The Closest Two Points Problem

Ahmad H. Saleh - 201709284
Frederik K. S. Nielsen - 201705351
Thomas U. Hansen - 201709065

Advisor
Peyman Afshani

Bachelor report (15 ECTS) in Computer Science
Department of Computer Science, Aarhus University
June 08, 2020

Abstract

In computational geometry there exists a $O(n \log(n))$ divide-and-conquer solution, to solving the closest points problem in any dimensional space as shown in the paper Bentley and Shamos 1976. In this project, we produce algorithms for solving the problem in 2 and 3 dimensional space as per Bentley and Shamos 1976, with the intend of analysing them in practice.

We initially see that our problem can be solved in any dimensional space using a brute force solution, which takes $\Theta(N^2)$ time. This can be improved by adapting a divide-and-conquer approach, which we then see that reduces the upper bound to $O(N \log^{k-1} N)$ initially, but by further improvements we can reduce it to $O(N \log N)$

In particular, we implement 2 algorithms for solving the problem in 2-dimensional space and 3 algorithms for solving the problem in 3-dimensional space. The first in 2-dimensional space being a basic version, which in theory runs in $O(N \log(N))$, secondly an improved version, which in theory runs the same, but chooses a good hyperplane cut. For 3-dimensional space, the first basic version, which in theory runs in $O(N \log^{k-1}(N))$. Secondly the semi-improved version, which in theory runs in $O(N \log^2(N))$, and thirdly the improved version, which in theory runs in $O(N \log(N))$.

Our analysis reveals several things of importance. The basic 2-dimensional version of the algorithms does less distance comparisons than expected in practice. For the 2-dimensional improved versions of our algorithms, the time spent finding good hyperplane cuts takes long enough time that the cost outweighs the benefit. The 3-dimensional versions work at their expected running time. We found that the basic version in practice has better running time than the semi-improved and improved versions for smaller data sets, we round off by hypothesizing that for far bigger data sets the semi-improved and improved versions would be better.

Contents

1 Introduction

The closest points problem deals with finding the two closest points of a set of N points in k-dimensional space. Naively this can be solved by looking through every possible pair of points and simply keeping track of the two points with the shortest distance according to some distance metric, this brute-force algorithm would have a running time of $\Theta(n^2)$. We can do better than this, Bentley and Shamos 1976 introduces a divide-and-conquer strategy that solves the closest points problem in any k-space with an upper bound of $O(N \log N)$ running time.

We show that this problem can be solved in k-dimensional space, and from this we implement the problem in 2-dimensional space, expanding it to 3-dimensional space using the same divide-and-conquer technique, with the purpose that this covers the steps towards solving the problem in any dimension with a running time of $O(N \log(N))$. When expanding to 3-dimensional space, an additional divide-and-conquer method is introduced to find a solution, namely sparse fixed-radius-near-neighbor covered by Bentley and Shamos 1976, which solves the problem that occur when the subproblem of (k-1)-dimension is not of 1-dimensional space. It works by finding pairs within a given distance of each other, which then can be iterated to find the closest pair.

At first we implement algorithms that solves the problem in 2-dimensional space. We call the first 2d basic, this solves the 2-closest points problem in 2-dimensional space with a theoretical running time of $O(N \log(N))$. An improvement is made to this algorithm by choosing a hyperplane cut that limits the amount of points close to the hyperplane, we call this 2d improved. Now expanding the problem into 3-dimensional space, we implement a basic version as before. This works similar to the one in 2-dimensional space, by choosing the hyperplane cut to split space in two half-spaces on the X-axis, and was found in theory to have a running time of $O(N \log^{k-1}(N))$. The semi-improved version makes an improvement to the sparse fixed-radius-near-neighbor part, where the chosen hyperplane cut in this problem gives an upper bound on the amount of points close to the hyperplane. This makes the theoretical running time of the semi-improved version $O(N \log^2(N))$. At last the 3d improved version extends the semi-improved with the addition that it chooses a good hyperplane cut when splitting the points.

After implementing the algorithms to solve the closest points problem, we hypothesise that these algorithms running time compares to the theoretical running time introduced by Bentley and Shamos 1976, this is investigated using an experimental approach. This procedure also allows for a practical analysis of our algorithms, where we can examine if improvements to the algorithms are visible in practice. Both by comparing how the algorithms runs on different data sets and how they compare to each other.

When hypothesizing, we have limited time to analyse, therefore when a hypothesis is wrong, we can not always pursue the reason for this and make another hypothesis. For this reason we can expect a few loose ends, where we might have been able to investigate further if we had more time and data.

2 Review of Literature

In the paper by Bentley and Shamos 1976, they show efficient solutions to three of the closest points problems in k-dimensions, the three problems being, 2-closest points, sparse fixed-radius-near-neighbor, and all-closest points problem.

In this paper, we will make use of the proof and implement the 2-closest points, which can be solved in $O(N \log N)$ time, by following the solution from Bentley and Shamos 1976. In this review we will give an overview of the different proofs required to reach the solution of closest points problem in a running time of $O(N \log N)$.

Introduction to the 2-closest points problem

The problem we are studying in this paper is the 2-closest points problem, given a collection of points, which two points are the closest to each other.

Proof of lower bound of the running time

First we want to prove, that we have a lower bound on running time to solve this problem. We note, that if we solve 2-closest points problem, we can also solve point uniqueness problem, by examining if the distance between the two closest points are 0 or not. Therefore 2-closest points problem is at least as hard to solve as point uniqueness. We note, that point uniqueness has a lower bound running time $O(N \log N)$, and therefore so does 2-closest points problem (Shamos and Hoey 1975).

Solution to the 2-closest points problem in the plane

To solve the 2-closest points problem in the plane, we can use a straightforward divide-and-conquer method. We make a hyperplane (a line) on our plane, splitting our points up in two half-spaces, and then solve for each half, and then for the pairs crossing the split. Now the recurrence relation will look as follows

$$P(N, 2) = 2P\left(\frac{N}{2}, 2\right) + O\left(N^2\right)$$

Where $P(N, k)$ is the minimax complexity of the closest points problem, N is the number of points, and k is the dimension. Using the master theorem for divide-and-conquer recurrences, we get that this has a running time of $O\left(N^2\right)$. Though we can decrease the $O\left(N^2\right)$ factor, which comes from comparing all the pairs who crosses the hyperplane, by reducing the amount of pairs we measure. Let δ_A and δ_B be the respective minimal distances between a pair points in the two half-spaces, and let $\delta = \min(\delta_A, \delta_B)$, then we can ignore all points which are more than δ distance away from the hyperplane.

We call the area that is within δ distance away from the hyperplane the "slab". We know that each point can at most be δ distance away from each other on each side. This feature is known as sparsity, within any δ-radius ball, there will be at most a constant amount of points in that ball $c \geq 1$. Then by projecting our points in the slab onto the hyperplane, and sorting our points, we can give an upper bound to the amount of points we need to compare each point to, since we are given sparsity.

Now we know, that each point from each side, will at most need to be compared to a constant amount of points, and therefore our upper bound on comparing pairs crossing the hyperplane turns to $O(N)$ instead, and our formula looks like:

$$P(N, 2) = 2P\left(\frac{N}{2}, 2\right) + O(N)$$

Which we can calculate the upper running time with the master theorem, which shows a running time of $O(n \log n)$. This proof comes from the Bentley and Shamos 1976 paper. How this can be implemented is shown in the Section 3. Implementation of Closest Points in the Plane.

Expanding 2-closest points to k-dimensions

Expanding this problem to k-dimensions requires an alternative way to solve for the slab, since we no longer can sort our points on the projection, as the projection is no longer only 1 dimension, but $k-1$ dimensions.

Instead we search for all the pairs within the projection, which are closer than δ distance from each other. This problem is called sparse fixed-radius-near-neighbor, and we denote its minimax complexity as $S(n,k)$. This allows us to write our running time of closestpoints in k dimensions as:

$$P(N,k) = 2P\left(\frac{N}{2},k\right) + S(N,k-1)$$

Solving sparse fixed-radius-near-neighbor

Solving sparse fixed-radius-near-neighbor can also be solved through divide-and-conquer, partitioning in two half-spaces, and solving for the pairs with one point in each half. Similarly to 2-closest points we make a hyperplane, which partitions our data, then we make a slab with all points within δ distance from the hyperplane.

To solve for the slab, we project the points onto our hyperplane, and run sparse fixed-radius-near-neighbor in one dimension less. Though we require an additional check, that our pairs are within the radius outside of the projection. We keep going one down in dimension until we end with 1 dimension, where finding the pairs takes $O(N)$ time, since we keep the sparsity from our original space, which can be found to $c \leq 4\left(3^{k-1}\right)$. The recurrence relation of this implementation of sparse fixed-radius-near-neighbor will be

$$S(N,k) = 2S\left(\frac{N}{2},k\right) + S(N,k-1) + O(N)$$

Where O(N) is the overhead in partitioning our points. We can then through induction on k find the running time of $S(N,k) \leq O\left(N\log^{k-1}(N)\right)$. Using this implementation to calculate $P(N,k)$ gives us

$$P(N,k) \leq 2P\left(\frac{N}{2},k\right) + S(N,k-1) \leq 2P\left(\frac{N}{2},k\right) + O\left(N\log^{k-2}(N)\right)$$

Which through the master theorem tells us that the running time is $P(N,k) \leq O\left(N\log^{k-1}(N)\right)$. The proof comes from Bentley and Shamos 1976, and an implementation of it in 3 dimensions will be discussed in Section 4. Implementation of Closest Points in Higher Dimensions.

Improving running time of sparse fixed-radius-near-neighbor

The running time can then be improved through carefully choosing where to partition our collection of points. We first look at sparse fixed-radius-near-neighbor, and show how to improve its running time to $O(n\log(n))$. We need to find a hyperplane cut, which is perpendicular to one of the original coordinate axes, where there is at least $\frac{N}{4k}$ in each sub-collection, and there are at most $kcN^{1-1/k}$ points within δ distance from the hyperplane cut. This hyperplane cut can be proven to exist through contradiction, by showing, that if this hyperplane cut doesn't exist, then our collection will not be sparse, which we know it is. A more formal proof can be seen in Bentley and Shamos 1976.

When using this hyperplane cut, the recurrence relation will change from this

$$S(N,k) = 2S\left(\frac{N}{2},k\right) + S(N,k-1) + O(N)$$

to this

$$S(N,k) = S\left(\frac{N}{4k},k\right) + S\left(N\left(1-\frac{1}{4k},k\right)\right) + O(kN) + S\left(kcN^{1-1/k},k-1\right)$$

Which can be proven to be $S(N,k) \leq O(N\log N)$, through induction on k, and the fact:
$S(N,k-1) \leq O(N\log N) \Rightarrow S\left(kcN^{1-1/k}\right) \leq O(N)$, and with an induction basis in $S(N,2) \leq O\left(N\log^{2-1}N\right) = O(N\log N)$.

With this implementation of sparse fixed-radius-near-neighbor, we can improve the recurrence relation of 2-closest points to

$$P(N,k) = 2P\left(\frac{N}{2},k\right) + S(N,k-1) \leq 2P\left(\frac{N}{2},k\right) + O(N\log N)$$

Using master theorem on this recursion, we get a running time of $P(N,k) \leq O\left(N\log^2 N\right)$.

Reducing running time of 2-closest points in higher dimensions

To reduce the worst case running time of 2-closest points problem in higher dimensions, we can try to find a good hyperplane cut again. We just need to ensure that, each sub-collection has at least $\frac{N}{4k}$ points, and after we find δ, there should at most be $kcN^{1-1/k}$ points within δ distance from the hyperplane cut.

This hyperplane cut can be found through a linear scan of our points on each axis in a sorted order, starting from the $\frac{N}{4k}$'th point, where we measure the "axis" distance of 2 points with $kcN^{1-1/k}$ points in between, and save the largest distance seen so far.

When we have done this for each axis, we choose the plane perpendicular to the axis which had the largest distance, which crosses the middle of the 2 points, which made up the largest distance, to be our hyperplane cut. It can then be shown, that this distance is greater than 2δ, and therefore will there at most be $kcN^{1-1/k}$ points in the slab.

Using this hyperplane cut, we can then rewrite the recurrence relation to

$$P(N,k) = P\left(\frac{N}{4k},k\right) + P\left(1-\frac{N}{4k},k\right) + S\left(kcN^{1-1/k},k\right) + O(N)$$

This can then be shown to have an upper bound running time of: $P(N,k) \leq O(N\log N)$. These proofs all come from Bentley and Shamos 1976. Description of how this improved version of 2-closest points in 3-dimensional space can be implemented will be shown in Section 4. Implementation of Closest Points in Higher Dimensions.

The upper bound of comparisons

Bentley and Shamos 1976 gives an upper bound on the number of interpoint distance calculations in the execution of the algorithms. Let $P_C(N,k)$ and $S_C(N,k)$ denote the worst case number of pointwise comparisons for the best algorithm, then Bentley and Shamos 1976 theorem 10 states that $P_C(N,k) \leq O(N)$ and $S_C(N,k) \leq O(N)$. The proof goes by induction, clearly with presorting we have $P_C(N,1) \leq O(N)$ and $S_C(N,1) \leq O(N)$, now assume that $S_C(N,k-1) \leq O(N)$, then we get the following recurrence relation

$$\begin{aligned} S_C(N,k) &\leq S_C(N/4k,k) + S_C(N(1-1/4k),k) + S_C(kcN^{1-1/k},k-1) \\ &= S_C(N/4k,k) + S_C(N(1-1/4k),k) + O(N^{1-1/k}) \end{aligned}$$

Which gives $S_C(N,k) = O(N)$. An analogous argument would show the same for $P_C(N,k) = O(N)$

3 Implementation of Closest Points in the Plane

In this section our implementation of the Bentley and Shamos solution to the closest points problem in the plane is presented in detail.

3.1 Implementing Closest Points, Basic Version

Here we present the implementation details of Closest Points in the plane where we do not attempt to find a good cut for the hyperplane. The exact source code can be seen in Appendix 10.2.4. Closest points 2D Implementation.

We use several helper classes to describe the spatial properties of a Euclidean plane. Point2D is as the name implies a 2-dimensional tuple (x, y) that describes a specific point in 2D space, in our implementation we use Euclidean distance calculated as $\sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}$ for two points $(p_1, p_2), (q_1, q_2)$ in the plane. The implementation code for Point2D can be found in Appendix 10.2.1. Point 2D. Note that Bentley and Shamos 1976 used the maximum coordinate metric L_∞ to measure the distance between two points in their proofs, but since Euclidean distance is an L^p metric this will only change the time complexity of the algorithm by a multiplicative constant. We also added a helper class that describes a pair of points in the plane, our algorithms will be outputting objects of this type as seen in Appendix 10.2.2. Pair 2D

We now move on to describe the actual steps that the algorithm takes to reach an output. We've split the algorithm into 4 steps. Firstly, before running the actual algorithm, we sort the points by X-coordinate and Y-coordinate. Then we split our points into two sub-collections without breaking the order such that they are still sorted. We recursively find the closest points in the sub-collections, and lastly we analyze the "slab" around the splitting line for points across the two sub-collections that may have even shorter distance than what was found in the two sub-collections.

1. Sorting points

In order to find a split in the middle of our points, and when finding the closest point in the slab, we need to keep 2 lists of our points, one sorted in X-coordinate, and one in Y-coordinate. Initially we use a normal sorting algorithm from the java library, taking $O(N \log N)$ time. We require this invariant to be kept true through the different recursions, and using an sorting algorithm in every iteration takes a long time, and it can be shown, our solution would take $O(N \log^2 N)$ time, if we did.

2. Splitting our points

To keep the invariant, of having our points sorted in both the X-coordinate and Y-coordinate, we need to secure our points in a way, that keeps them sorted. In this variant, we always split in the middle of X. It can easily be done for the X sorted list, since we just partition it in two parts, and both partitions are sorted on X. To sort Y, we take all elements in one of the partitions of X, and give them a variable "isLeft", and set it to true, and set it to false for the other partition. We then create 2 empty lists, and iterate through Y sorted list, and append them to one of the lists depending on their "isLeft" value. How we implemented this can be seen in the code below:

```java
int mid = n/2;

Point2D[] pointsLeftOfCenter = Arrays.copyOfRange(pointsSortedByX, 0, mid);
Point2D[] pointsRightOfCenter = Arrays.copyOfRange(pointsSortedByX, mid, n);

for(int i = 0; i < mid; i++){
    pointsSortedByX[i].isLeft = true;
}
for(int i = mid; i < n; i++){
    pointsSortedByX[i].isLeft = false;
}
int countLeft = 0, countRight = 0;

```

```java
Point2D[] pointsLeftOfCenterSortedByY = new Point2D[mid];
Point2D[] pointsRightOfCenterSortedByY = new Point2D[n-mid];

for(int i = 0; i < n; i++){
    if(pointsSortedByY[i].isLeft)
        pointsLeftOfCenterSortedByY[countLeft++] = pointsSortedByY[i];
    else
        pointsRightOfCenterSortedByY[countRight++] = pointsSortedByY[i];
}
```

This sorting only takes the initial $O(N \log N)$ time to sort, and then in each iteration it takes $O(N)$ to keep this invariant up.

3. Finding the closest points in the sub-collections

Now that we have our points split on the X coordinate sorted list, we call the algorithm recursively on the two $N/2$ sub-collections of points. To deal with the end of the recursion, we call bruteforce on the sub-collection in the recursion if the amount of points is less than 3. This works since splitting the collection in half every time will eventually yield sub-collections of size either 2 or 3, the only time we could get to a sub-collection of size 1 is if we exactly split a collection of 2 or 3 points, therefore we do not have to worry about a case where a sub-collection in the recursion only has 1 point in it.

Once the recursive calls have returned to the top level, we exactly have two pairs of points, call them A and B, that are the closest points in their respective sub-collections. Call their distances δ_A and δ_B, we now set $\delta = \min(\delta_A, \delta_B)$. This is is reflected in the following code snippet.

```java
Pair closestPairLeft = bentleyShamos(pointsLeftOfCenter, pointsLeftOfCenterSortedByY);
Pair closestPairRight = bentleyShamos(pointsRightOfCenter,
    pointsRightOfCenterSortedByY);

Pair closestPair = closestPairLeft;
if (closestPairLeft.distance() > closestPairRight.distance()) {
    closestPair = closestPairRight;
}
```

Here, bentleyShamos is the recursive call, two pairs are returned and we select the closest pair which will have δ distance. For this specific case where we split points in half and recursively call the algorithm in a divide-and-conquer fashion, we have the recurrence relation $T(N) = 2T(N/2) + f(N)$, where $f(N)$ is some function $f(N) = bN^k$.

4. Finding the closest points in the slab

Lastly, we only need to find the closest points in the slab. We add all the points that are within δ distance from our hyperplane cut to a collection, call it the slab, the points will be added from the collection of points sorted by Y. We can then iterate though the slab.

For each point we iterate through the next points and measure their distance in the Y-axis. If their distance on the Y-axis is less than δ, then we may have a pair that has distance shorter than δ, since a projection to a lower dimension cannot reasonably encode the same distance as on the plane, we check their actual distance on the plane and update the δ as it finds a better solutions. Each point only iterates through the next points, until it finds the Y-distance between itself and the next point is greater than δ, since this means, that all the following points will be greater than δ distance away anyway. The implementation, of the iteration can be seen below:

```java
for (int i = 0; i < size - 1; i++) {
    for (int j = i + 1; j < size; j++) {
        Pair newPair = new Pair(slab[i], slab[j]);
        CompCounter.addComp();
        if ((newPair.a.y - newPair.b.y) >= distance) break;
        if (newPair.distance() < closestPair.distance()) {
            distance = newPair.distance();
            closestPair = newPair;
        }
    }
}
```

This calculation will take $O(N)$ running time, since each point will at most be compared to $c = 12$ points, due to the sparsity in 2-dimensions, which was proven in Section 2. Review of Literature.

As both our upholding of the sorted partitions and slab calculations are $O(N)$, and all the overhead calculations are $O(N)$, we find that our $f(N) = bN$, in accordance to recurrence relation, we end up with a running time of $O(N \log N)$.

3.2 Implementing Closest Points, Improved Version

We learned that if we carefully choose a hyperplane cut, we could improve the running time in k-dimensions, so in addition to the basic version of closest points in the plane, we tried finding a good hyperplane cut, so we in Section 5. Analysis of Running Time in Practice, could see, if finding a good hyperplane cut was beneficial in 2 dimensions. We know from our review, that this only decreases the number of points in our slab, and our partitioning and looking through the list still takes $O(N)$ time, and therefore our running time will still be $O(N \log N)$.

This change requires a change in partitioning our data. We iterate through the middle $N - \frac{N}{8} - \frac{N}{8} = \frac{3N}{4}$ points in both axis, and measure the axis distance between two points, which are $2 \cdot 12 \cdot N^{\frac{1}{2}}$ points apart from each other. Once found these 2 points, and the axis they were found on, we choose our hyperplane cut to be the line perpendicular to the axis, crossing the middle point of the 2 points. The implementation can be seen below:

```
int minInSides = (int) Math.ceil(n / 8.0);
int mustContain = (int) Math.ceil(2*12*Math.pow(n,1/2));
double largestSoFar = 0;
Cutplane2D cut = new Cutplane2D(axis.X, pointsSortedByX[n/2].x);


for (int i = minInSides; i<n-minInSides-mustContain; i++ ){
    double size = -pointsSortedByX[i].x+pointsSortedByX[i+mustContain].x;
    if(size > largestSoFar){
        largestSoFar = size;
        cut = new Cutplane2D(axis.X, pointsSortedByX[i].x + size/2);
    }
}
for (int i = minInSides; i<n-minInSides-mustContain; i++ ){
    double size = -pointsSortedByY[i].y+pointsSortedByY[i+mustContain].y;
    if(size > largestSoFar){
        largestSoFar = size;
        cut = new Cutplane2D(axis.Y, pointsSortedByY[i].y + size/2);
    }
}
```

This change does require some changes to the rest of the code, e.g. partitioning, which can be seen in Appendix 10.2.5. Closest points 2D Improved Implementation. The change in running time will be discussed in Section 5. Analysis of Running Time in Practice.

4 Implementation of Closest Points in Higher Dimensions

In this section our implementation of the Bentley and Shamos solution to the closest points problem in 3-dimensional space is presented in detail. Note that we only present the 3-dimensional version, but abstracting to k-space is not very difficult.

4.1 Implementing Closest Points in 3D, Basic Version

In this basic version of the closest points solution in 3D we do not attempt to find a good cut for the hyperplane. Instead we do as in the 2D version and split the collections in two halves on the X-axis. The exact source code can be seen in Appendix 10.3.5. Closest points 3D implementation. Similarly to the 2D version of the algorithm, we use 3-tuples (x, y, z) to represent points in 3D space with their respective coordinates on the XYZ axes. Our choice of distance metric will again be the Euclidean distance, which is calculated in much the same way as in 2D except we now have an extra coordinate to add to the equation between two points. Of course, for k-dimensional space we can simply use k-tuples to represent points with k dimensions, the formula for Euclidean distance is similarly defined for k-dimensional Euclidean space. Once again we split the procedure into four total steps.

1. Sorting points

Similarly to 2 dimensions, we wish to keep our points sorted throughout our implementation, so we start out with an initial sort, taking $O(N \log N)$, and henceforth we keep this invariant valid by partitioning our already sorted lists, taking only $O(N)$ time. Expanding to k-dimension will require us to sort the new dimension, therefore our initial sorting will take $O(kN \log N)$ time.

2. Splitting our points

When we split our points, we split in the middle of one of the axes, we've chosen the X-axis. When we partition our X-list which is sorted, we get 2 partitions sorted by X again. We wish to keep the invariant, that all our lists are sorted, so we similarly to 2-dimensions, make an variable "isLeft", set to true for one of the partitions, and false for the other, and iterate through both the Y-sorted list and the Z-sorted list, and partition them into a total of 4 lists, a left and right sorted by Y list, and similarly for Z. This way we can keep up our invariant of sorted list in $O(N)$ time. Expanding this to k-dimensions can be easily, this just requires us to iterate through the new dimensions in equal fashion to how we did for Y and Z, taking a total of $O(kN)$ time.

3. Finding the closest points in the sub-collections

We've split our points into two sub-collections. Now comes the divide-and-conquer part of the algorithm. This is completely analogue to the basic 2D version. We have our lists, each list denotes a sub-collection that is either left or right of the hyperplane cut, and then each list is additionally sorted on one of the X,Y,Z axes, giving a total of six lists. We pass the three lists that denote the left collection to a recursive call, and then the three lists that denote the right collection are also passed to a recursive call.

We may expand this procedure quite easily to k-dimension space by having $2k$ lists in a similar fashion, where k of the lists denote the points in one half and each list is sorted by a unique axis in k-space, it is completely analogue for the other k lists for the other sub-collection of points. We then call the procedure recursively on these two sets of k lists.

4. Finding the closest points in the slab

Finding the closest point in the slab is a bit more difficult, since we no longer project our points down to 1 dimension, so we can't directly iterate through our points in order. Instead we find all the pairs, which are within δ distance of each other, and due to our sparsity, we know we can at most find $c \cdot N$ pairs. We can then iterate through all the pairs, and finding the closest pair. Finding the pairs is the hard part. We project our points in the slab down to our hyperplane cut, and run sparse fixed-radius-near-neighbor. Solving sparse fixed-radius-near-neighbor in 2-dimensions is done similarly to solving closest points, except we don't find the nearest points, but all the pairs within δ distance from each other.

We get our slab sorted by Y and Z, and the distance of the closest pair so far. We split our data into two partitions, maintain the sorted partitions in $O(N)$ time, find the pairs in both partitions through recursion, and lastly project our points down to 1 dimension, and iterate through our points, finding all pairs within δ distance of each other. Meaning the only notable difference between sparse fixed-radius-near-neighbor and closest points 2d is solving its slab. The solving of the slab can be seen below:

```java
for (int i = 0; i < slab.size() - 1; i++) {
    for (int j = i + 1; j < slab.size(); j++) {
        Pair newPair = new Pair(slab.get(i), slab.get(j));
        CompCounter.addComp();
        if ((newPair.a.y - newPair.b.y) >= delta) break;
        if (newPair.distance() <= delta) {
            result.add(newPair);
        }
    }
}
```

Instead of rewriting our closestpair, when we find a pair within δ of each other, we add it to all the pairs found so far. Through recurrence relation we can find sparse fixed-radius-near-neighbors running time to be $O(N \log N)$.

Expanding this to k-dimensions, require us, much similar to expanding closest points to k-dimensions, partition our data set into 2 equal size partitions, keep both partitions sorted in all dimensions, solve each partition recursively. Solving the slab require us to project our points down to k-1 dimensions, and recursively solve sparse fixed-radius-near-neighbor for our points in one dimension smaller. This can be shown to take $O(N \log^{k-1} N)$.

4.2 Implementing closest points in 3d semi-improved version

Our semi-improved implementation works like the basic implementation, but with the added feature that we now find a good hyperplane cut in our sparse fixed-radius-near-neighbor algorithm. Simply put, the algorithm runs as the basic version until we reach step 4 where we build the slab. We then project the points onto the plane and run the sparse fixed nearest neighbors algorithm on the plane.

The sparse fixed-radius-near-neighbor algorithm works by finding a good hyperplane cut in the space of the algorithm (in this case the space is 2-dimensional since we projected our points down onto a plane). The sub-collections induced by the hyperplane cut will contain at least $N/8$ points, and there are at most $2cN^{1/2}$ points within distance δ of the hyperplane cut. By applying theorem 5 of Bentley and Shamos 1976, we can abstract the finding of the hyperplane cut to arbitrary k-space.

The algorithm will then find the nearest neighboring points in each of its sub-collections, and finally see if there is a pair of points across the sub-collections with an even shorter distance by projecting onto a lower dimension. In this case it will project the points in the "slab" (note that this is a different slab pertaining to the fixed-radius-near-neighbor algorithm) onto a line. In which case it can find the nearest neighbor in linear time, giving the sparse fixed radius nearest neighbor algorithm a running time of $O(N \log N)$.

When we expand this algorithm to k-dimensions, we also reach a running time of $O(N \log N)$, even though we run it recursively in a lower dimension, since we will at most run on $kcN^{1-1/k}$ points, which gives the recursion a running time of $O(N)$. To implement this in k-dimensions, we would just make a sparse fixed-radius-near-neighbor which kept projecting down, while finding good hyperplane cuts, until it reached 1 dimension, and it becomes trivial to solve.

The overall runtime of the closest points in 3d semi-improved algorithm will thus still be $O(N \log^2 N)$, even when expanding to k-dimensions, as the initial cut in our closest-point could contain all points in the slab.

4.3 Implementing Closest Points in 3D, Improved Version

The semi-improved version of closest points finds a solution in $O(N \log^2 N)$ time, but as showed in the review, we can improve this to $O(N \log N)$ by finding a good hyperplane cut. We will show here how this could be implemented.

In the same fashion to closest point in 2d improved version, we start iterating through the points in every axis in a sorted order, measuring the distance between the i'th and $i + 2 \cdot 12 \cdot N^{2/3}$th point on the given axis, saving the most distant pair. Again, we start iterating from the $\frac{N}{12}$ point, until the $\frac{11N}{12}$ point, so we ensure that in each partition, there is at least $\frac{N}{4k}$ points. Making the hyperplane cut the perpendicular plane to the axis, which we found the most distant pair, and the plane goes through the middle of two points.

This way we ensure that when we calculate the sparse fixed-radius-near-neighbor, that there is at most $k \cdot c \cdot N^{1-1/k}$ points, which makes calculating the slab take $O(N)$ time, since calculating fixed-radius-near-neighbor for N points takes $O(N \log N)$. The full implementation of this can be seen in appendix 10.4.2. Closest points 3D implementation improved version.

5 Analysis of Running Time in Practice

Now that we've implemented several different solutions, we first want to look at, if this solution is even an improvement, when applied to actual data in practice. First we will look at the 2d implementations, and analyse their running time, and see how they run over different data sets. We wish to see, if our worst case running time is equal to the running time in practice. For the practical side of things, we will be using the same machine and environment to perform all statistical tests, the specifications of the platform that the tests will be run on is included in 10.1. Technical details, and it also runs unoptimized.

To also get a computer specification independent measure, we also counted number of comparisons. This measure counts every time we compare two points distance.

Data Distributions

For the different data sets we generate each of them according to its own probability distribution. In particular we use Gaussian Distribution, Uniform distribution and a Special distribution.

For instance when generating points using the Gaussian Distribution in 2d, we use mean 0 and standard deviation 1, an example if this is seen on Figure 1

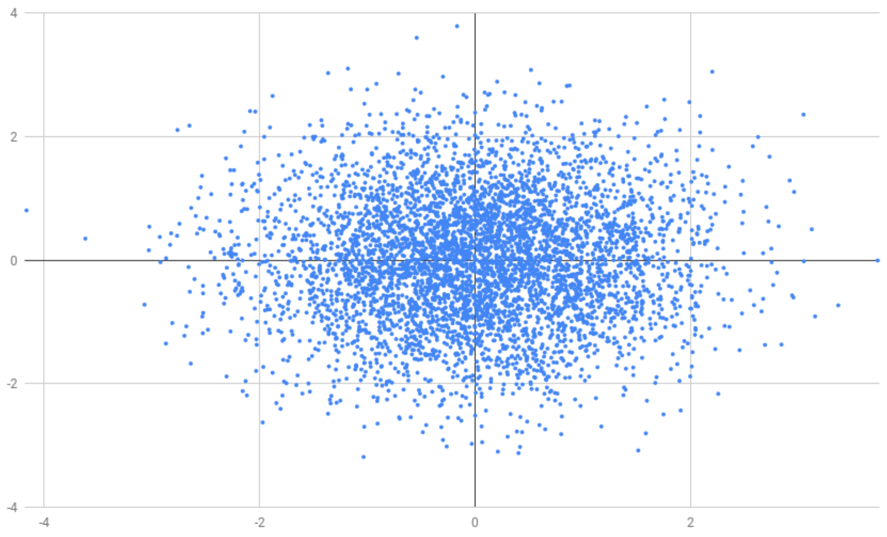


Figure 1: Gaussian distributed points

For Uniform distribution each point (x,y) is uniformly distributed in the interval $x, y \in [0, 1]$. The special distribution distributes the X axis gaussian with standard deviation $\frac{1}{20}$, and the Y axis is distributed with gaussian standard deviation 1.

Closest Points 2D Basic Version Compared to Brute-Force

To give a motivation on why it is necessary to actually find more efficient solutions, we will first compare our Closest points 2d basic version, and compare to the simple brute force. The data is Gaussian distributed.

As seen on Figure 2, the closest points implementation we used, is faster after only reaching a few points, and while our closest points implementation can calculate 10 000 000 in less than 30 seconds, the brute force takes over a minute for only 100 000. This analysis just goes to motivate, that these extra steps to calculate closests points does pay off. From here on, we will look at analysing the different implementations, and compare their running times, and how many comparisons they make.

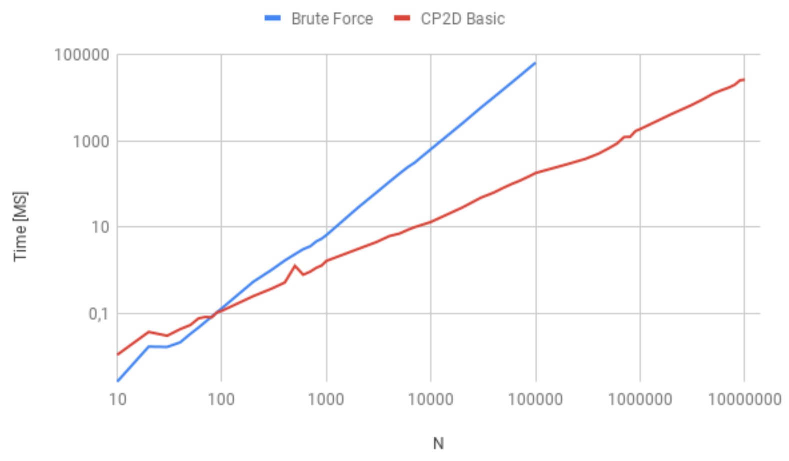
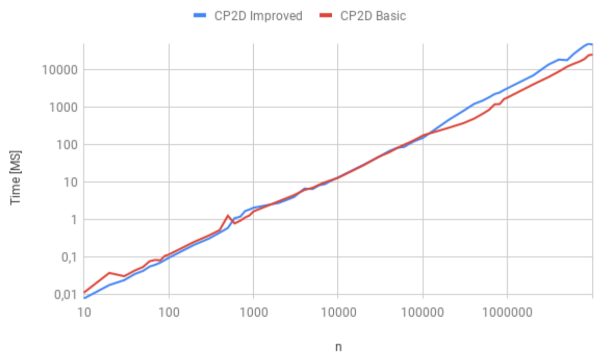


Figure 2: Closest points 2D basic version compared to Brute force

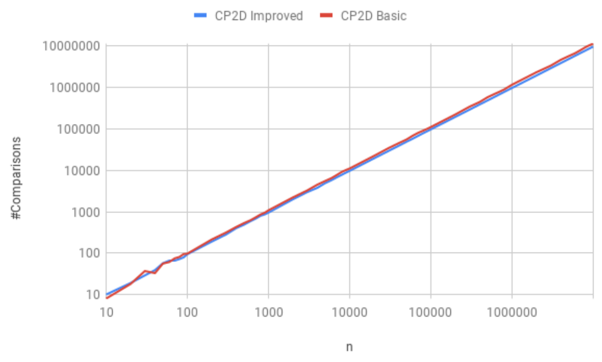
5.1 Analysing Closest points 2D

5.1.1 Closest points 2d basic version compared to Improved Version

We will now check, if finding the good cut in 2-dimensions is time efficient. We hypothesise that finding a good cut, will decrease the slab size, and therefore we would require less comparisons, which in turn increases the running speed. We expect our improved version to be faster than the basic version in worst case, if the calculations for the slab is greater than the overhead calculations for finding the good cut.



(a) Comparing running time of CP2D Basic version to Improved version, Gaussian distributed

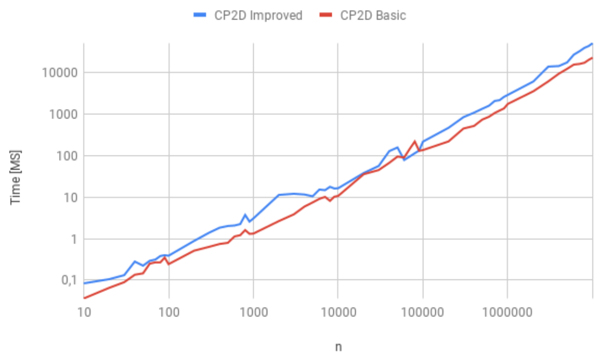


(b) Comparing #comparisons of CP2D Basic version to Improved version, Gaussian distributed

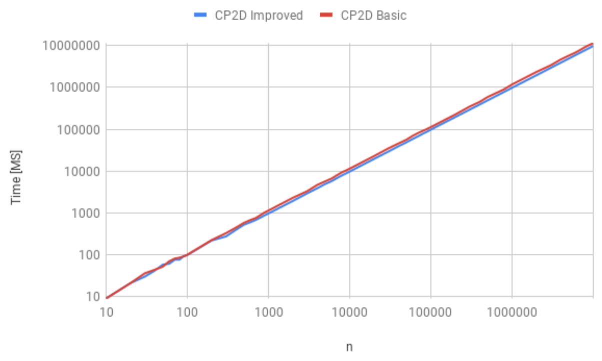
Figure 3: Comparing CP2D basic version with CP2D improved version

Figure 3 compares the running time of Closest Points 2d basic version to the improved version, which finds the good cut. We see in Figure 3a, that the Basic version runs faster than the Improved version by a significant amount, when it calculates on a large collection. To find the reason in this finding, we look at Figure 3b, which shows that while we hypothesised, that the number of comparisons would be far less for the improved version, while they in practice are very similar. This means, that we do a large overhead calculation in finding the good cut, without gaining much in reducing the amount of comparisons. The reason for the small difference in comparisons is hard to determine.

Though this is a pretty neutral distribution, we also wanted to check, if we made a distribution, which should be running worse for the basic version, is it still faster than the improved version. This distribution is the special, which will cause the number of points in the slab to be increased. In Figure 4 we see the exact



(a) Comparing running time of CP2D Basic version to Improved version, Special distributed



(b) Comparing #comparisons of CP2D Basic version to Improved version, Special distributed

Figure 4: Comparing CP2D basic version with CP2D improved version

same pattern as we did for Figure 3. The improved version runs slower than the basic version, and they have similar number of comparisons. This doesn't match our hypothesis, and we will in the next 2 sections look into why, our hypothesis and practical experiment doesn't match.

5.1.2 Closest Points 2D Basic Version Analysis

First we take a look at the closest points 2d basic version, see its running time over a few distributions, and how its change in running time over the number of points.

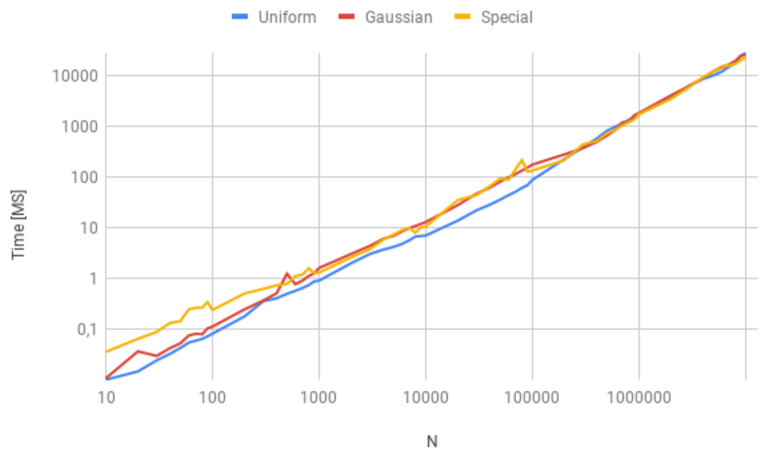
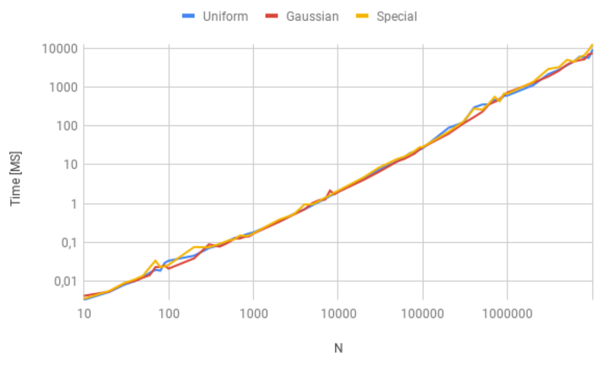


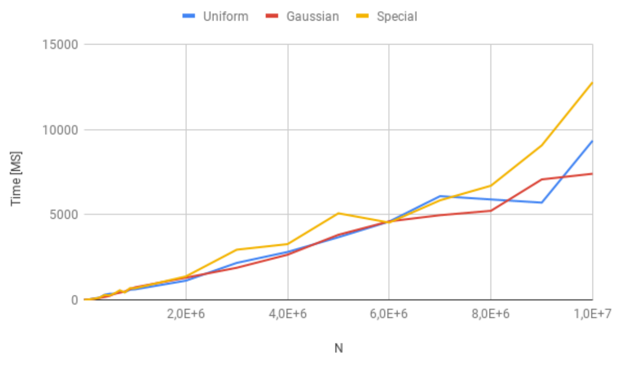
Figure 5: Running time in MS relative to number of points

As seen in Figure 5, the running time has a small dependency on the data distributions, and in this exact case, there is an increase in running time for small number of points. The special distribution which is distributed by Gaussian with standard derivation $\frac{1}{20}$ on the X-axis, and Gaussian distributed with standard derivation 1 on the Y-axis, runs slower for small amount of points, compared to the other distributions. We hypothesise this happens, due to the increased amount of points in the slab, as our spread in the X-axis is much smaller, meanwhile our δ doesn't decrease by as much. We do see, as we continue running, that they begin to line up, and take equal amount of time, which is due to long sorting time, and we would expect, that when we run on bigger collections, we are bounded by the sorting time.

Now that we know, we are bounded by the sorting time, we still wish to know, what the running time of our implementation is, if we exclude the sorting part. So we once again, measure the running time over the different distribution excluding the sorting.



(a) Linear scale

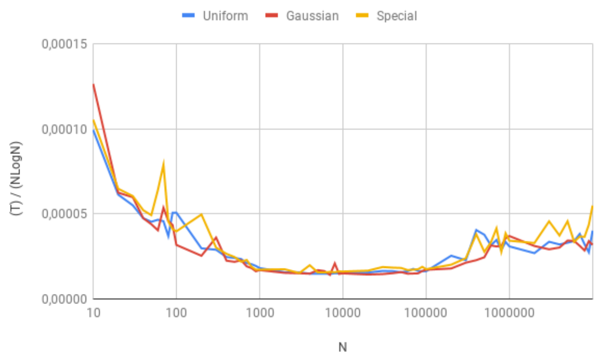


(b) Logarithmic scale

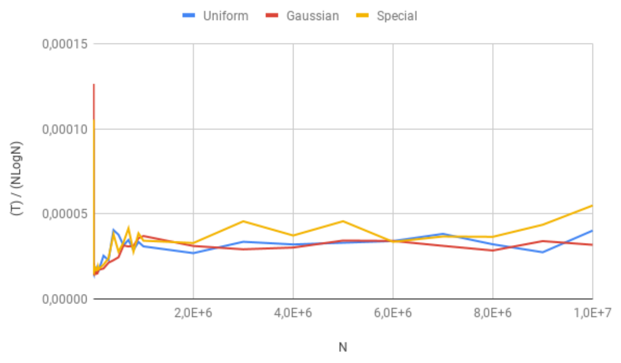
Figure 6: Closest points 2d basic version running time over distributions -sorting time

We once again, see a similar picture, they have close to the same running time, except there are more fluctuation in running time on larger sizes of N. We notice that there is a slight increased running time for our special distribution, as we had hypothesised, though we wish to learn, to what degree it affects its running time on even larger N.

To measure this, we assume it runs on the form of $T = O(N \log N) = cN \log N$, where c is some constant, N is the number of points and T is the running time. So we try to divide our running time by $N \log N$, and we would expect to see a straight line.



(a) Logarithmic scale



(b) Linear scale

Figure 7: Running time over N Log N relative to N

As we see in Figure 7, this is by no means a straight line, so our hypothesis is wrong, it is not a measured by a simple function as $T = cN \log N$. Though it does suggest, that we initially are much faster than $cN \log N$. As we increase our N, we start to line up to a straight line, which would suggest our function is on the form $T = c_1 \cdot f(N) + c_2 \cdot g(N)$, where our c_1 larger than c_2 and $O(f(N)) < O(g(N))$, so $f(N)$ is the dominant part of the running time for small N, and as N increases the $g(N)$ part is dominant. We suspect $g(N)$ is $g(N) = N \log N$. At the end we do see a small increase, which suggest that the running time is higher than $N \log N$, but it could also be random noise, though it is not clear enough to conclude it.

From here we take a look at the number of comparisons we've done, as we in theory know it is upper bounded by $O(N \log N)$, but in practice, we expect it to be less. We will therefore expect the number of comparisons to be on the form of $O(N \log^{\alpha} N)$, as it will both capture the $O(N)$ case and $O(N \log N)$ case.

So we write the number of comparisons as Y, and we expect it to have the equation $Y = O(N\log^\alpha N) = cN\log^\alpha N$, which we rewrite to $\log\frac{Y}{N} = \alpha\log\log N + \log c$. If we plot this, we would expect a trendline, with a derivative of α, which is bounded by $\alpha \leq 1$ As we see in Figure 8, the number of comparisons for

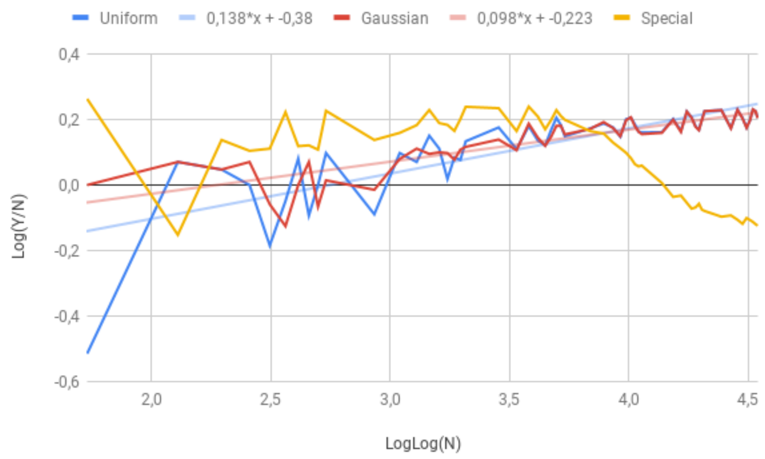


Figure 8: Basic 2D estimating number of comparisons

Gaussian and Uniform increases like $Y_{uniform} \approx 2^{-0,38}N\log^{0.14}N$ and $Y_{Gaussian} \approx 2^{-0.22}N\log N$, as they both seem to converge to a straight line. For our special distribution, we see another picture, as it doesn't converge, and instead starts dropping on larger sizes of N. This is in opposition with our original hypothesis, as we also expected it to converge to a straight line, and have a bigger slope than the Gaussian and Uniform distribution.
Though we did see, that we have a $f(N)$, for which it holds that $O(f(N)) \leq O(N\log N)$, which comes partly in the form of comparisons. We are still left with a question of, why does the special distribution take longer time to run, when it had fewer comparisons.

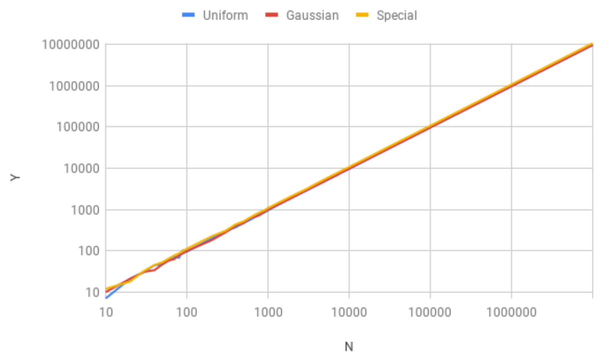
We originally expected a larger slab correlated to more comparisons. Though this is not necessarily true, we may actually archive a bigger slab, but we don't make as many pointwise comparisons, but our slab can be sparser. So we make an hypothesis, that they do not correlate, since a bigger slab requires more sparse data on one of the axes, which means our c is smaller. To test this hypothesis, we could measure the sum of the sizes of the slabs for different distributions and the number of comparisons.

5.1.3 Closest Points 2D Improved Version Analysis

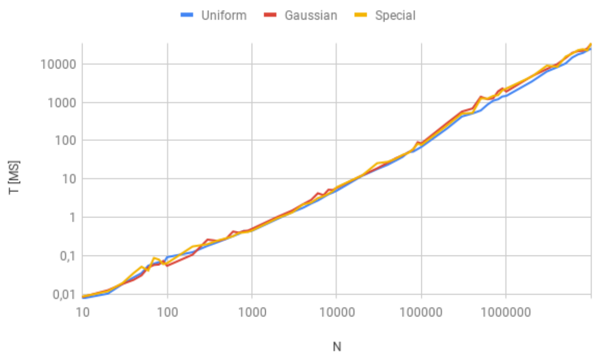
The 2D improved version will attempt to find a good hyperplane P cut such that the two half-spaces contain at least $N/8$ points, and there are at most $2cN^{1/2}$ points within distance δ of P. We hypothesize that the amount of comparisons we'll have to do is upper bounded by $O(N)$ since solving the problem in the two half-spaces induced by P has a bound of $O(N)$ comparisons (see theorem 10, Bentley and Shamos 1976), and the sub-problem to be solved in 1-space will have at most $2cN^{1/2}$ points which bounds the sub-problem by $O(N)$. We also hypothesize that we'll be doing some amount of hidden work in having to find P, which the comparisons counting will not be able to take into account. As before we will not be taking pre-sorting time into account.

Figure 9a shows the number of comparisons (we call it Y) relative to the number of points tested for, since the comparisons is a machine-independent measurement we see that all the tests in every distribution behaves nicely. Figure 9b looks at time in milliseconds (we call it T) relative to the number of points tested for, and has some more variance, but we can still see a clear trendline and the distributions roughly follow the same line.

As stated our hypothesis is that the comparisons $Y = cN$ for some constant c. We can check this hypothesis by dividing Y by N, giving us a graph $\frac{Y}{N} = c$ that will simply be a horizontal line if our



(a) Number of comparisons



(b) Time taken in milliseconds

Figure 9: Improved 2D implementation, N is number of points

hypothesis is correct. Figure 10 shows that indeed when we divide by N, each distribution converges towards a horizontal line with $c = \frac{Y}{N}$ on the vertical axis.

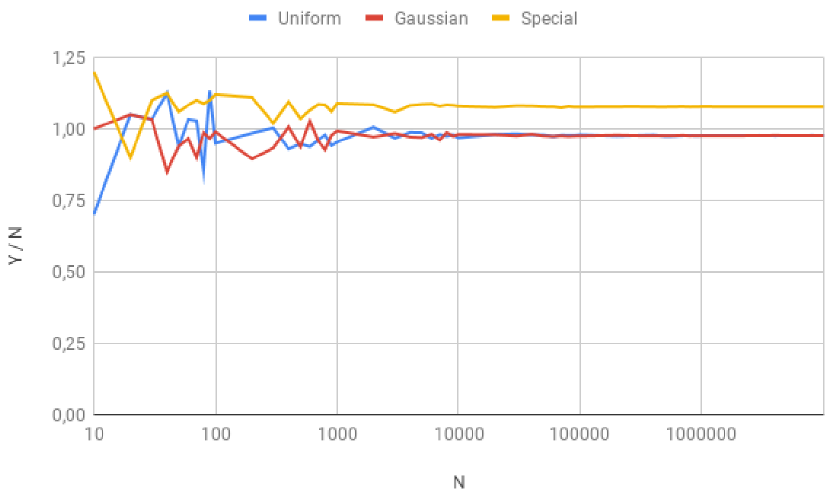
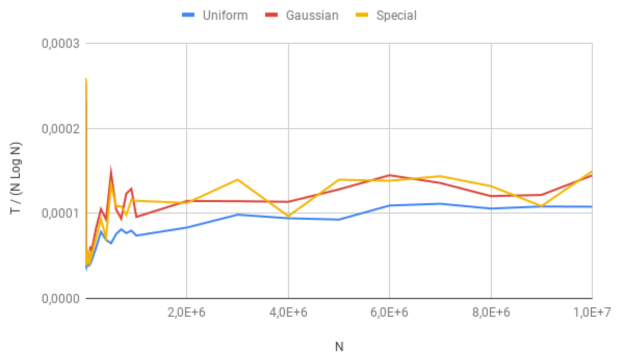


Figure 10: Comparisons over amount of points

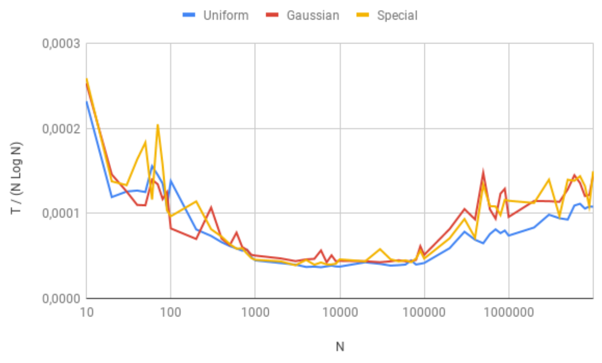
We can see that the uniform and Gaussian distributions both end up with a constant very close to $c = 1$, while the special distribution has a constant slightly higher than 1. We do not have conclusive data to find out why this is the case, but one possibility may be that the properties of the special distribution lends itself to having more comparisons, as the cutline will likely be perpendicular to the Y-axis and when we then project the points onto the cutline they will seem closer to each other as the deviation on the X-axis will be $\frac{1}{20}$.

We expect the whole runtime of the algorithm to be bounded by $N \log N$. Specifically, we hypothesize that $T = c(N \log N)$ milliseconds, and that $\frac{T}{N \log N} = c$, in a completely similar way as we did with the comparisons. Plotting $\frac{T}{N \log N}$ on the vertical axis gives figure 11, which seems to be different to what we expected. Looking closely, we see that the graph curves downwards first, and then seems to start curving upwards towards the end. This would imply that for small N the algorithm seems to run better than $N \log N$.

At larger N we see on figure 11a that the lines seem to fit with our idea of getting horizontal lines towards the end, there is some noise particularly at the end that we cannot determine the cause for as the dataset is too small. As said in the beginning the running time is better than $N \log N$, plotting the runtime over N



(a) Linear scale



(b) Logarithmic scale

Figure 11: $\frac{T}{N \log N}$ relative to input size N

gives figure 12 which shows that for the small N we seem to have a linear time algorithm. We believe the case here is that the runtime of the algorithm can be modeled as $T = c_1 f(N) + c_2 g(N)$ where $f(N) = N$ and $c_2 < c_1$ such that the linear term with $f(N)$ is dominant for small N, as N increases $g(N) = N \log N$ becomes the dominant part.

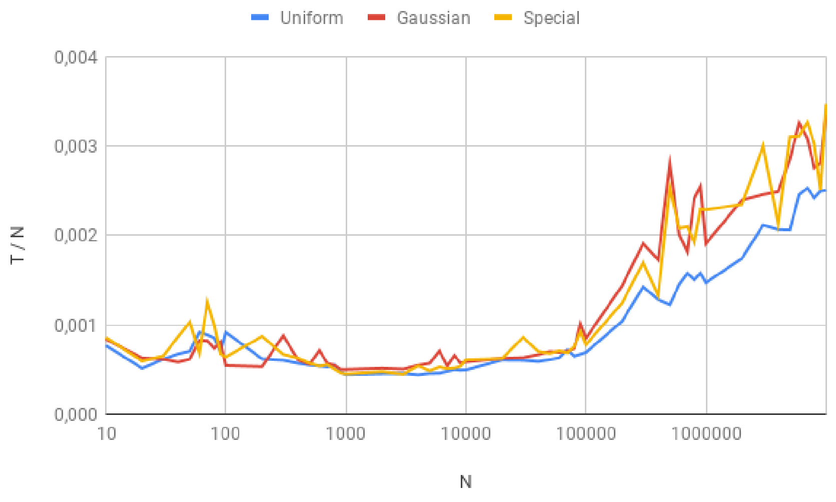


Figure 12: $\frac{T}{N}$ relative to input size N

5.1.4 Conclusion for Closest Points 2D

We had an original hypothesis, that for both implementation, even after excluding the time for presorting, we would have a running time to be $O(N \log N)$. In our experiments, we could conclude this running time, as both our implementations did converge to a horizontal line, when we graphed the time as $\frac{T}{N \log N}$, though we did experience some noise.

We also see the number of comparisons follow our hypothesis, as our basic version archived to have an upper bound of $O(N \log N)$ comparisons, and our improved versions number of comparison diverged towards $O(N)$ comparisons. This was achieved for all our distributions, even the Special, which was made to simulate a bad case.

When we look which one runs better in the general case, we would on our test set conclude the basic version runs faster, even though the improved one consistently had a slower growing number of comparisons, with an exception on the Special distribution. This is due to the overhead work, in finding the best cut far out weights the benefits in fewer comparisons. As we see in the number of comparisons, the theoretically worst case of number of comparisons is far off from the average case, meaning improving it for 2-Dimensions doesn't amount to much for large sizes of N.

5.2 Analysing Closest Points 3D

5.2.1 Closest Points 3D Comparing Implementations

We now go from 2-dimensions to 3-dimensions, here we will check the different implementations. The Basic Version, which is expected to take $O(N \log^{k-1} N)$ time, the Semi-improved, which is expected to take $O(N \log^2 N)$ time, and at last the improved, with expected running time of $O(N \log N)$. We first plot the running time for the different implementation, run on Gaussian distribution on all axes. We have subtracted the sorting time for all graphs, as they are already well studied, and we know they take $O(N \log N)$.

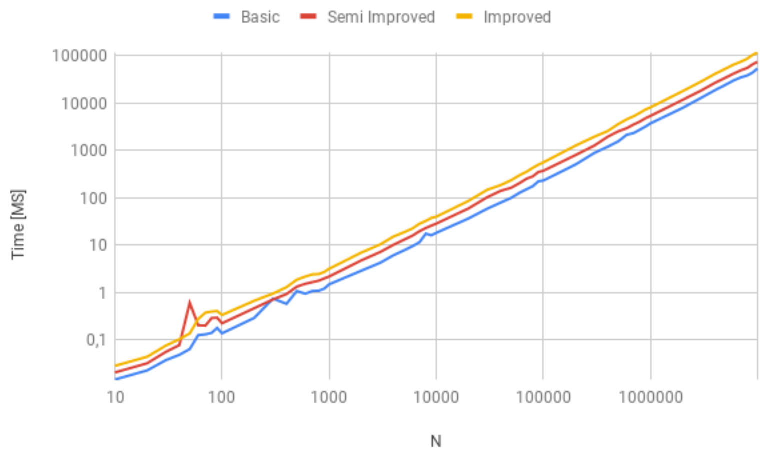


Figure 13: Running time of Closests points 3D for different implementations

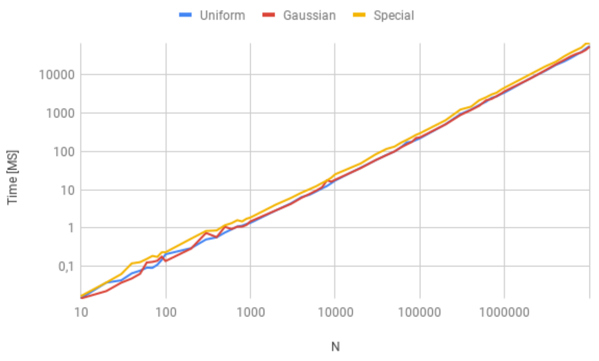
In Figure 13, we see the running time of Closest points 3D for our 3 implementations. We see a clear running speed difference. The Basic version, which in theory should have the worst running time is the fastest, meanwhile the improved version is the slowest, which in theory should be the fastest. We will do an analysis each of the implementations, to give an hypothesis on this observation.

5.2.2 Closest Points 3D Basic Version Analysis

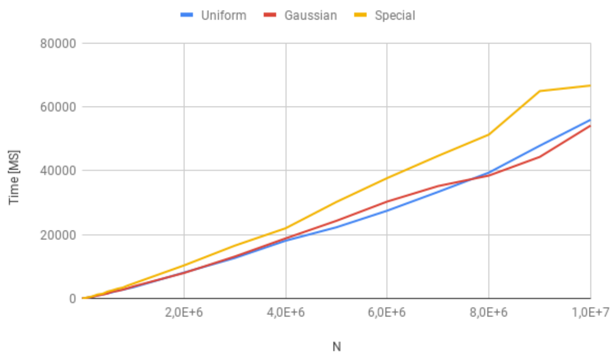
To analyse the running time of our Closest points 3D basic version, we first measure how it runs on our different distributions. On Figure 14 we see the running time of Closest points 3D basic version, on our 3 different distributions. Special is distributed by Gaussian on all axes, with standard deviation 1 on Y- and Z-axis, and standard deviation $\frac{1}{20}$ on the X-axis.

We see the Uniform and Gaussian distribution have similar running time, while our special distribution is slower. This is actually what we expected, as a dense X-axis will give a large slab, and therefore increase the running time. From here, it is interesting to see, if our distribution is upper bounded by our theoretical bound of $O(N \log^{k-1} N) = O(N \log^2 N)$, which can be tested by dividing our graph by $N \log^2 N$. On Figure 15, we see the different distributions fall, which means we can conclude it is $O(N \log^2 N)$, for our 3 distributions, for up to N=10 000 000. Though we do want to find a tighter bound. We expect our running time to be on the form of $T = c \cdot N \log^{\alpha} N$, where c and α are constants, from which we derive $\log \frac{T}{N} = \alpha \log \log N + \log c$. If we transform our data to this form, we can approximate our α and c through regression.

As we see in Figure 16, none of the distribution converge to a line, which suggest we again are in a case, where $T = c_1 \cdot f(N) + c_2 \cdot g(N)$, where $f(N)$ comes from the intermediate calculations like comparisons. In which case, we again end in an inconclusive state. To estimate $g(N)$ we would require additional tests for



(a) Logarithmic scale



(b) Linear scale

Figure 14: Closest points 3d basic version running time over distributions

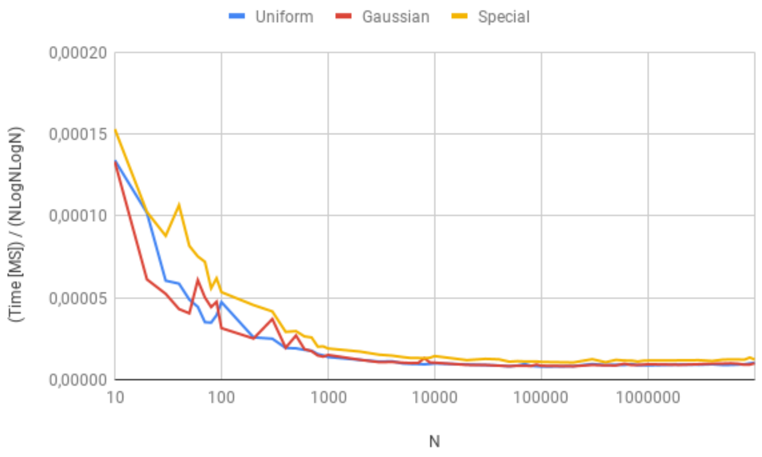


Figure 15: Closest points 3D Basic version running time divided by $N \log^2 N$

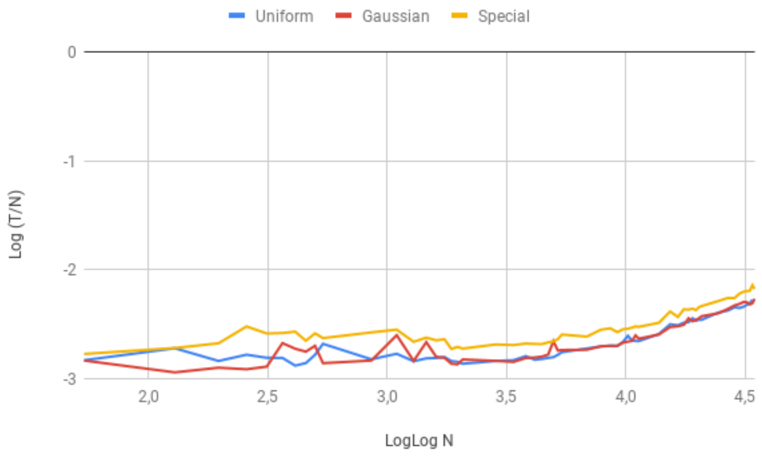
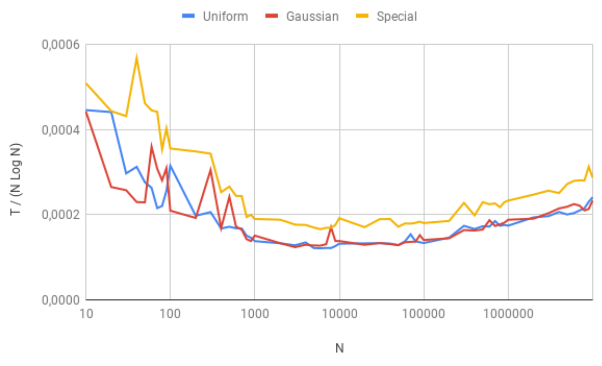
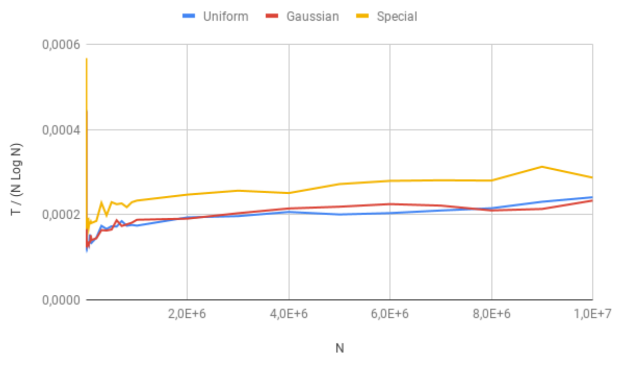


Figure 16: Basic 3D estimating running time

bigger sizes of N. Though we can see, if we have a lower bound of running time on $O(N \log N)$, by dividing by $N \log N$ and see if it converges towards a horizontal line, or diverge upwards or downwards.



(a) Logarithmic scale



(b) Linear scale

Figure 17: Closest points 3D running time divided by N log N

We see on Figure 17, the graph starts diverging upwards in the end, giving us a lower bound for running time in practice. This allows us to conclude, that the running time in practice is lower bounded by $\Omega(N \log N)$, and upper bounded by $O(N \log N)$, though we cannot give an conclusive estimate on it.

We also take a quick look at the number of comparisons, to check if we as expected are limited by $O(N \log^2 N)$ comparisons, as there could potentially be $O(N)$ points in all our slabs. Though we do expect this to be a lower, as $O(N)$ is an upper bound for extreme cases, which does not define the average case. To give a measure of the average case, we expect it to be on the form of $Y = c \cdot N \log^{\alpha} N$, and we try to plot $\log \frac{Y}{N} = \alpha \log \log N + \log N$.

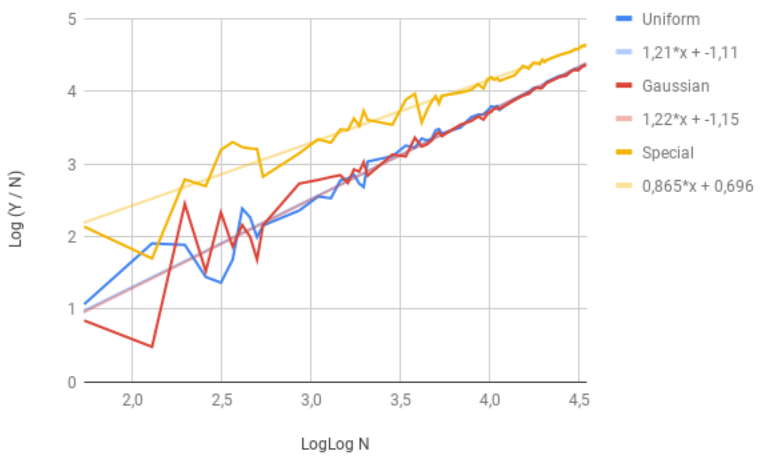


Figure 18: Closest points 3D estimating number of comparisons

In Figure 18, we see they all converge to a straight line, giving us an expected trend line, though we cannot guarantee to see a similar pattern as in Figure 8, where one of graph suddenly diverges for a large N. Though currently, we would expect the number of comparisons to approximately follow the lines $Y_{Uniform} \approx 2^{-1.11} \cdot N \log^{1.21} N$, $Y_{Gaussian} \approx 2^{-1.15} \cdot N \log^{1.22} N$ and $Y_{special} \approx 2^{0.67} \cdot N \log^{0.87} N$. So we can say, that this test followed our hypothesis, as expected, but the number of comparisons are dependent on our data set, as the number of comparisons consistently differed for the 3 different distributions. We do see, that for our special distribution, it has at least initially more comparisons than both Gaussian and Uniform by a significant amount. This is in contradiction with what we saw in the 2D basic version. This contradiction can be explained by the increased slab size. As we in 2-dimensions get a relative smaller δ, by reducing the deviation on one axis, meanwhile our δ doesn't decrease by as much, when we are working on a higher dimension.

5.2.3 Closest Points 3D Semi-Improved Version Analysis

To start off the analysis of closests points 3D semi-improved, we take a look at the running time on our 3 distribution.

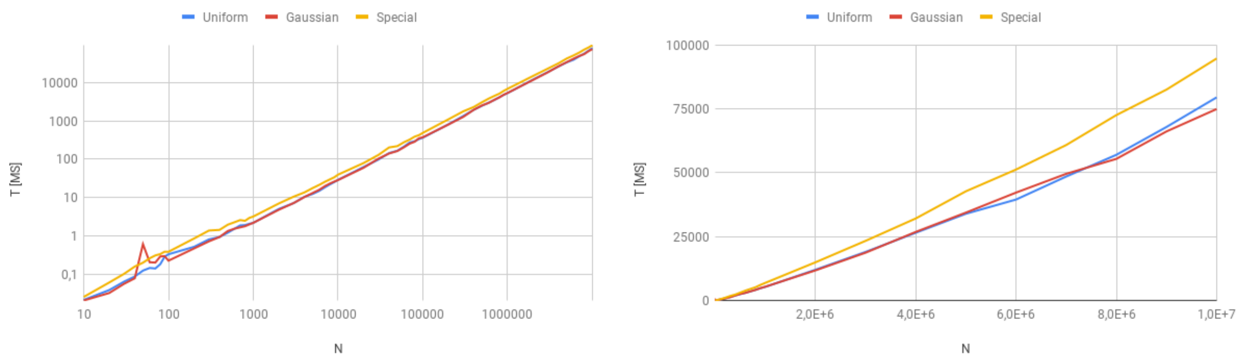
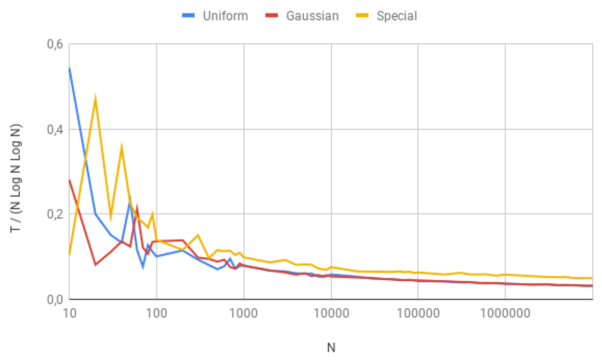
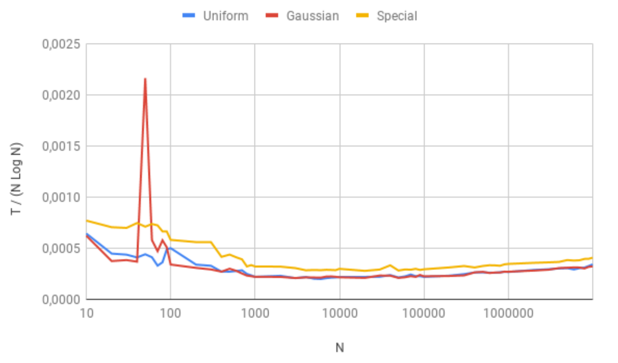


Figure 19: Closest points 3D semi-improved version running time over distributions

As we see in Figure 19, the running time is similar for the 3 distributions, except for Special distributions, which is somewhat slower. Now we will look at the running time, which we would expect to be $O(N \log^2 N)$. We will first check this, by dividing our running time by $N \log^2 N$, and see if it converges down to a line, to 0 or it diverges. We also look at if its running time is lower bounded by $\Omega(N \log N)$.



(a) Semi-improved running time over $N \log^2 N$



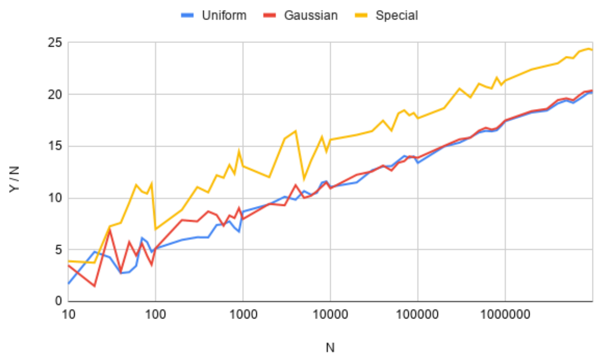
(b) Semi-improved running time over $N \log N$

Figure 20: Closest points 3D semi-improved version running times bounds

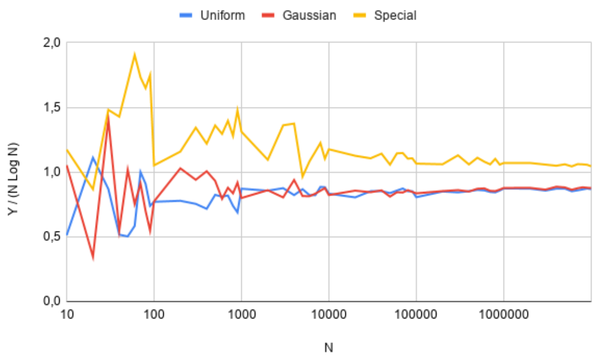
We see on Figure 20a, that we are converging towards 0, which means we have a running time below $O(N \log N)$ for all 3 distributions. Meanwhile we see on Figure 20b, that we initially are faster than $O(N \log N)$, but we end up diverging upwards, meaning we in practice are lower bounded by $\Omega(N \log N)$.

From here it is hard to analyse the actual running time, as we on Figure 20b only see the initial diverging on the graph, which could escalate or stay stable, so we can't give an conclusive analysis on the running time, except giving an expected upper and lower bound on running time of $\Omega(N \log N)$ and $O(N \log^2 N)$, and that it is dependent on the distribution of the data set, it is run on.

We now take a look at the number of comparisons. We have an expected number of comparisons on $O(N \log N)$, as we in worst case send $O(N)$ into the sparse fixed-radius-near-neighbor algorithm, meanwhile the near-neighbor algorithm, will at most send $O(\sqrt{N})$ into its slab to be compared. So if we plot $\frac{Y}{N \log N}$ we would expect line converging to a horizontal line.



(a) Semi-improved comparisons



(b) Semi-improved comparisons over $N \log N$

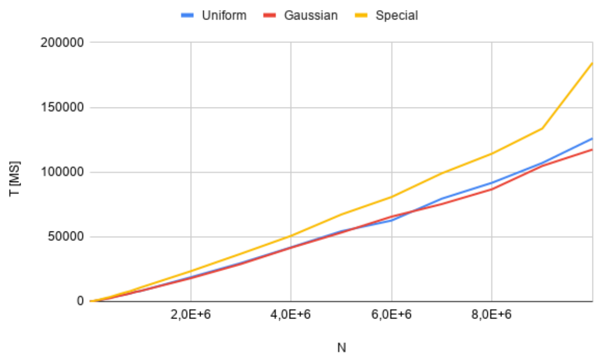
Figure 21: Closest points 3D semi-improved version comparisons

We see on Figure 21b, that for all our distribution, we start to converge towards a horizontal line, meaning we actually have approximately $O(N \log N)$ comparisons, as we expected. We do see an increased amount of comparisons for our special distributions, which can be explained by the increased size of the slab, in a similar fashion to the reason in 3D basic version.

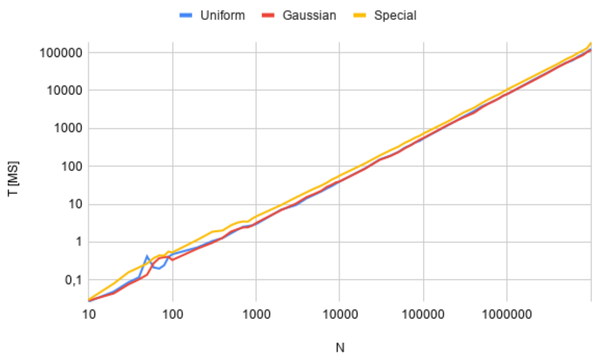
5.2.4 Closest Points 3D Improved Version Analysis

We now move on to the analysis of the 3D improved version. We measure how it runs on the different distributions in figure 22. As before, the Special distribution is simply Gaussian distribution on all axes with standard deviation 1 on YZ and standard deviation $\frac{1}{20}$ on the X-axis.

Since the X-axis is densely populated with points, we expect the hyperplane cut to be perpendicular to either the Y or the Z axis. Our implementation code for the sparse fixed radius near neighbor algorithm will go through sorted sequences on the axes in XYZ order, but since the X-axis is densely populated the algorithm will most likely not find a collection of $kcN^{1/2}$ contiguous points and will always have to move on to scanning the two other axes. This effectively means that the special distribution will increase the running time by a linear factor on N. That is our educated guess (as the implementers) for why the special distribution seems to be slower than the others on the figure, and we will not investigate this further.



(a) Linear scale



(b) Logarithmic scale

Figure 22: Closest points 3D improved version running time

By the theory we expect the implementation to have a running time of $O(N \log N)$, so we hypothesize that we can parametrize the running time T as $T = c(N \log N)$ for constant c, if we plot $\frac{T}{N \log N}$ then we would expect a horizontal line. If we look at figure 23, we see that indeed other than some strange variance for very small N it would seem like we have a horizontal line with the Special distribution having a higher constant c. Looking closely it would also seem as if the lines increase the further we come along N (as well as a jump up from the Special distribution), but we will not investigate this matter further.

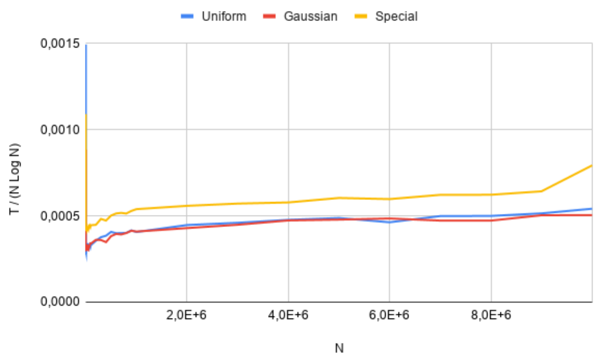
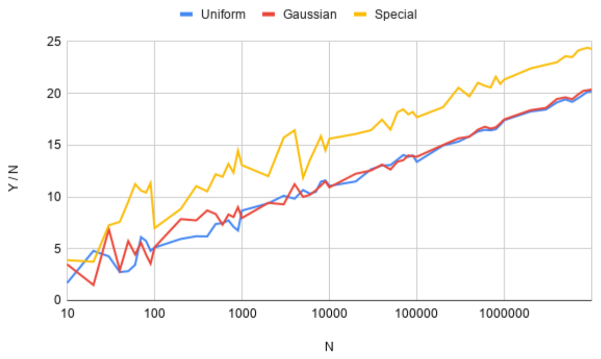


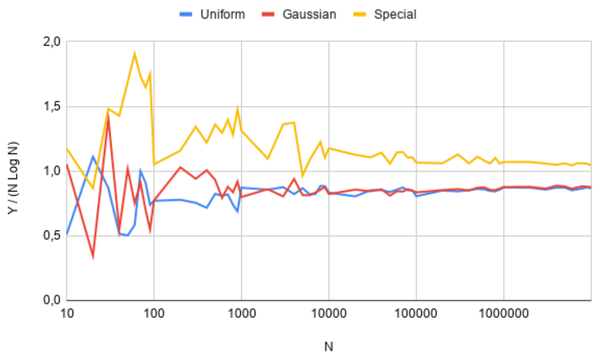
Figure 23: Time over $N \log N$

As before we also expect the number of comparisons to be bounded by $O(N)$. This means we can write the number of comparisons Y as $Y = cN$ and divide it by N to get a constant c. If we look at figure 24a we see something unexpected, instead of a flat horizontal line we get an increasing line. This is interesting, there seems to be some problem with our implementation such that we do too many comparisons. Looking at figure 24b we see that our comparisons are seemingly bounded by $O(N \log N)$.

We hypothesize here that there might be some N_0 such that the number of comparisons is bounded by $O(N \log N)$ for $N < N_0$, and when N is raised high enough such that $N > N_0$ our algorithm switches to $O(N \log N_0)$, such that the logarithmic factor becomes a constant, giving a linear number of comparisons. This hypothesis is formed on the basis that N needs to be raised to a high enough number such that $kcN^{1-1/k} > N(1 - \frac{1}{2k})$ for the improved version to actually be better than the basic version. This criteria is first fulfilled once we reach $N >$2 176 782, and our tests sizes only reach $N =$10 000 000. Therefore we do not have enough data to verify that our hypothesis is actually correct.



(a) Comparisons over N



(b) Comparisons over $N \log N$

Figure 24: Closest points 3D improved version comparisons graphs

5.2.5 Conclusion for Closest points 3D

We found that for all our implementations, they were bounded by their theoretically upper bound, but there is some noise. We did find that for our improved version, that it didn't differ so much from our semi-improved version, as it doesn't choose a good hyperplane cut, when there are less than $\approx 2\,000\,000$ points. For N, it should run exactly as fast as semi-improved, as it splits like it does. The reason for the difference in running speed is due to implementation differences. We do see, that the semi-improved and improved does look closer to a $O(N \log N)$ running speed than Basic does, and we hypothesise, that for large enough N, semi-improved and improved will run faster than the basic version.

We see in the number of comparisons, that we followed our expected upper bound of comparisons, with the exception for improved version. For which we argued, that it initially have $O(N \log(N))$ comparisons for $N < N_0$, and afterwards it will be bounded by $O(N \log N_0 + N)$ comparisons, which in theory is $O(N)$. This was reasoned by the additional improvement from semi-improved, which first had an impact after when working on $N \gtrapprox 20\,000\,000$.

5.3 Conclusion of Analysis

Our analysis of the implementations has shown a few things. The most obvious bit is the difference in improved versions between different dimensions. For $k = 2$ we saw that the improved closest points algorithm has $O(N)$ bound for comparisons in practice, but for $k = 3$ the comparisons had a lower bound of $N \log N$ comparisons for "small" N. We hypothesized that this is due to the fact that the improved version is only useful when we have $N(1 - 1/2k) > kcN^{1-1/k}$. For 2D this happens at $N > 1\,024$, for 3D this is at $N > 2\,176\,782$. One can imagine that as we go into even higher spaces of k, this requirement will increase drastically. We let the analysis end with an open question of whether it's actually practical to use the improved closest points algorithm over the basic one for $k > 2$.

For running time, we do see, that it takes longer to solve for higher dimensions, even though their upper bounds are both $O(N \log N)$. This is due to the large constant, that increase on bigger sizes of dimensions. We see an increased slab size on $kcN^{1-1/k}$ points, where c is $4 \cdot (3^{k-1})$, which means the constants increases exponentially on k. This means that in theory we have a good upper bound, but in practice the constants are too big, to be a tight bound on running time. This means it will require large sizes of N, for the running time to converge to $O(N \log N)$ in large dimensions.

6 Comparisons to other work and ideas for future work

In Shamos and Hoey 1975 the divide-and-conquer algorithm to solve the closest points problem in 2 dimensional space is given. Later in Ge, Wang, and Zhu 2006 an improvement is made, where the complexity of computing distance is reduced from $3n \log n$ to $\frac{3n \log n}{2}$ by smartly discarding points to compare in the slab, making it possible that in worst case one only has to compute half the amount of distance calculations compared to the original solution. Improving our implementation with this, opens up for future work when considering 2 dimensional space. It is possible to extend the algorithms from 3 dimensional space to k dimensional space through abstraction, this opens up for additional analysis, where we could check if this abstraction presents us with complications to the running time.

7 Conclusion

In this project, we looked into the problem "Closest points", which given a collection of points, would find the two points closest to each other. We initially see the running time of an brute force solution would take $\Theta(N^2)$ time, as it would require a point wise comparison between all points. Instead by adapting to a divide and conquer approach, we could initially reduce it to $O(N \log^{k-1} N)$ running time for k-dimensional point set, which we with further improvements could reduce to $O(N \log N)$ for all dimensions.

To test our theory, we implemented this approach for closest points in 2-dimension and 3-dimension. In addition to the theoretical best implementation, we also tested some of the implementations, which excluded some improvements, as it could happen, that we use too much energy in improving the worst case running time, that the average run time worsen.

To generate our data set, we used some of the more common distributions, to simulate what the average case looks like, in the form of Uniform and Gaussian distributed on all axes, and a special distribution, which was designed to have a worse running time.

We had our initial hypothesis, that our new approach to solving closest points was faster than the brute force approach. We found that when working with a data set containing more than 100 points, our divide and conquer approach was faster, and that brute force took over a minute for 100 000 points, meanwhile it took less than 30 seconds for our divide and conquer approach to solve for 10 000 000 points.

When analysing our two 2-dimensional versions, the basic version which chooses a hyperplane cut in the middle of the X-axis, and the improved version, which tried to find a good hyperplane cut, we initially hypothesised that for the 2-dimensional implementation, the improved version would be faster, if we spared enough time in solving for the slab, than time used in finding the good cut. What we found was that the basic version was faster than the improved version, meaning the time spent on finding the good cut took longer than the time we spared by having a good hyperplane cut. We also found, that our theoretical bound on number of comparisons. We expected it to be upper bounded by $O(N \log(N))$ for the basic implementation, and we found it to be approximately $O(N \log^{0.1}(N))$ in practice, for data sets distributed with Gaussian and Uniform distributions, and even less for our special distribution. We did however hypothesise there to be more comparisons for our special distribution, since we assumed that a larger slab meant more comparisons, but that isn't necessarily true, as it also depends on how the points are distributed in the slab.

For our improved version, we were expecting the number of comparisons to be upper bounded by $O(N)$. In practice we saw the number of comparisons converge towards a line on the form of $c \cdot N$, when increasing our test size, which aligns with our hypothesis.

For our 3-dimensional implementation, we tested 3 different implementation, the basic and semi-improved versions with expected running time of $O(N \log^2(N))$, and the improved version with expected running time of $O(N \log(N))$. We saw in our tests, that the basic version ran the fastest on the Gaussian distribution, out of the 3 implementations, at least up to a collection of 10 000 000 points. We saw they all ran within their expected worst bound running time for all distributions. We also see, that our basic version gets relatively slower running speed when solving for larger collections compared to the other solutions. So we hypothesise, that for larger collections, eventually both the improved and semi-improved will be faster than the basic version.

As for the number of comparisons, we hypothesised that for the basic versions we are bounded by $O(N \log^2(N))$, semi-improved $O(N \log N)$, and for the improved we are bounded by $O(N)$. For the basic version, we found it easily in bound, and in our tests, we found it to be in worst case $O(N \log^{1.2} N)$, but it differs between the distribution of points. Meanwhile the semi-improved directly converged towards $O(N \log N)$, which follows our hypothesis. Though our improved version also converged towards $O(N \log N)$, which is in opposition to our hypothesis. Though it can be argued, that it could converge towards $O(N)$, when reaching larger N, as the improved version has limited improvement for small N.

We also see a common theme for the running time of all the implementations, which is usually on the form of $T = c_1 f(N) + c_2 g(N)$, for where it holds $c_2 < c_1$ and $O(f(N)) < O(g(N))$. Reasoned by how all the graphs are plotted, where they initially have what looks like linear running time, but as time progresses its running time increases. Which is explained by, it first is bounded by the running time of $O(f(N))$, but as we increase the size of our set of points, eventually $c_2 g(N)$ becomes the dominant part, and we are bounded by $O(g(N))$.

8 Acknowledgements

We want to give a special thanks to our advisor Peyman Afshani, whom helped us through the process of making this project. Even in the time of a pandemic, which affected us all, he took time to meet with us.

9 References

Shamos, Michael Ian and Dan Hoey (1975). “Closest-point Problems”. In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. IEEE, pp. 151–162.

Bentley, Jon Louis and Michael Ian Shamos (1976). “Divide-And-Conquer in multidimensional space”. In: *STOC ’76: Proceedings of the eighth annual ACM symposium on Theory of computing*. Association for Computing Machinery, New York, NY, United States, pp. 220–230. ISBN: 978-1-4503-7414-9.

Ge, Qi, Haitao Wang, and Hong Zhu (Jan. 2006). “An Improved Algorithm for Finding the Closest Pair of Points”. In: *J. Comput. Sci. Technol.* 21, pp. 27–31. DOI: 10.1007/s11390-006-0027-7.

10 Appendix

10.1 Technical details

The implementation code is written in the Java programming language (specifically Java 12.0.1). We ran all statistical tests on the same machine, a computer with a 2.8GHz quad-core Intel I7 CPU with 16GB of 2133MHz DDR3 RAM, running macOS Catalina version 10.15.4.

10.2 Closest points 2D Implementation

10.2.1 Point 2D

```java
public class Point2D {
    public final double x, y;
    public boolean isLeft = false;

    public Point2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public static double distance(Point2D a, Point2D b) {
        double x = a.x - b.x;
        double y = a.y - b.y;
        return Math.sqrt(x*x + y*y);
    }
}
```

10.2.2 Pair 2D

```java
public class Pair {
    public Point2D a, b;

    public Pair(Point2D a, Point2D b) {
        this.a = a;
        this.b = b;
    }

    public double distance() {
        return Point2D.distance(a, b);
    }
}
```

10.2.3 Cutplane2D

```java
public class Cutplane2D {
    public axis axis;
    public double cutpos;
    public Cutplane2D(axis ax, double cut){
        axis = ax;
        cutpos = cut;
    }
}
```

Bruteforce 2D implementation

```java
    public static Pair bruteForce(Point2D[] points) {
        int n = points.length;
        if (n < 2) {
            return null;
        }

        Pair closestPair = new Pair(points[0], points[1]);
        if (n == 2) {
            return closestPair;
        }

        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 1; j < n; j++) {
                Pair newPair = new Pair(points[i], points[j]);
                if (newPair.distance() < closestPair.distance()) {
                    closestPair = newPair;
                }
            }
        }

        return closestPair;
    }
```

10.2.4 Closest points 2D Implementation

```java
public class ClosestPoint2D{
   public static Pair bentleyShamos(Point2D[] pointsSortedByX, Point2D[]
       pointsSortedByY) {
        int n = pointsSortedByX.length;
        if (n <= 3) {
            CompCounter.addComp(n-1);
            return ClosestPoints2D.bruteForce(pointsSortedByX);
        }

        int mid = n/2;

        Point2D[] pointsLeftOfCenter = Arrays.copyOfRange(pointsSortedByX, 0, mid);
        Point2D[] pointsRightOfCenter = Arrays.copyOfRange(pointsSortedByX, mid, n);

        for(int i = 0; i < mid; i++){
            pointsSortedByX[i].isLeft = true;
        }
        for(int i = mid; i < n; i++){
            pointsSortedByX[i].isLeft = false;
        }
        int countLeft = 0, countRight = 0;

        Point2D[] pointsLeftOfCenterSortedByY = new Point2D[mid];
        Point2D[] pointsRightOfCenterSortedByY = new Point2D[n-mid];

        for(int i = 0; i < n; i++){
            if(pointsSortedByY[i].isLeft)
                pointsLeftOfCenterSortedByY[countLeft++] = pointsSortedByY[i];
            else
                pointsRightOfCenterSortedByY[countRight++] = pointsSortedByY[i];
        }

        Pair closestPairLeft = bentleyShamos(pointsLeftOfCenter,
            pointsLeftOfCenterSortedByY);
        Pair closestPairRight = bentleyShamos(pointsRightOfCenter,
            pointsRightOfCenterSortedByY);

        Pair closestPair = closestPairLeft;
        if (closestPairLeft.distance() > closestPairRight.distance()) {
            closestPair = closestPairRight;
        }

        int size = 0;
        Point2D[] slab = new Point2D[n];
        double distance = closestPair.distance();
        double middleLine = pointsRightOfCenter[0].x;

        for (Point2D point : pointsSortedByY) {
            if (abs(middleLine - point.x) < distance) {
                slab[size++] = point;
            }
        }
        for (int i = 0; i < size - 1; i++) {
            for (int j = i + 1; j < size; j++) {
                Pair newPair = new Pair(slab[i], slab[j]);

                if (abs(newPair.a.y - newPair.b.y) >= distance){ break;}
                CompCounter.addComp();
                if (newPair.distance() < closestPair.distance()) {
                    distance = newPair.distance();
                    closestPair = newPair;
                }
            }
        }

        return closestPair;
    }

    public static Pair bentleyShamos(Point2D[] points) {
        Point2D[] pointsSortedByX = points.clone();
        Point2D[] pointsSortedByY = points.clone();
        sortByX(pointsSortedByX);
        sortByY(pointsSortedByY);
        return bentleyShamos(pointsSortedByX, pointsSortedByY);
    }

    // The functions below are abstracted away to hide usage of the standard Java
        library.

    private static double abs(double value) {
        return Math.abs(value);
    }

    private static void sortByX(Point2D[] points) {
        Arrays.sort(points, (p1, p2) -> Double.compare(p1.x, p2.x));
    }

    private static void sortByY(Point2D[] points) {
        Arrays.sort(points, (p1, p2) -> Double.compare(p1.y, p2.y));
    }
}
```

10.2.5 Closest points 2D Improved Implementation

```
public class ClosestPoints2DImproved {
    public static Pair bentleyShamos(Point2D[] pointsSortedByX, Point2D[]
        pointsSortedByY) {
        int n = pointsSortedByX.length;
        if (n <= 3) {
            CompCounter.addComp(n-1);
            return ClosestPoints2DImproved.bruteForce(pointsSortedByX);
        }

        int minInSides = (int) Math.ceil(n / 8.0);
        int mustContain = (int) Math.ceil(2*12*Math.pow(n,1/2));
        double largestSoFar = 0;
        Cutplane2D cut = new Cutplane2D(axis.X, pointsSortedByX[n/2].x);


        for (int i = minInSides; i<n-minInSides-mustContain; i++ ){
            double size = -pointsSortedByX[i].x+pointsSortedByX[i+mustContain].x;
            if(size > largestSoFar){
                largestSoFar = size;
                cut = new Cutplane2D(axis.X, pointsSortedByX[i].x + size/2);
            }
        }
        for (int i = minInSides; i<n-minInSides-mustContain; i++ ){
            double size = -pointsSortedByY[i].y+pointsSortedByY[i+mustContain].y;
            if(size > largestSoFar){
                largestSoFar = size;
                cut = new Cutplane2D(axis.Y, pointsSortedByY[i].y + size/2);
            }
        }
        ArrayList<Point2D> pointsLeftOfCenterX = new ArrayList<>(),
            pointsRightOfCenterX = new ArrayList<>();
        ArrayList<Point2D> pointsLeftOfCenterY = new ArrayList<>(),
            pointsRightOfCenterY = new ArrayList<>();

        switch (cut.axis){
            case X:
                for(int i = 0; i < n; i++){
                    if(cut.cutpos > pointsSortedByX[i].x)
                         pointsLeftOfCenterX.add(pointsSortedByX[i]);
                    else pointsRightOfCenterX.add(pointsSortedByX[i]);
                    if(cut.cutpos > pointsSortedByY[i].x)
                        pointsLeftOfCenterY.add(pointsSortedByY[i]);
                    else pointsRightOfCenterY.add(pointsSortedByY[i]);
                } break;
            case Y:
                for(int i = 0; i < n; i++){
                    if(cut.cutpos > pointsSortedByX[i].y)
                        pointsLeftOfCenterX.add(pointsSortedByX[i]);
                    else pointsRightOfCenterX.add(pointsSortedByX[i]);
                    if(cut.cutpos > pointsSortedByY[i].y)
                        pointsLeftOfCenterY.add(pointsSortedByY[i]);
                    else pointsRightOfCenterY.add(pointsSortedByY[i]);
                } break;
            default: break; //Not possible
            }

        int leftSize = pointsLeftOfCenterX.size();
        int rightSize = pointsRightOfCenterX.size();
        Point2D[] leftX = new Point2D[leftSize], leftY = new Point2D[leftSize];
        pointsLeftOfCenterX.toArray(leftX); pointsLeftOfCenterY.toArray(leftY);
        Point2D[] rightX = new Point2D[rightSize], rightY = new Point2D[rightSize];
        pointsRightOfCenterX.toArray(rightX); pointsRightOfCenterY.toArray(rightY);

        Pair closestPairLeft = bentleyShamos(leftX, leftY);
        Pair closestPairRight = bentleyShamos(rightX, rightY);

        Pair closestPair = closestPairLeft;
        if (closestPairLeft.distance() > closestPairRight.distance()) {
            closestPair = closestPairRight;
        }

        int size = 0;
        Point2D[] slab = new Point2D[n];
        double distance = closestPair.distance();

        for (Point2D point : pointsSortedByY) {
            boolean inSlab = false;
            switch(cut.axis){
                case X: inSlab = abs(cut.cutpos - point.x)< distance; break;
                case Y: inSlab = abs(cut.cutpos - point.y)< distance; break;
                default: break;
            }
            if (inSlab) {
                slab[size++] = point;
            }
        }

        for (int i = 0; i < size - 1; i++) {
            for (int j = i + 1; j < size; j++) {
                Pair newPair = new Pair(slab[i], slab[j]);
                boolean inDistance = false;
                switch(cut.axis){
                    case X: inDistance = abs(newPair.a.y - newPair.b.y) >= distance;
                        break;
                    case Y: inDistance = abs(newPair.a.x - newPair.b.x) >= distance;
                        break;
                    default: break;
                }
                if (inDistance) break;

                CompCounter.addComp();
                if (newPair.distance() < closestPair.distance()) {
                    distance = newPair.distance();
                    closestPair = newPair;
                }
            }
        }

        return closestPair;
    }

    public static Pair bentleyShamos(Point2D[] points) {
```

```java
        Point2D[] pointsSortedByX = points.clone();
        Point2D[] pointsSortedByY = points.clone();
        sortByX(pointsSortedByX);
        sortByY(pointsSortedByY);
        return bentleyShamos(pointsSortedByX, pointsSortedByY);
    }

    // The functions below are abstracted away to hide usage of the standard Java
        library.

    private static double abs(double value) {
        return Math.abs(value);
    }

    private static void sortByX(Point2D[] points) {
        Arrays.sort(points, (p1, p2) -> Double.compare(p1.x, p2.x));
    }

    private static void sortByY(Point2D[] points) {
        Arrays.sort(points, (p1, p2) -> Double.compare(p1.y, p2.y));
    }
}
```

10.3 Closest points 3D Implementation

10.3.1 Point3D

```java
public class Point3D {
    public final double x, y, z;
    public boolean isLeft = false, inSlab = false;


    public Point3D(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public static double distance(Point3D a, Point3D b) {
        double x = a.x - b.x;
        double y = a.y - b.y;
        double z = a.z - b.z;
        return Math.sqrt(x*x + y*y + z*z);
    }
}
```

10.3.2 Pair 3D

```java
public class Pair {
    public Point3D a, b;
    public Pair(Point3D a, Point3D b) {
        this.a = a;
        this.b = b;
    }
    public double distance() {
        return Point3D.distance(a, b);
    }
}
```

10.3.3 Cutplane3D

```
public class Cutplane3D {
    public axis axis;
    public double cutpos;
    public Cutplane3D(axis ax, double cut){
        axis = ax;
        cutpos = cut;
    }
}
```

10.3.4 Bruteforce 3D implementation

```
public static Pair bruteForce(Point3D[] points) {
    int n = points.length;
    if (n < 2) {
        return null;
    }

    Pair closestPair = new Pair(points[0], points[1]);
    if (n == 2) {
        return closestPair;
    }

    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            Pair newPair = new Pair(points[i], points[j]);
            if (newPair.distance() < closestPair.distance()) {
                closestPair = newPair;
            }
        }
    }

    return closestPair;
}
```

10.3.5 Closest points 3D implementation

```
public class ClosestPoints3D {
    public static Pair bentleyShamosHelper(Point3D[] pointsSortedByX, Point3D[]
        pointsSortedByY, Point3D[] pointsSortedByZ) {
        int n = pointsSortedByX.length;
        if (n <= 3) {
            CompCounter.addComp(n-1);
            return ClosestPoints3D.bruteForce(pointsSortedByX);
        }
        //split the point set for X
        int mid = n/2;

        Point3D[] pointsLeftOfCenterX = Arrays.copyOfRange(pointsSortedByX, 0, mid);
        Point3D[] pointsRightOfCenterX = Arrays.copyOfRange(pointsSortedByX, mid, n);

```

```java
        //Split the point set for Y and Z and keep it sorted
        for(int i = 0; i < n; i++){
            pointsSortedByX[i].isLeft = i < mid;
        }

        int leftCountY=0, leftCountZ=0, rightCountY=0, rightCountZ=0;

        Point3D[] pointsLeftOfCenterY = new Point3D[mid],   pointsLeftOfCenterZ  = new Point3D[mid],
                  pointsRightOfCenterY = new Point3D[n-mid], pointsRightOfCenterZ = new Point3D[n-mid];

        for(int i = 0; i < n; i++){
            if (pointsSortedByY[i].isLeft)   pointsLeftOfCenterY[leftCountY++] = pointsSortedByY[i];
            else                           pointsRightOfCenterY[rightCountY++] = pointsSortedByY[i];
            if (pointsSortedByZ[i].isLeft)   pointsLeftOfCenterZ[leftCountZ++] = pointsSortedByZ[i];
            else                           pointsRightOfCenterZ[rightCountZ++] = pointsSortedByZ[i];
        }

        //Recursive call to archieve divide and conquer
        Pair left = bentleyShamosHelper(pointsLeftOfCenterX, pointsLeftOfCenterY, pointsLeftOfCenterZ);
        Pair right = bentleyShamosHelper(pointsRightOfCenterX, pointsRightOfCenterY, pointsRightOfCenterZ);

        //Find best solution so far
        Pair closestPair = left;
        if (left.distance() > right.distance()) {
            closestPair = right;
        }

        //Solve for the slab
        int size = 0;
        Point3D[] slab = new Point3D[n];
        double distance = closestPair.distance();
        double middleLine = pointsRightOfCenterX[0].x;
        //Find the slab points
        for (Point3D point : pointsSortedByY) {
            if (Math.abs(middleLine - point.x) < distance) {
                slab[size++] = point;
            }
        }
        //Get the sorted slab for y and z
        Point3D[] slabY = Arrays.copyOfRange(slab, 0, size);
        for(int i = 0; i < n; i++) pointsSortedByY[i].inSlab = false;
        for(int i = 0; i < size; i++) slabY[i].inSlab = true;
        Point3D[] slabZ = Arrays.stream(pointsSortedByZ).filter(p -> p.inSlab).toArray(Point3D[]::new);
        //Run the fixed radius neighbors algorithm
        Pair pairs[] = S(slabZ, slabY, closestPair.distance());

        Pair lowest = closestPair;
        for (Pair pair : pairs) {
            if (pair.distance() < lowest.distance()) {
                lowest = pair;
            }
        }

        return lowest;
    }

    public static Pair bentleyShamos(Point3D[] points) {
        Point3D[] pointsSortedByX = points.clone();
        Point3D[] pointsSortedByY = points.clone();
        Point3D[] pointsSortedByZ = points.clone();
        Arrays.sort(pointsSortedByX, (p1, p2) -> Double.compare(p1.x, p2.x));
        Arrays.sort(pointsSortedByY, (p1, p2) -> Double.compare(p1.y, p2.y));
        Arrays.sort(pointsSortedByZ, (p1, p2) -> Double.compare(p1.z, p2.z));
        return bentleyShamosHelper(pointsSortedByX, pointsSortedByY, pointsSortedByZ);
    }

    public static Pair[] S(Point3D[] pointsSortedByZ, Point3D[] pointsSortedByY,
        double delta) {
        int n = pointsSortedByZ.length;
        if (n <= 1) {
            CompCounter.addComp(n-1);
            return new Pair[0];
        }

        int mid = n/2;

        Point3D[] pointsLeftOfCenter = Arrays.copyOfRange(pointsSortedByZ, 0, mid);
        Point3D[] pointsRightOfCenter = Arrays.copyOfRange(pointsSortedByZ, mid, n);

        for(int i = 0; i < mid; i++){
            pointsSortedByZ[i].isLeft = true;
        }
        for(int i = mid; i < n; i++){
            pointsSortedByZ[i].isLeft = false;
        }
        int countLeft = 0, countRight = 0;

        Point3D[] pointsLeftOfCenterSortedByY = new Point3D[mid];
        Point3D[] pointsRightOfCenterSortedByY = new Point3D[n-mid];

        for(int i = 0; i < n; i++){
            if(pointsSortedByY[i].isLeft)
                pointsLeftOfCenterSortedByY[countLeft++] = pointsSortedByY[i];
            else
                pointsRightOfCenterSortedByY[countRight++] = pointsSortedByY[i];
        }

        Pair neighborLeft[] = S(pointsLeftOfCenter, pointsLeftOfCenterSortedByY, delta
            );
        Pair neighborRight[] = S(pointsRightOfCenter, pointsRightOfCenterSortedByY,
            delta);


        ArrayList<Point3D> slab = new ArrayList<>();
        double middleLine = pointsRightOfCenter[0].x;

        for (Point3D point : pointsSortedByY) {
            if (Math.abs(middleLine - point.x) < delta) {
                slab.add(point);
            }
        }

        ArrayList<Pair> result = new ArrayList<>();
        result.addAll(Arrays.asList(neighborLeft));
        result.addAll(Arrays.asList(neighborRight));

        for (int i = 0; i < slab.size() - 1; i++) {
            for (int j = i + 1; j < slab.size(); j++) {
                Pair newPair = new Pair(slab.get(i), slab.get(j));
                if (Math.abs(newPair.a.y - newPair.b.y) >= delta) break;
                CompCounter.addComp();
                if (newPair.distance() <= delta) {
                    result.add(newPair);
                }
            }
        }

        return result.toArray(new Pair[result.size()]);
    }
}
```

10.4 Closest points 3D implementation improved version

10.4.1 Sparse fixed radius nearest neighbor implementation

```
public static Pair[] SHelper(Point3D[] points, axis ax, double delta) {
    Point3D[] pointsSortedByX = points.clone();
    Point3D[] pointsSortedByY = points.clone();
    Point3D[] pointsSortedByZ = points.clone();
    Arrays.sort(pointsSortedByX, (p1, p2) -> Double.compare(p1.x, p2.x));
    Arrays.sort(pointsSortedByY, (p1, p2) -> Double.compare(p1.y, p2.y));
    Arrays.sort(pointsSortedByZ, (p1, p2) -> Double.compare(p1.z, p2.z));
    switch (ax){
        case X: return S(pointsSortedByY, pointsSortedByZ, delta, ax);
        case Y: return S(pointsSortedByX, pointsSortedByZ, delta, ax);
        case Z: return S(pointsSortedByX, pointsSortedByY, delta, ax);
        default: return null; //This case shouldn't happen, just returning null to
            satisfy compiler
    }
}

public static Pair[] S(Point3D[] pointsSortedByA, Point3D[] pointsSortedByB, double
    delta, axis ax) {
    int n = pointsSortedByA.length;
    if (n <= 1) {
        return new Pair[0];
    }

    int minInSides = (int) Math.ceil(n / 8.0);
    int atMostContain = (int) Math.floor(2*36*Math.pow(n,1-1/2));
    Cutplane3D cut = new Cutplane3D(axis.X,0);

    switch (ax){
        case X:
            for(int i = minInSides; i < n-minInSides+1; i++){
                double sizey = pointsSortedByA[Math.min(i+atMostContain, n-minInSides
                    -1)].y-pointsSortedByA[i].y;
                double sizez = pointsSortedByB[Math.min(i+atMostContain, n-minInSides
                    -1)].z-pointsSortedByB[i].z;
                if(sizey > 2*delta || (atMostContain>n-n/4)){
                    cut = new Cutplane3D(axis.Y, (pointsSortedByA[i].y+sizey/2));
                    break;
                }else if (sizez > 2*delta){
                    cut = new Cutplane3D(axis.Z, (pointsSortedByB[i].z+sizez/2));
                    break;
                }
            } break;
        case Y:
            for(int i = minInSides; i < n-minInSides; i++){
                double sizex = pointsSortedByA[Math.min(i+atMostContain, n-minInSides
                    -1)].x-pointsSortedByA[i].x;
                double sizez = pointsSortedByB[Math.min(i+atMostContain, n-minInSides
                    -1)].z-pointsSortedByB[i].z;
                if(sizex < 2*delta || atMostContain > n-n/4){
                    cut = new Cutplane3D(axis.X, (pointsSortedByA[i].x+sizex/2));
                    break;
                }else if (sizez < 2*delta){
                    cut = new Cutplane3D(axis.Z, (pointsSortedByB[i].z+sizez/2));
                    break;
                }
            } break;
        case Z:
            for(int i = minInSides; i < n-minInSides; i++){
                double sizex = pointsSortedByA[Math.min(i+atMostContain, n-minInSides
                    -1)].x-pointsSortedByA[i].x;
                double sizey = pointsSortedByB[Math.min(i+atMostContain, n-minInSides
                    -1)].y-pointsSortedByB[i].y;
                if(sizey > 2*delta || atMostContain > n-n/4){
                    cut = new Cutplane3D(axis.Y, (pointsSortedByA[i].y+sizey/2));
                    break;
                }else if (sizex > 2*delta || atMostContain > n-n/4){
                    cut = new Cutplane3D(axis.X, (pointsSortedByA[i].x+sizex/2));
                    break;
                }
            } break;
    }

    ArrayList<Point3D> pointsLeftA = new ArrayList<>(), pointsRightA = new ArrayList
        <>(), pointsLeftB = new ArrayList<>(), pointsRightB = new ArrayList<>();
    switch (ax){
        case X:
            switch (cut.axis){
                case X: //Impossible case, since this is cutplane from 3D
                case Y:
                for(int i = 0; i < n; i++){
                    if(pointsSortedByA[i].y<cut.cutpos)
                         pointsLeftA.add(pointsSortedByA[i]);
                    else pointsRightA.add(pointsSortedByA[i]);
                    if(pointsSortedByB[i].y<cut.cutpos)
                         pointsLeftB.add(pointsSortedByB[i]);
                    else pointsRightB.add(pointsSortedByB[i]);
                }break;
                case Z:
                for(int i = 0; i < n; i++){
                    if(pointsSortedByA[i].z<cut.cutpos)
                         pointsLeftA.add(pointsSortedByA[i]);
                    else pointsRightA.add(pointsSortedByA[i]);
                    if(pointsSortedByB[i].z<cut.cutpos)
                         pointsLeftB.add(pointsSortedByB[i]);
                    else pointsRightB.add(pointsSortedByB[i]);
                }break;
            }   break;
        case Y:
        switch (cut.axis){
            case X:
                for(int i = 0; i < n; i++){
                    if(pointsSortedByA[i].x<cut.cutpos)
                        pointsLeftA.add(pointsSortedByA[i]);
                    else pointsRightA.add(pointsSortedByA[i]);
                    if(pointsSortedByB[i].x<cut.cutpos)
                        pointsLeftB.add(pointsSortedByB[i]);
                    else pointsRightB.add(pointsSortedByB[i]);
                }break;
            case Y:break;

            case Z:
                for(int i = 0; i < n; i++){
                    if(pointsSortedByA[i].z<cut.cutpos)
                        pointsLeftA.add(pointsSortedByA[i]);
                    else pointsRightA.add(pointsSortedByA[i]);
                    if(pointsSortedByB[i].z<cut.cutpos)
                        pointsLeftB.add(pointsSortedByB[i]);
                    else pointsRightB.add(pointsSortedByB[i]);
                }break;
        }break;
        case Z:
        switch (cut.axis){
            case X:
                for(int i = 0; i < n; i++){
                    if(pointsSortedByA[i].x<cut.cutpos)
                        pointsLeftA.add(pointsSortedByA[i]);
                    else pointsRightA.add(pointsSortedByA[i]);
                    if(pointsSortedByB[i].x<cut.cutpos)
                        pointsLeftB.add(pointsSortedByB[i]);
                    else pointsRightB.add(pointsSortedByB[i]);
                }break;
            case Y:
                for(int i = 0; i < n; i++){
                    if(pointsSortedByA[i].y<cut.cutpos)
                        pointsLeftA.add(pointsSortedByA[i]);
                    else pointsRightA.add(pointsSortedByA[i]);
                    if(pointsSortedByB[i].y<cut.cutpos)
                        pointsLeftB.add(pointsSortedByB[i]);
                    else pointsRightB.add(pointsSortedByB[i]);
                }break;
            case Z: break;
        }break;
    }
    int leftSize = pointsLeftA.size(), rightSize = pointsRightA.size();
    Point3D[] leftA = new Point3D[leftSize], leftB = new Point3D[leftSize], rightA =
        new Point3D[rightSize], rightB = new Point3D[rightSize];
    pointsLeftA.toArray(leftA); pointsLeftB.toArray(leftB); pointsRightA.toArray(
        rightA); pointsRightB.toArray(rightB);


```

```java
    Pair neighborLeft[] = S(leftA, leftB, delta, ax);
    Pair neighborRight[] = S(rightA, rightB, delta, ax);

    ArrayList<Point3D> slab = new ArrayList<>();
    double middleLine = cut.cutpos;

    for (Point3D point : pointsSortedByA) {
        double p = 0;
        switch (cut.axis){
            case X: p = point.x; break;
            case Y: p = point.y; break;
            case Z: p = point.z; break;
        }
        if (Math.abs(middleLine - p) < delta) {
            slab.add(point);
        }
    }

    ArrayList<Pair> result = new ArrayList<>();
    result.addAll(Arrays.asList(neighborLeft));
    result.addAll(Arrays.asList(neighborRight));

    for (int i = 0; i < slab.size() - 1; i++) {
        for (int j = i + 1; j < slab.size(); j++) {
            Pair newPair = new Pair(slab.get(i), slab.get(j));
            double a = 0;
            switch (cut.axis){
                case X: a = newPair.a.x - newPair.b.x; break;
                case Y: a = newPair.a.y - newPair.b.y; break;
                case Z: a = newPair.a.z - newPair.b.z; break;
            }
            CompCounter.addComp();
            if ((a) >= delta) break;
            if (newPair.distance() <= delta) {
                result.add(newPair);
            }
        }
    }

    return result.toArray(new Pair[result.size()]);
}
```

10.4.2 Closest points 3D implementation improved version

```java
public class ClosestPoints3DImproved {
    public static Pair bentleyShamosHelper(Point3D[] pointsSortedByX, Point3D[]
        pointsSortedByY, Point3D[] pointsSortedByZ) {
        int n = pointsSortedByX.length;
        if (n <= 3) {
            CompCounter.addComp(n-1);
            return ClosestPoints3DImproved.bruteForce(pointsSortedByX);
        }

        int minInSides = (int) Math.ceil(n / 12.0);
        int mustContain = (int) Math.ceil(3*36*Math.pow(n,1-1/3));
        double largestSoFar = 0;
        Cutplane3D cut = new Cutplane3D(axis.X, pointsSortedByX[n/2].x);
        for (int i = minInSides; i<n-minInSides-mustContain; i++ ){
            double size = -pointsSortedByX[i].x+pointsSortedByX[i+mustContain].x;
            if(size > largestSoFar){
                largestSoFar = size;
                cut = new Cutplane3D(axis.X, pointsSortedByX[i].x + size/2);
            }
        }
        for (int i = minInSides; i<n-minInSides-mustContain; i++ ){
            double size = -pointsSortedByY[i].y+pointsSortedByY[i+mustContain].y;
            if(size > largestSoFar){
                largestSoFar = size;
                cut = new Cutplane3D(axis.Y, pointsSortedByY[i].y + size/2);
            }
        }
        for (int i = minInSides; i<n-minInSides-mustContain; i++ ){
            double size = -pointsSortedByZ[i].z+pointsSortedByZ[i+mustContain].z;
            if(size > largestSoFar){
                largestSoFar = size;
                cut = new Cutplane3D(axis.Z, pointsSortedByZ[i].z + size/2);
            }
        }
        ArrayList<Point3D> pointsLeftOfCenterX = new ArrayList<>(),
            pointsRightOfCenterX = new ArrayList<>();
        ArrayList<Point3D> pointsLeftOfCenterY = new ArrayList<>(),
            pointsRightOfCenterY = new ArrayList<>();
        ArrayList<Point3D> pointsLeftOfCenterZ = new ArrayList<>(),
            pointsRightOfCenterZ = new ArrayList<>();
        switch (cut.axis){
            case X:
                for(int i = 0; i < n; i++){
                    if(cut.cutpos > pointsSortedByX[i].x)
                         pointsLeftOfCenterX.add(pointsSortedByX[i]);
                    else pointsRightOfCenterX.add(pointsSortedByX[i]);
                    if(cut.cutpos > pointsSortedByY[i].x)
                        pointsLeftOfCenterY.add(pointsSortedByY[i]);
                    else pointsRightOfCenterY.add(pointsSortedByY[i]);
                    if(cut.cutpos > pointsSortedByZ[i].x)
                        pointsLeftOfCenterZ.add(pointsSortedByZ[i]);
                    else pointsRightOfCenterZ.add(pointsSortedByZ[i]);
                } break;
                case Y:
                for(int i = 0; i < n; i++){
                    if(cut.cutpos > pointsSortedByX[i].y)
                        pointsLeftOfCenterX.add(pointsSortedByX[i]);
                    else pointsRightOfCenterX.add(pointsSortedByX[i]);
                    if(cut.cutpos > pointsSortedByY[i].y)
                        pointsLeftOfCenterY.add(pointsSortedByY[i]);
                    else pointsRightOfCenterY.add(pointsSortedByY[i]);
                    if(cut.cutpos > pointsSortedByZ[i].y)
                        pointsLeftOfCenterZ.add(pointsSortedByZ[i]);
                    else pointsRightOfCenterZ.add(pointsSortedByZ[i]);
                } break;
            case Z:
                for(int i = 0; i < n; i++){
                    if(cut.cutpos > pointsSortedByX[i].z)
                        pointsLeftOfCenterX.add(pointsSortedByX[i]);
                    else pointsRightOfCenterX.add(pointsSortedByX[i]);
                    if(cut.cutpos > pointsSortedByY[i].z)
                        pointsLeftOfCenterY.add(pointsSortedByY[i]);
                    else pointsRightOfCenterY.add(pointsSortedByY[i]);
```

```java
                if(cut.cutpos > pointsSortedByZ[i].z)
                    pointsLeftOfCenterZ.add(pointsSortedByZ[i]);
                else pointsRightOfCenterZ.add(pointsSortedByZ[i]);
            } break;
        }

        int leftSize = pointsLeftOfCenterX.size();
        int rightSize = pointsRightOfCenterX.size();
        Point3D[] leftX = new Point3D[leftSize], leftY = new Point3D[leftSize], leftZ
            = new Point3D[leftSize];
        pointsLeftOfCenterX.toArray(leftX); pointsLeftOfCenterY.toArray(leftY);
            pointsLeftOfCenterZ.toArray(leftZ);
        Point3D[] rightX = new Point3D[rightSize], rightY = new Point3D[rightSize],
            rightZ = new Point3D[rightSize];
        pointsRightOfCenterX.toArray(rightX); pointsRightOfCenterY.toArray(rightY);
            pointsRightOfCenterZ.toArray(rightZ);
        Pair left = bentleyShamosHelper(leftX, leftY, leftZ);
        Pair right = bentleyShamosHelper(rightX, rightY, rightZ);


        Pair closestPair = left;
        if (left.distance() > right.distance()) {
            closestPair = right;
        }

        int size = 0;
        Point3D[] slab = new Point3D[n];
        double distance = closestPair.distance();
        double middleLine = cut.cutpos;

        for (Point3D point : pointsSortedByX) {
            double p = 0;
            switch (cut.axis){
                case X: p = point.x; break;
                case Y: p = point.y; break;
                case Z: p = point.z; break;
            }
            if (Math.abs(middleLine - p) < distance) {
                slab[size++] = point;
            }
        }

        Point3D[] trimmedSlab = Arrays.copyOfRange(slab, 0, size);


        Pair pairs[] = SHelper(trimmedSlab, cut.axis, closestPair.distance());

        Pair lowest = closestPair;
        for (Pair pair : pairs) {
            if (pair.distance() < lowest.distance()) {
                lowest = pair;
            }
        }

        return lowest;
    }

    public static Pair bentleyShamos(Point3D[] points) {
        Point3D[] pointsSortedByX = points.clone();
        Point3D[] pointsSortedByY = points.clone();
        Point3D[] pointsSortedByZ = points.clone();
        Arrays.sort(pointsSortedByX, (p1, p2) -> Double.compare(p1.x, p2.x));
        Arrays.sort(pointsSortedByY, (p1, p2) -> Double.compare(p1.y, p2.y));
        Arrays.sort(pointsSortedByZ, (p1, p2) -> Double.compare(p1.z, p2.z));

        return bentleyShamosHelper(pointsSortedByX, pointsSortedByY, pointsSortedByZ);
    }
}
```

10.5 Closest points 3D Implementation semi-improved version

```java
public static Pair bentleyShamosHelper(Point3D[] pointsSortedByX, Point3D[]
    pointsSortedByY, Point3D[] pointsSortedByZ) {
        int n = pointsSortedByX.length;
        if (n <= 3) {
            CompCounter.addComp(n-1);
            return ClosestPoints3DSemiImproved.bruteForce(pointsSortedByX);
        }
        //split the point set for X
        int mid = n/2;

        Point3D[] pointsLeftOfCenterX = Arrays.copyOfRange(pointsSortedByX, 0, mid);
        Point3D[] pointsRightOfCenterX = Arrays.copyOfRange(pointsSortedByX, mid, n);

        //Split the point set for Y and Z and keep it sorted
        for(int i = 0; i < n; i++){
            pointsSortedByX[i].isLeft = i < mid;
        }

        int leftCountY=0, leftCountZ=0, rightCountY=0, rightCountZ=0;

        Point3D[] pointsLeftOfCenterY = new Point3D[mid],   pointsLeftOfCenterZ  = new
             Point3D[mid],
                  pointsRightOfCenterY = new Point3D[n-mid], pointsRightOfCenterZ = new
                      Point3D[n-mid];

        for(int i = 0; i < n; i++){
            if (pointsSortedByY[i].isLeft)   pointsLeftOfCenterY[leftCountY++] =
                pointsSortedByY[i];
            else                            pointsRightOfCenterY[rightCountY++] =
                pointsSortedByY[i];
            if (pointsSortedByZ[i].isLeft)   pointsLeftOfCenterZ[leftCountZ++] =
                pointsSortedByZ[i];
            else                            pointsRightOfCenterZ[rightCountZ++] =
                pointsSortedByZ[i];
        }

        //Recursive call to archieve divide and conquer
        Pair left = bentleyShamosHelper(pointsLeftOfCenterX, pointsLeftOfCenterY,
            pointsLeftOfCenterZ);
        Pair right = bentleyShamosHelper(pointsRightOfCenterX, pointsRightOfCenterY,
            pointsRightOfCenterZ);

        //Find best solution so far
        Pair closestPair = left;
        if (left.distance() > right.distance()) {
            closestPair = right;
        }

        //Solve for the slab
        int size = 0;
        Point3D[] slab = new Point3D[n];
        double distance = closestPair.distance();
        double middleLine = pointsRightOfCenterX[0].x;
        //Find the slab points
        for (Point3D point : pointsSortedByY) {
            if (Math.abs(middleLine - point.x) < distance) {
                slab[size++] = point;
            }
        }
        //Get the sorted slab for y and z
        Point3D[] slabY = Arrays.copyOfRange(slab, 0, size);
        for(int i = 0; i < n; i++) pointsSortedByY[i].inSlab = false;
        for(int i = 0; i < size; i++) slabY[i].inSlab = true;
        Point3D[] slabZ = Arrays.stream(pointsSortedByZ).filter(p -> p.inSlab).toArray
            (Point3D[]::new);
        //Run the fixed radius neighbors algorithm
        Pair pairs[] = S(slabY, slabZ, closestPair.distance(), axis.X);

        Pair lowest = closestPair;
        for (Pair pair : pairs) {
            if (pair.distance() < lowest.distance()) {
                lowest = pair;
            }
        }

        return lowest;
    }

    public static Pair bentleyShamos(Point3D[] points) {
        Point3D[] pointsSortedByX = points.clone();
        Point3D[] pointsSortedByY = points.clone();
        Point3D[] pointsSortedByZ = points.clone();
        Arrays.sort(pointsSortedByX, (p1, p2) -> Double.compare(p1.x, p2.x));
        Arrays.sort(pointsSortedByY, (p1, p2) -> Double.compare(p1.y, p2.y));
        Arrays.sort(pointsSortedByZ, (p1, p2) -> Double.compare(p1.z, p2.z));
        return bentleyShamosHelper(pointsSortedByX, pointsSortedByY, pointsSortedByZ);
    }
}
```