem 1: `find_closest_pair` Finds Closest Pair

If the closest pair is within distance $(1, c]$, `find_closest_pair` finds (x, y) such that $\mathrm{CP}_P(x, y)$.

Figure 2 gives an overview of our proof, highlighting the results of interest and capturing the dependencies between lemmas.

Function Name	Signature	Description
`find_closest_pair`	$(c : \mathbb{N}^+) \to (P : [\mathbb{Z}^2]) \to (P \times P)^?$	The entry point to our algorithm. Constructs the grid, then calls `aux`. If the result is further than distance c, returns none.
`aux`	$(g : \mathsf{Grid}) \to (P : [\mathbb{Z}^2]) \to (P \times P)^?$	The core subroutine of the algorithm. Recurses over P, using `mdp_with` at each point p to find points in $B_P(p, c)$.
`grid_points`	$(P : [\mathbb{Z}^2]) \to (c : \mathbb{N}^+) \to \mathsf{Grid}$	Creates the grid structure, inserting points according to `get_grid_idx`.
`get_grid_idx`	$(p : \mathbb{Z}^2) \to \mathbb{Z}^2$	Converts a point to its grid index (the identity mapping under the current set of restrictions).
`min_dist_pair_aux`	$(p : \mathbb{Z}^2) \to (Q : [\mathbb{Z}^2]) \to (P \times P)^?$	Given p and a list of points in a ball around it, returns the closest point to p.
`min_dist_pair`	$(p : \mathbb{Z}^2) \to (g : \mathsf{Grid}) \to (P \times P)^?$	Collects points around p via `get_neighbs`, then uses `min_dist_pair` to find the closest among them.
`get_hypercube`	$(n : \mathbb{N}) \to [\mathbb{Z}^2]$	Given a radius n, returns the set of grid cell indices that intersect a hypercube of radius n.
`get_idxs`	$(p : \mathbb{Z}^2) \to (c : \mathbb{N}^+) \to [\mathbb{Z}^2]$	Returns a list of indices in a hypercube of radius $\sqrt{c}^{\mathbb{N}} + 1$ around p, making use of `get_hypercube`.
`lift_option_list`	$\forall(\alpha : \mathsf{Type}).\ [\alpha]^? \to [\alpha]$	Converts all none lists into empty lists.
`get_neighbs`	$(p : \mathbb{Z}^2) \to (g : \mathsf{Grid}) \to [\mathbb{Z}^2]$	Uses `get_idxs` to collect indices around p, then fetches cells at those indices from g, excludes p from the result, then returns.

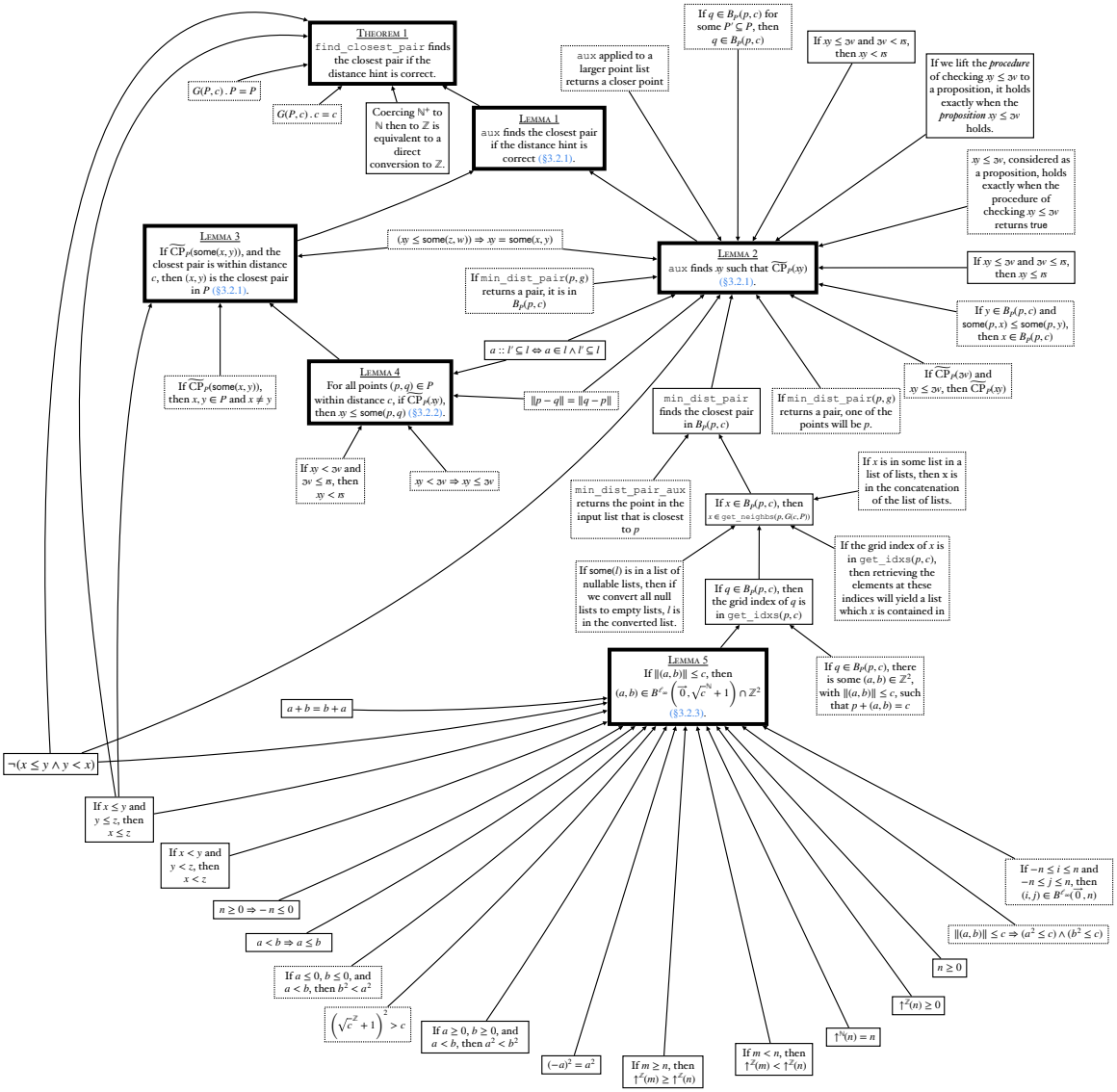

Figure 2: Graph capturing the dependency of each lemma/theorem on other lemmas. An edge exists from a to b when a is used by b (e.g., a lemma with high in-degree uses many lemmas). A dashed border means the lemma has not been proven. A bold border means the lemma is important/interesting/difficult; we cover some of them in Section 3.2.

3.1 General Strategy

Starting with Theorem 1, our general approach was:

1. Work through the proof.
2. Each time it seems you need a lemma pertaining to another function, create an empty definition for that lemma, assuming it to be true. The lemma can now be used in the main proof.
3. When the proof is finished, choose one of the posited lemmas and repeat this process.

This kind of top-down strategy is uniquely attuned to the realm of proofs, because we only care about *whether* a lemma is true—not *how* it's true. In programming, one can stub out methods, but ultimately, to test the program, method stubs need bodies.

3.2 War Stories

In this section, we cover proofs which were either conceptually interesting or highlight some of the difficulties of working with Lean.

3.2.1 `aux` Finds the Closest Pair (in the Union of Balls)

As mentioned in Section **??**, `aux` is a recursive function that performs the core iteration over the list of points P—`find_closest_pair` simply creates the grid and filters the result of `aux` if it's not within distance c. Thus, the most important property to prove is that `aux` finds the closest pair, given that the distance hint holds true.

Lemma 1: `aux` Finds Closest Pair

If the closest pair is within distance $(1, c]$, `aux` finds (x, y) such that $\mathrm{CP}_P(x, y)$.

When you prove facts about a recursive function, you often prove them by induction. If we want to prove our algorithm finds the closest pair, a naive induction on P would have us prove the following in the induction step:

> Suppose we know the recursive subcall on the sublist P' gives us the closest pair in P'.
> Then the current call gives us the closest pair for the larger list $P = p :: P'$.

But the antecedent doesn't hold for our algorithm. In particular, the closest pair might contain points we haven't recursed over yet. We can see the issue in step 3 of Figure 3 (right). The closest pair at this step includes a point outside the list of processed points, identifying a case where the naive invariant fails.

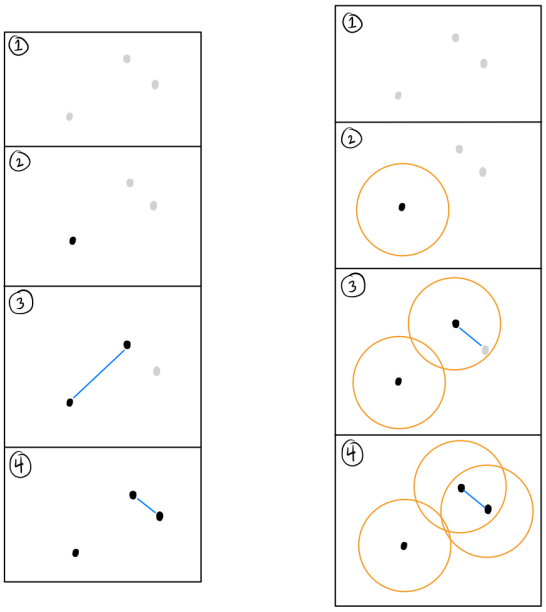

Figure 3: Depiction of the pairs satisfying the inductive hypothesis at each step of the algorithm, for both the naive and correct invariants. Black dots represent points in $P' \subseteq P$ that have been processed and gray dots represent unprocessed points in P. **(Left)** Pairs satisfying the naive induction invariant, expressing that the algorithm maintains the closest pair in P'. **(Right)** Pairs satisfying the correct induction invariant, expressing that the algorithm maintains the closest pair in $\bigcup_{p \in P'} B_P(p, c)$.

We need a proof goal that provides us with a different, stronger induction hypothesis. Our algorithm, conceptually speaking, draws balls around each point and finds the closest pair among all balls seen so far. To capture this process in a predicate, we create a recursively defined predicate $\widetilde{\mathrm{CP}}_P(xy)$ satisfying the following rules:

- **No points:** $\widetilde{\text{CP}}_P(\mathsf{none}, [])$
- **Recursive point closer:** If $\widetilde{\text{CP}}_P(xy, P')$ and xy is closer than all points in $B_P(p, c)$, then $\widetilde{\text{CP}}_P(xy, p :: P')$.
- **Current point closer:** If $\widetilde{\text{CP}}_P(xy, P')$, but there is a point $q \in B_P(p, c)$ closer to p than any other point in $B_P(p, c)$, and (p, q) is closer than xy, then $\widetilde{\text{CP}}_P(\mathsf{some}(\mathsf{p}, \mathsf{q}), p :: P')$.

Now, instead of proving `aux` finds (x, y) such that $\text{CP}_P(x, y)$, we prove the following.

Lemma 2: `aux` Finds Closest Pair in Ball Union

`aux` finds (x, y) such that $\widetilde{\text{CP}}_P(\mathsf{some}(x, y))$.

This lemma provides the correct induction hypothesis, allowing the proof to go through.

With Lemma 2 proven, we still need to prove Lemma 1. To do so, we prove the lemma below, which combines Lemma 2 with the distance hint. For brevity, we omit discussion of the proof, though we discuss one of the lemmas it relies on in Section 3.2.2.

Lemma 3: ($\widetilde{\text{CP}}_P$ and Distance Hint) $\Rightarrow \text{CP}_P$

`aux` finds (x, y) such that $\widetilde{\text{CP}}_P(\mathsf{some}(x, y))$.

3.2.2 Closest Pair In Ball Union Closer Than All Points Within Distance c

In proving Lemma 3, we require the following lemma.

Lemma 4: $\widetilde{\text{CP}}_P$ Flattening

If $\widetilde{\text{CP}}_P(\mathsf{some}(x, y))$ and $q \in \bigcup_{p \in P} B_P(p, c)$, then (x, y) is closer than (p, q).

After staring for a bit, you might think this lemma should hold trivially[2]. After all, isn't this exactly what $\widetilde{\text{CP}}_P$ is meant to express? Indeed it is, but the proof of this fact is not automatic. The reason is that $\widetilde{\text{CP}}_P$ is more specific than the simpler statement of being the closest pair in $\bigcup_{p \in P} B_P(p, c)$, because it enforces a particular *process* by which the closest pair was arrived at. This process is recursive, so to arrive at the "flattened" proposition above, we need to "unroll" it via induction. But induction on what?

We may wish to induct on the list of points P, but this causes an issue in the case where the current point is closer (see 3.2.1). Since we're inducting on P, all other variables remain fixed in the induction hypothesis. So if we're trying to prove that (x, y) is closer than (p, q) and we want to apply the induction hypothesis, we need to provide a proof that (x, y) satisfies $\widetilde{\text{CP}}_{P'}(\mathsf{some}(x, y))$. However, in the case where the current point is closer, we have that some other point pair xy' was the closest pair in the ball union for the sublist P'. Thus, we only have $\widetilde{\text{CP}}_{P'}(xy')$, and we are unable to apply the IH.

The issue is that inducting on P leaves (x, y) fixed and does not account for the fact that the closest pair changes as we recurse. What else can we induct on then? Well, induction can be performed on any type satisfying certain conditions[3], and it so happens that our recursive predicate $\widetilde{\text{CP}}$ satisfies them. This induction target solves our problem because the inductive hypothesis changes depending on whether we are in the case where the recursive result remains the closest or a pair in the current point's ball is closer.

There are two takeaways from this proof:

[2] The reader may be thinking this of many results in this report... and they are not alone. Such is life in formal verification, it seems.

[3] In particular, we must be able to impose a ***well-founded order*** upon it.

- In verification, you often define predicates that are aligned with the recursion structure of an algorithm, for the reasons outlined in Section 3.2.1. However, this often leads to an abstraction mismatch, because these predicates don't directly encode high-level mathematical properties. Thus, it is common to prove lemmas that lift these low-level predicates to intuitive statements you can work with in other proofs.
- Induction on recursively defined predicates can yield stronger induction hypotheses or, at least, hypotheses that are more closely aligned with the proof goal.

3.2.3 Bounded Norm ⇒ In ℓ_∞ Ball

On the path to proving Lemma 2, we need lemmas showing that we check the correct grid indices. The lemma below says that if the distance between two points is bounded, then checking in a hypercube of a certain radius around one of them suffices to find one of the points.

Lemma 5: Hypercube Containment for Bounded Norm

If $\|(a,b)\| \le c$, then $(a,b) \in \texttt{get\_hypercube}(\sqrt{c}^{\mathbb{N}} + 1)$.

There's nothing conceptually tricky about this lemma. However, the approach that felt natural made it an absolute pain to prove in Lean.

We first broke down the proof goal with a lemma showing it suffices to prove

$$-\uparrow^{\mathbb{Z}}(\sqrt{c}^{\mathbb{N}} + 1) \le a, b \le \uparrow^{\mathbb{Z}}(\sqrt{c}^{\mathbb{N}} + 1).$$

Recall, $\uparrow^{\mathbb{Z}}(n)$ represents a coercion of $n \in \mathbb{N}$ into an integer. This coercion is a trivial inclusion, but even so, it obstructed the application of lemmas, because Lean doesn't know which properties the coercions preserve.

To get around these obstructions, we separately proved that relations like $<, \le, >$, and $\ge$ were preserved by coercion, applied those lemmas, then applied the previously obstructed lemma. The proofs of preservation were trivial, but requiring the separate proofs still added friction to the process.

We now focus on proving only $-\uparrow^{\mathbb{Z}}(\sqrt{c}^{\mathbb{N}} + 1) \le a$. The other cases are similar. We proceed by contradiction, giving us the hypothesis that $\neg(-\uparrow^{\mathbb{Z}}(\sqrt{c}^{\mathbb{N}} + 1) \le a)$. To get this into the more useful form $a < -\uparrow^{\mathbb{Z}}(\sqrt{c}^{\mathbb{N}} + 1)$ required 4 lines of code.

We then used lemmas to establish the following facts, with later facts potentially depending on earlier ones:

- $a^2 \le c$ and $b^2 \le c$
- $\sqrt{c}^{\mathbb{N}} + 1 \ge 0$
- $-(\uparrow^{\mathbb{Z}}(\uparrow^{\mathbb{N}}(\sqrt{c}^{\mathbb{N}}) + 1) \le 0$
- $a \le -(\sqrt{c}^{\mathbb{N}} + 1)$
- $-(\uparrow^{\mathbb{N}}(\sqrt{c}^{\mathbb{N}}) + 1) \cdot -(\uparrow^{\mathbb{N}}(\sqrt{c}^{\mathbb{N}}) + 1) < a^2$
- $(-a)^2 = a^2$
- $(\sqrt{c}^{\mathbb{N}} + 1)^2 > c$
- $\uparrow^{\mathbb{Z}}(\sqrt{c}^{\mathbb{N}}) < a^2$

With the final fact, we were able to prove a contradiction by using the lemma $\neg(x \le y \wedge y < x)$ applied to $a^2 \le c$ and $\uparrow^{\mathbb{Z}}(\sqrt{c}^{\mathbb{N}}) < a^2$.

Recall, we have only proven a portion of the original proof goal, and we stated the other three cases were similar. To prove the other cases, we copied the first case three times and made slight modifications. We did not sink much time into it, but in the time we did spend, it was nonobvious how to do anything better.

We have no doubts there are tactics that could make the proof above less painful. The takeaway from this story is that a clean proof did not seem attainable from *basic* usage of Lean.

Potential Improvements. There exists a `wlog` tactic that captures the informal concept of WLOG often used in proofs, by considering permutations of variables in the proof goal. This tactic could potentially reduce the code duplication across the four cases.

There exists a `calc` mode that is able to perform chains of inequality reasoning, which could dramatically condense the proof of Lemma 5. However, we were unable to get it to work in the simple cases we tried.

3.3 Admitted Lemmas

Almost all of the lemmas we did not prove were unique to the machinery we set up for the closest pair algorithm. Most of the admitted lemmas were simply due to time constraints. Below, we discuss a few notable classes of admitted lemmas.

Comparisons In $(P \times P)^?$. In defining the comparison predicates $xy \leq zw$ and $xy < zw$ between pairs of points, we did not automatically gain access to lemmas on linear orders. We believe if we proved that it forms a linear order, then we would gain access to a number of theorems in Mathlib that would be helpful in some of the admitted proofs.

Norm. We can't directly leverage the infrastructure and theorems surrounding normed spaces, because the norm typeclass requires a codomain of $\mathbb{R}$, destroying computability properties.

In principle, a norm instance could be provided whereby the output would be lifted from $\mathbb{Z}$ to $\mathbb{R}$. Then, perhaps we could transport theorems from normed spaces. This might work, but it could also introduce a lot of overhead, where we intersperse coercions between $\mathbb{Z}$ and $\mathbb{R}$ throughout our proofs.

3.4 Statistics

In this section, we compare lines of code in our implementation (Table 4) to lines of code in our verification (Table 5).

Function Name	Lines of Code
`find_closest_pair`	11
`aux`	7
`grid_points`	12
`get_grid_idx`	2
`min_dist_pair_aux`	5
`min_dist_pair`	3
`get_hypercube`	6
`get_idxs`	5
`lift_option_list`	3
`get_neighbs`	6
$xy \leq zw$	7
Total	67

Figure 4: Breakdown of lines of code for each function in our implementation.

Theorem/Lemma	Lines of Code
Theorem 1	81
Lemma 1	13
Lemma 2	199
Lemma 3	66
Lemma 4	71
Lemma 5	257
Other	446
Total	1,133

Figure 5: Breakdown of lines of code for the main results in our verification and a summary of other smaller results.

We have a startling 17× blowup in the size of our verification, relative to our implementation! Some of this blowup could be due to my being an experienced functional programmer but a novice in formal verification. As noted earlier, there are facilities in Lean that I am aware of and that could have drastically reduced the size of certain proofs, but I did not have time to investigate them.

4 Conclusion

While the existence of Mathlib alleviates the burden of proving basic facts about mathematical objects, the constructions necessary to prove correctness of algorithms are often as complex or more than constructions in mathematics, and they are far more bespoke. With all the diversity in algorithms across computer science, formal verification of a particular implementation of a particular algorithm is a tough sell.

We are led to the conclusion that, until a breakthrough in proof automation is achieved or we discover new abstractions for mathematical construction of software, the average computational geometer, and the average programmer, can continue to disregard formal verification. Despite my lack of expertise in formal verification, the broad strokes of the picture are clear: the level of formality that verification entails leads to reasoning far below the level of abstraction computer scientists care to think at.