TIME-OPTIMAL HEURISTIC ALGORITHMS FOR FINDING CLOSEST-PAIR OF POINTS IN 2D AND 3D

Mashilamani Sambasivam

(formerly) Department of Computer Science, Texas A&M University, USA
kandan1976@outlook.com

ABSTRACT

Given a set of n points in 2D or 3D, the closest-pair problem is to find the pair of points which are closest to each other. In this paper, we give a new O(n log n) time algorithm for both 2D and 3D domains. In order to prove correctness of our heuristic empirically, we also provide java implementations of the algorithms. We verified the correctness of this heuristic by verifying the answer it produced with the answer provided by the brute force algorithm, through 600 trial runs, with different number of points. We also give empirical results of time taken by running our implementation with different number of points in both 2D and 3D.

KEYWORDS

Closest-pair, Algorithm, Heuristic, Time-Optimal, Computational Geometry, 2D, 3D

1. INTRODUCTION

The closest-pair solution has many applications in real-life. It forms a main step in many problem-solving procedures. These include applications in air/land/water traffic-control systems. A traffic control system can use the solution in order to avoid collisions between vehicles. The algorithm has applications in detecting collisions after they happen. There are also applications in self-navigating vehicles. The solution also has applications in bodies which must always keep close to particular other bodies. The problem also has applications in imaging technologies, pattern recognition, CAD, VLSI.

2. PREVIOUS WORK

The most popular algorithm in 2D appears in the book by Cormen et al[1] and is due to Preparata and Shamos[2]. The algorithm divides the problem spatially and uses a divide-and conquer method. Following this algorithm, many similar divide-and-conquer algorithms have been devised for 3D by dividing the points spatially by a plane.[3][4][5] contain a good survey of computational geometry algorithms. Our algorithm differs from previous algorithms in that it is much simpler and therefore much easier to implement practically. The previous best algorithms for 2D have a time bound of O(n log n) similar to our 2D algorithm. However, I am unable to establish the best time bound achieved by previous algorithms for 3D. I think the best time bound achieved by previous algorithms for 3D is $O(n*\log^2 n)$.

3. OUR ALGORITHMS

We present the 2D and 3D algorithms separately for clarity.

3.1. Algorithm for 2D

Algorithm 2D-ClosestPair( )

Given: n – number of points, p[1..n] – points array

Data structures used by algorithm:

d1[1..n] , d2[1..n], d3[1..n], d4[1..n] - distance arrays

sum[1..n] – sum array

index[1..n] – index array

1. a. Find point p1 such that its x coordinate is lower or equal to any other point in the array of points p.

   b. Find point p2 such that its x coordinate is higher or equal to any other point in the array of points p.

   c. Find point p3 such that its y coordinate is lower or equal to any other point in the array of points p.

   d. Find point p4 such that its y coordinate is higher or equal to any other point in the array of points p.

2. a. Find distance of each point in the p array from p1 and put its square in the d1 array.

   For i=1..n, $d1[i] = (\text{distance between } p1 \text{ and } p[i])^2$

   b. Find distance of each point in the p array from p2 and put its square in the d2 array.

   For i=1..n, $d2[i] = (\text{distance between } p2 \text{ and } p[i])^2$

   c. Find distance of each point in the p array from p3 and put its square in the d3 array.

   For i=1..n, $d3[i]=(\text{distance between } p3 \text{ and } p[i])^2$

   d. Find distance of each point in the p array from p4 and put its square in the d4 array.

   For i=1..n, $d4[i] = (\text{distance between } p4 \text{ and } p[i])^2$

3. Calculate the sum array using the following formula:

   For i=1..n, sum[i] = 11* d1[i] + 101 * d2[i] + 1009* d3[i] + 10007 * d4[i]

4. Initialise the index array to contain the indexes.

   For i=1..n, index[i] = i

5. Mergesort the sum array. While mergesorting, if you exchange any 2 indices i and j of sum array, be sure to exchange the corresponding entries i and j of index array.

6. For i=1..(n-1), Compare each point p[index[i]] to the 10 next points (if they exist).

   ie. p[index[i+1]], p[index[i+2]]..p[index[i+10]]

   If the 2 points being compared is the closest pair found so far, then store the 2 points.

7. Output the closest pair of points found.

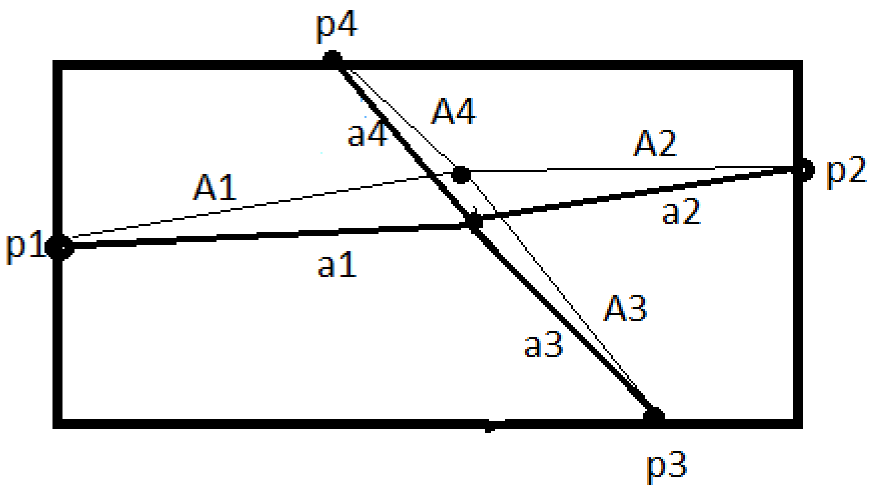

Figure 1. Closest Pair

Assume p1, p2, p3, p4, seen in the Figure 1 above, are the extreme points found in step 1 of our algorithm. Then the basic idea of our algorithm is that the closest pair of points (the 2 points inside the rectangle) should be almost equidistant from each of the 4 points (see Figure 1 above). That is, a1 should be near A1 numerically, and a2 should be near A2 numerically, and a3 should be near A3 numerically, and a4 should be near A4 numerically.

So what our algorithm does is that it calculates the distance of each point from the 4 extreme points and puts its square in the corresponding d array. We wish to find (d1,d2,d3,d4) of a point x such that it almost equals (d1,d2,d3,d4) of a point y. The closer the match of the d's, the closer the points are in the 2D plane.

So what we do to find the closest match of (d1, d2, d3, d4) among all points in the d array, is that we multiply each by a prime number and add them to get the sum array. The closer the (d1, d2, d3, d4) of point x is to (d1,d2,d3,d4) of point y, the closer will be the sum numerically. Multiplying by prime numbers gives us a unique signature of each point in the sum array. Note that the prime numbers are all different from each other.

So then, we let the index array carry the index of the point corresponding to the sum array. We then mergesort the sum array, taking care to exchange corresponding entries of index array when we exchange 2 elements of the sum array.

Now, we have the sorted sum array, and the points they represent are in the index array. Now, all we have to do is compare each point in the index array with 10 points that follow it. If the distance between 2 points being compared is the closest pair we have so far, it get stored. The closest pair of points is then output.

3.2. Algorithm for 3D

Algorithm 3D-ClosestPair( )

Given: n – number of points, p[1..n] – points array

Data structures used by algorithm:

d1[1..n] , d2[1..n], d3[1..n], d4[1..n], d5[1..n], d6[1..n] - distance arrays

sum[1..n] – sum array, index[1..n] – index array

1. a. Find point p1 such that its x coordinate is lower or equal to any other point in the array of points p.

   b. Find point p2 such that its x coordinate is higher or equal to any other point in the array of points p.

   c. Find point p3 such that its y coordinate is lower or equal to any other point in the array of points p.

   d. Find point p4 such that its y coordinate is higher or equal to any other point in the array of points p.

   e. Find point p5 such that its z coordinate is lower or equal to any other point in the array of points p.

   f. Find point p6 such that its z coordinate is higher or equal to any other point in the array of points p.

2. a. Find distance of each point in p array from p1 and put its square in the d1 array.
   For i=1..n, $d1[i] = (\text{distance between } p[i] \text{ and } p1)^2$

b. Find distance of each point in p array from p2 and put its square in the d2 array
   For i=1..n, d2[i] = (distance between p[i] and p2)^2

c. Find distance of each point in p array from p3 and put its square in the d3 array.
   For i=1..n, d3[i] = (distance between p[i] and p3)^2

d. Find distance of each point in p array from p4 and put its square in the d4 array.
   For i=1..n, d4[i] = (distance between p[i] and p4)^2

e. Find distance of each point in p array from p5 and put its square in the d5 array.
   For i=1..n, d5[i] = (distance between p[i] and p5)^2

f. Find distance of each point in p array from p6 and put its square in the d6 array.
   For i=1..n, d6[i] = (distance between p[i] and p6)^2

3. Calculate the sum array using the following formula:

   For i=1..n,

   sum[i] = 11*d1[i] + 101* d2[i] + 547*d3[i] + 1009*d4[i] + 5501*d5[i] + 10007*d6[i]

4. Initialise the index array to contain the indexes.

   For i=1..n, index[i] = i

5. Mergesort the sum array. While mergesorting, if you exchange any 2 indices i and j of

   sum array, be sure to exchange the corresponding entries i and j of index array.

6. For i=1..(n-1), Compare each point p[index[i]] to the 100 next points (if they exist)

   ie. p[index[i+1]], p[index[i+2]]..p[index[i+100]]

   if the 2 points being compared is the closest pair found so far, then store the 2 points.

7. Output the closest pair of points found.

Assume p1, p2, p3, p4, p5, p6 are the extreme points found in step 1 of our 3D algorithm. Then the basic idea of our algorithm is that the closest pair of points should be almost equidistant from each of the 6 points.

So what our algorithm does is that it calculates the distance of each point from the 6 extreme points and puts its square in the corresponding d array. We wish to find (d1,d2,d3,d4,d5,d6) of a point x such that it almost equals (d1,d2,d3,d4,d5,d6) of a point y. The closer the match of the d's, the closer the points are in 3D.

So what we do to find the closest match of (d1, d2, d3, d4, d5, d6) among all points in the d array, is that we multiply each by a prime number and add them to get the sum array. The closer the (d1, d2, d3, d4, d5, d6) of point x is to (d1, d2, d3, d4, d5, d6) of point y, the closer will be the sum numerically. Multiplying by prime numbers gives us a unique signature of each point in the sum array. Note that the prime numbers are all different from each other.

So then, we let the index array carry the index of the point corresponding to the sum array. We then mergesort the sum array, taking care to exchange corresponding entries in the index array when we exchange 2 elements of the sum array.

Now, we have the sorted sum array, and the points they represent are in the index array. Now, all we have to do is compare each point in the index array with 100 points that follow it. If the distance between 2 points being compared is the closest pair we have so far, it get stored.The closest pair of points is then output.

3.3 Correctness of our Heuristic Algorithm

We implemented our algorithms in 2D and 3D in java. The programs can be downloaded from the private url: https://drive.google.com/file/d/0B2MLVfnv5msBVlBnWEthcjRkM00/view?usp=sharing. We ran 600 trial runs with number of points ranging from 1 hundred to 10 million. We verified the answer we got with the answer got from the brute force algorithm of finding the closest pair. Our program got it right 100% of time.

The correctness of our heuristic is also intuitive—that the closest-pair of points will be almost equidistant from each of the extreme points found. Also, multiplying by a prime is intuitive in that it gives us a unique signature of each point in the sum array.

3.4 Running Time of our algorithm

Each of the steps in our algorithm takes O(n) time, except the mergesort step5. Mergesort step takes O(n log n) time. Note that the 6th step takes O(10n) for 2D algorithm and O(100n) for 3D algorithm, which is essentially O(n) time. So the total time taken by our algorithm is O(n log n).
The following tables gives the running time of our algorithm with varying number of points. It compares the running time against the running time of a brute force $O(n^2)$ algorithm. Each entry in the table (except brute-force algorithm entries for 1 million and 10 million points) is the average time of running the algorithm over 50 trial runs. The trials were run on a single-processor with base frequency of 1.6 GHz.

Table 1. Running time of our 2D algorithm and brute-force algorithm

Number of Points	**Our 2D algorithm time**	**Brute force algorithm time**
1000	17 millisecs	47 millisecs
10000	70 millisecs	1200 millisecs
100000	330 millisecs	99 secs
1 million	1.7 secs	> 12 hours
10 million	15.5 secs	>> 12 hours

Table 2. Running time of our 3D algorithm and brute-force algorithm

Number of Points	**Our 3D algorithm time**	**Brute force algorithm time**
1000	54 millisecs	49 millisecs
10000	165 millisecs	1700 millisecs
100000	731 millisecs	139 secs
1 million	5.2 secs	> 12 hours
10 million	47 secs	>> 12 hours

4. CONCLUSIONS

We found our heuristic algorithm gives the right answer 100% of time. Since the algorithm's correctness cannot be proved mathematically, it is still a heuristic. However, we have proved our algorithm's correctness empirically. Our algorithm is also time-optimal in that both the algorithms for 2D and 3D run in O(n log n) time. We verified empirically that our algorithm is time optimal.

Future work in finding closest pair of points can include finding the pair with multi-cores/multiprocessors, which are becoming more common day to day.

ACKNOWLEDGEMENTS

The author would like to thank Perumal. S, Sambasivam. K and Shankari. S for their support.

REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2009) Introduction to Algorithms, PHI Learning, Eastern Economy Edition.
[2] Franco P. Preparata & Michael Ian Shamos (1985) Computational Geometry: An Introduction, Springer.
[3] Herbert Edelsbrunner (1987) Algorithms in Combinatorial Geometry, Vol. 10 of EATCS Monographs on Theoretical Computer Science, Springer.
[4] Joseph O'Rourke (1998) Computational Geometry in C, Cambridge University Press.
[5] Mark de Berg, Otfried Cheong, Marc van Kreveld & Mark Overmars (2011) Computational Geometry: Algorithms and Applications, Springer.

AUTHORS

Mashilamani. S holds a Masters degree in Computer Science from Texas A&M University, College Station, USA and a Bachelors in Computer Science and Eng. from Madras University.