

# Simple Randomized Algorithm for Closest Pair Problems

Authors: M. Golin, R. Raman, C. Schwarz, M. Smid

Project By:

Prasad Khedekar (G00973357)

Sarthak Sheth (G00979607)

## Abstract

Simple randomized algorithm for closest pair problems is one of the solution for a basic computational geometry problem. This algorithm is used for finding closest pair from a set of  $n$  points where  $D \geq 2$ . Author explains two different implementations which are using Balanced binary search trees with time complexity  $O(n \log n)$  and Dynamic perfect hashing with time complexity  $O(n)$ . We have Implemented this algorithm using Balanced binary search tree in 2-dimensional case and compared it with a traditional brute force method to verify that this algorithm is simple, randomized, incremental and better than brute force. In addition, we will discuss about high probability bounds and some extensions like  $k$ -closest pair problem and fully dynamic problem.

## Table of Contents

Sr. No.	Topic	Page No.
1.	Introduction	4
2.	Simple Randomized algorithm for Closest Pair Problem	6
3.	Implementation Details	10
4.	Run time Analysis	18
5.	High Probability Bounds	22
6.	The K-closest pairs problem	24
7.	Summary	25
8.	Conclusions	26
9.	References	27

# 1. Introduction

Closest pair problem or closest pair of points problem is one of the basic computational geometry problem in which closest pair of points with a minimum Euclidian distance is calculated in given set of points. This closest pair computational problem is one of the first steps in in many other geometric problems such as minimal spanning tree, nearest neighbors etc. and this problem has many applications in areas such as molecular biology, statistics, pattern recognition and automatic navigation systems.

Many algorithms have already been proposed to solve the closest pair problem in  $\Omega(n \cdot \log n)$  time, but it is well known that this lower bound does not hold if the underlying model of computation is changed appropriately. There are so many methods and algorithms like brute force method, divide and conquer method, Rabin's algorithm, Khuller and Matia's algorithm to solve this problem but this algorithm is much simpler than all others.

Formula used for finding the minimum distance between two points p and q in two-dimensional case is:

$$d(p, q) = \sqrt{(p^x - q^x)^2 + (p^y - q^y)^2}$$

This algorithm takes the randomized permutation of input sequence  $p_1, p_2, p_3, \dots, p_n$  where sequence is a set of  $n$  points in  $D$  dimensions where  $D$  is greater than or equal to 2. This algorithm is a randomized incremental algorithm and uses floor function to calculate expected running time.

Randomized algorithm is an algorithm which takes source of random numbers and during execution it makes random choices which increases the chances of algorithm to achieve good performance in average case. Incremental algorithm is an algorithm where sequence of input is given and it finds a sequence of solutions that build incrementally

while adapting the changes in the input. Hence in this simple randomized algorithm for closest pair problem, we will take input sequence incrementally where first we will calculate minimum distance of first two points and then we will add next point and again apply algorithm to calculate new minimum distance. In computer science floor function maps a real number to the greatest preceding integer. In this algorithm floor function is used to calculate the index of the grid box.

This simple randomized algorithm for closest pair problem requires implementation of dynamic dictionary. Paper shows the implementation using Dynamic perfect hashing which gives overall expected running time of  $O(n)$  and using Balanced search tree which gives expected running time of  $O(n \log^* n)$ . While implementing this algorithm, we have used Red Black tree which is a Balanced binary search tree.

In next section, will discuss the algorithm idea, its notations and operation and then we will discuss the simple randomized algorithm for closest pair problem in detail.

In section 3 we will see the actual implementation details, implementation of this algorithm using balanced binary search tree and implementation using brute force method for closest pair problem. Then we will compare both implementations and their run times using graphs in section 4.

We will then discuss about high probability bounds of this algorithm and their time variants. And in the last we will discuss few extensions of this algorithm such as fully dynamic algorithm and k closest pair problem.

## 2. Simple Randomized Closest Pair Algorithm

The closest pair problem is to find a pair of points  $p, q \in S$  such that  $d(p, q) = \delta(S)$ .

$$\delta(S) = \min\{d(p, q) : p, q \in S \text{ such that } p \neq q\}$$

Let  $S_i = \{p_1, p_2, p_3, \dots, p_i\}$  be the set containing the first  $i$  points of  $S$ .  $G$  is a grid and  $b$  denotes a grid box. Let suppose a square grid with mesh size  $\delta(S_i)$ .

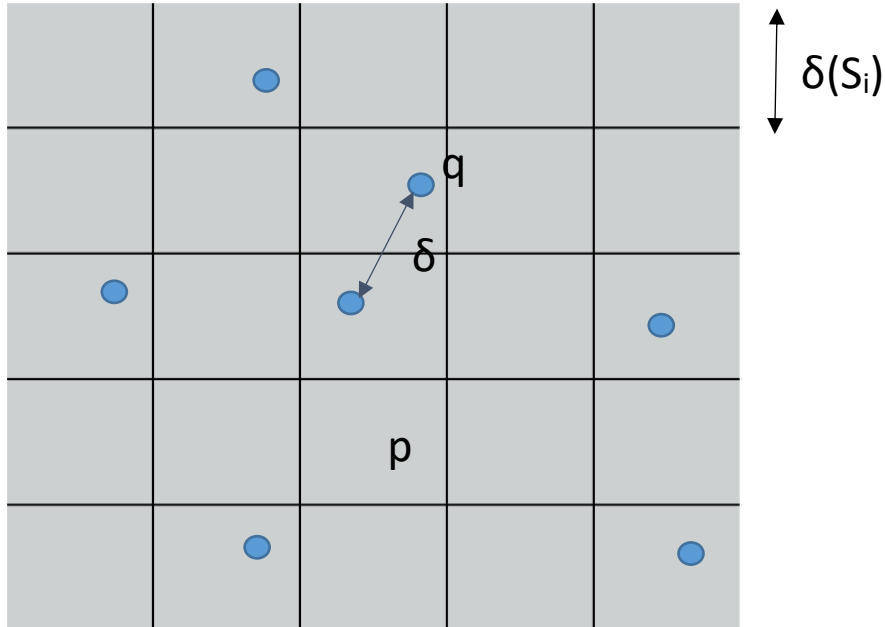


Figure 1 : Grid  $(S, \delta)$

Here each grid box can hold at-most 4 points from  $S_i$ .

Now suppose we insert new point  $p_{i+1}$ . Now we should calculate  $\delta(S_{i+1})$ . Let  $b$  be the grid box in which new point is located. Every point in  $S_i$  which is within the distance  $\delta(S_i)$  must be located within 9 neighbor boxes of  $b$ .

$\delta(S_{i+1}) = \min(d_{i+1}, \delta(S_i))$ . We will calculate  $\delta(S_{i+1})$  using this formula.

If  $d_{i+1} \geq \delta(S_i)$ , then  $\delta(S_{i+1}) = \delta(S_i)$  and hence we don't have to update grid. As shown in Figure 2. Here algorithm inserts  $P_{i+1}$  into the current grid.

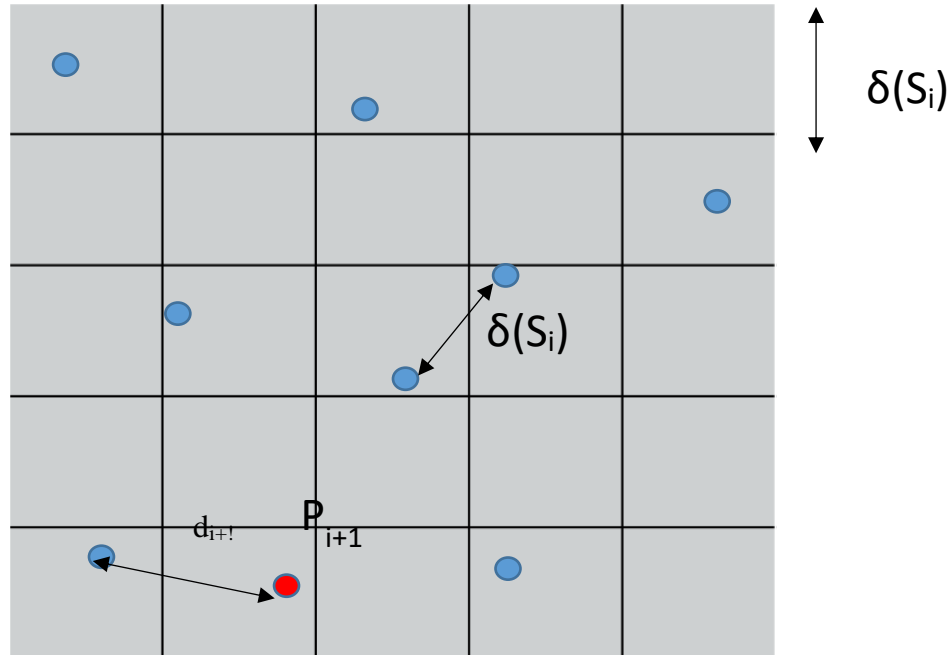


Figure 2 : Insert new point  $P_{i+1}$

Otherwise, if  $d_{i+1} < \delta(S_i)$ , then  $\delta(S_{i+1}) = d_{i+1} < \delta(S_i)$  and it means that  $d(p, p_{i+1}) < \delta(S_i)$  as shown in Figure 3. And

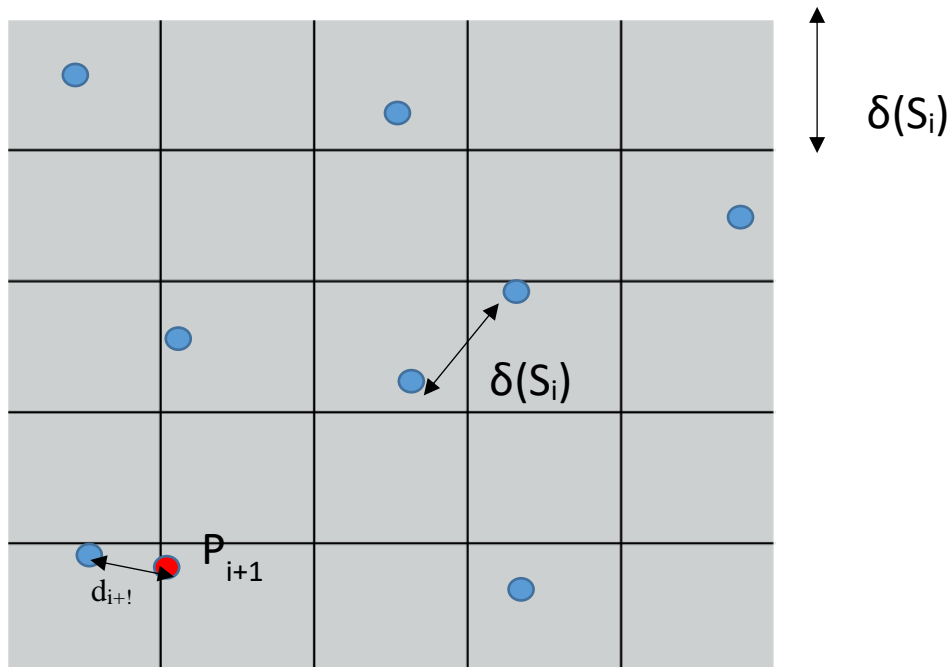


Figure 3 <sub>7</sub>

Here algorithm discards the old grid and creates a new grid with mesh size  $\delta(S_{i+1})$  and inserts new point into the grid as shown in figure 4.

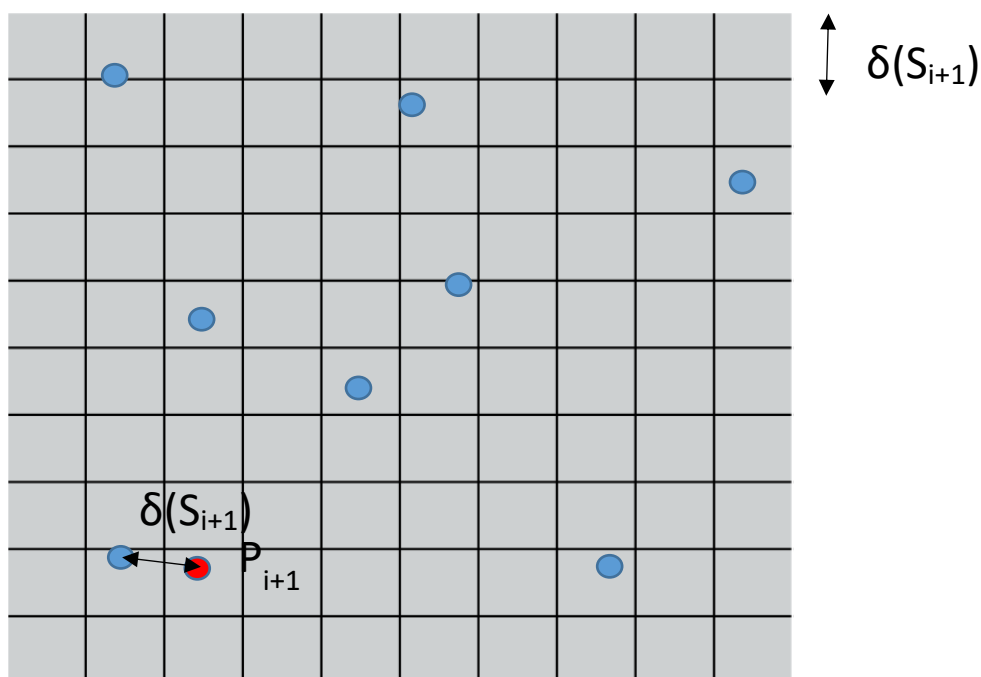


Figure 4

Pseudocode for the Algorithm:

**Algorithm**  $CP(p_1, p_2, \dots, p_n)$

- (1)  $\delta := d(p_1, p_2); \mathcal{G} := Build(S_2, \delta);$
- (2) **for**  $i := 2$  **to**  $n - 1$  **do**
- (3)     **begin**
- (4)          $V := \{Report(\mathcal{G}, b) : b \text{ is a neighbor of the box containing } p_{i+1}\};$
- (5)          $d := \min_{q \in V} d(p_{i+1}, q);$
- (6)         **if**  $d \geq \delta$  **then**  $Insert(\mathcal{G}, p_{i+1})$
- (7)             **else**  $\delta := d; \mathcal{G} := Build(S_{i+1}, \delta);$
- (8)     **end;**
- (9) **return**( $\delta$ ).

This is the actual pseudocode of the simple randomized algorithm for closest pair problem.



In this algorithm, there are 3 operations which we should focus on.

$\text{Build}(S, d) =$  Return a grid  $G$  with mesh size  $\delta$  that contains the points in  $S$ .

$\text{Insert}(G, p) =$  Insert point  $p$  into  $G$ .

$\text{Report}(G, b) =$  Return the list containing the points in grid box  $b$ .

In this above algorithm, first we will calculate minimum distance  $\delta(S_i)$  and build the grid with mesh size  $\delta(S_i)$ . In line 4 to 7, algorithm will insert  $p_{i+1}$  and compute  $\delta(S_{i+1})$ . Line 4 will call Report operation 9 times to find 36 points as each grid box contains at-most 4 points. Line 7 is called only if  $\delta(S_{i+1}) < \delta(S_i)$ .

Now let's see actual implementation of this algorithm using Balanced binary search tree and Hashing in the next section.

### 3. Implementation Details

We have implemented the algorithm presented in this paper and have also run an analysis for comparing the run times using the two variants of the underlying dictionary data structure against the brute force algorithm for finding the closest pair in a given set of points.

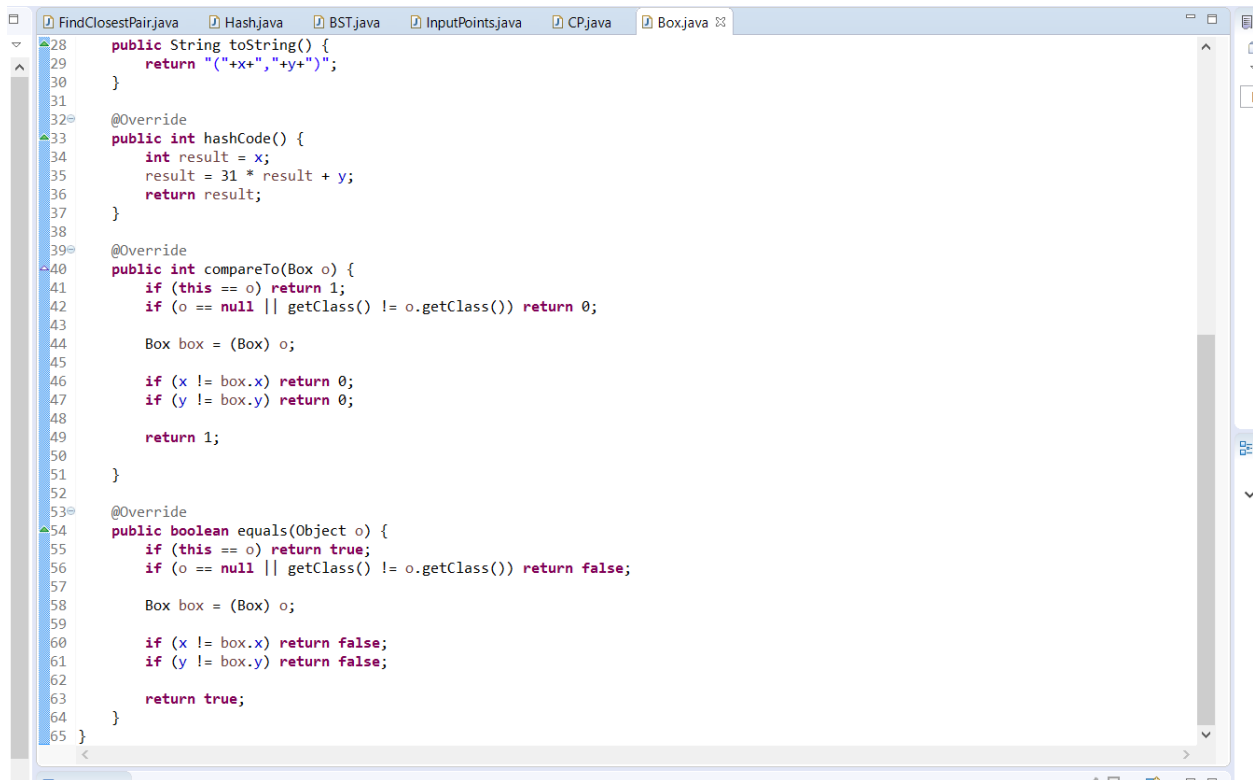
#### 3.1. Algorithm Code Details

This section explains in detail the code structure, data structures and the algorithm implementation and how they function together to compute the closest pair as well as the minimum distance in the given set of points.

##### 3.1.1. Supporting Classes and Data Structures

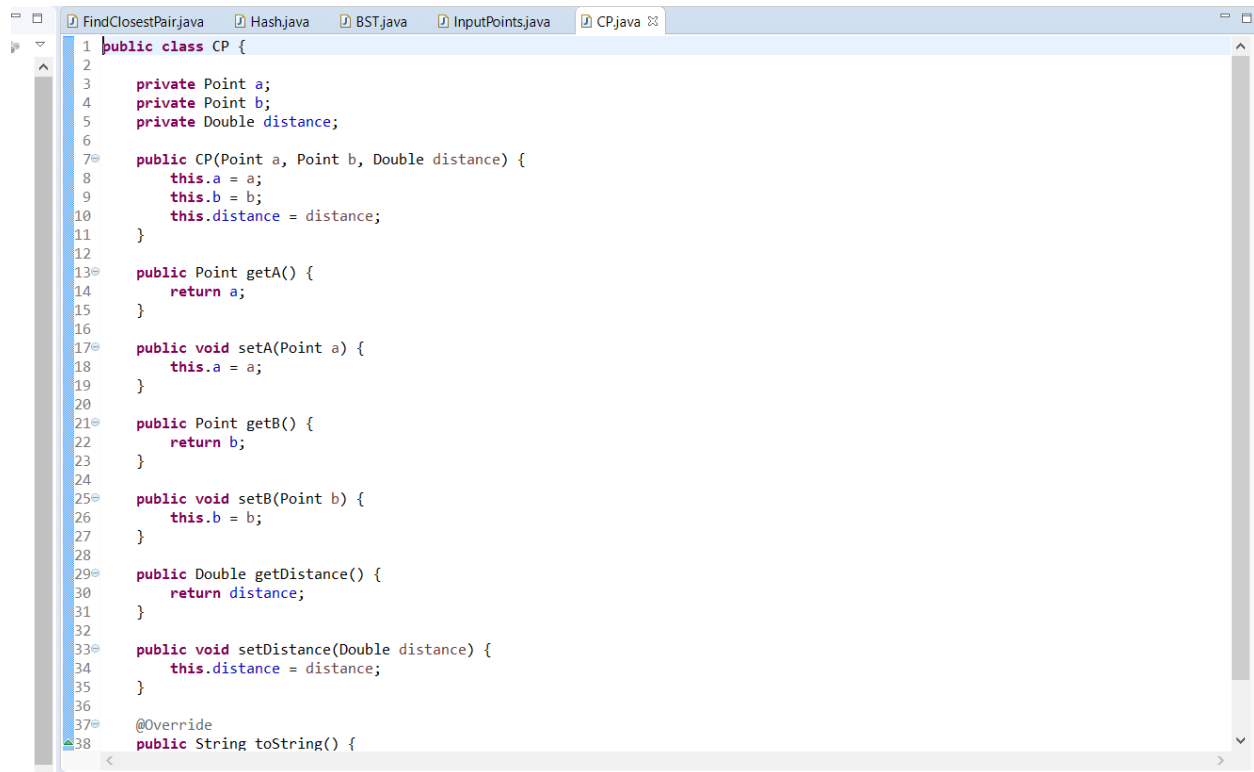
We have implemented the 2D version of the algorithm presented in the paper as the same algorithm is applicable to n-D problem as well. For storing the 2D points, we have defined a class named “Point” which simply includes x, y co-ordinates as its members along with the getters and setters for them. It also includes a “toString” method to return the x, y co-ordinates of the points as a String.

To denote the boxes in the grid, we have created a “Box” class which takes care of creating the required grid boxes as per the algorithm. Even this class contains x, y as member variables but they represent the box indices computed as  $x = p_x/d$  and  $y = p_y/d$ , where “d” is the minimum distance of the current closest pair in the system. It includes getters and setters for x, y along with other methods which are used by the BST and Hashing data structures for maintaining the dictionary.



```
28 public String toString() {
29     return "("+x+", "+y+")";
30 }
31
32 @Override
33 public int hashCode() {
34     int result = x;
35     result = 31 * result + y;
36     return result;
37 }
38
39 @Override
40 public int compareTo(Box o) {
41     if (this == o) return 1;
42     if (o == null || getClass() != o.getClass()) return 0;
43
44     Box box = (Box) o;
45
46     if (x != box.x) return 0;
47     if (y != box.y) return 0;
48
49     return 1;
50 }
51
52
53 @Override
54 public boolean equals(Object o) {
55     if (this == o) return true;
56     if (o == null || getClass() != o.getClass()) return false;
57
58     Box box = (Box) o;
59
60     if (x != box.x) return false;
61     if (y != box.y) return false;
62
63     return true;
64 }
65 }
```

Since we always maintain the closest pair of points and their minimum distance in the current pool of points, we have created a class called “CP” to perform this function. Its members include the two closest points and the minimum distance between them. It includes getters and setters along with a “toString” function for outputting the closest pair of points.



```
1 public class CP {
2
3     private Point a;
4     private Point b;
5     private Double distance;
6
7     public CP(Point a, Point b, Double distance) {
8         this.a = a;
9         this.b = b;
10        this.distance = distance;
11    }
12
13    public Point getA() {
14        return a;
15    }
16
17    public void setA(Point a) {
18        this.a = a;
19    }
20
21    public Point getB() {
22        return b;
23    }
24
25    public void setB(Point b) {
26        this.b = b;
27    }
28
29    public Double getDistance() {
30        return distance;
31    }
32
33    public void setDistance(Double distance) {
34        this.distance = distance;
35    }
36
37    @Override
38    public String toString() {
```

We make use of the class “InputPoints” for randomly generating a list of points which is used as an input for computing the closest pair. The “FindClosestPair” class includes a main() function and other variables for calculating the time cost of running the three algorithms. It also handles the final output of the entire program to the console.

### 3.1.2.Brute Force and Simple Randomized Algorithms

This section explains the entire implementation of the simple randomized algorithm using the two variants of Balanced Search Tree (BST) and Hashing as the underlying data structures for maintaining the dictionary of boxes and the points included in them.

Brute Force Algorithm:

The brute force algorithm simply computes and then compares the distance between a point and all the other points in the input to arrive at the final minimum distance and the closest pair. It involves two nested for loops which compute the Euclidean distance between each point and all

the other points in the input and set/change the minimum distance accordingly. This algorithm runs in  $O(n^2)$  time.

```

1 import java.util.List;
2
3 public class BruteForce {
4     /*
5      * Brute Force methodology for Closest Pair problem. Time complexity is  $O(n^2)$ .
6      */
7     public static CP findCP(List<Point> points) {
8         CP cP = new CP(points.get(0), points.get(1), Double.POSITIVE_INFINITY);
9         int counter = 0;
10        for(int i=0; i<points.size(); i++) {
11            for(int j=0; j<points.size(); j++) {
12                Point a = points.get(i);
13                Point b = points.get(j);
14                counter ++;
15                if(!a.equals(b)) {
16                    Double dist = FindClosestPair.EuclideanDistance(a, b);
17
18                    if(dist < cP.getDistance()) {
19                        cP.setDistance(dist);
20                        cP.setA(a);
21                        cP.setB(b);
22                    }
23                }
24            }
25        }
26        System.out.println("Iteration counter in Brute Force is : " + counter);
27        return cP;
28    }
29 }
30 }

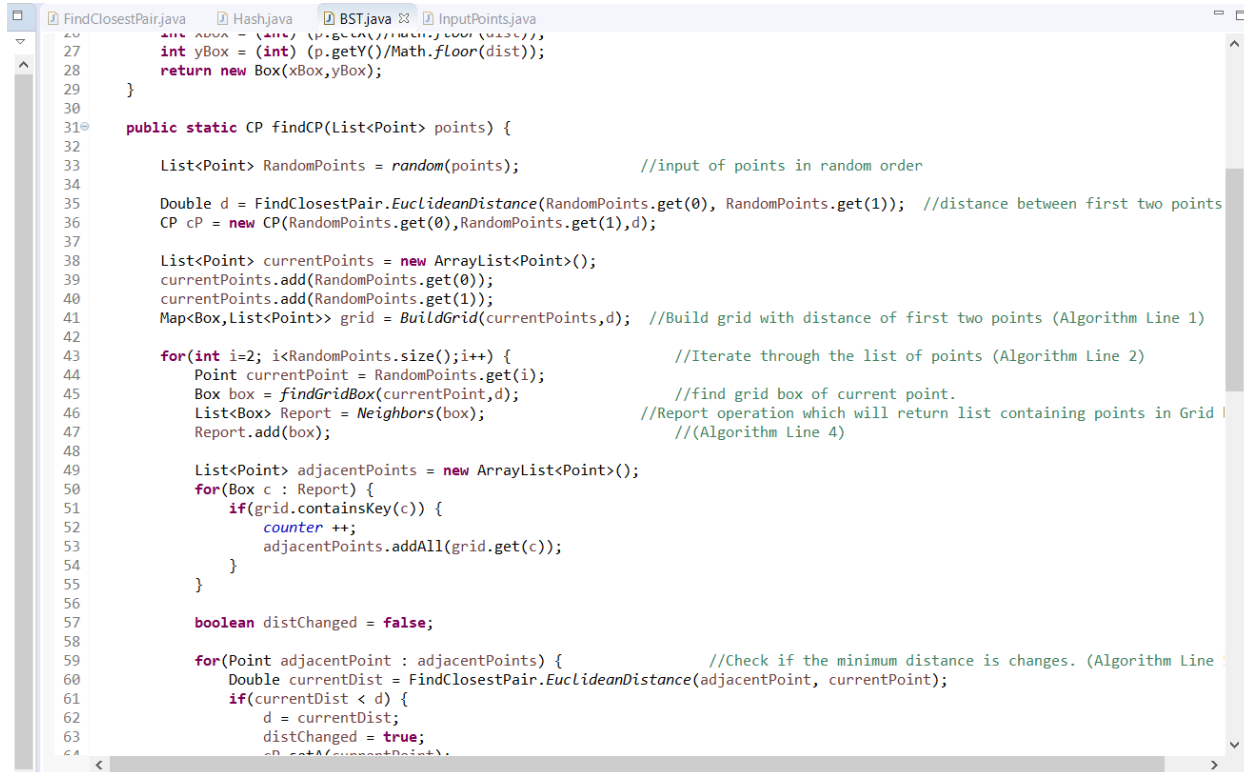
```

### Simple Randomized Algorithm using BST:

We have used the class “BST” to implement the simple randomized algorithm using the Balanced Binary Search Tree as the dictionary data structure.

Let us go through the supporting methods before diving into the actual algorithm. The random() method is used to randomize the input set of points before computing the closest pair of points. The findGridBox() method is used to compute the box indices for a given point by using the floor function. The BuildGrid() method takes the current list of points in the system and the minimum distance “d” to build out the entire grid of boxes based on the current minimum distance. We have made use of the Java collection “TreeMap” to simulate the functionality of a Balanced

Binary Search Tree, which is the dictionary data structure for this algorithm. The Neighbors() method simply returns the box indices of the 9 adjacent boxes of the given Box.



```

27     int yBox = (int) (p.getY()/Math.floor(dist));
28     return new Box(xBox,yBox);
29 }
30
31 public static CP findCP(List<Point> points) {
32     List<Point> RandomPoints = random(points);           //input of points in random order
33
34     Double d = FindClosestPair.EuclideanDistance(RandomPoints.get(0), RandomPoints.get(1)); //distance between first two points
35     CP cP = new CP(RandomPoints.get(0),RandomPoints.get(1),d);
36
37     List<Point> currentPoints = new ArrayList<Point>();
38     currentPoints.add(RandomPoints.get(0));
39     currentPoints.add(RandomPoints.get(1));
40     Map<Box,List<Point>> grid = BuildGrid(currentPoints,d); //Build grid with distance of first two points (Algorithm Line 1)
41
42     for(int i=2; i<RandomPoints.size();i++) {              //Iterate through the list of points (Algorithm Line 2)
43         Point currentPoint = RandomPoints.get(i);
44         Box box = findGridBox(currentPoint,d);             //find grid box of current point.
45         List<Box> Report = Neighbors(box);                 //Report operation which will return list containing points in Grid
46         Report.add(box);                                   //(Algorithm Line 4)
47
48         List<Point> adjacentPoints = new ArrayList<Point>();
49         for(Box c : Report) {
50             if(grid.containsKey(c)) {
51                 counter++;
52                 adjacentPoints.addAll(grid.get(c));
53             }
54         }
55
56         boolean distChanged = false;
57
58         for(Point adjacentPoint : adjacentPoints) {         //Check if the minimum distance is changes. (Algorithm Line
59             Double currentDist = FindClosestPair.EuclideanDistance(adjacentPoint, currentPoint);
60             if(currentDist < d) {
61                 d = currentDist;
62                 distChanged = true;
63                 cP.setA(currentPoint);

```

The method “findCP” takes the list of points as the argument and returns the closest pair object. This method implements the simple randomized algorithm for computing closest pair. The following steps explain the flow of the algorithm and how our implementation runs the algorithm.

1. The input list of points is first randomized using the random() method explained earlier.
2. It takes the first two points from the randomized list to compute the current minimum distance in the system. It also adds these two points in a separate list named “currentPoints”, which keeps track of the current set of points in the system.
3. The BuildGrid() method is then invoked to build out the entire grid structure based on the minimum distance and the current set of points, which in this case is just the first 2 points.

4. We then initiate a for loop running through 2 to  $n-1$ , where  $n$  is the size of the list of input points. For each point, we compute the box indices based on its  $x$ ,  $y$  co-ordinates using the `findGridBox()` method. We then report its 9 nearest neighboring boxes including that box.
5. We run another for loop to retrieve all the points stored in these boxes from the BST dictionary data structure. We store all these points in a list of points called “adjacentPoints”.
6. We now run another for loop to iterate over all the points in the “adjacentPoints” list and compute the Euclidean distances between the current point and all these points. We simultaneously check if we find a new minimum distance lower than the current minimum distance of the system.
7. We then add this point to the list of “currentPoints” and check if the minimum distance has changed. If the minimum distance has changed, we call “BuildGrid()” method, passing it the list of currentPoints and the new minimum distance, to build a new grid. If the minimum distance is unchanged, we simply add the current point to the list of points of the box to which it belongs in the BST dictionary. If the box is not already present in the BST, we add it by creating a new key value and add the current point to the list of points stored for that box.
8. Finally, after all the input points are added to the system, we return the closest pair, as it contains the minimum distance as well as the set of closest points.

This algorithm runs in  $O(n \log n)$  time as it makes use of the BST as the dictionary data structure.

#### Simple Randomized Algorithm using Hashing:

We have used the class “Hash” to implement the simple randomized algorithm using the Hashing technique to maintain the dictionary data structure.

```

83     }
84 }
85 }
86 System.out.println("Iteration counter in BST is : " + counter );
87 return cP;
88 }
89
90 private static Map<Box,List<Point>> BuildGrid(List<Point> points, Double dist) {           //function for Building new grid.
91     Map<Box,List<Point>> grid = new HashMap<Box, List<Point>>();
92
93     for(Point p : points) {
94         Box box = findGridBox(p,dist);
95
96         if(grid.containsKey(box)) {
97             List<Point> list = grid.get(box);
98             list.add(p);
99         }
100         else {
101             List<Point> list = new ArrayList<Point>();
102             list.add(p);
103             grid.put(box, list);
104         }
105     }
106     return grid;
107 }
108
109 private static List<Box> Neighbors(Box box) {                                           //function for finding 9 neighbors.
110     Box box2 = new Box(box.getX()+1,box.getY());
111     Box box3 = new Box(box.getX()+1,box.getY()-1);
112     Box box4 = new Box(box.getX(),box.getY()-1);
113     Box box5 = new Box(box.getX()-1,box.getY()-1);
114     Box box6 = new Box(box.getX()-1,box.getY());
115     Box box7 = new Box(box.getX()-1,box.getY()+1);
116     Box box8 = new Box(box.getX(),box.getY()+1);
117     Box box9 = new Box(box.getX()+1,box.getY()+1);
118
119     return new ArrayList<Box>(Arrays.asList(box2,box3,box4,box5,box6,box7,box8,box9));
120 }

```

The entire algorithm runs exactly as the steps outlined in the previous section. The only difference is in the underlying dictionary data structure which is now implemented using dynamic perfect hashing instead of balanced trees to organize the indices of the currently non-empty grid boxes. After going through the paper[7] which entails the intricacies of implementing dynamic perfect hashing, we started working on incorporating hashing as per that technique in our implementation. Unfortunately, after several attempts, we could not get the dynamic perfect hashing to function as per the technique given in paper[7]. This prompted us to implement hashing using the Java collection of HashMap to maintain the underlying dictionary data structure instead of BST. The Java collection HashMap provides a hashing function internally for storing unique keys for the box indices and we make use of list of points to store as values for those box indices.

Since we are making use of hashing technique to maintain the dictionary data structure, the run time of this algorithm improves to  $O(n)$ .



The next section explains the run time analysis of the simple randomized algorithm in detail along with observations and graphs from our code runs.

## 4. Run time Analysis

Now let's analyze the cost of this algorithm. Let  $Q(n)$  be time for Report,  $U(n)$  be time for Insert and  $P(n)$  be time for Build of the data structure of size  $n$ . Here lines 4-5 from Figure 1 use  $O(1 + Q(i)) = O(Q(i))$  time. Line 6 will take  $U(i)$  time and line 7 will take  $P(i+1)$  time. Note that line 7 is called only if we find a new minimum distance lower than the current minimum distance of the system. We capture this using the following definition:

DEFINITION 1. For any set  $V$  of points and any point  $p \in V$ , define

$$X(p, V) = \begin{cases} 1 & \text{if } \delta(V) < \delta(V \setminus \{p\}) \\ 0 & \text{otherwise.} \end{cases}$$

Denoting the running time for the  $i$ -th stage of the algorithm by  $T_i$ , we have

$$T_i = O(Q(i) + U(i) + X(p_{i+1}, S_{i+1}) \cdot P(i+1)). \quad (2.1)$$

An input sequence that causes the closest pair to change at each iteration will lead to quadratic running time as  $P(i+1) = \Omega(i)$  for both the dictionary implementations.

We now analyze the case that the points are fed to the algorithm in random order  $p_1, p_2, \dots, p_n$  i.e. each of the  $n!$  possible orders is equally likely. Then, for each stage  $i$ ,  $X(p_{i+1}, S_{i+1})$  is a random variable and by Definition 1,

$$E[X(p_{i+1}, S_{i+1})] = \Pr[\delta(S_{i+1}) < \delta(S_i)].$$

We will prove further ahead that this probability is at most  $2/(i+1)$ , and thus we get the following for the expected running time of the  $i$ -th stage:

$$E[T_i] = O(E[Q(i)] + E[U(i)] + 2/(i+1) \cdot E[P(i+1)]).$$

The random choices made by the hashing implementation have nothing to do with the variable  $X(p_{i+1}, S_{i+1})$ . Therefore, the two random variables are independent and we can multiply their expectations and get:

$$E[X(p_{i+1}, S_{i+1}).P(i + 1)] = E[X(p_{i+1}, S_{i+1})].E[P(i + 1)] = 2/(i + 1).( E[P(i + 1)] )$$

For the balanced trees, Insert and Report take logarithmic time, and Build runs in linear time, i.e.

$Q(n) = U(n) = O(\log n)$  and  $P(n) = O(n)$  for a set of  $n$  points

All running times are deterministic. Thus, the  $i$ -th stage takes  $O(\log i)$  deterministic time plus  $O(1)$  expected time in the tree implementation.

Using hashing, Report takes  $O(1)$  deterministic time, Insert takes  $O(1)$  expected time, and Build takes  $O(n)$  expected time. Therefore, we have  $E[T_i] = O(1)$

The table below represents the running times of the various stages for both the implementations.

	Balanced Binary Search Tree	Dynamic Perfect Hashing
Build( $P, \partial$ )	$O(n \cdot \log n)$ deterministic time	$O(n)$ expected time
Insert( $p, G$ )	$O(\log n)$ deterministic time	$O(1)$ expected time
Report( $G, b$ )	$O(\log n)$ deterministic time	$O(1)$ deterministic time

To summarize, the algorithm uses  $O(n \log n)$  expected time for the tree-based implementation, and  $O(n)$  expected time if we use dynamic perfect hashing.

We now prove the result which we used in the analysis of the run time.

Lemma 1: Let  $p_1, p_2, \dots, p_n$  be a random permutation of the points of  $S$ .

Let  $S_i = \{p_1, p_2, \dots, p_n\}$ , then  $\Pr[\partial(S_{i+1}) < \partial(S_i)] \leq 2/(i + 1)$

Proof:

We make use of Seidel's backwards analysis technique from papers [21] [22]. Consider  $S$  as a set of points and  $p_{i+1}$  where  $S_{i+1} = S \cup p_{i+1}$

$A$  is the set of points that are part of some closest pair in  $S_{i+1}$ . If  $|A| = 2$  then there is exactly one closest pair in  $S_{i+1}$  and we find a new minimum distance lower than the previous minimum distance if and only if  $p_{i+1} \in A$

If  $|A| > 2$  there are two possibilities. The first is that there is a unique  $p \in A$  that is a member of every closest pair in  $S_{i+1}$ .

The other possibility is that there is no such unique  $p$ . In that case,  $S_i$  must contain some pair of points from  $A$  and, therefore,  $\partial(S_{i+1}) = \partial(S_i)$ . We have just shown that regardless of the composition of  $S_{i+1}$ , there are at most 2 possible choices of  $p_{i+1}$  which will permit  $\partial(S_{i+1}) < \partial(S_i)$ .

Since  $p_1, p_2, \dots, p_n$  is a random permutation, the point  $p_{i+1}$  is a random point from  $S_{i+1}$ . Therefore, the probability that  $\partial(S_{i+1})$  is smaller than  $\partial(S_i)$  is at most  $2/(i+1)$ .

Observations:

Based on the many runs of our code implementation of closest pair problem using brute force, Simple randomized using BST and then Hashing, we observed that the computation times confirmed the theoretical running time bounds of all the algorithms. Below is a graph reflecting the running times for all the 3 algorithms.



## 5. High Probability Bounds

The closest pair algorithm given previously runs quickly with high probability bounds. These high probability bounds are proved in paper[3] by obtaining tail estimates on the space complexity of some randomized incremental constructions.

The most expensive operation in presented algorithm is the rebuilding of the grid, as the underlying dictionary data structure needs to be recomputed. The only time a rebuilding operation needs to be performed is the case where a new point added to the system leads to a new minimum closest pair distance due to which the entire grid is rebuilt using the new minimum distance. The underlying idea to improve the expected running time of the algorithm is to perform the expensive grid rebuilding operation with a lower probability, such that we achieve a tail estimate for the total cost of rebuilding operation.

From the running time analysis covered in the previous section, the rebuilding cost of the  $i$ -th stage of the algorithm is given by:

$$X(p_{i+1}, S_{i+1}) \cdot P(i+1), \text{ for } 2 \leq i \leq n-1.$$

We will now prove the tail estimate for the above rebuilding operation.

We take into consideration the following Lemma for proving the result.

*For all  $c \geq 1$ ,*

$$\Pr[Y_S \geq cn] \leq \frac{1}{e^2} \left( \frac{e^2}{c} \right)^c$$

where  $Y_s$  is a random variable for set  $S$  of  $n$  points.

The improved running time analysis based on the above result follows.

a. Tree-based implementation

Based on the running time analysis, the total work performed by the algorithm is  $O(n \log n + Y_s)$ . Let  $s$  be a positive integer. Using the Lemma with  $c = 2s \ln n / \ln \ln n$ . Then for sufficiently large  $n$ , we have:

$2c - c \ln(c) \leq -s \ln(n)$  and hence using the result of the Lemma,

$$\Pr[Y_s \geq 2sn \ln(n) / \ln \ln(n)] \leq (1/e^2)e^{-s \ln(n)} = O(n^{-s})$$

Hence, the tree-based implementation runs in  $O(n \log n)$  time with probability of  $1 - O(n^{-s})$  for any positive integer  $s$ .

b. Hashing-based implementation

Based on the results of Dietzfelbinger et. al, we obtain the following run times for our basic grid operations. Let  $1 \leq i \leq n$  be fixed. The time for Report is  $O(1)$  in the worst case, and the times for Build and Insert are  $O(i + \log^4 n)$  and  $O(1)$  respectively, each with a probability of  $1 - O(n^{-t})$  for any fixed integer  $t$ . To determine the running time for the whole closest pair algorithm, we simply add the failure probabilities occurring in each single stage. Therefore, the algorithm runs in  $O(n \log n / \log \log n)$  time with probability  $1 - O(n^{-t} + n^{-r})$  for any  $t, r \geq 1$ . So for any  $s \geq 1$ , we can choose  $t = s+1$  and  $r = s$  to obtain the running time with probability  $1 - O(n^{-s})$ .

To summarize,

		Run time	Probability
Balanced Search Tree	Binary	$O(n * \log n)$	$1 - O(n^{-s})$
Dynamic hashing	perfect	$O(n * \log n / \log \log n)$	$1 - O(n^{-s})$

## 6. The K-closest Pairs Problem

This algorithm has a simple extension to solve the k-closest pairs problem. Let  $S = \{p_1, p_2, \dots, p_n\}$  be a set of  $n$  points in the plane. We enumerate all the permutations and compute distances between pairs of points and sort them as  $e_1 \leq e_2 \leq \dots \leq e_{n^2}$ . Set  $\partial^k(S) = e_k$  to be the k-th closest pair distance in the set  $S$ . The k closest pairs problem is to find  $k$  pairs of points that are at most  $\partial^k(S)$  apart.

Let  $p_1, p_2, \dots, p_n$  be given to us in a random order. Our algorithm incrementally calculates  $\partial^k(S_i)$ . It differs from the closest pair algorithm only in that it uses a mesh size of  $\partial^k(S_i)$  instead of  $\partial(S_i)$  for the grid that stores  $S_i$ .

The tree based implementation of the k closest pairs algorithm runs in  $O(n(\sqrt{k} \log k + \log n))$  expected time, whereas the hashing based implementation runs in  $O(n\sqrt{k} \log k)$  expected time.



## 7. Summary

Closest pair of problem computes closest pair of points with minimum Euclidian distance in given set of points. There are many approaches to solve this problem. We have used Simple randomized algorithm for finding closest pair problem and implemented it using two approaches which are using Balanced binary search tree and using Dynamic perfect hashing. This algorithm is randomized, incremental and uses floor function for creating grid hence this algorithm is simpler than others. We have implemented Brute force approach as well to compare its run times with the algorithm mentioned in the paper and results are shown in earlier sections. Run time analysis says that Brute force takes  $O(n^2)$  time, binary search tree takes  $O(n \cdot \log n)$  time and dynamic hashing takes  $O(n)$  time.

## 8. Conclusions

Simple randomized algorithm for closest pair problem implementation using binary search tree takes  $O(n \cdot \log n)$  time and dynamic perfect hashing takes  $O(n)$  time. We have succeeded in matching those runtimes after implementing actual algorithm using Java. We used HashMap data structure for implementing this algorithm using Dynamic perfect hashing which is a different approach than paper. In future, we will try to implement this algorithm using the actual hashing function mentioned in paper.

## References

- [1] M. Golin, R. raman, C. Schwarz, M. smid. Simple randomized algorithm for closest pair problems, Nordic journal of Computing , 2:3-27, 1995.
- [2] M. Dietzfelbinger and F. Meyer. A new universal class of hash functions and dynamic hashing in real time. In proc. 17 internat volume 443. Of lecture notes in Computer Science pages 6-19, 1990.
- [3] R. Seidel, small dimensional linear programming and convex hull makes easy. Discrete Comput. Geom 6:423-434, 1991
- [4] R. Seidel, Backwards analysis of randomized geometric algorithms. TR-92-014, Dept. CS, UCB, Berkeley, CA, 1992