Verification of Closest Pair of Points Algorithm

Aditi Goyal
aditi.goyal@epfl.ch

Suhas Shankar
suhas.shankar@epfl.ch

Yatharth Goswami
yatharth.goswami@epfl.ch

January 6, 2023

Abstract

The divide and conquer algorithm for finding the closest pair of points in 2 dimensions is verified using the Stainless framework. We used a fully functional implementation of the famous divide and conquer algorithm for the purpose of verification. [4] serves as our reference paper for this project. The time complexity of the verified implementation is estimated through experiments. The final presentation can be found here [6]. The implementation can be found on GitHub here [5].

1 Introduction

In this report, we verify a purely functional implementation of the divide and conquer algorithm to solve the "Closest pair of points" problem in 2-dimensional space using Stainless [3]. This is one of the most fundamental problems in computational geometry and appears as a step in many algorithms. It 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. We take inspiration from the paper [4], which verifies a divide and conquer algorithm, the basis of which comes from the book [1]. After providing the preliminary definitions, we move on to a discussion of the verification of the algorithm. Next, we provide empirical evidence of time complexity bound in the form of benchmarking experiments in 4. We also list the times we get on the verified implementation which could be used further to compare with previous works and get the overhead (if any) for the verified functional implementation in 4. For the purpose of this report, reference to *distance* between points refers to the square of the Euclidean distance, *x-coordinate-distance* (*y-coordinate-distance*) refers to the square of the difference in $x - coordinates$ ($y - coordinates$).

2 Preliminaries

2.1 Stainless Verifier

Stainless [3] is a program verification tool which can be used for a subset of the Scala programming language. Similar to some other program verifiers like Dafny [2], a programmer can embed the specifications required for a function within the function itself. It does this in the form of requires and ensures clauses. Here is an example program in Stainless:

```scala
  def max(x: Int, y: Int): Int = {
     require(0 <= x && 0 <= y)
     val d = x - y
     if (d > 0) x
     else y
  } ensuring (res =>
     x <= res && y <= res && (res == x || res == y))
```

We will try to explain a bit of the concept behind Stainless using the above example. This is an example implementation of a max function. Stainless allows you to write the preconditions and postconditions that you want to be met. The above function takes two numbers x and y, and requires those to be non-negative. In the end, the function will ensure that the returned output is amongst x and y, and is greater than or equal to both of them, ensuring the functionality of the max of two numbers. Stainless verifies that the final postconditions will follow under the set of preconditions given, followed by executing the given piece of code. In addition, it will also prove that the function terminates.

From the above example, some benefits of using Stainless are clear.

- Firstly, most other theorem provers do not work with Scala programming language, which benefits from it's strong type system and ensures memory safety. Also, Scala allows programmers to write a higher level code than C, which is used by most other theorem provers.
- Secondly, it allows the specification of the functions to be written using Scala itself. Also, it allows embedding these inside the function itself, which allows for better readability and easier use.

2.2 Delta point sparsity

A set of points P is δ-point-sparse with respect to a point p_0 if the distance between p_0 and any point in P which is not same as p_0 is $\geq \delta$. Formally,

$$\mathit{sparse\_point}\,(\delta,\, p_0,\, P) = (\forall p_1 \in P \,.\, p_0 \neq p_1 \to \delta \leq d(\, p_0,\, p_1))$$

2.3 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,

$$\mathit{sparse}\,(\delta,\, P) = (\forall p_0 \in P \,.\, \forall p_1 \in P \,.\, p_0 \neq p_1 \to \delta \leq d(\, p_0,\, p_1))$$

In terms of point-sparsity, we can think of it as:

$$\mathit{sparse}\,(\delta,\, P) = (\forall p_0 \in P \,.\mathit{sparse\_point}\,(\delta,\, p_0,\, P))$$

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

$$p_0 \in P \wedge p_1 \in P \wedge p_0 \neq p_1 \wedge \mathit{sparse}\,(d(p_0, p_1)),\;\; P)$$

2.4 Algorithm overview

The algorithm is based on the divide and conquer paradigm. It starts by splitting the set of points into two sets by the median $x - coordinate$ (say l). Then, it recursively solves the problem for the two halves. Let δ_l (δ_r) denote the distance between the closest pair of points in the left (right) half. Now, we know that the closest pair of points will either lie on the left or right side, or there will be 1 point from the left and another 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 containing points having $x-$coordinate-distance of at most δ from $(l, 0)$ to find a better candidate for the closest points (see figure below).

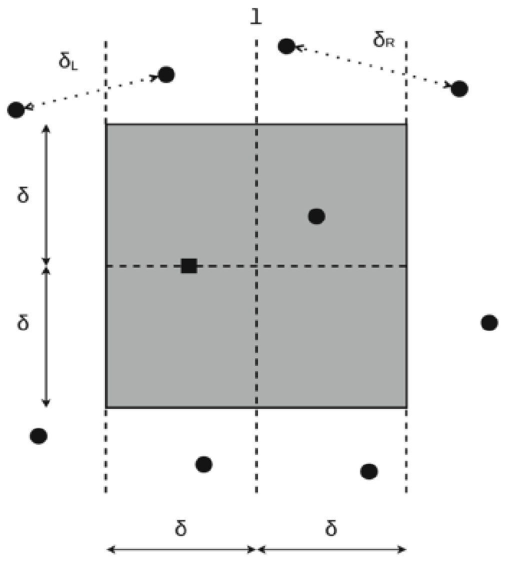


Figure 1: Combine step in the algorithm. Taken from [4]

2.5 Constraints and other details

For the purpose of verification, we have restricted ourselves to solving it in the domain of integer point coordinates (also referred to as BigInt in Scala). In the implementation, we have also omitted to take the square root for the computation of the Euclidean distance between two points since we only care about relative distances for finding the closest pair of points.

2.6 Previous Work

In [4], the authors have verified a fully functional implementation of the divide and conquer algorithm for finding the closest pair of points in a 2D plane using the Isabelle theorem prover. Our work builds up on their efforts, and we verified using the Stainless verifier. We used the reference paper[4] to get the basic structure for our implementation and the critical lemmas required to be proven. The authors have not provided how they proved those lemmas, so we came up with our own proofs, which we used for verification in Stainless.

The specifications of each function were also missing in the paper, so we developed our own set of preconditions and postconditions for each function.

3 Implementation and Functional Correctness

We will describe the implementation of the divide and conquer algorithm that we have verified along with the specifications that hold in terms of lemmas and theorems. As discussed in the overview 2.4, we solve for the 2 recursive cases by splitting points into 2 halves. Now, we discuss how we combine the solutions from these halves to obtain the solution for the original problem.

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 the distance between them (say δ) and a list of points (say l) which are sorted according to their $y-coordinates$. The combine step should return a pair of points having a distance $\delta' \leq \delta$ such that the list l is δ' sparse. For this, we start with the function $findClosestPointInStrip$.

3.1.1 findClosestPointInStrip

```
def findClosestPointInStrip(p: Point)(d: BigInt)(l: List[Point]):
   Point =
{
   require(!l.isEmpty && isSortedY(l) && p.y <= l.head.y)
   if l.tail.isEmpty then l.head
   else if d <= (l.head.y - p.y)*(l.head.y - p.y) then {
      ghost { transitiveDistanceProperty(p, d, l.head, l.tail) }
      l.head
   }
   else{
      val p1 = findClosestPointInStrip(p)(min(d,
         p.distance(l.head)))(l.tail)
      if p.distance(l.head) <= p.distance(p1) then l.head else p1
   }
}
.ensuring(res0 => deltaSparsePoint(min(p.distance(res0), d), p, l)
   && l.contains(res0))
```

This function takes in a point $p = (x, y)$ and a list of points (say l) (in the context of the algorithm, this contains points above p in the filtered strip), an integer number d (in context of the algorithm, δ). **It looks at points in the list having $y-$coordinate-distance less than d from p and returns the point closest to p amongst these. If no points are satisfying this condition, it returns the head of the list**. Though it looks like a function with linear time complexity, it is not! For our use case (delta-sparsity condition met on the left and right halves and l being sorted), only a constant number of points can have y-coordinate-distance less than d, refer to time complexity section in [7] for proof. Hence, this function runs in constant time complexity.

3.1.2 findClosestPairInStrip

```
def findClosestPairInStrip(x: PairPoint)(@induct l:
   List[Point]):PairPoint =
{
   require(isSortedY(l))
   if l.isEmpty || l.tail.isEmpty then x
   else {
      val p1 =
         findClosestPointInStrip(l.head)(pairDistance(x))(l.tail)
      assert(deltaSparsePoint(min(p1.distance(l.head),
         pairDistance(x)), l.head, l.tail))
      if pairDistance(x) <= p1.distance(l.head) then{
         val z = findClosestPairInStrip(x)(l.tail)
         ghost {
            reducingDeltaPreservesPointSparsity(pairDistance(x),
            pairDistance(z), l.head, l.tail) }
         z
      }
      else {
         val z = findClosestPairInStrip((l.head, p1))(l.tail)
         ghost {
            reducingDeltaPreservesPointSparsity(l.head.distance(p1),
            pairDistance(z), l.head, l.tail) }
         z
      }
   }
}
.ensuring(res0 => deltaSparse(pairDistance(res0), l) &&
   pairDistance(res0) <= pairDistance(x) && (res0 == x ||
   (l.contains(res0._1) && l.contains(res0._2))))
```

This function takes in a pair of points x (in the context of the algorithm, this is the closest pair of points found till now; initially it is the pair having the lesser distance from the two: closest pair in the left half and closest pair in the right half), and a list l (in context of the algorithm, this is the filtered strip sorted by $y - coordinates$). It traverses l from start to end, and for each point p, it checks for possible candidates for the closest pair with p. Here, we check all eligible candidates for the closest pair of points with one point as p, by calling findClosestPointInStrip (with distance d as the distance between the closest pair of points found till now and list as the points above p in the list l). Essentially, **this function finds the closest pair in the strip if this closest pair has a distance less than the distance between points in x, otherwise, it simply returns x.**

3.1.3 combine

```
def combine(lpoint: PairPoint)(rpoint: PairPoint)(div: BigInt)(l:
   List[Point]): PairPoint = {
   require(isSortedY(l) && l.contains(lpoint._1) &&
      l.contains(lpoint._2) && l.contains(rpoint._1) &&
      l.contains(rpoint._2))
   val z = compare(lpoint, rpoint)
   val d = pairDistance(z)
   val l2 = l.filter(p => p.distance(Point(div, p.y)) < d)
   ghost { filterSorted(l, p => p.distance(Point(div, p.y)) < d) }
   findClosestPairInStrip(z)(l2)
}.ensuring(res0 => deltaSparse(pairDistance(res0), l.filter(p =>
   p.distance(Point(div, p.y)) < pairDistance(compare(lpoint,
   rpoint)))) && pairDistance(res0) <=
   pairDistance(compare(lpoint, rpoint)) && l.contains(res0._1) &&
   l.contains(res0._2))
```

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, the median $x - coordinate$ and the original list sorted by $y - coordinates$. **It returns the closest pair of points in** l. It first calculates d 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 *findClosestPairInStrip* by filtering points in l based on their x-coordinate-distance from the median $x - coordinate$ (div)

3.1.4 Correctness

The following lemmas are used to verify the correctness of the combine step.

Lemma 3.1 $ps.contains(p_0) \land ps.contains(p_1) \land p_0 \neq p_1 \land d(p_0, p_1) < \delta \land$
$(\forall p \in P_L \,.\, p.x \leq l) \land sparse(\delta, P_L) \land$
$(\forall p \in P_R \,.\, p.x \geq l) \land sparse(\delta, P_R) \land$
$ps.content = (P_L + +P_R).content \land ps' = ps.filter((p) => d(p, (l, p.y)) < \delta)$
$\implies ps'.contains(p_0) \land ps'.contains(p_1)$

Lemma 3.1 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 filtered strip described before or in other words, **looking within the strip instead of the whole list suffices for the combine step**.
Now, we discuss how we proved this lemma in Stainless. Let's assume $p_0 \in P_L$. From this assumption, we have that $p_1 \in P_R$ because both p_0 and p_1 can't be in P_L as P_L is δ-sparse and the distance between p_0 and p_1 is less than δ. Now, let's assume that $p_0 \notin ps'$. The distance of any point in P_R from p_0 has to be at least δ because the x-coordinate-distance of p_0 from vertical line at median $x - coordiante$ was at least δ. This would result in a contradiction because p_1 was also in P_R. Hence, $p_0 \in ps'$. Similarly, we can conclude $p_1 \in ps'$. Hence, if $p_0 \in P_L$ then $p_0 \in ps'$ and $p_1 \in ps'$. Similar is the case when $p_0 \in P_R$.

Lemma 3.2 $sorted\_y(ps) \land ps.content = (P_L{+}{+}P_R).content \land$
$(\forall p \,.\, P_L.contains(p) \implies p.x \leq l) \land sparse(d(p_{0L}, p_{1L}), P_L) \land$

$(\forall p \,.\, P_R.contains(p) \implies p.x > l) \land sparse(d(p_{0R}, p_{1R}), P_R) \land$
$(p_0, p_1) = combine((p_{0L}, p_{1L}), (p_{0R}, p_{1R}), l, ps)$
$\implies sparse((d(p_0, p_1), ps))$

Lemma 3.2 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 pair of points in ps. **This is correctness of combine function**. Let's move on to verification of this lemma. This lemma can be proved by contradiction. Assume that ps is not δ-sparse where δ is the distance between pair of points returned by the function. Then, we can get a pair of points from the list having a distance less than the distance between returned points. On these points, we can apply lemma 3.1 to conclude that these points must lie within the strip. From the ensuring clause of *combine*, we have that the strip was in fact δ-sparse where δ is the distance between returned points. This leads to a contradiction; thus, our assumption was wrong (see Theorems.scala in [5]).

3.2 Divide and conquer algorithm

The divide step of the algorithm divides the points based on their $x - coordinates$ into two halves and solves the problem recursively on these halves, calling combine step afterwards.

3.2.1 findClosestPairRec

```
def findClosestPairRec(l: List[Point]): (List[Point], PairPoint) =
   {
   require(l.size >= 2 && isSortedX(l))
   decreases(l.size)
   if l.size <= 3 then bruteForce(l)
   else{
      val (left_half, right_half) = l.splitAtIndex(l.size/2)
      ghost { split(l, l.size/2) }
      val (lsorted, lpoint) = findClosestPairRec(left_half)
      val (rsorted, rpoint) = findClosestPairRec(right_half)
      val sortedList = mergeY(lsorted, rsorted)
      val res =
         combine(lpoint)(rpoint)(right_half.head.x)(sortedList)
      ghost {
         combineLemma(sortedList, left_half, right_half,
            right_half.head.x, lpoint, rpoint, res)
         subsetPreservesDeltaSparsity(pairDistance(res),
            sortedList, l)
      }
      (sortedList, res)
   }
}.ensuring(res0 => res0._1.content == l.content &&
   isSortedY(res0._1) && deltaSparse(pairDistance(res0._2), l) &&
   l.contains(res0._2._1) && l.contains(res0._2._2))
```

This function 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 the input list. It first calls itself recursively by splitting the original list into two 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 $y - coordinates$.

3.2.2 findClosestPair

```scala
  def findClosestPair(l: List[Point]): PairPoint = {
     require(l.size >= 2)
     val p = findClosestPairRec(mergeSortX(l))._2
     ghost { subsetPreservesDeltaSparsity(pairDistance(p),
        mergeSortX(l), l) }
     p
  }.ensuring(res0 => deltaSparse(pairDistance(res0), l) &&
     l.contains(res0._1) && l.contains(res0._2))
```

This function takes as input a list of points and returns the closest pair of points from the list. It does so by first sorting the points by their $x - coordinates$ and then calling *findClosestPairRec*.

3.2.3 Correctness

The following theorems ensure the correctness of *findClosestPair*. These are necessary and sufficient for proving correctness of the algorithm.

Theorem 3.3 $1 < ps.length \land (p_0, p_1) = findClosestPair(ps) \implies sparse(d(p_0, p_1), ps)$

Informally, this means the distance between the returned pair of points must be atleast as much as the distance between any 2 points in the list ps. This follows from correctness of the *combine* function.

Theorem 3.4 $1 < ps.length \land (p_0, p_1) = findClosestPair(ps) \implies ps.contains(p_0) \land ps.contains(p_1)$

This means that the returned points must be present in the original list ps.

Theorem 3.5 $1 < ps.length \land ps.distinct \land (p_0, p_1) = findClosestPair(ps) \implies p_0 \neq p_1$

Given a list of distinct points as input, the points returned should also be distinct! To prove this, we had to write lemmas to ensure properties for each function given a distinct list: given a distinct list, *findClosestPairInStrip* returns distinct pair of points, *findClosestPairRec* returns distinct pair of points, *mergeSortX* returns a distinct list and so on. Some of these can be seen below. For example - The first one proves that $findClosestPairInStrip$ returns distinct points, if given a list with distinct points.

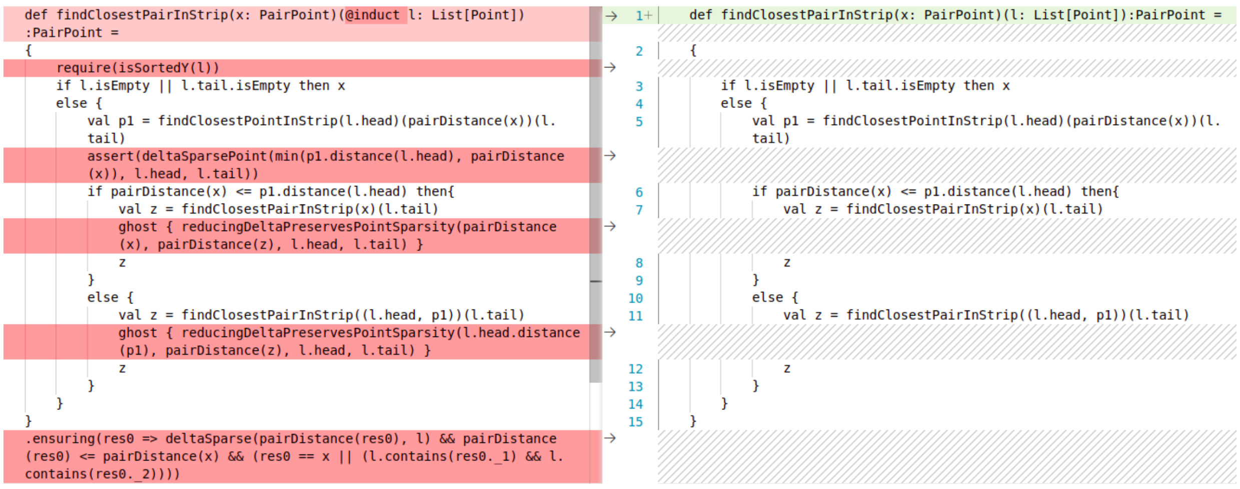
```
def closestPairDistinctLemma(x: PairPoint, l: List[Point], res:
   PairPoint): Unit = {
   require(isDistinct(l) && isSortedY(l) &&
      findClosestPairInStrip(x)(l) == res && x._1 != x._2)
   if(!l.isEmpty && !l.tail.isEmpty){
      val p1 =
         findClosestPointInStrip(l.head)(pairDistance(x))(l.tail)
      closestPointDistinctLemma(l.head, pairDistance(x), l.tail,
         p1)
      if(pairDistance(x) <= p1.distance(l.head)){
         val z = findClosestPairInStrip(x)(l.tail)
         closestPairDistinctLemma(x, l.tail, z)
      }
      else{
         val z = findClosestPairInStrip((l.head, p1))(l.tail)
         closestPairDistinctLemma((l.head, p1), l.tail, z)
      }
   }

}.ensuring(_ => res._1 != res._2)
```

The below function proves the distinctness properties for combine function.

```
def combineDistinctLemma(lpoint: PairPoint, rpoint: PairPoint,
   div: BigInt, l: List[Point], res: PairPoint): Unit = {
   require(lpoint._1 != lpoint._2 && rpoint._1 != rpoint._2 &&
      isDistinct(l) && isSortedY(l) && l.contains(lpoint._1) &&
      l.contains(lpoint._2) && l.contains(rpoint._1) &&
      l.contains(rpoint._2) &&res ==
      combine(lpoint)(rpoint)(div)(l))
   val z = compare(lpoint, rpoint)
   val d = pairDistance(z)
   val l2 = l.filter(p => p.distance(Point(div, p.y)) < d)
   filterSorted(l, p => p.distance(Point(div, p.y)) < d)
   filteringPreservesDistinct(l, p => p.distance(Point(div, p.y))
      < d)
   closestPairDistinctLemma(z, l2, res)
}.ensuring(_ => res._1 != res._2)
```

4 Benchmarks

In this section, we provide details about the methodology we employed to benchmark our verified implementation along with the time taken to run the algorithm for varying number of points. Since we have ghost annotations inside our Stainless implementation, we directly ran it through some tests after compiling with the Stainless compiler. We noticed that for a larger number of points, we were getting StackOverflow errors owing to the Stainless lists we had used in our code. We realised that most of the Stainless List functions like size and filter were not tail recursive and hence failed for larger-sized inputs. Owing to limited time, we thought of running the implementation through the Scala compiler which would have used more efficient Scala Lists. We simply removed the stainless constructs like requires, ensures, ghost annotations and assert statements from our code and then used this program for the final benchmarking. An example is shown below



```
def findClosestPairInStrip(x: PairPoint)(@induct l: List[Point])
:PairPoint =
{
    require(isSortedY(l))
    if l.isEmpty || l.tail.isEmpty then x
    else {
        val p1 = findClosestPointInStrip(l.head)(pairDistance(x))(l.
        tail)
        assert(deltaSparsePoint(min(p1.distance(l.head), pairDistance
        (x)), l.head, l.tail))
        if pairDistance(x) <= p1.distance(l.head) then{
            val z = findClosestPairInStrip(x)(l.tail)
            ghost { reducingDeltaPreservesPointSparsity(pairDistance
            (x), pairDistance(z), l.head, l.tail) }
            z
        }
        else {
            val z = findClosestPairInStrip((l.head, p1))(l.tail)
            ghost { reducingDeltaPreservesPointSparsity(l.head.distance
            (p1), pairDistance(z), l.head, l.tail) }
            z
        }
    }
}
.ensuring(res0 => deltaSparse(pairDistance(res0), l) && pairDistance
(res0) <= pairDistance(x) && (res0 == x || (l.contains(res0._1) && l.
contains(res0._2))))
```

```
def findClosestPairInStrip(x: PairPoint)(l: List[Point]):PairPoint =
{
    if l.isEmpty || l.tail.isEmpty then x
    else {
        val p1 = findClosestPointInStrip(l.head)(pairDistance(x))(l.
        tail)
        if pairDistance(x) <= p1.distance(l.head) then{
            val z = findClosestPairInStrip(x)(l.tail)
            z
        }
        else {
            val z = findClosestPairInStrip((l.head, p1))(l.tail)
            z
        }
    }
}
```

Figure 2: Difference between verified and benchmarked implementation

The figure below provides empirical evidence of the *nlogn* time complexity of our implementation.

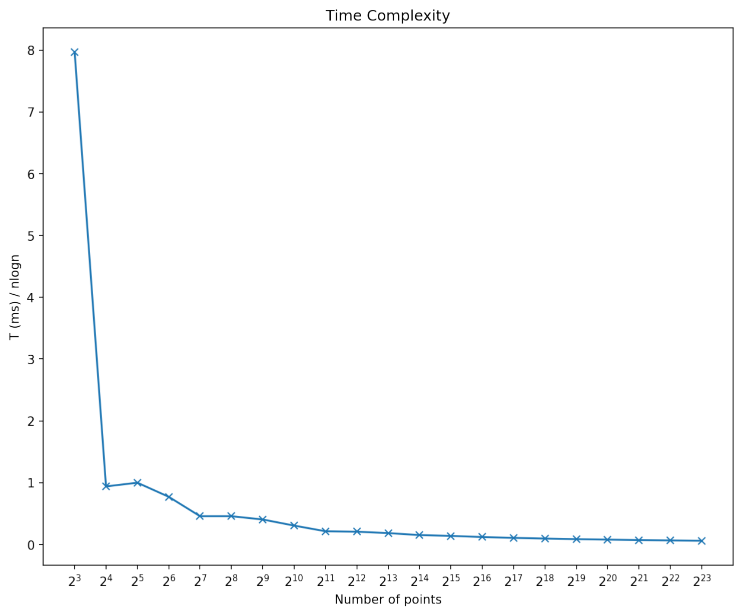


Figure 3: $\frac{T(\text{in ms})}{n \log n}$

The absolute running times are as follows:

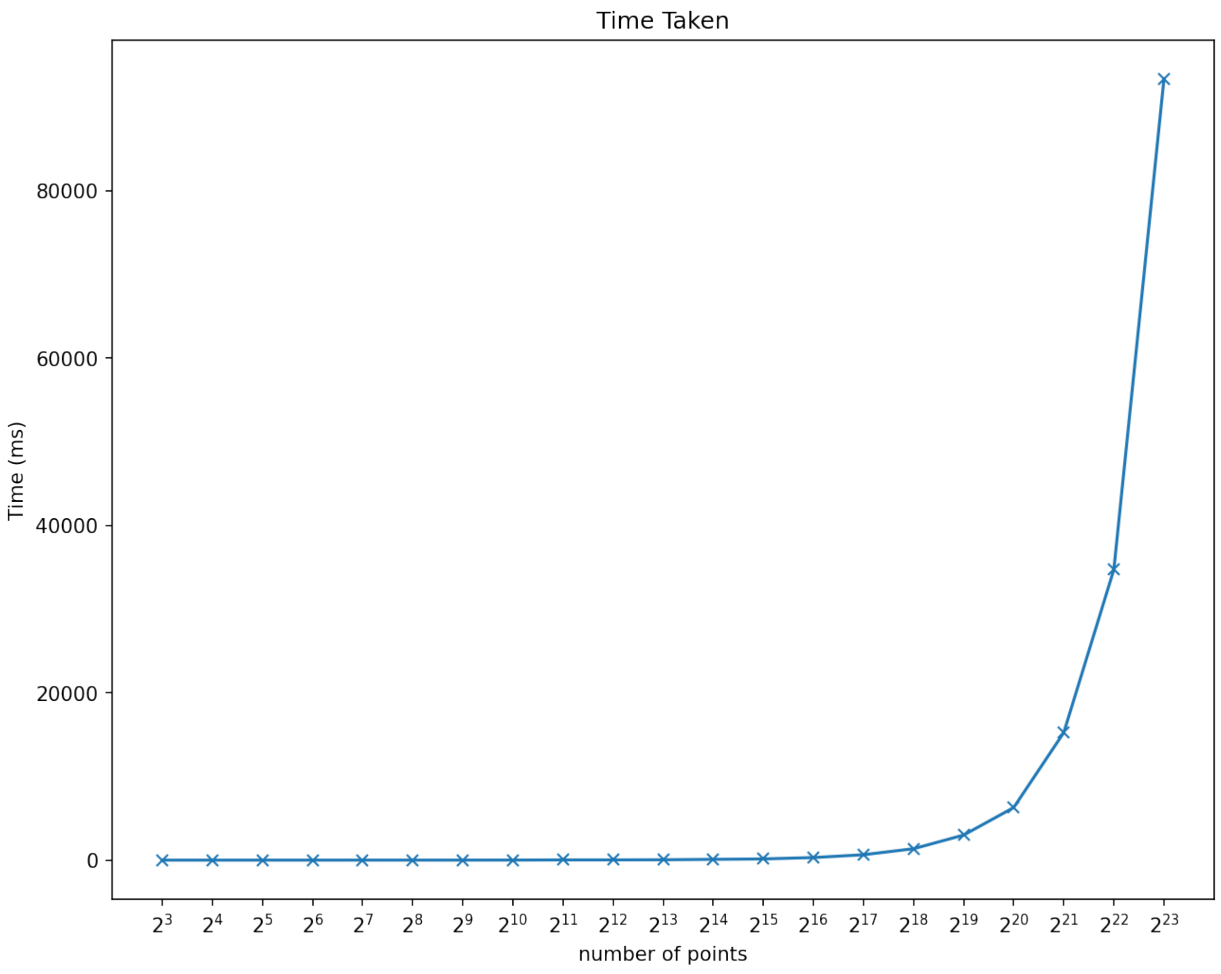


Figure 4: Time taken vs number of points

5 Conclusion

This report provides the formal verification of a functional implementation of the algorithm to solve the "Closest pair of points" in 2 dimensions using Stainless. We feel that it is one of the first attempt of verification of a geometric algorithm using Stainless. It describes in detail, the lemmas used and their corresponding proofs in Stainless. It also provides details about the runtimes of the algorithm in Scala as well as the time taken to verify the implementation in Stainless.
An extension of the report 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 have not yet been studied. Another future task can be formally verifying the time complexity of the implementation in Stainless.

References

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

[2] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning.* LPAR'10. Dakar, Senegal: Springer-Verlag, 2010, pp. 348–370. ISBN: 3642175104.

[3] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. "System FR: Formalized Foundations for the Stainless Verifier". In: *Proc. ACM Program. Lang* OOPSLA (Nov. 2019). DOI: `https://doi.org/10.1145/3360592`.

[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.

[5] Yatharth Goswami Aditi Goyal Suhas Shankar. *CS550-Formal-Verification-Project.* URL: `https://github.com/aditi-goyal-257/CS550-Formal-Verification-Project`.

[6] Yatharth Goswami Aditi Goyal Suhas Shankar. *CS550-Formal-Verification-Project-Presentation.* URL: `https://docs.google.com/presentation/d/1Y_OateO78_VOEX0brfk0SBsZbBsbPrLylhMzajE6aao/edit?usp=sharing`.

[7] Yatharth Goswami Aditi Goyal Suhas Shankar. *Review: Verification of Closest Pair of Point Algorithms.* URL: `https://github.com/aditi-goyal-257/CS550-Formal-Verification-Project/blob/main/CS550_Paper_Review.pdf`.