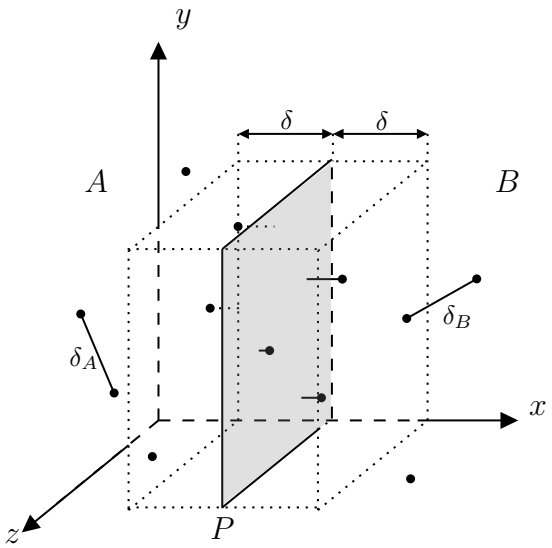
Exploring Closest Points Algorithms: Complexity, Practical Implementations, and Benchmarking

Nærmeste Punkter Algoritmer: Kompleksitet, Praktiske Implementeringer og Benchmarking

Authors:
Adam Yasser Tallouzi, 202004765
Jacob Lind, 201304090

Advisor:
Peyman Afshani



Bachelor report (15 ECTS) in Computer Science
Department of Computer Science
Aarhus University
June 8, 2023

Abstract

This report investigates the closest-points problem, based on the development by Bentley and Shamos [1976]. In their paper, they initially introduce a 2D algorithm as a foundation and subsequently build upon it to achieve an $O(n \log n)$ algorithm in k-dimensions. The objective of this report is to closely examine and replicate these advancements by implementing the algorithms in Python and conducting experiments to evaluate their practical performance.

The implementation of the algorithm on the plane finds a practical time complexity that aligns with its theoretical worst-case time complexity. However, when generalizing the algorithm to k-dimensions, a slower running time than the theoretically expected time is achieved.

The report discusses these findings and gives possible explanations as to why the practical experiments deviate from the claims by Bentley and Shamos [1976].

Contents

1 Introduction

The closest points problem is a well-known computational geometry problem that has a wide range of applications in various fields. The problem is as follows: Given a set of n points P in $\mathbb{R}^k$, find the two points $p, q \in P$ such that the Euclidean distance between them is the smallest. A closely related problem is the fixed-radius near-neighbor problem, which, given a fixed distance δ asks for all pairs of points within δ of one another. For the sake of analysis, we will focus on a specific variant of this problem called the sparse fixed-radius near-neighbor problem, in which it is guaranteed that no hypercube of side 2δ in the space contains more than a constant $c \geq 1$ number of points. This condition is referred to as sparsity.

The closest points problem can clearly be solved by investigating all pairs of points and returning the pair with the smallest distance, but this method has time complexity $\Theta(n^2)$. Previous work by Shamos and Hoey [1975] shows that the problem can be solved on the plane in $O(n \log n)$ time by utilizing techniques involving the application of Voronoi structures. Later algorithms by Bentley and Shamos [1976] demonstrate that the problem can be solved in $O(n \log n)$ time both on the plane and in k-space. In their paper, they initially introduce a divide-and-conquer algorithm to solve the problem on the plane in $O(n \log n)$ time, which they subsequently build upon to achieve an $O(n \log n)$ algorithm in k-space. To validate their findings, we implement these algorithms in Python and conducted tests to confirm that their theoretical time complexity matches their performance in practice. The first algorithm finds the closest pair on the plane in $O(n \log n)$ time. This algorithm is then generalized to k-space, yielding an $O(n \log^{k-1} n)$ algorithm. Special properties of the problem allow for optimizations on the second algorithm, yielding a final $O(n \log n)$ algorithm.

We denote $P(n, k)$ as the worst-case time complexity of the algorithm by Bentley and Shamos [1976] for solving the problem with n points in k dimensions. We also let $S(n, k)$ denote the worst-case time complexity for their algorithm to solve the sparse fixed-radius near-neighbor problem.

The following sections will present a detailed overview of the algorithms by Bentley and Shamos [1976], discuss the methodology employed in our implementation and experiments, and present the results obtained.

2 Closest points on the plane

The initial algorithm by Bentley and Shamos [1976] uses a divide-and-conquer technique where the problem is decomposed into two subproblems with $n/2$ points and one subproblem with n points in on a line. The special structure of the subproblems allows the algorithm to run in $O(n \log n)$ time, i.e. $P(n, 2) = O(n \log n)$.

2.1 Algorithm Description

The algorithm works by dividing the points by a vertical line L, such that $n/2$ points lie on either side of L, as seen in Figure 1. We denote the leftmost pointset consisting of $n/2$ points by A, and similarly the rightmost pointset by B. The

algorithm is recursively used on both A and B, yielding $(p, q)_A$, the closest pair in A with distance δ_A, and $(p, q)_B$, the closest pair in B with δ_B. Let $\delta = \min(\delta_A, \delta_B)$. To find the pair with the *smallest* distance δ_{AB} for the pointset $A \cup B$, consider the vertical slab that extends δ distance from L on either side, as seen in Figure 1. If any pair of points $(p, q) \in A \times B$ has a distance closer than δ between them, they must lie within this vertical slab.

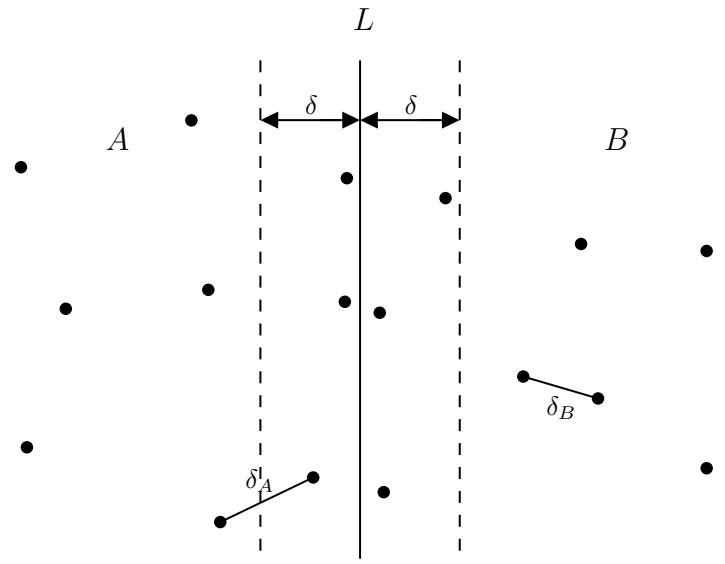


Figure 1: A rough illustration of how the algorithm works

Consider any square of sidelength 2δ centered around L, as seen on Figure 2. The square can contain at most a constant $c = 12$ amount of points (or $c = 9$ points if we disallow overlapping points), since each pair of points on either side of L can at minimum be δ distance apart.

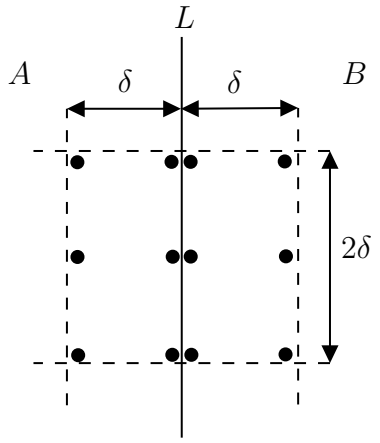


Figure 2: Square of sidelength 2δ centered around L

The points within the slab are projected onto L, such that only their y-coordinates are considered, reducing the problem down to a single dimension. The projection cannot increase the distance between the points. It follows that within any interval

of length 2δ along L, there can be at most 12 points (with 6 overlapping points in the middle). To find a pair closer than δ apart, investigate every point (x_i, y_i) within the slab and check the distance to every other point (x_j, y_j) that satisfies $|y_i - y_j| < \delta$. Since there are at most $12n$ pairs within δ in the projection, this check can be done in linear time if the points are checked in an increasing y-sorted order.

Since the algorithm reduces a problem into two subproblems on the plane, each of size $n/2$, and a subproblem on the line, which is solved in $O(n)$ time, we obtain the following recurrence

$$P(n, 2) = 2P(n/2, 2) + O(n)$$

which solves to $P(n, 2) = O(n \log n)$ by the master theorem.

2.1.1 Base Cases

In the above formulation of the algorithm, we disregard the base cases in favor of conciseness. For completeness, they are described here.

The two valid base cases are the two-point base case and the three-point base case. A single-point base case is not valid since we expect at least a *pair* of points. The two-point base case is trivial in that the closest pair is simply the only two points in the set. In the three-point base case, we cannot do another split since this would result in an invalid single-point base case. Instead, the closest pair is simply the pair with the minimum distance between the three points. Since both base cases contain a constant number of points, a constant number of distance calculations are performed.

2.2 Implementation

This section details our implementation of the algorithm, highlighting some of the challenges we encountered and the approaches we took to address them. We also characterize the algorithm's time complexity based on the source code. We refer to files that can be found in the Appendix throughout.

2.2.1 The index problem

One challenge in implementing the algorithm is that in order to do the linear time scan in the projection onto L, it is necessary to have the points sorted by both x-coordinate and y-coordinate at each step in the recursion. A straightforward approach of splitting the array sorted by x-coordinate during recursion and subsequently sorting by y at each step would result in a time complexity of $O(n \log^2 n)$. Therefore, in order to achieve the $O(n \log n)$ time complexity, it is necessary to maintain two separate arrays, each containing the points, with one array sorted by increasing x-coordinates and the other by increasing y-coordinates. Consequently, whenever the points are split along the x-axis, we must ensure that the points in one half of the array sorted by x-coordinates are also present in the corresponding array sorted by y-coordinates.

Our solution to this problem involves "index pointers". Essentially, each point in the x-sorted array is appended with the index of the same point in the y-sorted

array and vice versa. This allows us to implement a $O(n)$ function to split the arrays into two halves, such that their sorted order is maintained and that each resulting half contains the same points in both the x-sorted and y-sorted arrays.

With the above in mind, the algorithm now takes as input two arrays containing the points: `xsort`, which is sorted by increasing x-value, and `ysort`, which is sorted by increasing y-value. More specifically, the arrays contain 3-tuples with the following format:

$$\texttt{xsort} = [(x, y, y_{ptr}), ...]$$
$$\texttt{ysort} = [(x, y, x_{ptr}), ...]$$

where x and y are the coordinates for the point, y_{ptr} is the index of the same point in `ysort`, and vice versa for x_{ptr}. These arrays are set up in the preprocessing step of our implementation. A consequence of our choice of using index pointers is that overlapping points are tricky to index correctly in the preprocessing step and we, therefore, choose to explicitly disallow such points. This also reduces the maximum number of points contained within any square of sidelength 2δ to $c = 9$ (Figure 2).

2.2.2 Distance calculation

In this implementation we use the squared Euclidean distance, which for two points $p \in \mathbb{R}^2$ and $q \in \mathbb{R}^2$ is defined by

$$d^2(p, q) = (p_1 - q_1)^2 + (p_2 - q_2)^2.$$

Notice that we omit the square root calculation, which is normally used in the standard Euclidean distance. This is more convenient and slightly speeds up the calculation, but more importantly, it does not change the results when it comes to comparing distances. We must however be careful whenever we compare some smallest distance δ between a pair against some axis-wise distance. Since δ is the squared Euclidean distance and not the true distance, we must compare against $\sqrt{\delta}$ in these cases. We implement the distance calculation in Figure 3.

```
def sq_distance(p, q):
    x_diff = p[0] - q[0]
    y_diff = p[1] - q[1]
    return x_diff * x_diff + y_diff * y_diff
```

Figure 3: Euclidean Distance calculation

Bentley and Shamos [1976] use the Chebyshev Distance defined by

$$d_\infty(p, q) = \max(|p_1 - q_1|, |p_2 - q_2|),$$

but as they point out, the use of any other distance calculation only changes the time complexity by some multiplicative constant and does not affect the theoretical time complexity.

2.2.3 Code

We will refer to the pointsets A and B as `l` and `r` respectively in the code. Consider the algorithm in Figure 5. Lines 3-14 check if we are in a base case. If so, we return the closest pair by calculating the distance using the function in Figure 3.
If a base case is not yet reached, we split `xsort` and `ysort`, making sure the same points are in the two new arrays for the left pointset, `xl` and `yl`, as well as for the right pointset, `xr` and `yr` on line 17. The function is then called recursively on the left and right pointsets, yielding the pair of closest points in the left half: `pl`, `ql`, and similarly the right half: `pr`, `qr`. We let `delta` be the minimum distance between them and `best_pair` be the pair that is closest.

Consider the split function in Figure 4. When n is even, both arrays of points `xl` and `xr` will be exactly of size $n/2$. When n is odd, the extra point will belong to `xr` since `mid` is always rounded down due to the integer division (`//`), meaning `xl` will be even and `xr` will be odd. This is also taken care of when using the list slice operator `[n:m]`, where the index `n` is included and the index `m` is excluded.

```
1  def split(xsort, ysort, mid):
2    xl = xsort[:mid]
3    xr = xsort[mid:]
4    temp_yl = [None] * len(ysort)
5    temp_yr = [None] * len(ysort)
6
7    for i,(x,y,j) in enumerate(xl):
8      temp_yl[j] = (x,y,i)
9
10   for i,(x,y,j) in enumerate(xr):
11     temp_yr[j] = (x,y,i)
12
13   yl = [p for p in temp_yl if p is not None]
14   yr = [p for p in temp_yr if p is not None]
15
16   for i,(x,y,j) in enumerate(yl):
17     xl[j] = (x,y,i)
18
19   for i,(x,y,j) in enumerate(yr):
20     xr[j] = (x,y,i)
21
22   return xl, yl, xr, yr
```

Figure 4: Helper function for splitting the x- and y-sorted arrays while preserving sorted order

After `xsort` has been split in lines 2-3, two arrays are created of the same length as the original arrays with `None` values (Python equivalent of `null`). The entries from the new left x-sorted array `xl` are then added to `temp_yl`, at the same indices they have in `ysort`, using the y_{ptr} index pointers in lines 7-8. At the same time, Python enumeration allows for updating the x_{ptr} index pointers to point to the correct indices in `xl`. Some of the entries of `temp_yl` will now be `None`, and we can simply scan `temp_yl` for entries that are not `None` in line 13 and add them to `yl`,

effectively removing the `None` entries. This is repeated for the right pointset.

Since the y_{ptr} index pointers in `xl` and `xr` still have the indices of `ysort`, they are in lines 16-20 updated to be the correct indices in `yl` and `yr`, respectively. With this method, we avoid having to sort the points by increasing y-coordinate after projecting the points, which would significantly increase the running time.

```
1  def closest_points(xsort, ysort):
2      n = len(xsort)
3      if n == 2:
4          return tuple(xsort)
5      if n == 3:
6          d01 = distance(xsort[0], xsort[1])
7          d12 = distance(xsort[1], xsort[2])
8          d20 = distance(xsort[2], xsort[0])
9          if d01 <= d12 and d01 <= d20:
10             return xsort[0], xsort[1]
11         elif d12 <= d01 and d12 <= d20:
12             return xsort[1], xsort[2]
13         else:
14             return xsort[2], xsort[0]
15
16     mid = n // 2
17     xl, yl, xr, yr = split(xsort, ysort, mid)
18     pl, ql = closest_points(xl, yl)
19     pr, qr = closest_points(xr, yr)
20     dl, dr = distance(pl, ql), distance(pr, qr)
21     delta = min(dl, dr)
22     sqrt_delta = math.sqrt(delta)
23     best_pair = (pl, ql) if dl <= dr else (pr, qr)
24
25     L = (xsort[mid][0] + xsort[mid - 1][0]) / 2
26     candidates = [point for point in ysort if abs(L - point[0]) < sqrt_delta]
27
28     if len(candidates) < 2:
29         return best_pair
30
31     for i, point in enumerate(candidates):
32         j = i + 1
33         while j < len(candidates) and abs(point[1] - candidates[j][1]) < sqrt_delta:
34             d = sq_distance(point, candidates[j])
35             if d < delta:
36                 delta = d
37                 sqrt_delta = math.sqrt(delta)
38                 best_pair = (point, candidates[j])
39             j += 1
40     return best_pair
```

Figure 5: The algorithm

After having defined `delta` in Figure 5, the middle line, `L`, is calculated and a new array, `candidates`, is created by iterating through the y-sorted array, `ysort`, using Python list comprehension and adding the points that are within `sqrt_delta` (the Euclidean distance) distance of `L` on lines 25-26. This ensures that we can

iterate through the points within `sqrt_delta` from L, in a sorted y order

If there are less than two candidate points, the closest pair was either in A or in B and we simply return `best_pair`. Otherwise, we enumerate over `candidates` on lines 31-50, comparing each point to the subsequent points, until we are either further away than `delta` distance on the y-axis or until we reach the end of the array. When the loop is over, we have found the closest pair and return it.[1]

2.2.4 Time complexity

To characterize the running time of the implementation, consider the base cases in lines 3-14 in Figure 5. Since they contain a constant number of points each, it takes $O(1)$ time in total to find the closest pair. Now consider the split function call in line 17. In the split function in Figure 4, we go through n points a total of 8 times with the loop-bodies taking $O(1)$ time, and the list comprehensions in lines 4-5 and 13-14 taking $O(n)$ time resulting in a running time of $O(n)$ for the function. In lines 18-19 in Figure 5, the problem is divided into to subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively. In the list comprehension in line 26, n points are iterated through and a constant time check is made for each iteration, resulting in $O(n)$ time. In lines 31-39, `candidates` can at most contain n points, but recalling back to Figure 2, we find that at most a constant $c = 9$ number of points (since we disallow overlaps) are checked in each iteration, resulting in $O(n)$ running time for the loop. From the above, a recurrence describing the running time of the algorithm can be expressed as

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & n > 3 \\ O(1) & n = 2, n = 3 \end{cases}$$

which can be simplified to

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & n > 3 \\ O(1) & n = 2, n = 3 \end{cases}.$$

This can be solved using the master theorem, yielding the final running time of the algorithm

$$T(n) = \Theta(n \log n),$$

2.3 Testing

In this section, we experimentally test the time complexity of our implementation of the algorithm to determine if it matches our theoretical expectations. We also do secondary testing on the number of distance calculations performed by the algorithm in order to provide a more objective measure of how the algorithm behaves in practice since the number of distance calculations is not affected by noise to the time measurement or the reliability of the time-measurement methods in Python.

[1] In order to verify that our algorithm finds the correct closest pair, we run thorough tests by comparing it against a brute-force algorithm. The code for this can be found in the appendix.

2.3.1 Expected number of distance calculations

To determine the expected number of distance calculations, we can modify the recurrence described in Section 2.1 to account for the counting of distance calculations. Let $P_C(n, 2)$ be the number of distance calculations performed by the algorithm. For the distance calculations on the line, if the points are presorted by y, we do a constant number of distance calculations per point, resulting in $O(n)$ distance calculations. Thus, the modified recurrence for $P_C(n, 2)$ becomes:

$$P_C(n, 2) = 2P_C(n/2, 2) + O(n),$$

which solves to $P_C(n, 2) = O(n \log n)$ by master theorem.

2.3.2 Data generation

In order to test the algorithm, we first generate a number of data sets containing 2-dimensional points. We generate two types of data sets with varying point distributions. The first type of set has points uniformly distributed across the plane, while the second has points with uniform y-coordinates and fixed x-coordinates, rendering the points collinear along the y-axis. The uniform set represents the average case, while the collinear set represents the worst case, since the x-coordinates are the same, resulting in all of the points being within δ and thus being projected onto L during the merge step of the algorithm.

The code for generating the data sets can be found in the appendix. The data sets are generated with exponentially increasing sizes in order to verify that the running time is consistent through small and large n. For each n we generate 10 different data sets which will result in more reliable test results.

2.3.3 Setup

To measure the execution time of the algorithm we use the standard Python module `time`, with its `process_time` function, which measures CPU execution time and has a microsecond resolution on Windows 10 OS (our OS). This is better than other methods such as `perf_counter`, which measures wall-clock time and may be influenced by other processes on the same computer, introducing noise to time measurements. However, we still run the tests on computers that are not currently in use in order to minimize any potential noise. Furthermore, in order to verify that the algorithm runs in $O(n \log n)$ time, we normalize the measured execution time by dividing it by $n \log n$, which should leave us with some constant $c > 0$ as a result. This should manifest as a horizontal line when plotting the results. The code for measuring execution times can be found in Appendix 6.

Since we also measure the number of distance calculations, we increment a counter whenever the distance calculation function is called and save the result after the algorithm terminates. This result is also normalized by dividing it by $n \log n$.

2.3.4 Results

As seen in Figure 6, our results for the uniform data sets form an acceptable horizontal line when we normalize the running time from the tests. This indicates that

the algorithm does indeed run in the expected theoretical time with uniformly distributed points. The orange lines in the boxplots represent the mean and the circles represent outliers that may be a result of noise introduced during testing or a result of the data set being generated in a way that is likely to result in a slow running time compared to the rest of the data sets.

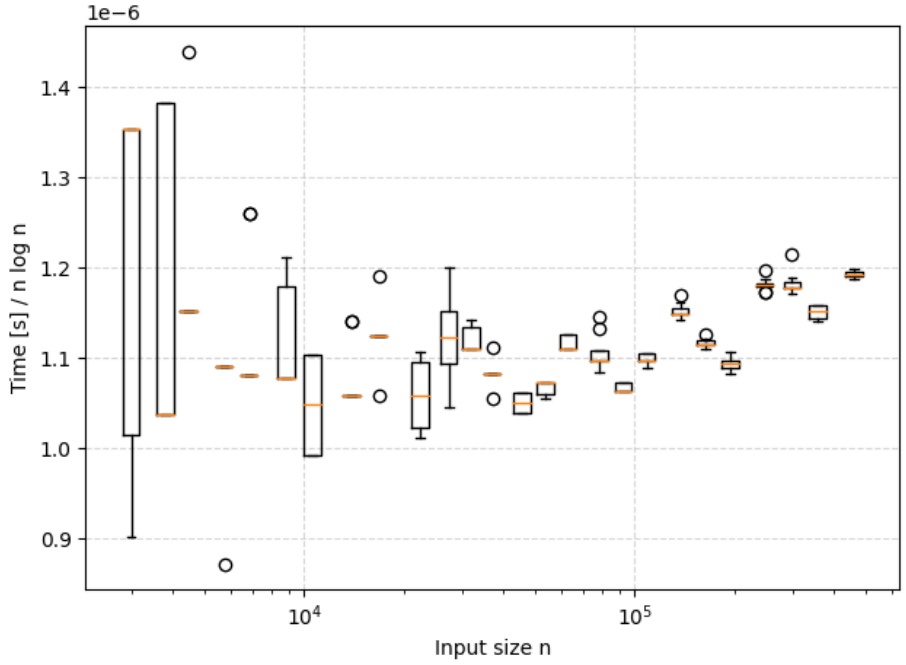


Figure 6: Running time for uniform points

Looking at the results for the distance calculations we surprisingly see a decreasing trend, indicating the number of distance calculations is actually less than we expect. If we instead normalize the distance calculations results by dividing by the input size n, in Figure 8, we see that the number of distance calculations appears to follow a $O(n)$ trend.

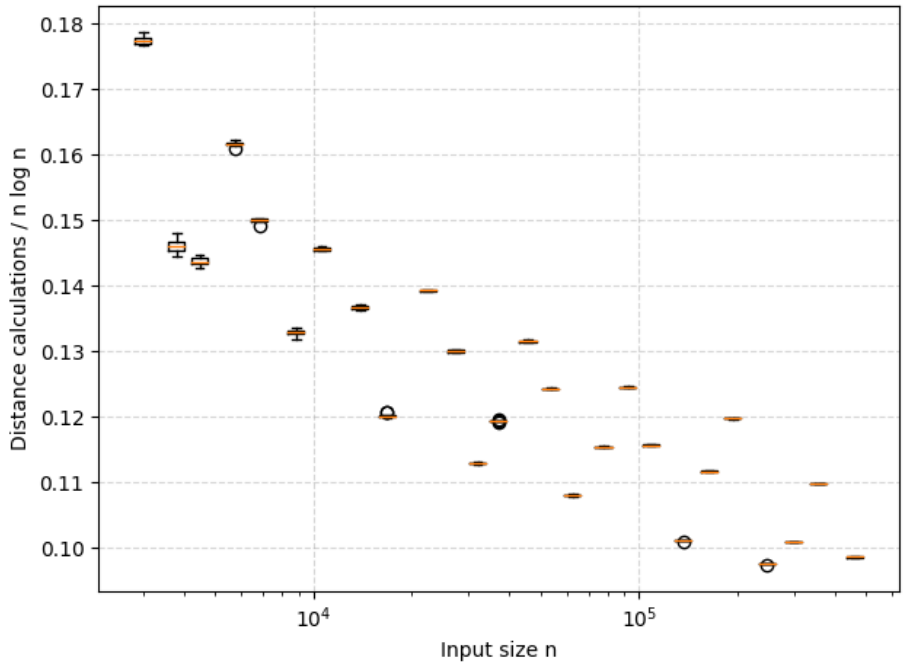


Figure 7: Distance calculations for uniform points

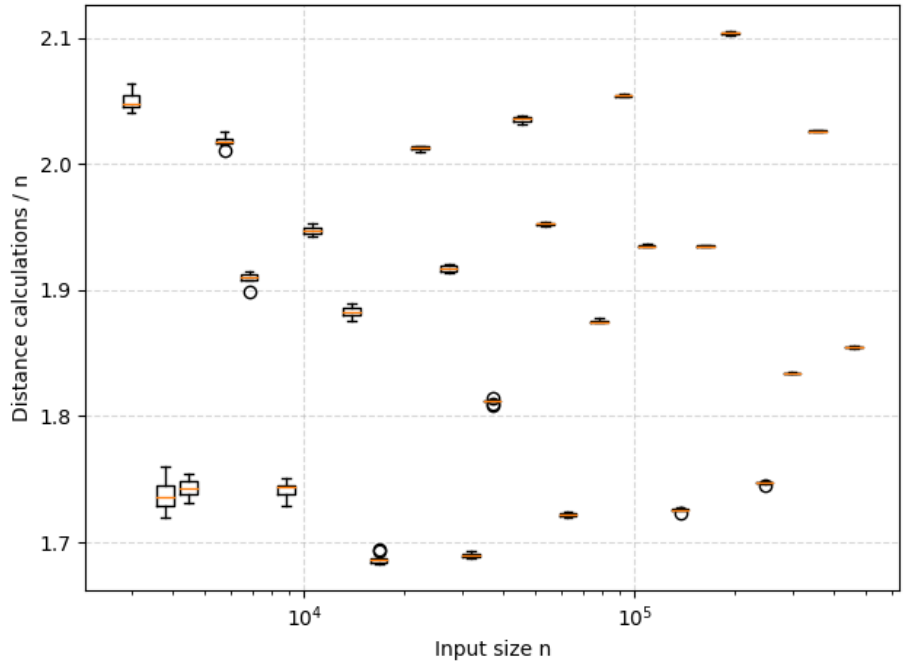


Figure 8: Distance calculations for uniform points divided by n

The results for the collinear tests in Figure 9 unexpectedly show a similar running time compared to the uniform results. We discuss this in Section 2.3.5. We see that the number of distance calculations in Figure 10 is also similar to the number of distance calculations for uniform points.

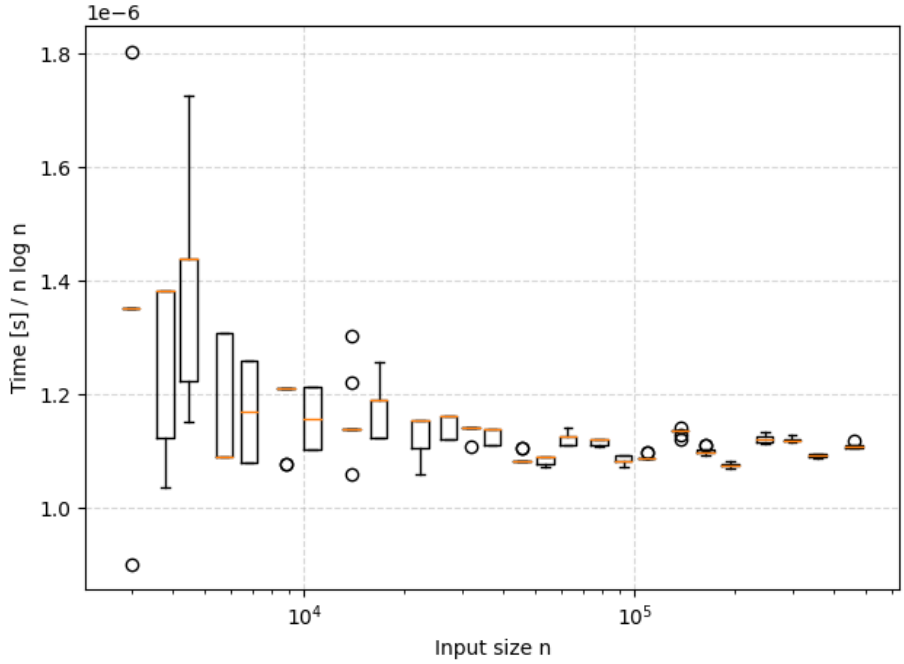


Figure 9: Running time for collinear points

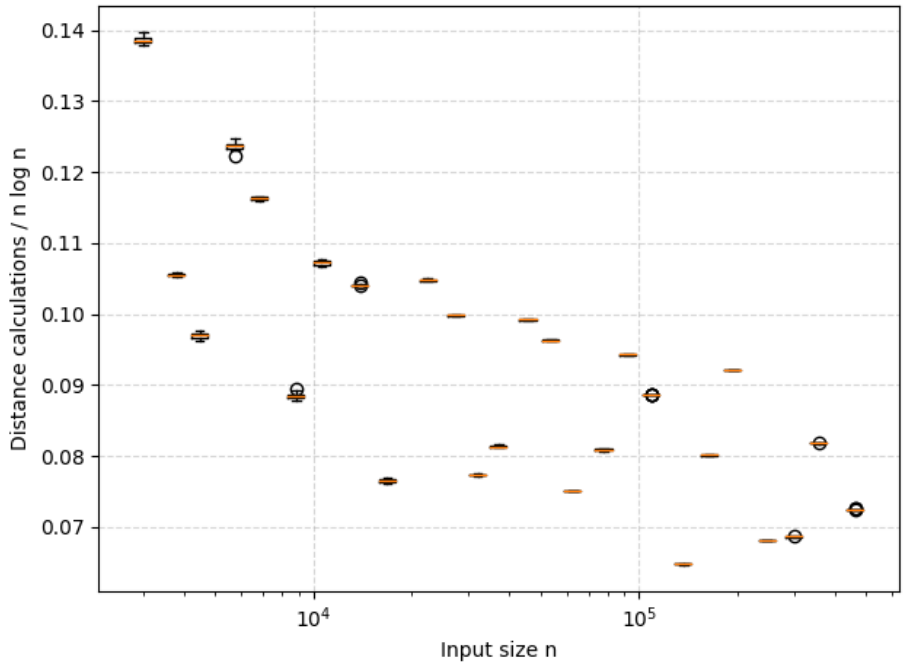


Figure 10: Distance calculations for collinear points

2.3.5 Discussion

Initially, we anticipated that the algorithm would perform significantly slower on collinear data sets compared to uniform data sets. However, we discovered that our

test sets were inadvertently generated in a way that favored the collinear sets. This advantage arises from an optimization in the data set generation code (see appendix), which significantly speeds up generation, in which we only perform sorting and the index pointer setup process on the y-sorted array, which we then copy over to the x-sorted array. Since the x coordinates for the points are all 0, they would still be sorted by x- coordinates by duplicating the y-sorted array. Consequently, when running the algorithm on such a data set, (close to) optimal cuts are consistently achieved since a cut on the x-sorted array translates to the same cut on the y-sorted array.

Additionally, as per the theory, we should compare the distance between one point and the following 8 points ($c = 9$) for every point along the y-axis, but we instead stop comparing when the distance on the y-axis exceeds `sqrt_delta`, as seen in Figure 5, on line 33. This optimization means that even though we include all points in our `candidate` array, we only use the distance calculation when two points are closer than `sqrt_delta` on the y-axis. Given that x is always 0, this implies that we will, at most, need to calculate one additional distance along L, as the two points closest to L are the only points that can have a distance smaller than δ.

A more naive approach, as seen in Figure 11, would involve iterating through 8 points ahead for each point in the `candidates` array, without considering that the y-distance might be further than `sqrt_delta`. This approach would be slower because it would force the algorithm to examine the maximum number of points in the worst-case scenario.

```
1 for i, point in enumerate(candidates):
2     for j in range(1, 9):
3         if i+j >= len(candidates):
4             continue
5         d = sq_distance(point, candidates[i+j])
6         if d < sqrt_delta:
7             delta = d
8             sqrt_delta = math.sqrt(delta)
9             best_pair = (point, candidates[i+j])
```

Figure 11: Slower linear scan on L

3 Closest points in k-space

In this section, the algorithm given in Section 2.1 is generalized to k-space. We also describe algorithms by Bentley and Shamos [1976] for the sparse fixed-radius near-neighbor problem on the plane and in k-space. These will in a later section be applied to obtain an optimized algorithm for the closest-points problem in k-space in $O(n \log n)$ time.

3.1 Sparse Fixed-radius near-neighbor on the plane

The algorithm is similar to the first algorithm and proceeds by partitioning the points into two pointsets A and B of size $n/2$ by a vertical line L. The algorithm

is recursively used to list all pairs within distance δ (note that δ is pre-determined) of one another in A and in B (with base cases similar to those cases in the first algorithm). The next step is to list all pairs $(p, q) \in A \times B$ within distance δ of one another. As in the first algorithm, consider all points within the slab that extends δ distance from either side of L. From the problem statement, it is apparent that this set of points is sparse since no square of sidelength 2δ contains more than a constant number of points. Next, the points are projected onto L, and all pairs within a distance of δ of each other on L are examined to determine if they are within δ of each other in the plane. As in the first algorithm, if the points are pre-sorted by y-coordinate, the checking could be done in linear time. From this, we get the recurrence

$$S(n, 2) = 2S(n/2, 2) + O(n),$$

which by the master theorem, solves to

$$S(n, 2) = O(n \log n),$$

giving an $O(n \log n)$ algorithm.

3.2 Sparse Fixed-radius near-neighbor in k-space

In the above algorithm, we reduce the sparse problem into two sparse problems of size $n/2$ and one sparse problem on the line. We extend this approach to k-space in a relatively straightforward manner: We once again partition the points into two pointsets A and B of size $n/2$, but now with a hyperplane P perpendicular to the x-axis (for $k = 2$, this was the line L). The algorithm is recursively used to list all pairs within distance δ of one another in A and in B. To list all pairs $(p, q) \in A \times B$ within distance δ of one another, we project all points within distance δ from P, resulting in a subproblem in $(k-1)$-space with up to n points. In the first algorithm, we showed that no square of sidelength 2δ within δ of L contains more than $c = 12$ points. This can be generalized to k-space with a sparsity constant of $c = 4(3^{k-1})$. This sparsity constant is preserved for the points projected onto P. We then apply the algorithm recursively to solve the subproblem.

3.2.1 Time complexity

To determine the time complexity of the algorithm, note that we reduce the original problem into two subproblems of $n/2$ in the same dimension and a subproblem of up to n points in $k - 1$ dimensions. We can describe this as a recurrence

$$S(n, k) = 2S(n/2, k) + S(n, k - 1) + O(n)$$

Notice that for any $k > 2$, we eventually reach a 2-dimensional subproblem, which can be solved in $O(n \log n)$ time. We can use this fact as a base case for induction on k to prove the following theorem:

Theorem 1: For any $k \geq 2$,

$$S(n, k) = O(n \log^{k-1} n).$$

Proof. For the base case $k = 2$, we have $S(n, 2) = O(n \log n)$ by the algorithm in Section 3.1, so the theorem holds.

Inductive step: Assume the theorem holds for some $k \geq 2$. Then for $k + 1$ we get

$$\begin{aligned} S(n, k+1) &= 2S(n/2, k+1) + S(n, k) + O(n) \\ &= 2S(n/2, k+1) + O(n \log^{k-1} n) && \text{(by induction hypothesis)} \\ &= O(n \log^k n) && \text{(by master theorem)} \end{aligned}$$

Thus, the theorem holds for $k+1$. By induction, the theorem holds for any $k \geq 2$. □

With the sparse problem in place, we can now describe the algorithm for the closest points problem in k-space.

3.3 Algorithm Description

The algorithm follows a similar strategy as the sparse problem in k-space. The points are partitioned into two pointsets A and B of size $n/2$ by a hyperplane P. The algorithm is recursively used on A and B (using the same base cases as in the 2-dimensional algorithm) to find the closest pair in A with distance δ_A and the closest pair in B with distance δ_B. The situation is depicted in Figure 12. Let $\delta = \min(\delta_A, \delta_B)$. All points within δ of P are projected onto P. We can now recursively use the algorithm in one lower dimension to find the closest pair. Since the points are projected, we need to keep track of all the coordinates of the points to ensure their distance is closer than δ in the original k-space.

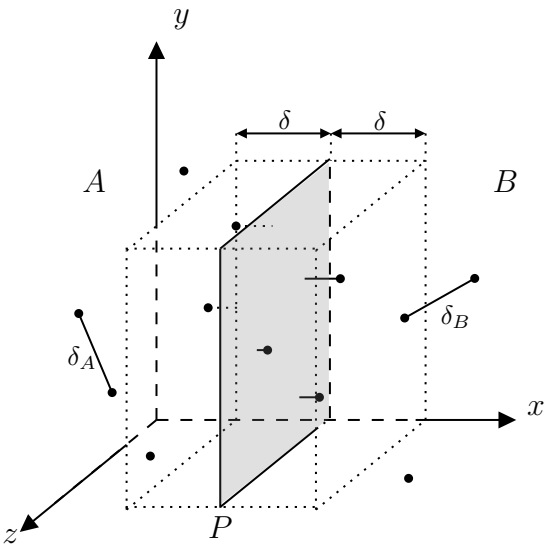


Figure 12: Illustration of how the algorithm works in three dimensions

3.3.1 Time complexity

In the above, we again reduce the problem into two subproblems of size $n/2$ and one subproblem in $k-1$ dimensions. Thus, the recurrence is

$$P(n,k) = 2P(n/2,k) + P(n,k-1) + O(n).$$

Similar to Theorem 1, we can use the algorithm in 2.1 as a base case for induction on k to prove the following:

Theorem 2: For any $k \geq 2$,

$$P(n,k) = O(n \log^{k-1} n).$$

Proof. For the base case $k = 2$, we have $P(n,2) = O(n \log n)$ by the algorithm in Section 2.1, so the theorem holds.

Inductive step: Assume the theorem holds for some $k \geq 2$. Then for $k+1$ we get

$$\begin{aligned} P(n,k+1) &= 2P(n/2,k+1) + P(n,k) + O(n) \\ &= 2P(n/2,k+1) + O(n \log^{k-1} n) && \text{(by induction hypothesis)} \\ &= O(n \log^{k} n) && \text{(by master theorem)} \end{aligned}$$

Thus, the theorem holds for $k+1$. By induction, the theorem holds for any $k \geq 2$. □

3.4 Implementation

This section details the implementation of the algorithm in Section 3.3. We generalize the techniques used in the 2D algorithm in order to handle k dimensions and also describe the challenges we encountered during this process.

3.4.1 Generalizing index pointers

Since the algorithm needs to handle k dimensions, we need k separate arrays containing the points, each sorted by some coordinate. We nest these arrays inside an outer array, `pointsets`. Consequently, we also need to manage an arbitrary number of index pointers connected to each point. This difference poses a challenge for our split function for the first algorithm, where each point has a single index pointer pointing to the other sorted array. Therefore, we require a new structure for our points. We now represent the points as arrays of size $2k$, where the first k entries are the coordinates of the points and the i'th entry with $i > k$ is the index of the same point in the i'th array within `pointsets`. While this implies that there is an unused index pointer connected to each point, it simplifies the new split process.

Consider the updated split function in Figure 13 (some functions are omitted in the figure, but they can be found in the appendix). The arguments are defined as follows: `cut_axis` represents the index of the sorted array in `pointsets` that we cut on. It determines the dimension used to find the cut plane P. `mid` is the index of the

midpoint where we split the pointsets into two pointsets A and B. `k_init` is used to distinguish between the coordinates of the points and index pointers. During the execution of the algorithm, we preserve the coordinates of the points but remove some index pointers.

In lines 2-12, we split the sorted points corresponding to `cut_axis` into two halves and initialize empty arrays for the remaining axes. In line 14, we call a function that sets the points in the other sorted arrays (other axes) according to the index pointers of the points in `cut_left`. This process ensures the relative ordering of the points is preserved, with some `None` entries in between. This operation has a time complexity of $O(k^2n)$. In lines 17, we remove the `None` entries, leaving us with the same points in `points_l`, but sorted by different axes. This step takes linear time. We now need to update the index pointers. In line 22, we update the index pointers of the points in the `cut_left` array by iterating through the points in the other sorted arrays (other axes). This ensures that the index pointers of the points corresponding to the cut-axis are correctly updated. We can then use these index pointers to correct the index pointers in the other arrays (other axes) in line 25, taking $O(k^2n)$ time. We perform all of the above with the right half as well.

```
1  def split_kd(pointsets, cut_axis, mid, k_init):
2      cut_left = pointsets[cut_axis][:mid]
3      cut_right = pointsets[cut_axis][mid:]
4
5      pointsets_l, pointsets_r = [], []
6      for axis in range(len(pointsets)):
7          if axis == cut_axis:
8              pointsets_l.append(cut_left)
9              pointsets_r.append(cut_right)
10         else:
11             pointsets_l.append([None] * len(pointsets[0]))
12             pointsets_r.append([None] * len(pointsets[0]))
13
14     set_axes_cut_iptr(pointsets_l, cut_axis, k_init)
15     set_axes_cut_iptr(pointsets_r, cut_axis, k_init)
16
17     pointsets_l = [[p for p in pointset if p is not None]
18                    for pointset in pointsets_l]
19     pointsets_r = [[p for p in pointset if p is not None]
20                    for pointset in pointsets_r]
21
22     set_cutaxis_iptrs(pointsets_l, cut_axis, k_init)
23     set_cutaxis_iptrs(pointsets_r, cut_axis, k_init)
24
25     set_axes_iptrs(pointsets_l, cut_axis, k_init)
26     set_axes_iptrs(pointsets_r, cut_axis, k_init)
27     return pointsets_l, pointsets_r
28
29 [...]
```

Figure 13: The new split function for arbitrary k

3.4.2 Distance calculation

In contrast to the distance calculation in the first algorithm, each point now has k coordinates. The squared Euclidean distance between two points $p \in \mathbb{R}^k$ and $q \in \mathbb{R}^k$ is defined by

$$d^2(p, q) = \sum_{i=1}^{k} p_i^2 - q_i^2.$$

We implement this in Figure 14 by summing over the coordinate-wise squared distance for each coordinate.

```
def sq_distance(p, q, k):
    return sum([(p[i] - q[i])*(p[i] - q[i]) for i in range(k)])
```

Figure 14: Euclidean Distance calculation for k dimensions

3.4.3 Code

One challenge this algorithm poses is how to cut away dimensions while retaining the information needed to calculate distances in the original space. To do this, we introduce two new arguments: `k_init`, which keeps track of the original dimension in order to properly calculate distances, since the points keep all of their original coordinates (the current k is decremented for each subproblem), and `cut_axis`, which keeps track of the current dimension we are using to find P. Since the algorithm always cuts away the first axis, we initialize `cut_axis` to 0. Consider the algorithm in Figure 15. Since we handle subproblems in a lower dimension, we need a base case for a 2-dimensional subproblem. We check if $k = 2$ in line 5, and if so, we apply the first algorithm. Here `closest_points_2d` is a slightly modified version of the first algorithm that uses the new split function and handles k-dimensional points. The base cases remain the same as in the first algorithm. Next, `pointsets` is split, and recursions are used to define `delta` and `best_pair`, followed by a definition of the cut plane `P`. Maintaining index pointers in k dimensions is a more costly process in k dimensions. Therefore, the algorithm first checks if there are at least two points within `sqrt_delta` in lines 11-26. If not, we just return the best pair. If there are at least 2 points on opposite sides to project onto P, the algorithm continues in lines 28-53 to create the `proj_points` array with one less dimension by excluding the first nested array in `pointsets` in the for-loop. It then calls itself recursively in line 52 to compare the current δ to the δ of the `other_pair`. It will then return the pair of points with the smaller δ in line 53.

```python
def closest_points_kd(pointsets, k_init, cut_axis=0):
    cut_axis_points = pointsets[0]
    k = len(pointsets)
    n = len(cut_axis_points)
    if k == 2:
        return closest_points_2d(pointsets, cut_axis, k_init)
    [...Base Cases...]
    mid = n // 2
    pointsets_l, pointsets_r = split_kd(pointsets, 0, mid, k_init)
    pl, ql = closest_points_kd(pointsets_l, k_init, cut_axis)
    pr, qr = closest_points_kd(pointsets_r, k_init, cut_axis)
    [...Define delta, best_pair, sqrt_delta...]

    P = (cut_axis_points[mid][cut_axis] +
         cut_axis_points[mid-1][cut_axis]) / 2

    nof_proj_points = 0
    exists_left = False
    exists_right = False
    for point in pointsets_l[0]:
        if abs(P - point[cut_axis]) < sqrt_delta:
            exists_left = True
            nof_proj_points += 1

    for point in pointsets_r[0]:
        if abs(P - point[cut_axis]) < sqrt_delta:
            exists_right = True
            nof_proj_points += 1

    if nof_proj_points < 2 or not(exists_left and exists_right):
        return best_pair

    candidates = [[None] * (k_init + k - 1) for _ in range(n)]
    for axis, pointset in enumerate(pointsets):
        if axis == 0: continue
        i = 0
        for point in pointset:
            if abs(P - point[cut_axis]) < sqrt_delta:
                coords, iptrs = split_point(point, k_init)
                cut_iptr = iptrs[0]
                cut_point = candidates[cut_iptr]
                if cut_point[0] is None:
                    cut_point[:k_init] = coords
                cut_point[k_init+axis-1] = i
                i += 1

    proj_points = [[None] * nof_proj_points for _ in range(k-1)]
    for point in candidates:
        if point[0] is None: continue
        for axis, iptr in enumerate(point[k_init:]):
            proj_points[axis][iptr] = point

    other_pair = closest_points_kd(proj_points, k_init, cut_axis + 1)
    return other_pair if sq_distance(*other_pair, k_init) <= delta else best_pair
```

Figure 15: The algorithm

3.5 Testing

This section details the experimental testing of the algorithm. We evaluate the running time of the algorithm as well as the number of distance calculations performed.

3.5.1 Data generation & Setup

Similar to the first algorithm, we generate a number of data sets containing 3-dimensional points with two different point distributions. The first point distribution has points uniform across the cube. The second point distribution has points uniform across the z-axis with fixed x- and y-coordinates, rendering the points collinear along the z-axis. We conduct the testing for the second algorithm using a similar setup to that of the first algorithm. However, since we expect a $O(n \log^2 n)$ running time, we now normalize the measured execution time by that value. A similar argument to that in Section 2.3.1 shows that we expect $O(n \log^2 n)$ distance calculations.

3.5.2 Results

When looking at Figure 16, we see that the running time decreases with increasing n when dividing by $n \log^2 n$, which indicates a better running time than in the worst case. We might suspect a running time closer to $n \log n$. However, if we normalize the results by $n \log n$ in Figure 17, we see an increasing trend. To rule out potential noise, we can look at the number of distance calculations made for more objective results, when normalizing by $n \log n$ in Figure 18. We see a concerning increasing trend for the running time in Figure 18. However, looking at the distance calculations for the collinear points normalized by $n \log n$ we see a similar pattern as for the uniform points.

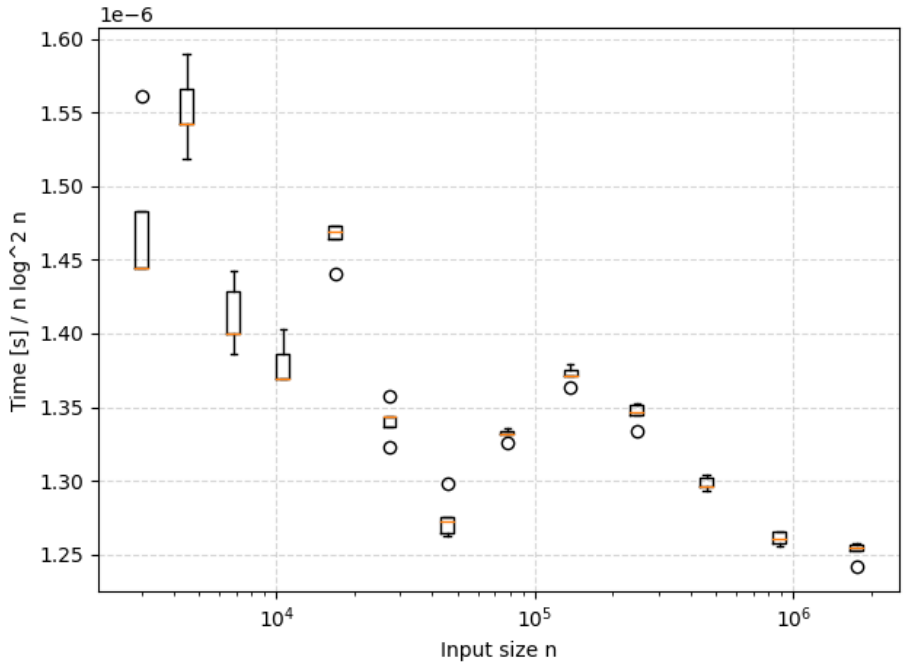


Figure 16: Running time for uniform points by $n \log^2 n$

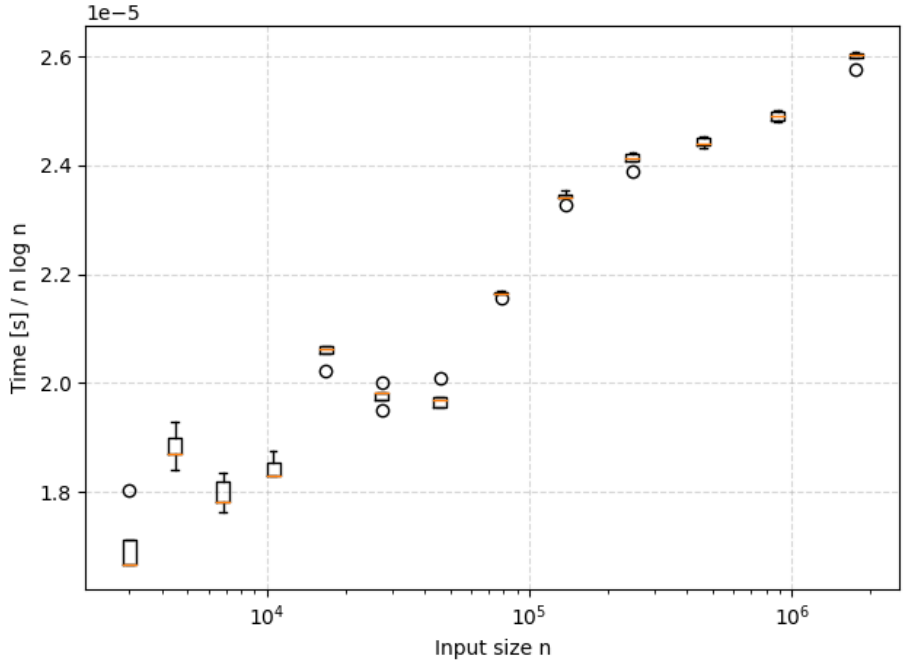


Figure 17: Running time for uniform points by $n \log n$

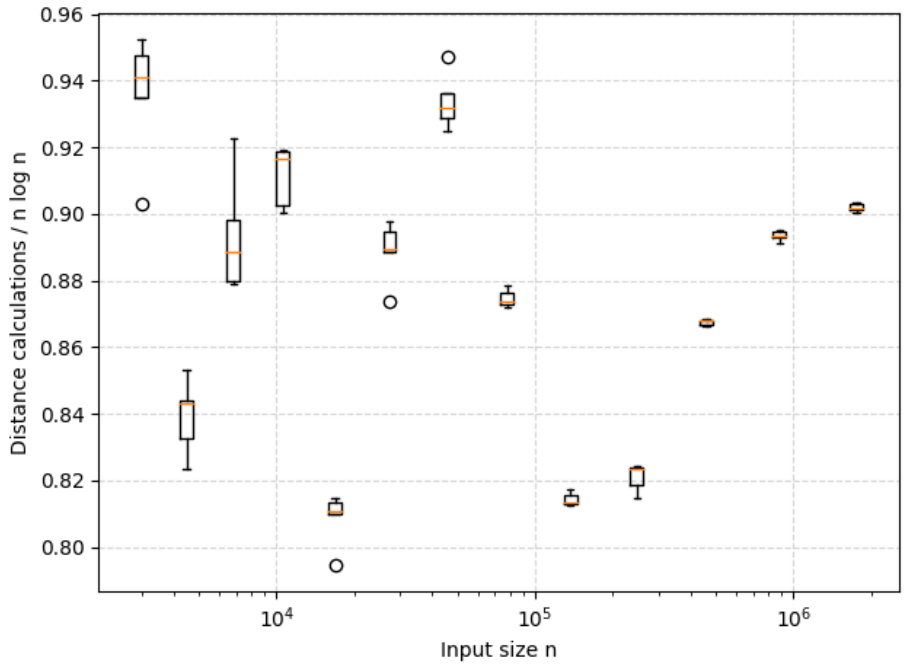


Figure 18: Distance calculations for uniform points by $n \log n$

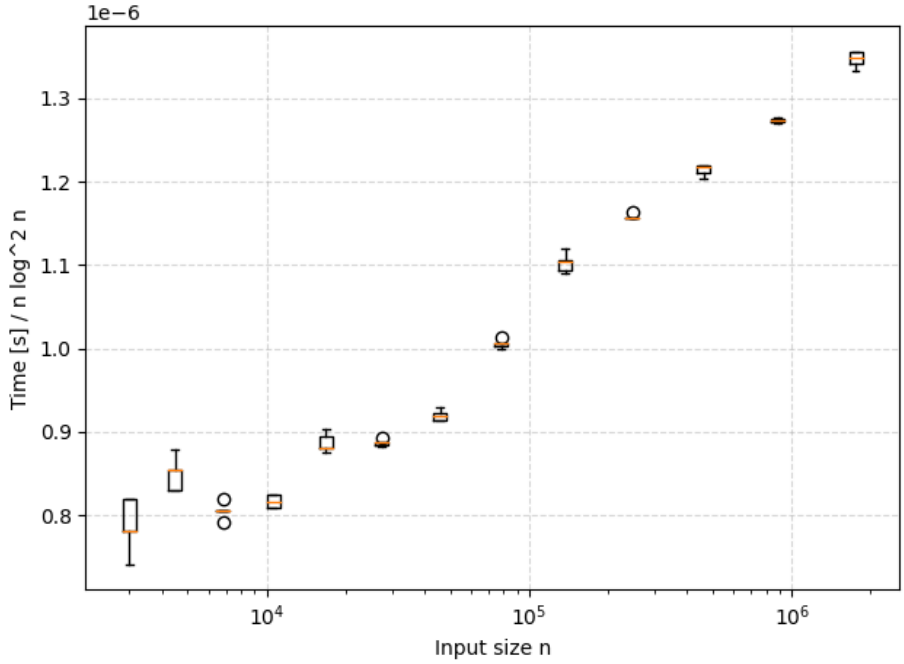


Figure 19: Running time for collinear points by $n \log^2 n$

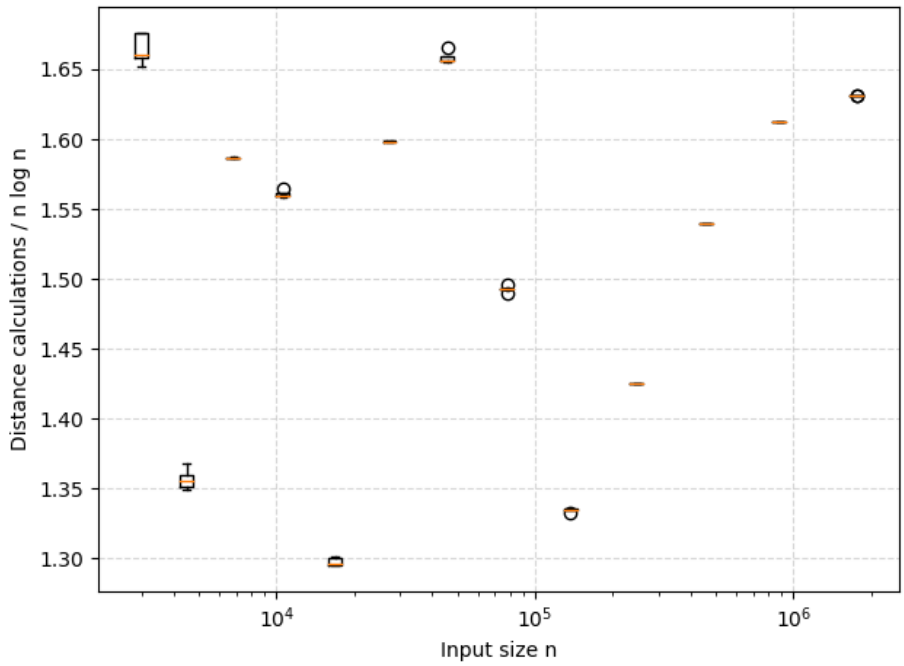


Figure 20: Distance calculations for collinear points by $n \log n$

3.5.3 Discussion

It is clear from the running time results that the algorithm is faster (with increasing n) in the uniform tests than in the collinear tests, which is as expected. We see

a great difference in the results when looking at Figure 16 and Figure 19, but looking at the corresponding distance calculation results, we see a lot of similarities, despite the noise and fluctuations. This could suggest that some noise might have been introduced during the testing process. Alternatively, it could indicate that a significant portion of the work done by the algorithm is spent maintaining the pointsets rather than calculating distances. The former explanation seems more plausible, primarily because Python can be unreliable when measuring execution times. One contributing factor specific to this case is the overhead associated with using Python lists to store the points, in which memory is overallocated, allowing for amortized $O(1)$ time appends. However, other libraries available in the Python Standard Library offer the capability to create fixed-size, efficient arrays, eliminating this overhead. It would be interesting to evaluate the algorithm's behavior without this overhead.

In conclusion, we observe that the algorithm yields faster running time results than the theoretical worst-case time for uniform points, while slower results are obtained for collinear points. However, due to the unreliable nature of measuring execution times in Python, additional considerations, such as the overhead associated with Python lists, need to be taken into account.

4 Optimized algorithm

The time complexity for the algorithm described in 3.3 has an exponential term in the logarithmic factor due to the fact that we are dealing with a subproblem involving (up to) all of the n points in $k-1$ dimensions. However, Bentley and Shamos [1976] have shown that the subproblem size for the sparse problem can be bounded by some function $f(n)$, such that $f(n)\log f(n) \leq O(n)$. The recurrence in Section 3.2 then becomes

$$\begin{aligned} S(n,k) &= 2S(n/2,k) + S(f(n),k-1) + O(n) \\ &= 2S(n/2,k) + O(n) \end{aligned}$$

allowing for an $O(n\log n)$ algorithm for the sparse problem. This can then be applied to obtain an $O(n\log n)$ algorithm for the closest-points in k-space.

4.1 Cut-planes

Bentley and Shamos [1976] show that bounding the size of the subproblem involves a more intelligent choice of the cut-plane P that partitions the pointset into two subsets A and B. The cut-plane must satisfy two conditions. Firstly, both A and B must not contain "too many" points, ensuring that the k-space subproblems maintain balance. Secondly, there should be at most $O(n/\log n)$ points within distance δ of P. The following proof by Bentley and Shamos [1976] demonstrates the existence of such a cut-plane on the plane, i.e., a cut-line:

Theorem 3: For a sparse set of n points in the plane, there exists a cut-line L

perpendicular to one of the original axes such that (1) both subsets A and B induced by L contain at least $n/8$ of the points and (2) there are at most $2cn^{1/2}$ points within distance δ of L.

Proof (see Bentley and Shamos [1976]). Assume, for the sake of contradiction, that there exists a set of points without the above properties. Consider the points sorted by increasing x-value. We focus on the middle $n-2n/8 = 3n/4$ points. Assuming the converse of the theorem implies that every set of $2cn^{1/2}$ points sorted by increasing x-value projects onto a segment at most 2δ in length on the x-axis (if the length was greater than 2δ it could be used to define a cut-plane that satisfies the conditions). The scenario is illustrated in Figure 21. The regions L_x and R_x contain the leftmost $n/8$ points and the rightmost $n/8$ points, respectively. The region C_x contains the middle $3n/4$ points. The region T contains a set of $2cn^{1/2}$ points. We observe that C_x is composed of $\frac{3n/4}{2cn^{1/2}}$ collections of $2cn^{1/2}$ points. Since each of these collections has a width of at most 2δ we can bound the width w of C_x:

$$w \leq \frac{3n/4}{2cn^{1/2}} 2\delta = \frac{3n^{1/2}\delta}{4c}$$

Similar arguments can be made for C_y, which consists of the middle $3n/4$ points sorted by increasing y-value.

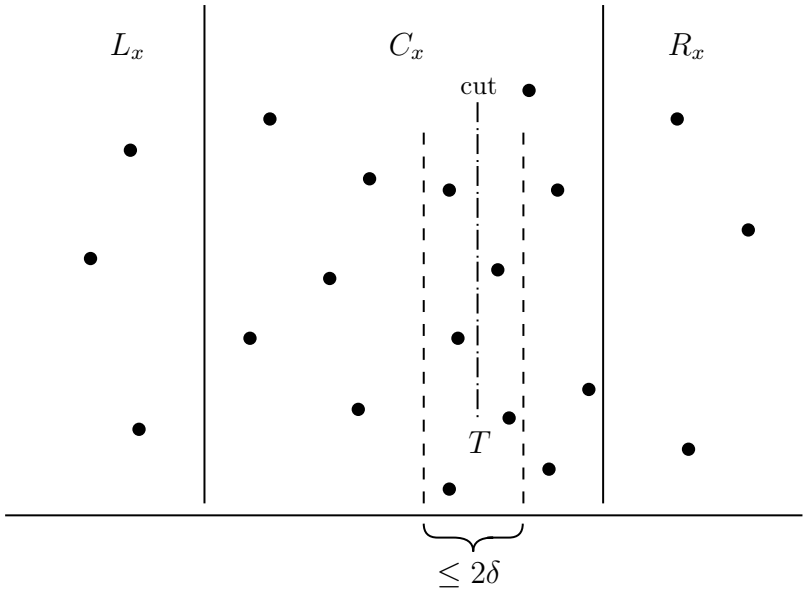


Figure 21: Points sorted by increasing x-value

Now, consider C_{xy}, the intersection of C_x and C_y. Since L_x, L_y, R_x and R_y contain at most $n/8$ points each, C_{xy} must contain at least $4n/8 = n/2$ points. However, since the length of the sides of C_{xy} is bounded by $\frac{3n^{1/2}\delta}{4c}$, the area of C_{xy} is bounded by that value squared $\frac{9n\delta^2}{16c^2}$. By sparsity, we know that the area associated with each point in a square of side 2δ is $c/4\delta^2$. It follows that the number of points in C_{xy} is bounded above by

$$\frac{9n\delta^2}{16c^2} \cdot \frac{c}{4\delta^2} = \frac{9n}{64c},$$

which for $c \geq 1$ is less than $n/2$, which is a contradiction. Thus, a cut-line satisfying the conditions exists. □

We can generalize this to k-space:

Theorem 4: For a sparse set of n points in k-space, there exists a cut-plane P perpendicular to one of the original coordinate axes such that (1) both subsets A and B induced by P contain at least $n/4k$ points and (2) there are at most $kcn^{1-1/k}cn$ points within distance δ of P.

Proof (see Bentley and Shamos [1976]). The proof follows a similar approach as in Theorem 2. Assuming the converse of the theorem leads us to believe that the k-dimensional hypercube $C_{x_1,\ldots,x_k}$, which is the intersection of $C_{x_1}, \ldots, C_{x_k}$, contains at least $n/2$ points but at most $\left(\frac{1-1/2k}{kc}\right)^k cn$ points. However, since

$$n/2 > \left(\frac{1-1/2k}{kc}\right)^k cn$$

for $k > 1$, $c \geq 1$, this is a contradiction. Thus, a cut-plane satisfying the conditions exists. □

4.2 Optimized Sparse Algorithm

The cut-planes in Theorem 4 can be applied in the algorithm in Section 3.2 to obtain an $O(n \log n)$ algorithm for the sparse fixed-radius near-neighbor problem. To find a suitable cut-plane P, the algorithm will scan each of the k sorted sequences of points until a point that defines P is found. This can be done in $O(kn)$ time. The scan involves investigating the middle $n(1 - 1/4k)$ points in each dimension until a sequence of $kcn^{1-1/k}$ points is found that projects onto an interval of length at most 2δ. Next, we recursively list all fixed-radius near-neighbor pairs in each of A and B. The cost of this step will be greatest when the problem sizes are the most unbalanced. Since A and B must contain at least $n/4k$ points by Theorem 4, this will be when $|A| = n/4k$ and $|B| = n - |A| = n - n/4k$, or vice versa. All points within distance δ of P are projected onto P, which by Theorem 4 is at most $kcn^{1-1/k}$ points. The recurrence is then

$$S(n, k) \leq S(n/4k, k) + S(n - n/4k, k) + S(kcn^{1-1/k}, k - 1) + O(kn)$$

which for fixed k solves to $O(n \log n)$. This follows from the fact that $S(kcn^{1-1/k}, k-1) \leq O(n)$, since $kcn^{1-1/k} \leq O(n/\log^x n)$ for a constant $x > 0$. The proof of this is based on the fact that for fixed $k > 1$ and $c \geq 1$, we have

$$n^{1-1/k} \leq O\left(\frac{n}{\log^x n}\right) \implies \frac{n}{n^{1/k}} \leq O\left(\frac{n}{\log^x n}\right) \implies \frac{1}{n^{1/k}} \leq O\left(\frac{1}{\log^x n}\right) \implies$$

$$n^{1/k} \geq O\left(\log^x n\right) \implies \log n^{1/k} \geq O(x \log\log n) \implies \frac{1}{k} \log n \geq O(\log\log n).$$

Since k is fixed, the above is asymptotically true.

4.3 Optimized Closest Points Algorithm

With the optimized algorithm for the sparse problem at our disposal, we can now utilize it to solve the $(k-1)$-dimensional subproblem for the closest points problem and, check each of the listed pairs that are within δ distance, and find the closest. If we also bound the size of the subproblems in the original problem by a well-chosen cut-plane P, we get a recurrence

$$\begin{aligned} P(n,k) &= P(n/4k,k) + P(n - n/4k,k) + S(kcn^{1-1/k}, k-1) + O(n) \\ &= P(n/4k,k) + P(n - n/4k,k) + O(n) \end{aligned}$$

which yields a time complexity of $O(n \log n)$. The cut-plane P should have the following properties: (1) The subsets A and B induced by P must contain at least $n/4k$ points. (2) When $\delta = \min(\delta_A, \delta_B)$ is found, there are at most $kcn^{1-1/k}$ points within distance δ of P. To determine such a cut-plane for $k = 2$ (a cut-line), given that the points are presorted along each axis, we perform a linear scan for each axis. We compare the distance, denoted as m, from one point to the point that is $2cn^{1/2}$ points ahead. The scan starts with the $(n/8)$-th point in the sequence and continues until the second point falls within the last $n/4k$ points, i.e., the rightmost pointset B (this will ensure property 1 is satisfied). The associated cut-line is then defined as the average of the two points. Among the cut-lines found this way, we select the one that maximizes m to divide the problem. The $k-1$ dimensional subproblem is solved using the algorithm in Section 4.2. To show that property 2 is satisfied, it suffices to show that $m > 2\delta$. If we assume $m \leq 2\delta$, then any set of $2cn^{1/2}$ points sorted by some axis, projects onto a segment of length at most 2δ along that axis. However, in Theorem 3, the same assumption leads to a contradiction. Employing a similar argument, we arrive at the same contradiction, thereby showing that $m > 2\delta$ and thus that the cut-line satisfies the required properties.

This approach can be extended to k-space. The linear scan now bounds the subproblem sizes by $n/4k$ and determines a cut-plane P that maximizes the distance m between two points that are $kcn^{1-1/k}$ points apart, where $c = 4(3^{k-1})$. Employing a similar argument as before, we find that once δ is found, then $m > 2\delta$, thereby bounding the $(k-1)$-dimensional subproblem size by $kcn^{1-1/k}$.

4.4 Implementation

In this section, we modify the implementation of the algorithm in Section 3.4 to make use of cut-planes with the properties described in Section 4.3.

4.4.1 Deviation from Theory

There are some significant changes when it comes to implementing the algorithm compared to the one in Section 4.3. One of these changes is that we do not implement an algorithm for the sparse problem (similar to the previous algorithm) and instead recursively use the same algorithm for the closest points to solve the subproblem. Another important change involves our approach in finding the cut-plane P. This is because for relatively small n, the point that is $kcn^{1-1/k}$ points ahead of the $(n/4k)$-th point in a sequence might already be within B or extend beyond the total number

of points, rendering the linear scan infeasible. Nevertheless, it remains possible to find the suitable cut-plane for large enough n,[2] given the following inequality is satisfied:

$$\begin{aligned} n - n/4k &\geq n/4k + kcn^{1-1/k} \\ \implies n &\geq 2n/4k + kcn^{1-1/k} \end{aligned}$$

with $c = 4(3^{k-1})$. For n that do not satisfy the above, a different approach to finding the cut-plane is needed. One approach might be to always cut in the middle but randomly decide which axis to cut on. However, a better approach is to go through each axis, note the interval between the extreme points of the middle $n - 2n/4k$ points and choose to cut in the middle of the axis with the largest such interval.

4.4.2 Code

Consider the algorithm in Figure 22. We omit parts of the code that are similar to Figure 15 (see appendix for the full code). This version of the algorithm introduces a slight change in the parameters compared to the algorithm described in Section 3.4. Since the algorithm now cuts on arbitrary axes, it becomes necessary to keep track of the remaining coordinate axes. To do this, we introduce a parameter `axes`, which is initially an array containing the numbers $0, \ldots, k-1$, representing the coordinate axes. Whenever a cut is made on a coordinate axis, the corresponding entry in `axes` is removed.

In lines 5-8, we define the constants necessary to find the cut-plane. Depending on the value of n, we utilize the method described in Section 4.3 in lines 10-21 if n is large enough. Otherwise, we employ our own method in lines 22-29. Furthermore, we define `cut_axis` as the index of the coordinate axis to be cut on within `pointsets`, and `cut_axis_coord` as the index of the corresponding axis in the actual points. Note that these indices differ because subarrays in `pointsets` are progressively removed in each $(k-1)$-dimensional subproblem, while the points themselves retain their original coordinates.

4.5 Testing

This section details the experimental testing of the algorithm. We evaluate the running time of the algorithm as well as the number of distance calculations performed. We use the same setup and data sets as in Section 3.5 but we instead divide the running time by $n \log n$ and the number of distance calculations by n (Bentley and Shamos [1976] show that $P_C(n, k) \leq O(n)$). We also expect the algorithm to run faster on collinear pointsets compared to uniform pointsets since there are now at most two points within distance δ of P.

[2] This turns out to be for $n \geq 2,176,783$.

```
def closest_points_kd(pointsets, k_init, axes):
    [...Base cases...]
    mid = n // 2
    cut_axis = 0
    least = math.ceil(n/(4*k_init))
    c = int(4*(3**(k_init-1)))
    dist = int(k_init*c*(n**(1-1/k_init)))
    if n - least >= least + dist:
        max_m = 0
        for axis in range(k):
            for i in range(least+1, n-least):
                if i + dist > n-least:
                    break
                m = abs(pointsets[axis][i][axes[axis]] -
                        pointsets[axis][i+dist][axes[axis]])
                if m > max_m:
                    max_m = m
                    mid = i + dist//2
                    cut_axis = axis
    else:
        max_interval = 0
        for axis in range(k):
            axis_interval = abs(
                pointsets[axis][least][axes[axis]] - pointsets[axis][-least][axes[axis]])
            if axis_interval > max_interval:
                max_interval = axis_interval
                cut_axis = axis

    cut_axis_coord = axes[cut_axis]
    cut_axis_points = pointsets[cut_axis]

    [...Split pointsets, find closest pair in AB...]
    P = (cut_axis_points[mid][cut_axis_coord] +
         cut_axis_points[mid-1][cut_axis_coord]) / 2

    [...Project points onto P...]
    axes = [axis for axis in axes if axis != cut_axis_coord]
    other_pair = closest_points_kd(proj_points, k_init, axes)
    return other_pair if sq_distance(*other_pair, k_init) <= delta else best_pair
```

Figure 22: The final algorithm with some omitted parts

4.5.1 Results

Looking at Figure 23, we see that results for the running time are increasing when normalized by $n \log n$, while the distance calculations normalized by n for the same distribution in Figure 25 show a fairly flat graph. We see a similar pattern for the collinear pointsets in Figure 24 and Figure 26.

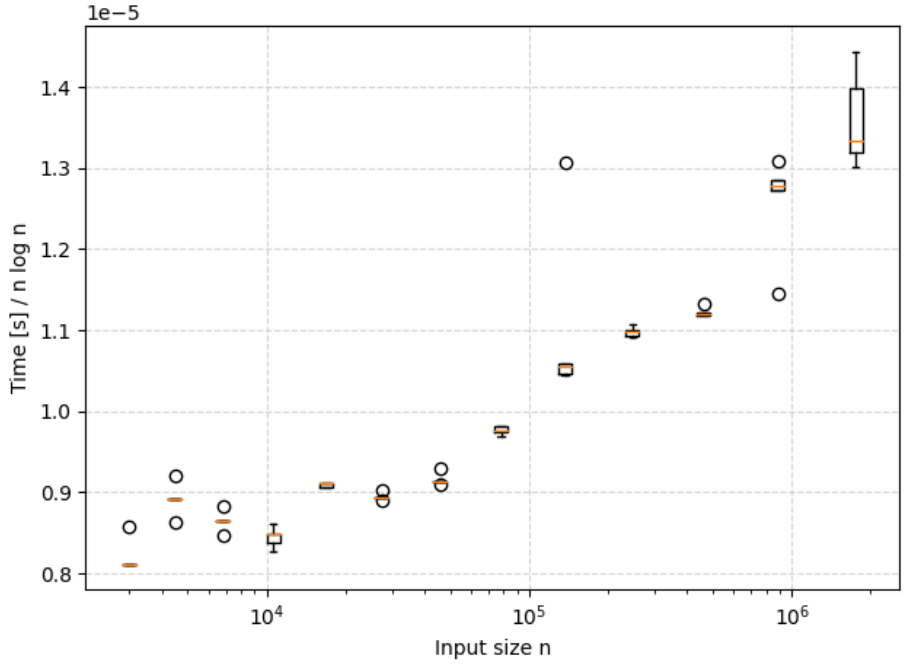


Figure 23: Running time for uniform points by $n \log n$

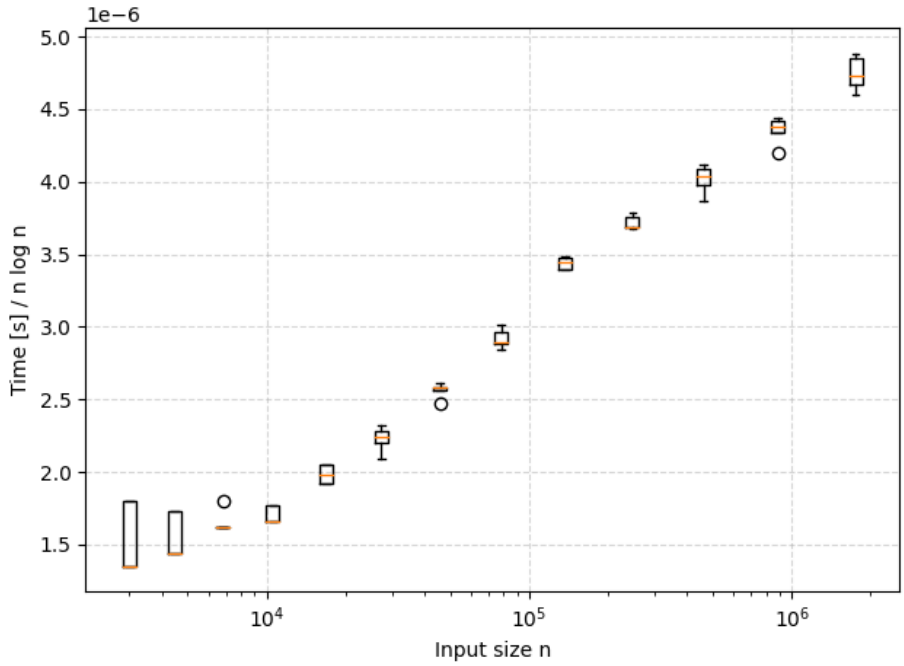


Figure 24: Running time for collinear points by $n \log n$

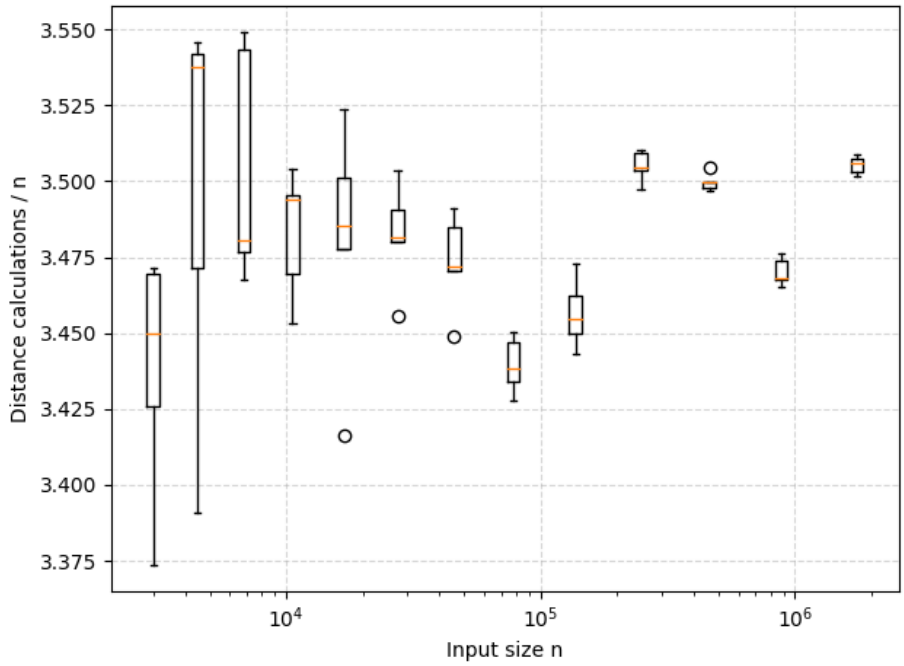


Figure 25: Distance calculations for uniform points by n

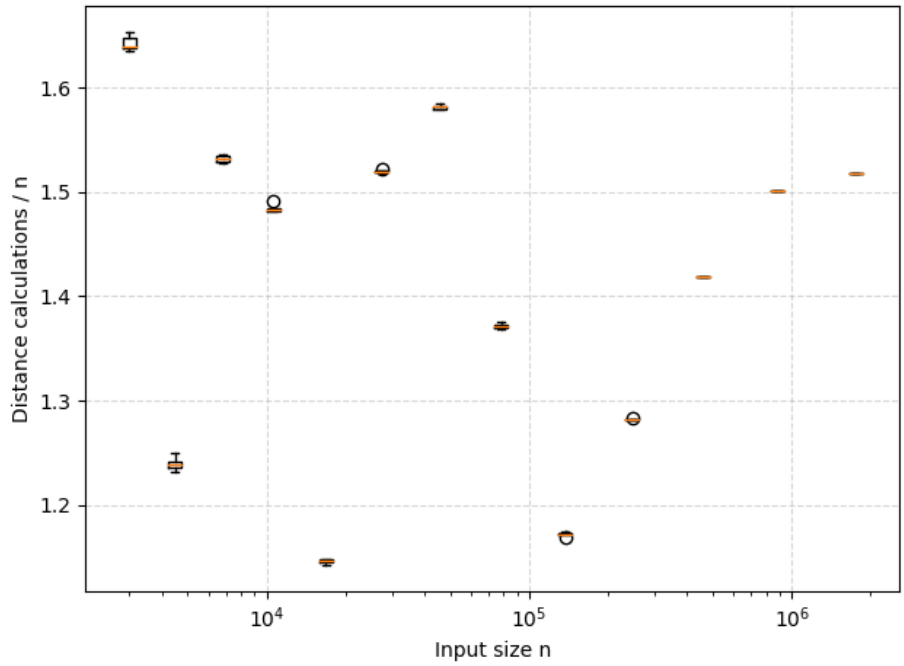


Figure 26: Distance calculations for collinear points by n

4.5.2 Discussion

The results in Figure 23 and 24 indicate that our algorithm runs slower than $O(n \log n)$. It is possible that the running time reaches a plateau as n increases

if we disregard the apparent noise in the measurements.

The results for the distance calculations in both cases seem to confirm that $P_C(n,k) \leq O(n)$ as noted by Bentley and Shamos [1976]. This suggests that much of the work done by the algorithm is spent on maintaining the pointsets in the recursion. This differs from the previous algorithms, where the distance calculations appeared to be the primary contributor to the time complexity, except for the results of the collinear datasets in Section 3.5. As discussed in Section 3.5.3, it is likely that the execution time measurements are unreliable due to other factors, such as overhead in Python lists. Nevertheless, our implementation of the algorithm appears to be slower than what is theoretically expected.

It is important to note that the input sizes of the pointsets in these experiments do not satisfy the inequality stated in 4.4.1, leading us to employ our own method for finding cut-planes. It would be interesting to see how the algorithm behaves for larger n, which would enable us to use the cut-plane method by Bentley and Shamos [1976]. This method guarantees sparsity in the $(k-1)$-dimensional subproblem, enabling the algorithm to achieve a time complexity of $O(n \log n)$.

5 Ideas for future work

This section highlights potential ideas for future work that build upon what we have discussed in the previous sections.

One potential path for future work is to test the running time of the optimized algorithm on large n as this allows us to find cut-planes using the method described by Bentley and Shamos [1976] rather than our own method. This was something we had planned to investigate, but unfortunately, due to time constraints, we were unable to pursue this part.

Another suggestion that, in hindsight, seems rather obvious is to evaluate the running time of the optimized algorithm on the 2-dimensional datasets and then compare the results with those obtained from the initial algorithm. This would require generating 2-dimensional collinear datasets again, as they are created in a manner that favors collinear points in terms of performance, as discussed in Section 2.3.5. It would be interesting to see whether the extra effort put into identifying a suitable cut-plane is worthwhile in the context of 2-dimensional data.

When considering potential factors that may be causing seemingly noisy data in our running time results, one suggestion is to analyze the cache performance during tests. Specifically, we can investigate whether potential cache misses have an impact on the observed behavior.

6 Conclusion

In conclusion, this report follows the developments presented in the paper by Bentley and Shamos [1976] and implements the corresponding algorithms. Starting with the 2D algorithm, which solves the problem in two dimensions in $O(n \log n)$, we progress towards the optimized version, which runs in $O(n \log n)$ replicating their findings and evaluating the practical performance of each stage. The analysis of the

results obtained from the first algorithm demonstrates a practical time complexity that aligns with its theoretical worst-case time complexity. However, the two algorithms for k-dimensions appear to have slower running times than theoretically expected. It is worth noting that the reliability of the running time results for the k-dimensional algorithms is questionable due to various factors such as the programming language used and the specific computing environment. Consequently, we rely more on the accuracy of the distance calculation results rather than the running time measurements. Moreover, the deviation between our implementation of the final algorithm and the algorithm presented by Bentley and Shamos [1976] could also be a contributing factor. Further investigation and testing on larger pointsets are, however, needed to confirm this.

Acknowledgements

We wish to thank our advisor Peyman Afshani for guiding us through the project during our many meetings and discussions.

References

Bentley, J. L. and Shamos, M. I. (1976). Divide-and-conquer in multidimensional space. In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, page 220–230, New York, NY, USA. Association for Computing Machinery.

Shamos, M. I. and Hoey, D. (1975). Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 151–162.

Appendix

There are some code changes compared to the ones in the figures. However, these changes are not important to the core of the algorithms.

Implementation of Closest Points on Plane

```
from util import *
import math

n_dist_calc = IntRef(0)

def closest_points(xsort, ysort, opt=True):
    n = len(xsort)
    if n == 2:
        return tuple(xsort)
    if n == 3:
        d01 = sq_distance(xsort[0], xsort[1])
        d12 = sq_distance(xsort[1], xsort[2])
        d20 = sq_distance(xsort[2], xsort[0])
        if d01 <= d12 and d01 <= d20:
            return xsort[0], xsort[1]
        elif d12 <= d01 and d12 <= d20:
            return xsort[1], xsort[2]
        else:
            return xsort[2], xsort[0]

    mid = n // 2
    xl, yl, xr, yr = split(xsort, ysort, mid)
    pl, ql = closest_points(xl, yl, opt)
    pr, qr = closest_points(xr, yr, opt)
    dl, dr = sq_distance(pl, ql), sq_distance(pr, qr)
    delta = min(dl, dr)
    sqrt_delta = math.sqrt(delta)
    best_pair = (pl, ql) if dl <= dr else (pr, qr)

    L = (xsort[mid][0] + xsort[mid - 1][0]) / 2
    candidates = [point for point in ysort if abs(L - point[0]) < sqrt_delta]

    if len(candidates) < 2:
        return best_pair

    if opt:
        for i, point in enumerate(candidates):
            j = i + 1
            while j < len(candidates) and abs(point[1] - candidates[j][1]) <
            ↪  sqrt_delta:
                d = sq_distance(point, candidates[j])
                if d < delta:
                    delta = d
                    sqrt_delta = math.sqrt(delta)
                    best_pair = (point, candidates[j])
                j += 1
    else:
        for i, point in enumerate(candidates):
```

```
            for j in range(1, 10):
                if i+j >= len(candidates):
                    continue
                d = sq_distance(point, candidates[i+j])
                if d < sqrt_delta:
                    delta = d
                    sqrt_delta = math.sqrt(delta)
                    best_pair = (point, candidates[i+j])
    return best_pair

def sq_distance(p, q):
    global n_dist_calc
    n_dist_calc.x += 1
    x_diff = p[0] - q[0]
    y_diff = p[1] - q[1]
    return x_diff * x_diff + y_diff * y_diff
```

Implementation of Closest Points in k-space

```
from util import *
import math

n_dist_calc = IntRef(0)


def closest_points_2d(pointsets, axis, k_init):
    xsort, ysort = pointsets
    n = len(xsort)
    if n == 2:
        return tuple(xsort)
    if n == 3:
        p0, p1, p2 = xsort
        d01 = sq_distance(p0, p1, k_init)
        d12 = sq_distance(p1, p2, k_init)
        d20 = sq_distance(p2, p0, k_init)
        if d01 <= d12 and d01 <= d20:
            return p0, p1
        elif d12 <= d01 and d12 <= d20:
            return p1, p2
        else:
            return p2, p0

    mid = n // 2
    pointsets_l, pointsets_r = split_kd(pointsets, 0, mid, k_init)
    pl, ql = closest_points_2d(pointsets_l, axis, k_init)
    pr, qr = closest_points_2d(pointsets_r, axis, k_init)
    dl, dr = sq_distance(pl, ql, k_init), sq_distance(pr, qr, k_init)
    delta = dl if dl <= dr else dr
    best_pair = (pl, ql) if dl <= dr else (pr, qr)
    sqrt_delta = math.sqrt(delta)

    L = (xsort[mid][axis] + xsort[mid - 1][axis]) / 2
    candidates = [point for point in ysort if abs(L - point[axis]) < sqrt_delta]

    if len(candidates) < 2:
```

```
37          return best_pair
38
39      for i, point in enumerate(candidates):
40          j = i + 1
41          while j < len(candidates) and abs(point[axis+1] - candidates[j][axis+1])
            ↪  < sqrt_delta:
42              d = sq_distance(point, candidates[j], k_init)
43              if d < delta:
44                  delta = d
45                  sqrt_delta = math.sqrt(delta)
46                  best_pair = (point, candidates[j])
47              j += 1
48
49      return best_pair
50
51
52  def closest_points_kd(pointsets, k_init, cut_axis=0):
53      cut_axis_points = pointsets[0]
54      k = len(pointsets)
55      n = len(cut_axis_points)
56
57      if k == 2:
58          return closest_points_2d(pointsets, cut_axis, k_init)
59      if n == 2:
60          return tuple(cut_axis_points)
61      if n == 3:
62          p0, p1, p2 = cut_axis_points
63          d01 = sq_distance(p0, p1, k_init)
64          d12 = sq_distance(p1, p2, k_init)
65          d20 = sq_distance(p2, p0, k_init)
66          if d01 <= d12 and d01 <= d20:
67              return p0, p1
68          elif d12 <= d01 and d12 <= d20:
69              return p1, p2
70          else:
71              return p2, p0
72
73      mid = n // 2
74      pointsets_l, pointsets_r = split_kd(pointsets, 0, mid, k_init)
75      pl, ql = closest_points_kd(pointsets_l, k_init, cut_axis)
76      pr, qr = closest_points_kd(pointsets_r, k_init, cut_axis)
77      dl, dr = sq_distance(pl, ql, k_init), sq_distance(pr, qr, k_init)
78      delta = dl if dl <= dr else dr
79      best_pair = (pl, ql) if dl <= dr else (pr, qr)
80      sqrt_delta = math.sqrt(delta)
81
82      P = (cut_axis_points[mid][cut_axis] +
83           cut_axis_points[mid-1][cut_axis]) / 2
84
85      # First check if there are atleast 2 points on opposite sides
86      nof_proj_points = 0
87      exists_left = False
88      exists_right = False
89
90      for point in pointsets_l[0]:
91          if abs(P - point[cut_axis]) < sqrt_delta:
```

```
            exists_left = True
            nof_proj_points += 1

    for point in pointsets_r[0]:
        if abs(P - point[cut_axis]) < sqrt_delta:
            exists_right = True
            nof_proj_points += 1

    if nof_proj_points < 2 or not(exists_left and exists_right):
        return best_pair

    # Next we basically pop the first axis
    # (recursion 1: pop x-axis, recursion 2: pop y-axis...)
    candidates = [[None] * (k_init + k - 1) for _ in range(n)]
    for axis, pointset in enumerate(pointsets):
        if axis == 0: continue
        i = 0
        for point in pointset:
            if abs(P - point[cut_axis]) < sqrt_delta:
                coords, iptrs = split_point(point, k_init)
                cut_iptr = iptrs[0]
                cut_point = candidates[cut_iptr]
                if cut_point[0] is None:
                    cut_point[:k_init] = coords
                cut_point[k_init+axis-1] = i
                i += 1

    proj_points = [[None] * nof_proj_points for _ in range(k-1)]
    for point in candidates:
        if point[0] is None: continue
        for axis, iptr in enumerate(point[k_init:]):
            proj_points[axis][iptr] = point

    other_pair = closest_points_kd(proj_points, k_init, cut_axis + 1)
    return other_pair if sq_distance(*other_pair, k_init) <= delta else best_pair

def sq_distance(p, q, k):
    global n_dist_calc
    n_dist_calc.x += 1
    return sum([(p[i] - q[i])*(p[i] - q[i]) for i in range(k)])
```

Implementation of Optimal Closest Points Algorithm

```
import math
from util import *
import time

n_dist_calc = IntRef(0)

def closest_points_2d(pointsets, axes, k_init):
    n = len(pointsets[0])
    if n == 2:
        return tuple(pointsets[0])
    if n == 3:
        p0, p1, p2 = pointsets[0]
```

```
        d01 = sq_distance(p0, p1, k_init)
        d12 = sq_distance(p1, p2, k_init)
        d20 = sq_distance(p2, p0, k_init)
        if d01 <= d12 and d01 <= d20:
            return p0, p1
        elif d12 <= d01 and d12 <= d20:
            return p1, p2
        else:
            return p2, p0

    mid = n // 2
    cut_axis = 0
    least = math.ceil(n/(4*k_init))
    c = int(4*(3**(k_init-1)))
    dist = int(k_init*c*(n**(1-1/k_init)))

    if n - least >= dist + least:
        max_m = 0
        for axis in range(2):
            for i in range(least+1, n-least):
                if i + dist > n-least:
                    break
                m = abs(pointsets[axis][i][axes[axis]] -
                        pointsets[axis][i+dist][axes[axis]])
                if m > max_m:
                    max_m = m
                    mid = i + dist//2
                    if cut_axis != axis:
                        cut_axis = axis
    else:
        max_interval = 0
        for axis in range(2):
            axis_interval = abs(
                pointsets[axis][least][axes[axis]] -
                ↪ pointsets[axis][-least][axes[axis]])
            if axis_interval > max_interval:
                max_interval = axis_interval
                cut_axis = axis

    cut_axis_points = pointsets[cut_axis]
    cut_axis_coord, other_axis_coord = axes[cut_axis], axes[1-cut_axis]

    pointsets_l, pointsets_r = split_kd(pointsets, cut_axis, mid, k_init)
    pl, ql = closest_points_2d(pointsets_l, axes, k_init)
    pr, qr = closest_points_2d(pointsets_r, axes, k_init)
    dl, dr = sq_distance(pl, ql, k_init), sq_distance(pr, qr, k_init)
    delta = dl if dl <= dr else dr
    best_pair = (pl, ql) if dl <= dr else (pr, qr)
    sqrt_delta = math.sqrt(delta)

    L = (cut_axis_points[mid][cut_axis_coord] +
         cut_axis_points[mid - 1][cut_axis_coord]) / 2
    candidates = [point for point in pointsets[1-cut_axis] if abs(L -
    ↪ point[cut_axis_coord]) < sqrt_delta]

    if len(candidates) < 2:
```

```
67          return best_pair
68
69      for i, point in enumerate(candidates):
70          j = i + 1
71          while j < len(candidates) and abs(point[other_axis_coord] -
            ↪  candidates[j][other_axis_coord]) < sqrt_delta:
72              d = sq_distance(point, candidates[j], k_init)
73              if d < delta:
74                  delta = d
75                  sqrt_delta = math.sqrt(delta)
76                  best_pair = (point, candidates[j])
77              j += 1
78      return best_pair
79
80  def closest_points_kd(pointsets, k_init, axes):
81      k = len(pointsets)
82      n = len(pointsets[0])
83
84      if k == 2:
85          return closest_points_2d(pointsets, axes, k_init)
86      if n == 2:
87          return tuple(pointsets[0])
88      if n == 3:
89          p0, p1, p2 = pointsets[0]
90          d01 = sq_distance(p0, p1, k_init)
91          d12 = sq_distance(p1, p2, k_init)
92          d20 = sq_distance(p2, p0, k_init)
93          if d01 <= d12 and d01 <= d20:
94              return p0, p1
95          elif d12 <= d01 and d12 <= d20:
96              return p1, p2
97          else:
98              return p2, p0
99
100     mid = n // 2
101     cut_axis = 0
102     least = math.ceil(n/(4*k_init))
103     c = int(4*(3**(k_init-1)))
104     dist = int(k_init*c*(n**(1-1/k_init)))
105
106     if n - least >= least + dist:
107         max_m = 0
108         for axis in range(k):
109             for i in range(least+1, n-least):
110                 if i + dist > n-least:
111                     break
112                 m = abs(pointsets[axis][i][axes[axis]] -
113                         pointsets[axis][i+dist][axes[axis]])
114                 if m > max_m:
115                     max_m = m
116                     mid = i + dist//2
117                     cut_axis = axis
118     else:
119         max_interval = 0
120         for axis in range(k):
121             axis_interval = abs(
```

```
122                 pointsets[axis][least][axes[axis]] -
                    ↪  pointsets[axis][-least][axes[axis]])
123             if axis_interval > max_interval:
124                 max_interval = axis_interval
125                 cut_axis = axis
126 
127     cut_axis_coord = axes[cut_axis]
128     cut_axis_points = pointsets[cut_axis]
129     pointsets_l, pointsets_r = split_kd(pointsets, cut_axis, mid, k_init)
130     pl, ql = closest_points_kd(pointsets_l, k_init, axes)
131     pr, qr = closest_points_kd(pointsets_r, k_init, axes)
132     dl, dr = sq_distance(pl, ql, k_init), sq_distance(pr, qr, k_init)
133     delta = dl if dl <= dr else dr
134     best_pair = (pl, ql) if dl <= dr else (pr, qr)
135     sqrt_delta = math.sqrt(delta)
136 
137     P = (cut_axis_points[mid][cut_axis_coord] +
138          cut_axis_points[mid-1][cut_axis_coord]) / 2
139 
140     nof_proj_points = 0
141     exists_left = False
142     exists_right = False
143 
144     for point in pointsets_l[0]:
145         if abs(P - point[cut_axis_coord]) < sqrt_delta:
146             exists_left = True
147             nof_proj_points += 1
148 
149     for point in pointsets_r[0]:
150         if abs(P - point[cut_axis_coord]) < sqrt_delta:
151             exists_right = True
152             nof_proj_points += 1
153 
154     if nof_proj_points < 2 or not (exists_left and exists_right):
155         return best_pair
156 
157     # project the points to a lower dimension by removing cut_axis
158     # we still keep the dimensionality of the points but remove the iptr
159     # corresponding to cut_axis
160     candidates =  [[None] * (k_init + k - 1) for _ in range(n)]
161     for axis, pointset in enumerate(pointsets):
162         if axis == cut_axis: continue
163         i = 0
164         for point in pointset:
165             if abs(P - point[cut_axis_coord]) < sqrt_delta:
166                 coords, iptrs = split_point(point, k_init)
167                 cut_iptr = iptrs[cut_axis]
168                 cut_point = candidates[cut_iptr]
169                 if cut_point[0] is None:
170                     cut_point[:k_init] = coords
171                 if axis > cut_axis:
172                     cut_point[k_init+axis-1] = i
173                 else:
174                     cut_point[k_init+axis] = i
175                 i += 1
176 
```

```
    proj_points = [[None] * nof_proj_points for _ in range(k-1)]
    for point in candidates:
        if point[0] is None: continue
        for axis, iptr in enumerate(point[k_init:]):
            proj_points[axis][iptr] = point

    axes = [axis for axis in axes if axis != cut_axis_coord]

    other_pair = closest_points_kd(proj_points, k_init, axes)
    return other_pair if sq_distance(*other_pair, k_init) <= delta else best_pair

def sq_distance(p, q, k):
    global n_dist_calc
    n_dist_calc.x += 1
    return sum([(p[i] - q[i])*(p[i] - q[i]) for i in range(k)])
```

Utility functions

```
class IntRef():
    def __init__(self, val=0):
        self.x = val

def binary_search(arr, key, axis):
    lo = 0
    hi = len(arr) - 1

    while hi - lo > 1:
        mid = (hi + lo) // 2
        if arr[mid][axis] < key[axis]:
            lo = mid + 1
        else:
            hi = mid

    if arr[lo][axis] == key[axis]:
        return lo
    elif arr[hi][axis] == key[axis]:
        return hi
    else:
        raise Exception("Key not found")

def find_element(arr, key, axis):
    indx = binary_search(arr, key, axis)
    i = 0
    while indx + i >= 0 and indx + i < len(arr):
        if arr[indx+i] == key:
            return indx + i
        elif arr[indx-i] == key:
            return indx - i
        i += 1
    raise Exception("Key not found")

def split(xsort, ysort, mid):
    xl = xsort[:mid]
    xr = xsort[mid:]
    temp_yl = [None] * len(ysort)
```

```
38      temp_yr = [None] * len(ysort)
39
40      for i, (x, y, j) in enumerate(xl):
41          temp_yl[j] = (x, y, i)
42
43      for i, (x, y, j) in enumerate(xr):
44          temp_yr[j] = (x, y, i)
45
46      yl = [p for p in temp_yl if p is not None]
47      yr = [p for p in temp_yr if p is not None]
48
49      for i, (x, y, j) in enumerate(yl):
50          xl[j] = (x, y, i)
51
52      for i, (x, y, j) in enumerate(yr):
53          xr[j] = (x, y, i)
54
55      return xl, yl, xr, yr
56
57  def split_kd(pointsets, cut_axis, mid, k_init):
58      cut_left = pointsets[cut_axis][:mid]
59      cut_right = pointsets[cut_axis][mid:]
60
61      pointsets_l, pointsets_r = [], []
62      for axis in range(len(pointsets)):
63          if axis == cut_axis:
64              pointsets_l.append(cut_left)
65              pointsets_r.append(cut_right)
66          else:
67              pointsets_l.append([None] * len(pointsets[0]))
68              pointsets_r.append([None] * len(pointsets[0]))
69
70      # update cut axis index pointer in other axes
71      set_axes_cut_iptr(pointsets_l, cut_axis, k_init)
72      set_axes_cut_iptr(pointsets_r, cut_axis, k_init)
73
74      # remove None-entries
75      pointsets_l = [[p for p in pointset if p is not None]
76                     for pointset in pointsets_l]
77      pointsets_r = [[p for p in pointset if p is not None]
78                     for pointset in pointsets_r]
79
80      # update index pointers in cut axis
81      set_cutaxis_iptrs(pointsets_l, cut_axis, k_init)
82      set_cutaxis_iptrs(pointsets_r, cut_axis, k_init)
83
84      # update index pointers for the other axes using pointers in cut axis
85      set_axes_iptrs(pointsets_l, cut_axis, k_init)
86      set_axes_iptrs(pointsets_r, cut_axis, k_init)
87
88      return pointsets_l, pointsets_r
89
90  def set_axes_cut_iptr(pointsets, cut_axis, k_init):
91      for i, point in enumerate(pointsets[cut_axis]):
92          for axis, iptr in enumerate(get_iptrs(point, k_init)):
93              if axis == cut_axis: continue
```

```
            pointsets[axis][iptr] = point.copy()
            pointsets[axis][iptr][k_init+cut_axis] = i

def set_cutaxis_iptrs(pointsets, cut_axis, k_init):
    for axis, pointset in enumerate(pointsets):
        if axis == cut_axis: continue
        for i, point in enumerate(pointset):
            cut_iptr = point[k_init+cut_axis]
            cut_point = pointsets[cut_axis][cut_iptr]
            cut_point[k_init+axis] = i


def set_axes_iptrs(pointsets, cut_axis, k_init):
    for axis, pointset in enumerate(pointsets):
        if axis == cut_axis: continue
        for point in pointset:
            cut_iptr = point[k_init+cut_axis]
            cut_point = pointsets[cut_axis][cut_iptr]
            point[k_init:] = get_iptrs(cut_point, k_init)
            point[k_init+cut_axis] = cut_iptr

def get_iptrs(point, k_init):
    return point[k_init:]

def split_point(point, k_init):
    return point[:k_init], point[k_init:][fontsize=\footnotesize,
    ↪  linenos]{Python}
```

Dataset Generation Code

```
import random
import math
import util
import numpy as np
import copy

def generate_collinear_2d_pointset(size):
    points = [(0, random.randrange(-size*10, size*10)) for _ in range(size)]
    points = list(set(points))

    y_sorted = sorted(points, key=lambda p: p[1])
    y_sorted_lbl = [(*pos, i) for i, pos in enumerate(y_sorted)]
    x_sorted_lbl = y_sorted_lbl.copy()

    return x_sorted_lbl, y_sorted_lbl

def generate_uniform_2d_pointset(size):
    points = [(random.randrange(-size*10, size*10),
               random.randrange(-size*10, size*10)) for _ in range(size)]
    points = list(set(points))

    x_sorted = sorted(points, key=lambda p: p[0])
    y_sorted = sorted(points, key=lambda p: p[1])

    x_sorted_lbl = [None] * len(points)
    y_sorted_lbl = [None] * len(points)
```

```
27
28      for i, pos in enumerate(x_sorted):
29          y_ptr = util.find_element(y_sorted, pos, axis=1)
30          x_sorted_lbl[i] = (*pos, y_ptr)
31
32      for i, pos in enumerate(y_sorted):
33          x_ptr = util.find_element(x_sorted, pos, axis=0)
34          y_sorted_lbl[i] = (*pos, x_ptr)
35
36      return x_sorted_lbl, y_sorted_lbl
37
38  def generate_uniform_3d_pointset(size):
39      points = [(random.randrange(-size*10, size*10),
40                 random.randrange(-size*10, size*10),
41                 random.randrange(-size*10, size*10)) for _ in range(size)]
42
43      points = list(set(points))
44
45      x_sorted = sorted(points, key=lambda p: p[0])
46      y_sorted = sorted(points, key=lambda p: p[1])
47      z_sorted = sorted(points, key=lambda p: p[2])
48
49      x_sorted_lbl = [None] * len(points)
50      y_sorted_lbl = [None] * len(points)
51      z_sorted_lbl = [None] * len(points)
52
53      for i, pos in enumerate(x_sorted):
54          y_ptr = util.find_element(y_sorted, pos, axis=1)
55          z_ptr = util.find_element(z_sorted, pos, axis=2)
56          x_sorted_lbl[i] = [*pos, i, y_ptr, z_ptr]
57
58      for i, pos in enumerate(y_sorted):
59          x_ptr = util.find_element(x_sorted, pos, axis=0)
60          z_ptr = util.find_element(z_sorted, pos, axis=2)
61          y_sorted_lbl[i] = [*pos, x_ptr, i, z_ptr]
62
63      for i, pos in enumerate(z_sorted):
64          x_ptr = util.find_element(x_sorted, pos, axis=0)
65          y_ptr = util.find_element(y_sorted, pos, axis=1)
66          z_sorted_lbl[i] = [*pos, x_ptr, y_ptr, i]
67
68      return x_sorted_lbl, y_sorted_lbl, z_sorted_lbl
69
70  def generate_collinear_3d_pointset(size):
71      points = [(0, 0, random.randrange(-size*10, size*10), 0, 0, 0) for _ in
        ↪  range(size)]
72
73      points = list(map(list, set(points)))
74
75      x_sorted = points
76      for i, p in enumerate(x_sorted):
77          p[3] = i
78
79      y_sorted = points
80      random.shuffle(y_sorted)
81      for i, p in enumerate(y_sorted):
```

```
82          p[4] = i
83
84      z_sorted = points
85      z_sorted.sort(key=lambda p: p[2])
86      for i, p in enumerate(z_sorted):
87          p[5] = i
88
89      x_sorted_lbl = [None] * len(points)
90      for p in x_sorted:
91          x_sorted_lbl[p[3]] = p.copy()
92
93      y_sorted_lbl = [None] * len(points)
94      for p in y_sorted:
95          y_sorted_lbl[p[4]] = p.copy()
96
97      z_sorted_lbl = [None] * len(points)
98      for p in z_sorted:
99          z_sorted_lbl[p[5]] = p.copy()
100
101     return x_sorted_lbl, y_sorted_lbl, z_sorted_lbl
102
103 def test_iptr_correctness_3d(xs, ys, zs):
104     for i, p in enumerate(xs):
105         assert p[3] == i
106         assert p[:3] == ys[p[4]][:3]
107         assert p[:3] == zs[p[5]][:3]
108
109     for i, p in enumerate(ys):
110         assert p[4] == i
111         assert p[:3] == xs[p[3]][:3]
112         assert p[:3] == zs[p[5]][:3]
113
114     for i, p in enumerate(zs):
115         assert p[5] == i
116         assert p[:3] == xs[p[3]][:3]
117         assert p[:3] == ys[p[4]][:3]
118
119 def generate_files(min_points, max_points, number_of_duplicates, dim,
    ↪ distribution):
120     i = min_points
121     while i < max_points:
122         print(
123             f"Generating {number_of_duplicates} {distribution} pointsets in
                ↪ {dim}d with: {i} points")
124         for j in range(number_of_duplicates):
125             outfile = f"test_points/{dim}d_{distribution}/{str(i)}-{str(j)}"
126             points = None
127             if distribution == "collinear":
128                 if dim == 2:
129                     points = generate_collinear_2d_pointset(i)
130                 if dim == 3:
131                     points = generate_collinear_3d_pointset(i)
132             elif distribution == "uniform":
133                 if dim == 2:
134                     points = generate_uniform_2d_pointset(i)
135                 if dim == 3:
```

```
                    points = generate_uniform_3d_pointset(i)
            else:
                raise Exception("Unknown distribution: " + distribution)
            if dim == 2:
                np.save(outfile, np.array(points, dtype=object),
                ↪  allow_pickle=True)
            else:
                np.save(outfile, np.array(points))

        i = math.floor(i**1.05)

def generate(distribution, dim):
        #generate_files(100, 500000, 10, dim, distribution)
        generate_files(3000, 10000000, 5, dim, distribution)
        #generate_files(32000, 500000, 10, dim, distribution)

        # Continuation of min points = 3000
        #generate_files(3600246, 10000000, 5, dim, distribution)

#generate("collinear", 3)
#generate("uniform", 3)
```

Benchmarking Code

```
import time
import os
import matplotlib.pyplot as plt
import numpy as np
from algorithm1 import closest_points as algorithm1, n_dist_calc as
↪  algo1_n_dist_calc
from algorithm2 import closest_points_kd as algorithm2, n_dist_calc as
↪  algo2_n_dist_calc
from algorithm3 import closest_points_kd as algorithm3, n_dist_calc as
↪  algo3_n_dist_calc

def get_file_info(file):
    n, id = file.split("-")
    return int(n), int(id.split(".")[0])

def process_files(path, name, max_duplicates, opt):
    clock_info = time.get_clock_info('process_time')

    nof_points = []
    running_times = []
    dist_calcs = []
    files = sorted(os.listdir(path), key=lambda f: get_file_info(f)[0])

    prev_n = 0
    i = -1
    for file in files:
        n, id = get_file_info(file)
        if id >= max_duplicates:
            continue
        if prev_n != n:
```

```
28              nof_points.append(n)
29              running_times.append([])
30              dist_calcs.append([])
31              i += 1
32
33          print(f"Timing {n} points with {name},
            ↪  time_resolution={clock_info.resolution}")
34          if name == "algorithm1":
35              xsort, ysort = np.load(path + file, allow_pickle=True).tolist()
36              xsort, ysort = list(map(tuple, xsort)), list(map(tuple, ysort))
37
38              algo1_n_dist_calc.x = 0
39              start = time.process_time()
40              _ = algorithm1(xsort, ysort, opt=opt)
41              end = time.process_time()
42
43              dist_calc = algo1_n_dist_calc.x
44
45          elif name == "algorithm2":
46              pointsets = np.load(path + file, allow_pickle=True).tolist()
47
48              algo2_n_dist_calc.x = 0
49              start = time.process_time()
50              _ = algorithm2(pointsets, 3)
51              end = time.process_time()
52
53              dist_calc = algo2_n_dist_calc.x
54
55          elif name == "algorithm3":
56              pointsets = np.load(path + file, allow_pickle=True).tolist()
57
58              axes = [0,1,2]
59              algo3_n_dist_calc.x = 0
60              start = time.process_time()
61              _ = algorithm3(pointsets, 3, axes)
62              end = time.process_time()
63
64              dist_calc = algo3_n_dist_calc.x
65
66          dist_calcs[i].append(dist_calc)
67          running_times[i].append((end - start))
68          prev_n = n
69
70      return nof_points, running_times, dist_calcs
71
72
73  def nlogn(n):
74      return n * np.log2(n)
75
76  def nlogsqn(n):
77      return n * np.log2(n) ** 2
78
79  def test(name, max_duplicates, opt=True):
80      if name == "algorithm1":
81          dim = "2d"
82          prefix = "opt_" if opt else "no_opt_"
```

```
83      else:
84          dim = "3d"
85          prefix = ""
86
87      path = f"test_points/{dim}_collinear/"
88      nof_points, running_times, dist_calcs = process_files(
89          path, name, max_duplicates, opt)
90      np.save(f"test_results/{name}/{prefix}collinear_nof_points",
        ↪  np.array(nof_points))
91      np.save(f"test_results/{name}/{prefix}collinear_runtime",
        ↪  np.array(running_times))
92      np.save(f"test_results/{name}/{prefix}collinear_dist_calcs",
        ↪  np.array(dist_calcs))
93      print(f"Saved results for {dim} collinear test on {name}")
94
95      path = f"test_points/{dim}_uniform/"
96      nof_points, running_times, dist_calcs = process_files(
97          path, name, max_duplicates, opt)
98      np.save(f"test_results/{name}/{prefix}uniform_nof_points",
        ↪  np.array(nof_points))
99      np.save(f"test_results/{name}/{prefix}uniform_runtime",
        ↪  np.array(running_times))
100     np.save(f"test_results/{name}/{prefix}uniform_dist_calcs",
        ↪  np.array(dist_calcs))
101     print(f"Saved results for {dim} uniform test on {name}")
102
103
104 def flatten(l):
105     return [item for sublist in l for item in sublist]
106
107 def boxplot(name, result_type, complexity, opt=True):
108     if name == "algorithm1":
109         prefix = "opt_" if opt else "no_opt_"
110     else:
111         prefix = ""
112
113     if complexity == "nlogn":
114         suffix = "n log n"
115     elif complexity == "nlogsqn":
116         suffix = "n log^2 n"
117     else:
118         suffix = "n"
119
120     if result_type == 'runtime':
121         ylabel = f'Time [s] / {suffix}'
122     else:
123         ylabel = f'Distance calculations / {suffix}'
124
125     fig_collinear, ax_collinear = plt.subplots()
126     nof_points = np.load(f"test_results/{name}/{prefix}collinear_nof_points.npy")
127     result = np.load(f"test_results/{name}/{prefix}collinear_{result_type}.npy")
128
129     for i in range(len(result)):
130        n = nof_points[i]
131        result[i] = np.divide(result[i], n*np.log2(n)**2)
132
```

```
133     def width(p, w): return 10**(np.log10(p)+w/2.)-10**(np.log10(p)-w/2.)
134     ax_collinear.boxplot(result.tolist(), positions=nof_points.tolist(),
        ↪  widths=width(nof_points, 0.05))
135     ax_collinear.set_xscale("log")
136     ax_collinear.set_xlabel("Input size n")
137     ax_collinear.set_ylabel(ylabel)
138     title = f"{result_type}_{prefix}collinear_{complexity}"
139     fig_collinear.tight_layout()
140     ax_collinear.grid(alpha=0.5, linestyle='--')
141     fig_collinear.savefig(f'figures/{name}/' + title + ".png")
142 
143     fig_uniform, ax_uniform = plt.subplots()
144     nof_points = np.load(f"test_results/{name}/{prefix}uniform_nof_points.npy")
145     result = np.load(f"test_results/{name}/{prefix}uniform_{result_type}.npy")
146 
147     # for i in range(len(result)):
148     #     n = nof_points[i]
149     #     result[i] = np.divide(result[i], n*np.log2(n)**2)
150 
151     ax_uniform.boxplot(result.tolist(), positions=nof_points.tolist(),
        ↪  widths=width(nof_points, 0.05), vert=True)
152     ax_uniform.set_xscale("log")
153     ax_uniform.set_xlabel("Input size n")
154     ax_uniform.set_ylabel(ylabel)
155     title = f"{result_type}_{prefix}uniform_{complexity}"
156     ax_uniform.grid(alpha=0.5, linestyle='--')
157     fig_uniform.tight_layout()
158     fig_uniform.savefig(f'figures/{name}/' + title + ".png")
159 
160     plt.show()
161 
162 test("algorithm3", 5)
163 
164 #boxplot(name='algorithm2', result_type='dist_calcs', complexity="nlogsqn")
```

Algorithm Correctness Test Code

```
1  import random
2  import numpy as np
3  import copy
4  from util import find_element
5  
6  def sq_distance(p, q, k_init):
7      return sum([(p[i] - q[i])*(p[i] - q[i]) for i in range(k_init)])
8  
9  def brute_force(pointset, k_init):
10     delta = 1000000000
11     for p1 in pointset:
12         for p2 in pointset:
13             if p1 == p2:
14                 continue
15             delta = min(delta, sq_distance(p1, p2, k_init))
16     return delta
17 
18 def basic_test_2d():
```

```
    for _ in range(100):
        size = 100
        pointset = [(random.randrange(-size, size),
                     random.randrange(-size, size)) for _ in range(size)]
        pointset = list(set(pointset))
        xs = sorted(pointset, key=lambda p: p[0])
        ys = sorted(pointset, key=lambda p: p[1])
        xs_lbl, ys_lbl = [], []

        for p in xs:
            xs_lbl.append((*p, find_element(ys, p, 1)))

        for p in ys:
            ys_lbl.append((*p, find_element(xs, p, 0)))

        p, q = algo1(xs_lbl, ys_lbl)
        assert np.isclose(sq_distance(p, q, 2), brute_force(pointset, 2))

    print("2D test passed")

def basic_test_kd(k, n, runs):
    for _ in range(runs):
        pointset = [(tuple(random.randrange(-n*10, n*10)
                     for _ in range(k))) for _ in range(n)]
        pointset = list(set(pointset))
        points_sorted = [sorted(pointset, key=lambda p: p[i])
                         for i in range(k)]
        points_labeled = []

        for i, pointset in enumerate(points_sorted):
            pointset_label = []
            for p in pointset:
                pointset_label.append(
                    [*p, *tuple(find_element(points_sorted[j], p, j) if j != i
                    ↪  else i for j in range(k))])
            points_labeled.append(pointset_label)

        res = brute_force(pointset, k)
        p, q = algo2(copy.deepcopy(points_labeled), k)
        assert np.isclose(sq_distance(p, q, k), res)

        p, q = algo3(points_labeled, k, list(range(k)))
        assert np.isclose(sq_distance(p, q, k), res)

    print(f"{k}D test passed")

basic_test_2d()
basic_test_kd(k=2, n=100, runs=100)
basic_test_kd(k=3, n=100, runs=100)
basic_test_kd(k=4, n=100, runs=100)
basic_test_kd(k=5, n=100, runs=40)
basic_test_kd(k=6, n=100, runs=40)
```