

Closest Pair Problem

This project is based on the algorithm proposed by M. Golin, R. Raman, C. Schwarz, M. Smid [4]. Authors presented a simple randomized algorithm for finding closest pair in dimensional space D greater than and equal to 2, for a given set of n points. Two approaches of implementing this algorithm are Binary search tree and Dynamic perfect hashing which run in $O(n \log n)$ and $O(n)$ expected time respectively. We present the experimental results of these approaches comparing it with naïve implementation of Brute-force technique. Author also extends the algorithm to fully dynamic closest pair problem and to k -closest pair problem.

Table of Contents

1. Motivation.....	4
2. Problem Description... ..	4
3. Related Work	5
4. Algorithm.....	6
4.1 Algorithm Idea	6
4.2 Algorithm Details.....	9
5. Data Structure	10
6. Cost and Runtime analysis	11
7. Fully Dynamic Algorithm	
8. K-Closest Pair Algorithm	
9. Implementation	
Details	14
9.1 Brute-Force	
9.2 Binary Search Tree	
9.3 Dynamic Perfect Hashing	
10. Experiment Execution.....	18
11. Results	20
12. Conclusion	21
13. References.....	22

1. Motivation

Closest pair problem is one of the basic problems in computation geometry. It is building block for various geometry algorithms and data structures. Solution to this problem is a main step in problem solving procedures like Nearest Neighbor, Minimal Spanning Trees etc. Commonly, it is well known that a practical way of solving closest pair problem for two dimension is to apply $O(n \log n)$ implementation of Divide-and-Conquer technique by Bentley and Shamos [1].

Real Life Applications:

This problem has many applications in many different domains. For example, a sea or air traffic control system may have to identify two closest vehicles for detecting possibility of potential collisions. It can also be extended to usage in self driving vehicles and project like “Waymo” by Google are evident of that. In the geographical field this problem has implementation in Geographical information systems which even extends to missile detection system. All these examples show that it is very interesting area of research as the solution to this problem directly affects the efficiency of real time systems. In next sections, we will see the work done in this area. Other applications from various domains are listed below:

- Molecular modeling
- Graphics and Computer vision
- Imaging technologies
- Pattern recognition etc.

2. Problem Description:

In a layman terminology, closest pair problem is to find two points closest to each other when given a set S of n points on a multidimensional plane. For understanding the problem, we will continue with the problem in 2 dimensions. Let the Euclidean Distance between two points is given by $d(p,q)$ then,

$$d(p,q) = \sqrt{((p_x - q_x)^2 + (p_y - q_y)^2)}$$

The closest pair distance is given by $\delta(S)$ where set S is the set of points $\{p_1, p_2 \dots, p_n\}$.

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

Mathematically, the problem can be defined as to find two closest points $p,q \in S$ such that distance between two points equals the minimum distance.

$$d(p,q) = \delta(S)$$

Proposed Solution:

Authors propose a solution to solve this problem in logarithmic $O(n \log n)$ and linear $O(n)$ time. They propose a Simple Randomized incremental algorithm and we will see the algorithm details and implementations in next sections. Also, the high probability bounds have been explored. The linear expected time algorithm runs in $O(n \log n / \log \log n)$ time with high probability and the logarithmic expected time algorithm runs in $O(n \log n)$ time with high probability. The solution to this problem also extends to fully dynamic algorithm and k-closest pair algorithm and also to algorithm which does not consider non-algebraic functions.

It is worth mentioning that floor functions used in above algorithm assumes to be calculated at unit cost.

3. Related Work:

A lot of work has been done in past to solve closest pair problem. Solutions have been proposed from quadratic time implementation to logarithmic time implementation over a course of years.

Quadratic time algorithm: Brute Force $O(n^2)$:

Brute force is the most naïve and straightforward approach of implementation for these problems. It is an in-efficient solution to this problem which runs in quadratic time. We will see the basic pseudo code in implementation section.

Logarithmic time algorithm: Divide-and-Conquer $O(n \log n)$:

In 1976 Bentley and Shamos [1] proposed a divide and Conquer approach for multidimensional closest pair problem. They used a recursion method and achieved $O(n \log n)$ the time bounds in multidimensional space.

Linear time algorithm: $O(n)$

- **Rabin** [2] in 1976 proposed a Probabilistic Algorithms to solve this problem in linear time. Rabin achieved this linear running time by combining floor function with randomization technique.
- **Khuller-Matias** [3] in 1991 proposed a simple randomized sieve algorithm for closest pair problem. They introduced a filtering process to the algorithm. In such process points are removed from the set and at the end an approximation is obtained to closest pair distance. Then closest pair is calculated using the approximation.

This Paper: $O(n)$

In 1995 authors of [4] paper proposed a “Simple Randomized Algorithms for Closest Pair Problems”. They used similar floor function and randomization but this algorithm has an edge over other algorithms as it is incremental in nature. Its incremental nature is based on the process by

which new incoming points to set are treated and grid structure is updated accordingly. We will discuss more about grid structure in algorithm section.

4. Algorithm

4.1 Algorithm Idea:

To begin with, Let $S_i = (p_1, p_2, p_3, \dots, p_i)$ be the set containing i points of S . where $\delta(S_i)$ is the closest distance of set S_i . Suppose a square grid with a mesh size $\delta(S_i)$ is laid over the plane and each point is stored in the grid box.

P_1			P_2
	P_3		
P_7			P_4
P_6		P_5	

Now we insert p_{i+1} into the grid and want to compute $\delta(S_{i+1})$. If $\delta(S_{i+1}) < \delta(S_i)$, then the grid updates, holds if and only if there is some point $p \in S_i$ such that $d(p, p_{i+1}) < \delta(S_i)$. When we insert a new point p_{i+1} in a grid box, let it be 'b'. Then every point in S_i that is within distance of p_{i+1} must be located in one of the 9 grid boxes that are adjacent to grid box b. These 9 boxes are called neighbors of b. Whenever we insert a point into the grid box, then it finds at most 36 points in these 9 neighbors in which the point is located and computes the minimum distance between the point and these at most 36 points. This is because if it contains more than four points then some of its pair would be less than $\delta(S_i)$, contradicting the definition of $\delta(S_i)$.

From the discussion above we know that $\delta(S_{i+1}) = \min(d_{i+1}, \delta(S_i))$. If $d_{i+1} \geq \delta(S_i)$ then $\delta(S_{i+1}) = \delta(S_i)$ and the algorithm inserts p_{i+1} into the current grid. Otherwise, $\delta(S_{i+1}) = d_{i+1} < \delta(S_i)$ and the algorithm discards the old grid, creates a new one with mesh size $\delta(S_{i+1})$, and inserts the new points of S_{i+1} into this grid. We can also incrementally maintain a list of all point pairs in S_i with distance $\delta(S_i)$, for $2 \leq i \leq n$, without additional cost. Let's consider i th stage of the algorithm, when

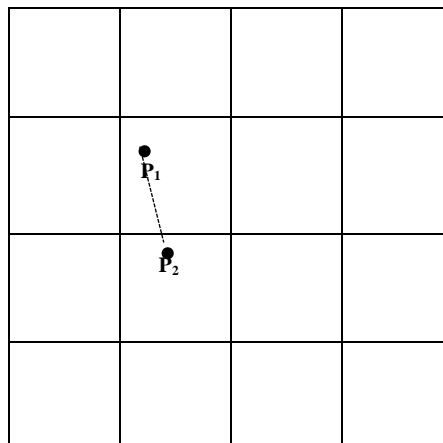
we add p_{i+1} to S_i . the algorithm checks all points within the distance $\delta(S_i)$ of p_{i+1} , we find all points q in S_i such that $d(q, p_{i+1}) = \delta(S_i)$, then the pairs (q, p_{i+1}) can be only $O(1)$ of them are exactly new closest pair after i th stage of algorithm. If $\delta(S_{i+1}) = \delta(S_i)$, then we add these new pairs to the closest pair list.

If $\delta(S_{i+1}) < \delta(S_i)$, then we discard the old list and create a new list containing only these pairs.

An example of our algorithm working is shown below:

Let S be a set of points $S = (p_1, p_2, p_3, \dots, p_n)$, where S_2 be a subset of S and $S_2 = (p_1, p_2)$. The minimum distance between two points $\delta(S_2)$ is calculated. Where $\delta(S) = \min (d(p, q) : p, q \in S, p \neq q)$, $d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$. The closest pair problem is to find a pair of points $p, q \in S$ such that $d(p, q) = \delta(S)$.

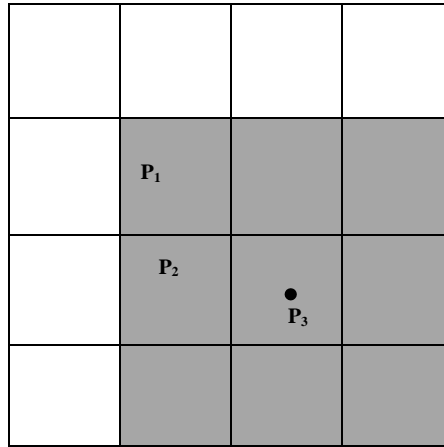
$$\delta(S_2) = d(p_1, p_2)$$



$$\delta(S_2)$$

Now add a point p_3 to the plane, compute $\delta(S_3) = d(p_3, p_?)$, the new point p_3 checks the 9 grid boxes adjacent to p_3 , which are called neighbors of p_3 . So, it finds the points in these 9 neighbors and compute the minimum distance between the point and these 9 neighbors. If calculated $d(p_3, p_?) < \delta(S_2)$,

Then $\delta(S_3) = d(p_3, p_?)$ else $\delta(S_3) = \delta(S_2)$.



Since $d(p_3, p_?) > \delta(S_2)$, $\delta(S_3) = \delta(S_2)$. There is no change in the grid.

Now continue to add p_4 to the grid and compute $\delta(S_4)$, Similarly it checks the 9 neighbors of point p_4 and compute the minimum between the points in these 9 neighbors, this is shown in *Figure A* below. Here $d(p_4, p_1) < \delta(S_3)$, then $\delta(S_4)$ gets updated to $d(p_4, p_1)$. This results in rebuild of the grid with new mesh size $\delta(S_4)$, this is shown in *Figure B* below.

$$\delta(S_3) = d(p_1, p_2)$$

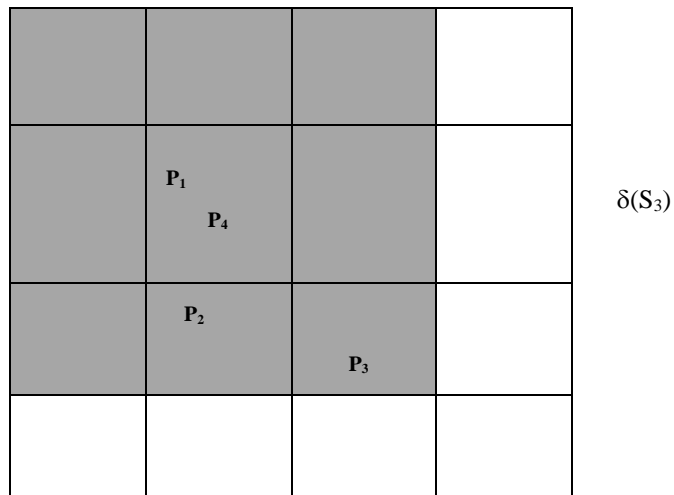


Figure A

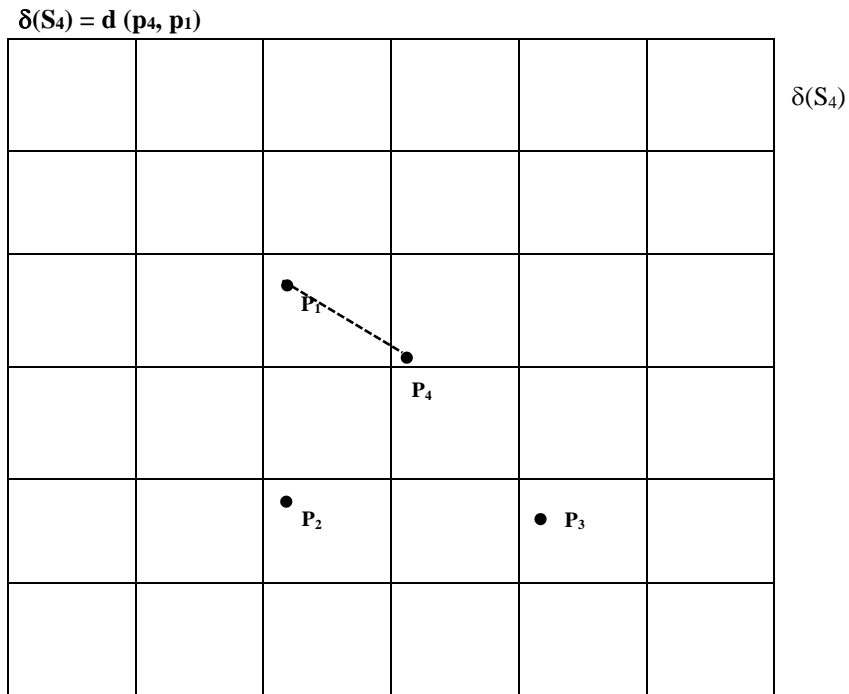


Figure B

4.2 Algorithm Detail:

Pseudocode

Algorithm: S: Set of Points ($p_1, p_2, p_3, \dots, p_n$)

- (1) $\delta = d(p_1, p_2);$
- (2) $Grid = Build(S_2, \delta);$
- (3) for $i = 2$ to $n-1$ do
- (4) begin
- (5) $pointsList = Report(Grid, b);$
- (6) $d = \min d(p_{i+1}, pointsList);$
- (7) if $d \geq \delta$ then
- (8) $Grid = Insert (Grid, p_{i+1});$
- (9) else
- (10) $\delta = d;$
- (11) $Grid = Build (S_{i+1}, \delta);$
- (12) end;
- (13) return (δ) .

To Implement the above algorithm, we need following: Let S be a set of points, d is a positive real number i.e. it denotes the Euclidean distance between the points. p is any point in a set S , Grid is a grid dictionary with mesh size d , pointList is List of points in the grid box and let b denote a grid box containing point p . The main three operations in the algorithm are

- Build (S, d): Returns a grid 'Grid' that contains the points S with a mesh size d .
- Insert (Grid, p): Insert a point p into grid 'Grid'.
- Report (Grid, b): Returns the list of points in the grid box b .

5. Data Structures:

In order to implement the grid operations, we need some data structures to store point sets, consider a grid G with a mesh size d , let the point $p = (p^x, p^y)$ in the plane and b denote a box containing point p in the grid G . The index of b is the integer pair divided by the mesh size i.e. $(\lfloor p^x/d \rfloor, \lfloor p^y/d \rfloor)$. To store these point sets we create a dictionary called box dictionary, using their indices as search key. We can implement this box dictionary using two ways, one standard way is to use balanced binary search trees, which takes $O(n \log n)$ to run and another way to implement is by using dynamic perfect hashing to store the indices of current grid boxes, this is expected to run in $O(n)$ time.

Next Important operation is to insert a point into grid, to insert a point in the grid we perform floor function to compute the index of the grid box that contains this point. Then by Lookup operation in box dictionary using this index. If this box is already present then we insert a new point into the list that is already store, else we insert a new grid box with a list containing the new point into the box dictionary.

5.1 Balanced Search tree:

To implement Box Dictionary using balanced binary search tree, let the points in two lists X and Y are sorted by their x and y coordinates respectively where each point in Y contains a pointer to its X pair. For each point p , compute the value $\lfloor p^x/d \rfloor$. Initialize an empty bucket $B(\lfloor p^x/d \rfloor)$ for all distinct values, now give each point in X a pointer to its bucket. A bucket contains at least one point in P . Go through the list Y to its occurrence in X by its pointer and follow the point to its bucket, then add $(\lfloor p^x/d \rfloor, \lfloor p^y/d \rfloor)$ at the end of bucket, if it's not stored. This list contains the indices of grid boxes sorted in lexicographical order. This implementation takes logarithmic time to run i.e. $O(n \log n)$.

5.2 Dynamic Perfect Hashing:

Another way to implement Box Dictionary is by using dynamic perfect hashing, which runs in $O(n)$ expected time. We know that insertion of totally arbitrary points in lookup table is not possible by dynamic perfect hashing. It requires to know the universe containing the elements in advance and also a prime exceeding the size of the universe. In terms of grid, the first requirement is to know about the bound on indices of possible grid boxes. We need to know the range of all positive real numbers that contains all input points of our algorithm. This range is referred as Frame containing the values. Then for a grid of mesh size d , the box indices will be in range $[\lfloor x^-/d \rfloor \dots \lfloor x^+/d \rfloor] \times [\lfloor y^-/d \rfloor \dots \lfloor y^+/d \rfloor]$. so, before we start preparing grid we have to know the possible bound of indices that will occur in box dictionary. The second requirement is knowing the prime which has to exceed the universe. In moderate universe size, we can use a precomputed lookup table to find the desired prime due to limitations of d . If such table is not available, then we can build a box dictionary containing n elements in two steps. First apply universe reduction to our input keys, by which it reduces collision. And second use the reduced key in hashing, by this there is a balance in degree of compression and reliability in reduction. The only precondition that we need for this technique is to know a power of two exceeding all input keys in advance. Let us know that the additional effort made in each dictionary operation to fulfill all the requirements of dynamic perfect hashing does not affect the complexities of our grid operation.

6. Cost and Runtime Analysis:

As we have seen the algorithm in function in previous sections, in this section we will analyze cost for this algorithm.

Let's denote the important steps algorithm as:

- $Q(n)$ – Time for operation *Report* on grid.
- $U(n)$ – Time for operation *Insert* on grid.
- $P(n)$ – Time for operation *Build* on grid.

Run time at i th iteration of algorithm is denoted by T_i ,

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

Where, X is a random variable which is defined as follows.

For any set V and $p \in V$,

$$\text{if } (\delta(V) < \delta(V \setminus \{p_{i+1}\})$$

$$X(p_{i+1}, V) = 1$$

$$\text{else } X(p_{i+1}, V) = 0$$

We can also infer a drawback for algorithm from this result. In the worst this algorithm may run in quadratic time as at each stage the grid structure is updated when each input is in order of updating the grid.

Let us analyze the case when input is available in random order. In such case, we will calculate the expected running time as:

$$E[X(p_{i+1}, S_{i+1})] = Pr[\delta(S_{i+1}) < \delta(S_i)] , s.t Pr \text{ is at most } 2/(i+1))$$

Proof:(Lemma 1 [4]): We shall prove the probability that new point p_{i+1} will make the closest pair and update the grid is less then equal to $2/(i+1)$.

$$Pr[\delta(S_{i+1}) < \delta(S_i)] \leq 2/(i+1))$$

We consider that $S_{i+1} = S_i \cup \{p_{i+1}\}$. Also, let A is set of closest pair in S_{i+1} then to permit $\delta(S_{i+1}) < \delta(S_i)$, at most 2 possible choices of p_{i+1} are available.

If $|A| = 2$ then,

$$p_{i+1} \in A$$

If $|A| > 2$ then,

$$p_{i+1} = p$$

In this case we have assumed that points are in random order so point p_{i+1} is a random point from S_{i+1} , therefore the probability that $\delta(S_{i+1})$ is less than $\delta(S_i)$ is at most $2/(i+1)$.

Based on above results the expected running time for algorithm in i^{th} iteration is:

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

Run time analysis for both implementations discussed in this paper [4].

Binary Search Tree analysis:

At i^{th} iteration this algorithm takes $O(\log i)$ deterministic time plus $O(1)$ expected time. Running time for important operations of this algorithm will be:

- Insert – $O(\log n)$
- Report – $O(\log n)$
- Build - $O(n)$

Therefore, using BST for box dictionary implementation, the algorithm will run in $O(n \log n)$ expected time.

Dynamic Perfect Hashing analysis:

If use this data structure than our operations will run in:

- Insert – $O(1)$
- Report – $O(1)$
- Build - $O(n)$

Therefore, using Dynamic Perfect Hashing analysis for dictionary implementation, the algorithm will find closest pair in $O(n)$ expected time.

7. Fully Dynamic Algorithm:

The algorithm which we discussed in previous sections is not truly dynamic in design. I would become truly dynamic in nature if we don't care about creating random permutation of inputs when the algorithm starts. Also, if deletion of points is also available similar to insertions of points. This dynamic algorithm has two significant parts as:

Deletion:

The deletion algorithm works as follows:

If closest point deleted, then

If no other pair can take minimal distance, then

Recompute the closest pair from scratch

Minimal distance is increased

New grid is build

Else

Point is deleted from grid and closest pair updated.

Running time for this obtained as:

$$O(Q(n)) + U(n) + X(p,s) \cdot P(n)$$

Random Update:

In fully dynamic algorithm a process of update sequence is introduced which takes care of the input order. Such a change was required in algorithm because the run time of randomized algorithm is directly proportional to closest pair change. This update sequence is called random if First, each points p is equally likely to be incoming point p_i . Second, the updated set S_i is random subset of set S of size m_i . Using this technique, the probability of closest pair change will be small and ultimately gives better results.

This fully dynamic algorithm randomization is not based on location of points in dimensional space rather it is in respect to the order of random updates.

8. The k closest pair problem:

This problem is an extension to our previously discussed randomized incremental algorithm. This problem is to find k number of pair of points which are at minimum distance $\delta^k(S)$. We are assuming the set S as previously described. Let us consider that at i^{th} iteration the mesh size in this algorithm is $\delta^k(S)$ and has set S_i saved in grid. Also, the current closest pairs available in grid are saved in binary search tree (D-tree). The algorithm works as follows:

At i^{th} iteration point p_{i+1} in incoming point in grid.

Distance between $\delta^k(S_i)$ and p_{i+1} is calculated.

If $(\delta^k(S_{i+1}) = \delta^k(S_i))$ the distance is equal to the mesh size then,

the D-tree is updated with new point and at same time removing an old point to maintain k .

If $(\delta^k(S_{i+1}) < \delta^k(S_i))$ the new distance is less than current mesh size then,

new grid structure is build.

Cost Analysis:

This algorithm takes following running time based on the data structure used on insertion of $(i+1)^{st}$ point.

- Binary Search Tree: $O(k + \log i)$

After a minor change in algorithm the running time is calculated as:

Amortized: $O(\log i + \sqrt{k} \log k)$

Expected: $O(n(\sqrt{k} \log k + \log n))$

- Dynamic Perfect Hashing: $O(k)$

Similarly running time for this data structure is :

Amortized: $O(\sqrt{k} \log k)$

Expected: $O(n \sqrt{k} \log k)$

9. Implementation Details:

We have implemented the randomized incremental algorithm in 2D with Binary Search Tree data structure and compared the results with the naïve implementation of Brute Force. For implementation, we have used C++ as programming language and Microsoft Visual Studio as development IDE.

9.1 Brute Force:

We implemented the naïve approach for solving this problem to compare the efficiencies with randomized incremental algorithm. Below is the pseudocode for Brute Force:

$d_{min} = \infty$

for $i = 1$ to inputs - 1

for $j = i + 1$ to inputs

let $p = P[i]$, $q = P[j]$

if $dist(p, q) < d_{min}$:

$d_{min} = dist(p, q)$

closestPair = (p, q)

return closestPair

9.2 Randomized Incremental algorithm:

We have implemented both the data structure for this algorithm. The basic algorithm is implemented as below with important operations of Build, Report and Insert.

MAIN PROGRAM

```
{
    [READ ALL POINTS FROM A FILE INTO MEMORY]

    - minDistance = max possible value for double (real) type in C++

    - for every newPoint
    {
        - x grid index = GET_X_INDEX (point x coordinate)
        - y grid index = GET_X_INDEX (point x coordinate)

        - creating collection of 9 neighbour boxes
        - adding [x,y] box of the current grid into the collection
        - adding [x-1,y-1] box of the current grid into the
          collection
        - adding [x-1,y] box of the current grid into the
          collection
        - adding [x-1,y+1] box of the current grid into the
          collection
        - adding [x+1,y-1] box of the current grid into the
          collection
        - adding [x+1,y] box of the current grid into the
          collection
        - adding [x+1,y+1] box of the current grid into the
          collection
        - adding [x,y-1] box of the current grid into the
          collection
        - adding [x,y+1] box of the current grid into the
          collection

        - minDistanceToNeighbour = minDistance

        - for every box in the collection of neighbour box {
            - for every pointInTheBox {
                - distance = distance between newPoint and
                  the pointInTheBox
                - if distance < minDistanceInTheBox {
                    minDistanceToNeighbour = distance
                }
            }
        }

        - if minDistanceToNeighbour < minDistance {
            minDistance = minDistanceToNeighbour
            currentGrid = BUILD(distance, currentGrid)
        }

        INSERT(thw newPoint) in the current grid
    }
}
```

```

    }
}

```

- All points are read out from an input file and stored in memory before processing in order to make time measuring more accurate.
- Grids store pointers (references to the point instances), but not store the values it selves
Only pointers (references) to all objects (collections) are sent between functions/methods due to performance reasons; objects themselves are not copied when working.
- REPORT provides a single box; this is how we can avoid an overhead of implementation in C++

Build Operation:

```

BUILD (meshSize) {
//called once, initially
    - save meshSize value
    - create an empty BST <KEY: composite index of a box, VALUE: list
      of points related to the box>
}

```

```

BUILD (meshSize, pointer (reference) to a previous grid)
//called when a new grid is required
{
    - save meshSize value
    - create an empty BST <KEY: composite index of a box, VALUE: list
      of points related to the box>
    - for every box in the previous grid {
        - for every point(reference) in the box {
            INSERT (the point)
        }
    }
}

```

```

GET_X_INDEX (x coordinate)
{
    return x coordinate divided by mesh size
}

```

```

GET_Y_INDEX (y coordinate)
{
    return y coordinate divided by mesh size
}

```

Insert Operation:

```
INSERT (new point)
{
    - find a box (list of points in the box) in the BST according to a
      composite index (x and y of the new point)
    - add the new point to the box (list)
}
```

Report Operation:

```
REPORT_A_LIST_OF_POINTS_FOR_A_BOX (x box index, y box index)
{
    - find and return a box (list of points in the box) in the BST
      according to a composite index (provided x and y)
}
```

9.2.1 Binary Search tree:

- We have used the Balanced binary search tree implementation of C++. It is called *std::map*. This map is well-debugged and tested container, and it is a good implementation of BST in C++ coding style.
- In the implementation there is a single BST ('std::map<CompositeIndex, Box> boxes;') which “stores whole points”.
(Grid::Grid(double meshSize, const Grid & previousGrid)) takes O(n) time as required.

9.2.2 Dynamic Perfect Hashing:

Our implementation of hashing technique is not exactly same as mentioned in [4]. We started with the implementation as mentioned in [5] by H. Dietzfelbinger but it became too complex for us to implement. We were stuck with below mentioned implementation details.

1. vEB (van Emde Boas) structure.
2. y-fast Tries
3. FKS hashing (static perfect hashing)
4. Combining those into Dynamic perfect hashing structure
5. Adaptor for mapping pairs of double coordinates into known universe size required in Dynamic perfect hashing.

We have used C++ default hash table implementation of `unordered_map`. Below is the sample code we used for hash chaining:

```
// resulting seed is a hash of chain
void hash_combine(std::size_t &seed, T const &key) {
    std::hash<T> hasher;
    seed ^= hasher(key) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
}
```


10. Experiment Execution:

All the implementations were executed in below environment and the results were obtained.

Processor: Intel Core i5 – 6200 U CPU 2.30 GHz

Memory: 12.0 GB

Operating System: 64-bit Windows 10 OS

Input:

Our implementations take input from a plain text file in form of x y points. A sample input file is displayed in *Figure 1*. All the three implementation were provided with same input points.

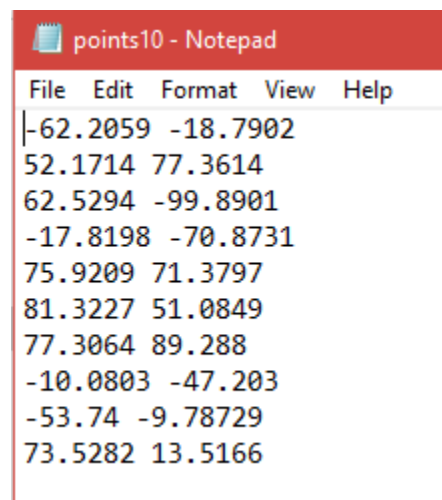


Figure 1. Sample Input

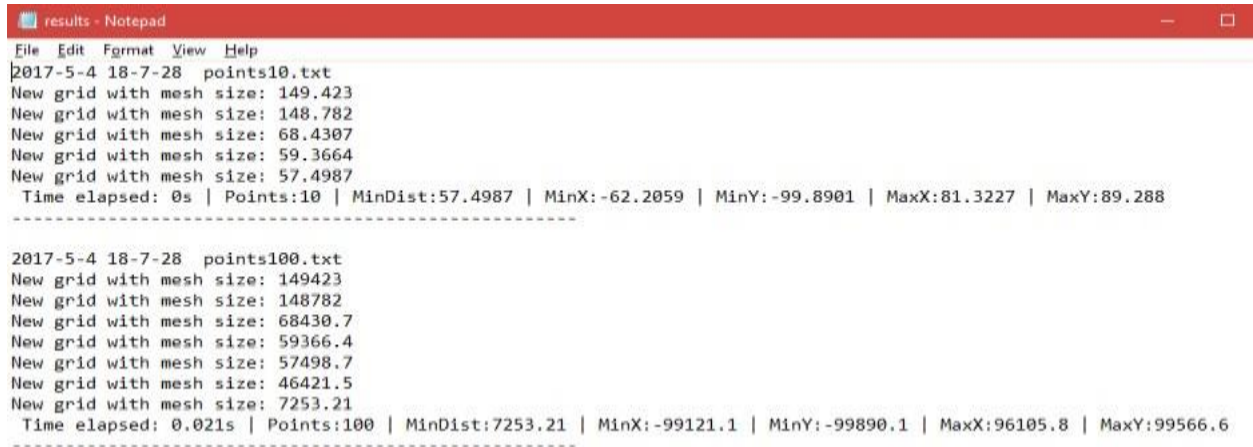
We have developed a Utility program to generated the input file. The points are generated randomly as in *Figure 1*. This utility can generate any number of points based on the passed parameter in generateInputFilesTest1.bat file. A sample input generator is displayed in Figure 2:

```
GenerateInputFile.exe points10.txt 10 -100 -100 +100 +100
GenerateInputFile.exe points100.txt 100 -100000 -100000 +100000 +100000
GenerateInputFile.exe points1000.txt 1000 -100000 -100000 +100000 +100000
```

Figure 2. Input Generator

Output:

When closest pair algorithm program is executed it generates an output result file which contains the description of closest pair distance and the pair points. It also gives the details of whenever the grid update operation takes place or in other words whenever the grid size changes. A sample output is displayed in Figure 3.



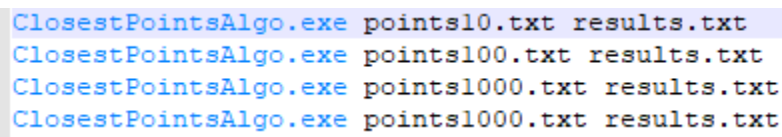
```
File Edit Format View Help
2017-5-4 18-7-28 points10.txt
New grid with mesh size: 149.423
New grid with mesh size: 148.782
New grid with mesh size: 68.4307
New grid with mesh size: 59.3664
New grid with mesh size: 57.4987
Time elapsed: 0s | Points:10 | MinDist:57.4987 | MinX:-62.2059 | MinY:-99.8901 | MaxX:81.3227 | MaxY:89.288
-----
2017-5-4 18-7-28 points100.txt
New grid with mesh size: 149423
New grid with mesh size: 148782
New grid with mesh size: 68430.7
New grid with mesh size: 59366.4
New grid with mesh size: 57498.7
New grid with mesh size: 46421.5
New grid with mesh size: 7253.21
Time elapsed: 0.021s | Points:100 | MinDist:7253.21 | MinX:-99121.1 | MinY:-99890.1 | MaxX:96105.8 | MaxY:99566.6
-----
```

Figure 3. Sample output

Execution Steps:

We have published the code to executable file which can be run on any Windows OS. We have created a bat script file which will execute the executable files and generates output file. Below are the steps to follow:

1. Modify the generateInputFilesTest1.bat to provide the number of points and range. The default file is configured to different files with points such as 10, 100, 1000 etc. in increasing order. You can use the default file also to generate numbers. Figure 2 displays sample input generator.
2. Modify the testPoints.bat file and provide input(points.txt) and output(result.txt) file name. This .bat file refers to the current directory. Figure 4 displays sample .bat file.



```
ClosestPointsAlgo.exe points10.txt results.txt
ClosestPointsAlgo.exe points100.txt results.txt
ClosestPointsAlgo.exe points1000.txt results.txt
ClosestPointsAlgo.exe points1000.txt results.txt
```

Figure 4. Sample testPoints.bat file

3. Execute the testPoints.bat file. It will start reading and processing the inputs.
4. Successful completion of process will result in an output result.txt similar to Figure 3.

11. Results:

We have performed experiment on Brute Force algorithm, Binary Search Tree and Hashing for various data points. Below is the result analysis graph for 2D.

For Data Points $\times 10^4$:

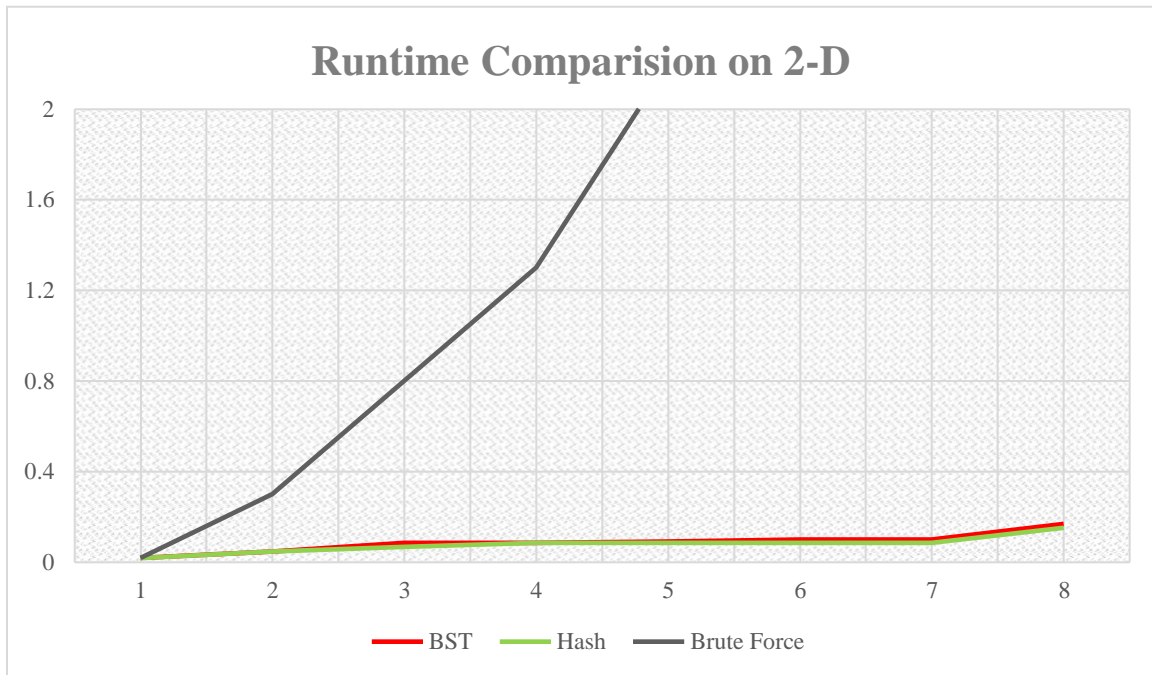


Figure 5.

In Figure 5 the X axis represents **Data points** and Y axis represents **Running Time (Seconds)**. In $\times 10^4$ data point experimentation both BST and Hashing are nearly close to each other because the number of points are less. As we increase the data points, we will see the logarithmic and linear curve difference. The brute force algorithm increases exponentially.

For Data Points $\times 10^5$:

In Figure 6 the X axis represents **Data points** and Y axis represents **Running Time (Seconds)**. As compared to previous chart, the chart displayed in Figure 6 is for data points $\times 10^5$. At higher data points, we can clearly see the difference in run time between BST and Hashing curve. The Brute force run time increases drastically in such high ranges. X axis represents **Data points** and Y axis represents **Running Time (seconds)** in Figure 6

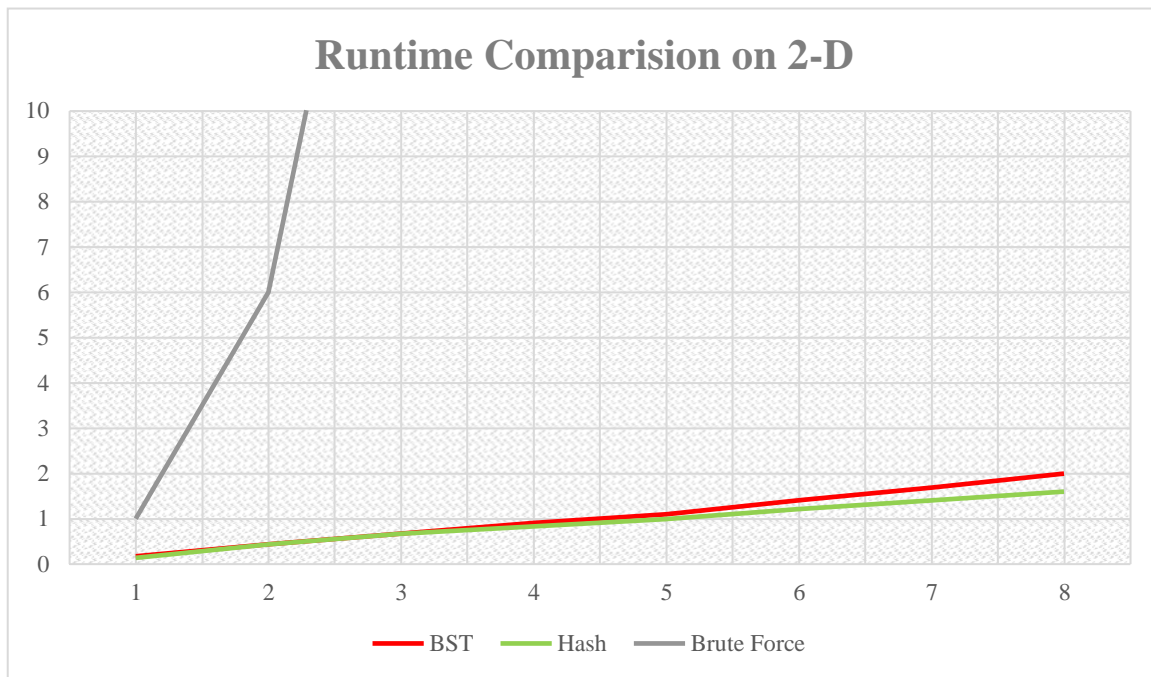


Figure 6.

11. Conclusion:

From the experiments conducted above we have verified that randomized incremental algorithm runs logarithmic and linear time as expected. Below are the findings for two data structure.

Binary Search tree:

Using BST data structure for implementing box dictionary, this algorithm takes logarithmic time of $O(n \log n)$. This graphical depiction for this we have seen in results section.

Dynamic Perfect Hashing:

Using Hashing data structure for implementing box dictionary, this algorithm runs in linear time of $O(n)$.

We have also seen the result for Brute Force implementation which runs in $O(n^2)$ time and is drastically slow. The fastest implementation for this algorithm is Dynamic Perfect Hashing.

12. References

- [1] Last Name, F. M. (Year). Article Title. Journal Title, Pages From - To. Last Name, F. M. (Year).
Book Title. City Name: Publisher Name.
- [2] M.O Rabin Probabilistic algorithms J.F Traub (Ed.),
Algorithms and Complexity, Academic Press, New York (1976), pp. 21-30
- [3] S. Khuller and Y. Matias. A simple randomized sieve algorithm for closest pair problem. In
Proc. 3rd canad. Conf. Comput. Geom., pp130-134, 1991
- [4] M. Golin, R. Raman, C. Schwarz, and M. Smid, "Simple randomized algorithms for closest
pair problems", Nordic Journal of Computing, 2:3-27, 1995
- [5] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F Mayer, H. Rohnert, R. Tarjan "Dynamic Perfect
hashing: Upper and Lower Bounds" SIAM L. Computing, 1990