Efficient Approximate Algorithms for the Closest Pair Problem in High Dimensional Spaces

Xingyu Cai, Sanguthevar Rajasekaran*, and Fan Zhang

Dept. of CSE, University of Connecticut, Storrs, CT, United States
{xingyu.cai, sanguthevar.rajasekaran}@uconn.edu
Zhejiang University, Hangzhou, China
fanzhang@zju.edu.cn

Abstract. The Closest Pair Problem (CPP) is one of the fundamental problems that has a wide range of applications in data mining, such as unsupervised data clustering, user pattern similarity search, etc. A number of exact and approximate algorithms have been proposed to solve it in the low dimensional space. In this paper, we address the problem when the metric space is of a high dimension. For example, the drug-target or movie-user interaction data could contain as many as hundreds of features. To solve this problem under the ℓ_2 norm, we present two novel approximate algorithms.Our algorithms are based on the novel idea of projecting the points into the real line. We prove high probability bounds on the run time and accuracy for both of the proposed algorithms. Both algorithms are evaluated via comprehensive experiments and compared with existing best-known approaches. The experiments reveal that our proposed approaches outperform the existing methods.

Keywords: closest pair, high dimension, approximate algorithms

1 Introduction

Similarity search has been widely used in data mining. Example applications include finding the similarity between user patterns from online merchant transactions, analysis of social media connections, unsupervised data clustering, knowledge discovery from semantic data, etc. Two of the fundamental problems in data mining are finding the Nearest Neighbor (NN) and finding the Closest Pair (CP). These two problems are closely related. For instance, CP could be seen as an extension of NN, which requires more computation and thus is more challenging. In general, multi-feature data could be modeled as points in a high dimensional metric space. Among all the different similarity measurement metric, ℓ_p norm is commonly used. In this paper, ℓ_2 norm, or Euclidean distance, is employed as it is one of the most widely applicable measurements.

The Closest Pair Problem (CPP) we are addressing is that of identifying the closest pair of points from a given set of N points $\in \Re^m$ when m is not small. This classical problem has been studied extensively [19]. A straightforward algorithm for solving this problem takes $O(N^2m)$ time, where m is the dimension of the input space. Research works have been carried out in different domains for different

* Corresponding author; Sanguthevar.Rajasekaran@uconn.edu

purposes to solve this problem in an efficient way. In 1979, Fortune and Hopcroft presented a deterministic algorithm with a run time of $O(N \log \log N)$ assuming that the floor operation takes $O(1)$ time [8]. In [1], another divide-and-conquer deterministic algorithm was introduced. Later improvements include [18, 9, 11, 17]. In his seminal paper, Rabin proposed a randomized algorithm with an expected run time of $O(N)$ [7] (where the expectation is in the space of all possible outcomes of coin flips made in the algorithm). Rabin's algorithm also used the floor function as a basic operation. In 1995, the sieve method was proposed to eliminate points in a randomized way such that the actual comparison of the remaining candidates could be dramatically reduced [14]. A sample-based randomized approach was proposed in [6] in 1997 to solve several issues existing in [7]. Yao has proven a lower bound of $\Omega(N \log N)$ on the algebraic decision tree model (for any dimension) [21]. All these algorithms assume a constant dimensional space (i.e., $m = O(1)$), and the run times are exponentially dependent on the dimension, making them not applicable for a dimension of several hundreds.

In recent years, database applications have driven the research on CPP. By exploring the connection between CPP and matrix multiplication, Indyk [10] has presented an $O(N^{(w+3)/2})$ time algorithm for CPP in ℓ_1 and ℓ_∞ norms, where $O(N^w)$ is the time needed to multiply two $N \times N$ matrices. This algorithm is not applicable for ℓ_2 norm. Corral et al. [3] have provided a method that uses tree data structures. Besides exact algorithms, Lopez et al. [15] have provided an approximate algorithm to address this problem by making copies of the original data and employing random shifting on each copy. More recently, Locally Sensitivity Hashing (LSH) has gained attention in solving the NN problem. As a consequence, approximate algorithms based on LSH for CPP are proposed in the literature. Datar [5] has proposed a sub-quadratic time algorithm using LSH that solves the c-approximate problem (output neighbors that are no further than c times the distance between the nearest neighbors). Later Tao [20] improved Datar's algorithm and extended it to out-of-core CPP, where the I/O costs are optimized. The comparison in [20] shows that their algorithm outperforms the methods in [15, 3]. These algorithms mainly focus on the NN problem, or address the problem for efficiency in I/O cost, making them fit for out-of-core computation with many applications such as in database query processing. However, they are not very suitable for in-memory computation. Also, the construction of special data structures (such as LSB tree in [20]) will bring significant overhead for in-core tasks. In addition, the approximate methods in this domain are addressing the c-approximate problem that introduces a relaxation factor c.

Mueen, et al., have presented an elegant exact algorithm called MK for the CPP [16]. Though this algorithm was originally proposed to solve a special case of the CPP, known as the time series motif mining problem, it can be used to solve the CPP very well. Although MK is an $O(N^2 m)$ time algorithm, it improves the performance of the brute-force algorithm in practice using the triangular inequality and a technique called early-abandoning. MK is a deterministic algorithm that always finds the closest pair. To the best of our knowledge, MK is still one of the best performing algorithms for high dimensional CPP in practice,

even though it is no longer the state-of-the-art choice for time series motif mining problem. In this paper we use MK as the baseline to evaluate our proposed algorithms. To provide a fair comparison, we use the original MK code that is publicly available in http://alumni.cs.ucr.edu/mueen/MK/. The code for our proposed approaches can be found at https://github.com/TideDancer/ACPP.git.

In this paper we present two approximate algorithms for the CPP. One of them revisits the divide-and-conquer approach but modifies it to high dimensional settings. The other uses a novel idea in random projection: The original Johnson-Lindenstrauss Lemma shows the existence of a random projection of $O(\log N)$ dimension that preserves all pairwise distances with a high probability. For the CPP we only have to preserve the distance between the closest pair. We use random projection of points into 1D. We show that if we perform this projection $O(\log N)$ times, then the distance between the closest pair will be preserved at least once with a high probability. Note that although the proposed algorithms are sequential, all these algorithms along with MK, could be easily parallelized due to the independence of their subroutines.

The rest of this paper is organized as follows: In Section 2, we present two approximate algorithms for the high dimensional CPP. Running time and accuracy bounds are proved for both of the approaches (refer to Appendix for details). Comprehensive experiments are carried out to evaluate the performance of both algorithms in Section 3, and some conclusions are provided at the end.

2 Proposed Approximate Algorithms

Two approximate approaches for the high dimensional CPP are provided: ACP-P and ACP-D. Both algorithms always keep an upper bound δ_u on the distance δ^* between the closest pair of points. The common initial step for both algorithms is to obtain an upper bound on δ^* by picking a random sample of $\sqrt{N}$ points and identifying the distance between the closest pair of points in the sample. Even a brute-force algorithm will only take $O(N)$ time for doing this.

2.1 ACP-D

The divide-and-conquer algorithm of [1] performs well on low dimensional (e.g., 2D and 3D) data with a run time of $O(N \log N)$. Its performance degrades significantly on high dimensional data since the run time has an exponential dependence on the dimension. The divide-and-conquer algorithm proceeds by partitioning the input into two using the median along one of the dimensions. The closest pairs are recursively found for each of the two parts. Followed by this, we have to find the closest among the cross-part pairs. When the input is from 2D (i.e., $m = 2$), the number of cross-pairs that have to be considered is proved to be $O(N)$. When the input is from an m-dimensional space, the number of candidate cross-pairs to be considered goes up to $O(N \times 3^m)$. This number can be $\Omega(N^2)$ or worse. Thus the performance could be as bad as that of the brute force algorithm.

In this section we propose an enhanced divide-and-conquer algorithm. The idea is to choose the candidate cross-part pairs appropriately. Here again we

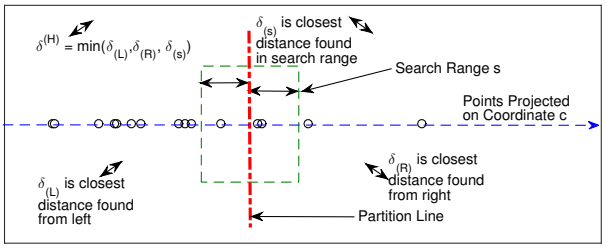

Fig. 1: Illustration of search within a range s around the partition line along a particular coordinate

Algorithm 1 ACP-D

Input: N points $p_i \in \Re^m$ $(1 \leq i \leq N)$, brute-force subset size T, search range constant α. Initialize left = 1, right = N, depth = 1

Output: function ACP-D(depth, left, right) finds the closest pair (l, r) with the smallest Euclidean distance $D(l, r)$

```
len = right - left + 1
if depth = 1 then
    Randomly select one coordinate c^(H), c^(H) ∈ [1, d];
    Sort the points based on values p_i[c^(H)] along the coordinate c^(H);
end if
if len ≤ T then
    Use brute-force to find best-so-far d(i, j) where left ≤ i ≤ j ≤ right;
    if d(i, j) < D(l, r) then
        l = i; r = j; D(l, r) = d(i, j);
    end if
    return D(l, r);
else
    mid = left + len/2;
    ACP-D(depth+1, left, mid);
    ACP-D(depth+1, mid+1, right);
    Obtain two sets of indices: S = {p_i}, i < mid, p_mid − p_i ≤ αD/√N
    and S' = {p_j}, j > mid, p_j − p_mid ≤ αD/√N;
    for i ∈ S do
        for j ∈ S' do
            if dist(p_i, p_j) < D(l, r) then
                l = i; r = j; D(l, r) = dist(p_i, p_j);
            end if
        end for
    end for
end if
```

partition the input into two and recursively find the closest pair in each part. To find the closest cross-part pair we do the following: Let $\mathcal{H}$ be the hyperplane that partitions the input into two (based on the median along one of the coordinates). We have to consider all pairs of the form (a, b) where a is one side of the hyperplane $\mathcal{H}$ and b is on the other side of $\mathcal{H}$. Instead of checking all such pairs we only consider pairs where a and b are on different sides of $\mathcal{H}$ but within a distance of s. We refer to s as the search range (see Figure 1). This procedure is repeated by partitioning along different coordinates to increase the chances of finding the closest pair.

To begin with, ACP-D randomly chooses a coordinate to do partition. It then recursively finds the closest pair's distance from the left and the right partitions (denote the distances as $\delta_{(L)}, \delta_{(R)}$). Next we look at all the points that

reside within a search range s around the partition hyperplane $\mathcal{H}$ along this coordinate, and find the closest pair (distance as $\delta_{(s)}$) among these candidates. Followed by this we update the pair with the minimum distance denoted as $\delta^{(\mathcal{H})} = \min(\delta_{(L)}, \delta_{(R)}, \delta_{(s)})$. The detailed pseudocode of ACP-D is given in Algorithm 1. An illustration of searching within a range s is shown in Figure 1. We establish the following theorem and provide the proof in Appendix. This theorem offers a probabilistic bound on success rate and run time of ACP-D. To boost the success rate, we repeat ACP-D and output the closest pair seen as the closest pair of points.

Theorem 1. *Let $p[c]$ represent vector p's c-th element, or equivalently p's c-coordinate value. Assume that the coordinate values in each dimension are uniformly distributed and let the spread length of points be $r = \max_i p_i[c^{(\mathcal{H})}] - \min_j p_j[c^{(\mathcal{H})}]$ on the partition coordinate $c^{(\mathcal{H})}$. Use a search range of $s = \sqrt{\alpha}\frac{\delta^{(\mathcal{H})}}{\sqrt{m}}$. As long as $r^2 = \Omega(N)$ where m is the dimension, ACP-D algorithm's expected run time will be $T(N) = O(N \log N)$, with a high probability.*

Corollary 1. *We have the following probability bound on the run time:*

$$Prob\{T(N) \geq (\beta + 1)\alpha(\delta^{(\mathcal{H})})^2 N \log N\} \leq e^{-\beta}, \text{ for any } \beta > 0.$$

2.2 ACP-P

Random projection lemma [12]states that pairwise distances are closely preserved in a random $O(\log N)$-dimensional space with a high probability. In this paper we prove that, if we repeat projecting the input points from $\Re^m$ to $\Re^d$ randomly ($d < m$) a total of k times, as long as kd satisfies a certain condition, the closest pair's distance will be closely preserved in at least one of the projections, with a high probability. In addition, $d = 1$ would significantly reduce the computation cost. We exploit this property in the ACP-P algorithm.

After the projection, all the pairs in the projected space that are within a distance of $\delta^{(P)} = (1 + \epsilon)\delta_u$ (in $\Re^d$) needs to be identified, where ϵ is a small constant. For the case of $d > 1$, identifying these close pairs in $\Re^d$ still remains a difficult task. One can use hyper-sphere centered at each point with a radius of $\delta^{(P)}$, and check if there are other points in the hyper-sphere. However, this might be even harder than directly computing all pairwise distances in $\Re^d$, which takes $O(N^2 d)$ running time. On the other hand, if in 1-D space ($d = 1$), the hyper-sphere becomes left and right intervals, making the job of identifying close points within an interval of $\delta^{(P)}$ extremely easy. To be specific, one can use any sorting algorithm to first sort all the projected points because all the points are identified by a scalar value in 1-D space. Then a scanning from left to right is performed and all the adjacent points within a certain range are detected. In total it only requires an $O(N \log N)$ running time. In fact, sorting could be replaced by a griding approach to identify pairs within an interval (see Appendix 4.4). After identifying these pairs, the Euclidean distance between each pair is computed in $\Re^m$. The pair with the least distance is kept and δ_u is updated.

The above projection-identification process is repeated k times. We show (in Appendix) that if $kd = \Theta(\log N)$, then the closest pair would come within a

Algorithm 2 ACP-P

Input: N points in $\Re^m$: $p_1, p_2, \ldots, p_N$.
Output: The closest pair of input points.
1: $j = 1$
2: **repeat**
3: Randomly generate a projection vector $\Phi \in \Re^{1\times n}$
4: **for** $i = 1$ to N **do**
5: $p'_i = \Phi p_i^T$
6: **end for**
7: Sort $p'_1, p'_2, \ldots, p'_N$;
8: **for** $i = 1$ to N **do**
9: Identify the interval that p'_i belongs to;
10: **end for**
11: **for** every interval **do**
12: Generate all possible pairs from the points that have fallen into this interval. These are candidate pairs;
13: **end for**
14: For each candidate pair compute the distance in $\Re^m$ and pick the pair with the least distance. Let this distance be δ_j;
15: $j = j + 1$;
16: **until** $j = k$
17: Find $\delta_o = \min\{\delta_1, \delta_2, \ldots, \delta_k\}$;
18: **return** δ_o

distance of $(1+\epsilon)\delta_u$ in the projected space at least once with a high probability. Note that in the original Johnson-Lindenstrauss Lemma, $d = O(\log N)$. **The reason that we can push the limit to $d = 1$ is because we only have to preserve the distance between the closest pair, and not all pairwise distances.** We provide the following theorems and the corresponding proofs (in Appendix).

Theorem 2. *Let the closest pair have a distance of δ^*. If we repeat the random projection $\Re^m \to \Re^d$ for a total of k times, then the probability that $(\delta^{(P)})^2 < (1+\epsilon)(\delta^*)^2$ at least once is high (i.e., $\geq 1 - N^{-\alpha}$ where α is some constant), as long as $dk \geq \frac{4\alpha}{\epsilon^2-\epsilon^3}\log N$, for any $\epsilon \in [0, 1]$.*

Corollary 2. *Let $d = 1$. In each iteration of ACP-P, let the projected points be quantized with intervals of length $2(1+\epsilon)\delta_u$. The probability that the closest pair (in $\Re^m$) will fall into the same interval is $\geq \frac{1}{2}[1 - e^{-(\epsilon^2-\epsilon^3)/4}]$. This in turn means that the number of iterations taken by ACP-P to identify the closest pair of points with a high probability is $k = O\left(\frac{\alpha \log N}{\epsilon^2-\epsilon^3}\right)$.*

In practice, we have found that the sorting based implementation in 1-D does not introduce an observable overhead. A detailed pseudocode of ACP-P that employs sorting is given in Algorithm 2. It is worth pointing out that the key difference between ACP-P and the LSH method used in [20] is after projection. The linear search based on the sorted list of points is much more efficient to identify each points' close neighbors, rather than a grid scheme using hashset technique. In our experiments we have realized that neither C++/boost hashset nor google's hashset could achieve desirable in-core performance, making the method in [20] not suitable for in-memory computations. Besides, the probability bound analyses in Appendix are also different.

3 Experiments

We have conducted experiments to evaluate the performance of ACP-P and ACP-D against MK. We have employed an Intel Xeon E5 CPU @ 3.2 GHz machine. The experiments have been performed on synthetic datasets, with different numbers of points and dimensions. Coordinate values have been generated uniformly randomly from the range: $[0, 1000]$. The following values have been used: $N = 10, 20, 30, 40, 50 \times 10^3$ and $m = 128, 256, 512, 1024$ and 2048.

To further boost the success rate, in each run we repeat the approximate algorithms Q times and output the best among them. Q is designed as $Q_{\text{ACP-D}} = h\frac{N}{10\times10^3}$ and $Q_{\text{ACP-P}} = h(\frac{N}{10\times10^3})^2$ for ACP-D and ACP-P, respectively. N is the input size and h is the hyper parameter. For instance if $N = 30k, h = 2$, then $Q = 6$ for ACP-D and $Q = 18$ for ACP-P. We perform 10 runs and provide the average running time, average rank and the hit rate (i.e., the fraction of the number of times the closest pair is found in 10 runs). Clearly, the larger the h, the better is the accuracy and the worse is the run time.

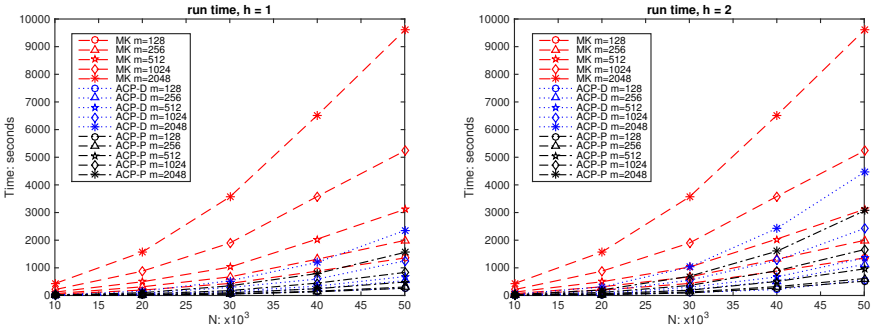

Fig. 2: Run time comparison

Figure 2 shows the run time comparison. Clearly, for all settings, ACP-D and ACP-P are significantly faster than the MK algorithm. As expected, the run time when $h = 2$ (the right plot) is longer than when $h = 1$ (the left plot) for both approximate algorithms. When N is smaller, ACP-D could be slightly faster, but when N is larger, ACP-P becomes the fastest. For instance, when $N = 50k, m = 1,024, h = 1$, ACP-D's run time is 1,252 seconds and ACP-P's is 832 seconds, while MK is much slower using 5,230 seconds.

To illustrate the accuracy, in Figure 3, the hit rate is presented. Again in the case of $h = 2$, the overall hit rate is higher as expected. When m is higher, the hit rate tends to be better than in smaller dimension cases. The average rank is also provided in Table 1. From the table we can see that for larger N, the proposed algorithms are more robust because the average ranks are closer to 1. And the overall average rank for $h = 2$ is also better than that for $h = 1$.

In addition to the rank of the best pair identified, we also report the difference between the output pair's distance and the true closest pair's distance. We define

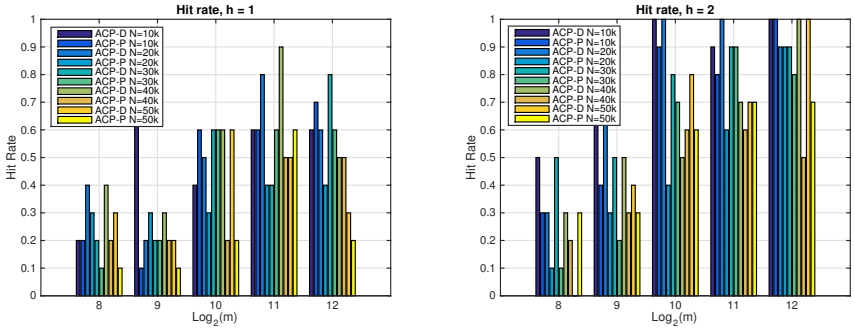

Fig. 3: Hit rate comparison

Table 1: Average Rank (the smaller the better)

	$m = 128$		$m = 256$		$m = 512$		$m = 1024$		$m = 2048$	
$h = 1$	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P
N=10k	2.7	7.1	4	6.2	4.4	8.5	4	6.6	3.9	7.9
N=20k	1.7	4.7	2	3.7	3.5	4.9	3.6	5	3.2	5.5
N=30k	2.1	1.9	1.7	2.7	1.9	2.3	1.5	3.2	1.8	2.9
N=40k	1.5	1.9	1.2	3.3	2.4	2	1.3	2.9	2.1	2.2
N=50k	1.6	1.5	1.4	2.6	1.2	1.4	1.7	2	2	3.5
$h = 2$	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P
N=10k	1.6	4.8	2.5	3.7	1.7	5.5	2.2	5.8	3.5	5.7
N=20k	1.5	1.9	1.2	3.3	1.8	2.4	1.5	4.6	2.2	3.1
N=30k	1	1.2	1	2.1	1.2	1.5	1.5	2	1.3	1.6
N=40k	1.1	1.2	1	1.4	1.1	1.1	1.3	1.7	1.3	1.3
N=50k	1	1	1.1	1.1	1.1	1.2	1	1.8	1	1.4

the distance ratio as $\rho = d/d^*$, where d^* is the distance between the closest pair of points and d is the distance between the output pair of points. In Table 2, we show the mean and variance of ρ when $h = 2$, and demonstrate that for our synthetic dataset, the distance ratio is very close to 1 with a small variance. This also proves the robustness of our proposed approximate algorithms.

4 Conclusions

In this paper we have offered two approximate algorithms for solving the CPP. Both of them are based on the idea of converting high dimensional search into line search. We provide theoretical bounds on the run time and prove the accuracy of ACP-D. For ACP-P, we exploit random projections but push the limit to 1-D space because we only identify the closest pair rather than preserving all pairwise distances. A theoretical analysis is also provided. In the experiments, we perform comprehensive simulations to evaluate both the run time and the accuracy for the proposed approximate algorithms. The results reveal that our approach runs much faster than the state-of-the-art method while still keeping a very good accuracy. Our algorithms could be easily parallelized for further speedups.

Table 2: Distance Ratio ρ (Mean and Standard Deviation) when $h = 2$

$h = 2$	$m = 128$		$m = 256$		$m = 512$		$m = 1024$		$m = 2048$	
Mean	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P
N=10k	1.010	1.034	1.004	1.008	1.002	1.012	1.001	1.004	1.006	1.006
N=20k	1.008	1.015	1.000	1.002	1.002	1.004	1.001	1.003	1.005	1.007
N=30k	1.000	1.004	1.000	1.002	1.000	1.001	1.000	1.001	1.001	1.002
N=40k	1.001	1.003	1.000	1.012	1.000	1.000	1.001	1.001	1.001	1.001
N=50k	1.000	1.000	1.002	1.002	1.000	1.000	1.000	1.005	1.000	1.001
Std	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P	ACP-D	ACP-P
N=10k	0.011	0.029	0.004	0.003	0.002	0.006	0.001	0.002	0.002	0.004
N=20k	0.015	0.015	0.000	0.003	0.003	0.003	0.001	0.003	0.005	0.005
N=30k	0.000	0.011	0.000	0.002	0.001	0.002	0.000	0.001	0.002	0.003
N=40k	0.004	0.006	0.000	0.014	0.001	0.001	0.001	0.002	0.001	0.001
N=50k	0.000	0.000	0.007	0.007	0.001	0.001	0.000	0.005	0.000	0.001

Acknowledgments

This work has been partly supported by NSF Grants 1447711 & 1743418 to SR; and the National Natural Science Foundation of China Grants 61472357 & 61571063 to FZ.

References

1. J. L. Bentley and M. I. Shamos. Divide-and-conquer in multidimensional space. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 220–230. ACM, 1976.
2. S. Chaudhuri and D. Dubhashi. Probabilistic recurrence relations revisited. *Theoretical computer science*, 181(1):45–56, 1997.
3. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *ACM SIGMOD Record*, volume 29, pages 189–200. ACM, 2000.
4. S. Dasgupta and A. Gupta. An elementary proof of a theorem of johnson and lindenstrauss. *Random Structures & Algorithms*, 22(1):60–65, 2003.
5. M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.
6. M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.
7. T. J. W. I. R. C. R. Division and M. Rabin. *Probabilistic algorithms*. 1976.
8. S. Fortune and J. Hopcroft. A note on rabin's nearest-neighbor algorithm. *Information Processing Letters*, 8(1):20–23, 1979.
9. Q. Ge, H.-T. Wang, and H. Zhu. An improved algorithm for finding the closest pair of points. *Journal of computer Science and Technology*, 21(1):27–31, 2006.
10. P. Indyk, M. Lewenstein, O. Lipsky, and E. Porat. Closest pair problems in very high dimensions. In *ICALP*, volume 3142, pages 782–792. Springer, 2004.
11. M. Jiang and J. Gillespie. Engineering the divide-and-conquer closest pair algorithm. *Journal of Computer Science and Technology*, 22(4):532–540, 2007.

12. W. B. Johnson and J. Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1, 1984.
13. R. M. Karp. Probabilistic recurrence relations. *Journal of the ACM (JACM)*, 41(6):1136–1150, 1994.
14. S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118(1):34–37, 1995.
15. M. A. Lopez and S. Liao. Finding k-closest-pairs efficiently for high dimensional data. 2000.
16. A. Mueen, E. Keogh, Q. Zhu, S. Cash, and B. Westover. Exact discovery of time series motifs. In *Proceedings of the 2009 SIAM International Conference on Data Mining*, pages 473–484. SIAM, 2009.
17. J. C. Pereira and F. G. Lobo. An optimized divide-and-conquer algorithm for the closest-pair problem in the planar case. *Journal of Computer Science and Technology*, 27(4):891–896, 2012.
18. F. P. Preparata and M. Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.
19. M. I. Shamos and D. Hoey. Closest-point problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 151–162. IEEE, 1975.
20. Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transactions on Database Systems (TODS)*, 35(3):20, 2010.
21. A. C.-C. Yao. Lower bounds for algebraic computation trees of functions with finite domains. *SIAM Journal on Computing*, 20(4):655–668, 1991.

Appendix

4.1 Proof of Theorem 1

Proof. In the recursive algorithm ACP-D, the initial data size is N and each recursion splits the input into two halves. Recursion ends when the size is below the threshold T. The brute-force or MK algorithm can be used when the final input size is $\leq T$. Thus the total number K of recursions is $K = \log(N/T)$.

In each iteration, the closest pair (p_a, p_b) may not be captured only if this pair is neither in the left half nor the right half, and also not captured by the search range in the middle. So the probability of failure in this iteration equals the probability that p_a, p_b is split into left and right halves (denote this probability as $P_{(sep)}$), multiplied by probability that (p_a, p_b) is missed in the search range s (denote this probability as $P_{(miss)}$) conditioned on $P_{(sep)}$. Denote iteration k with a superscript such as $P^k_{(sep)}$.

Assume a uniform distribution of points' coordinate values. In the randomly chosen coordinate $c^{(\mathcal{H})}$, denote p_M as the index of the median point along this coordinate. Then, for the first iteration,

$$P^1_{(sep)} = \text{Prob}\{p_a[c^{(\mathcal{H})}], p_b[c^{(\mathcal{H})}] \text{ are split by } p_M[c^{(\mathcal{H})}]\} = 1/2.$$

In the second iteration, probability that p_a, p_b are split again in both halves is

$$\begin{aligned} P^2_{(sep)} &= 2 \times (\text{Prob}\{p_a[c^{(\mathcal{H})}], p_b[c^{(\mathcal{H})}] \text{ in same half}\} \times \text{Prob}\{\text{They are separated}\}) \\ &= 2 \times (\frac{1}{4} \times \frac{1}{2}) = \frac{1}{4}. \end{aligned}$$

Thus we can easily see that $P^k_{(sep)} = \frac{1}{2^k}$.

Next let us compute $P_{(miss)}$. Let the distance between the closest pair (p_a, p_b) be δ^* and let $\delta^{(D)} = \min(\delta_{(L)}, \delta_{(R)})$. Clearly, $\delta^{(\mathcal{D})} \geq \delta^*$. Since there are m dimensions, the expected contribution of δ^* to coordinate $c^{(\mathcal{H})}$ is $E[l_{c^{(\mathcal{H})}}(p_a, p_b)] = \delta^*/\sqrt{m}$, which means there are at least $(m - \frac{m}{\alpha})$ coordinates for which $l_c(p_a, p_b) \leq \sqrt{\alpha}\frac{\delta^{(\mathcal{D})}}{\sqrt{m}}$, $\alpha > 1$.

If we set the search range as $s = \sqrt{\alpha}\frac{\delta^{(\mathcal{D})}}{\sqrt{m}}$, meaning we check pairs (p_i, p_j) such that $p_i[c] \in [p_M[c] - s, p_M[c]]$ and $p_j[c] \in [p_M[c], p_M[c] + s]$, then choosing a random coordinate would give Prob{Capture the closest pair | $P^1_{(sep)}$} $\geq 1 - 1/\alpha$. As a result, for any iteration k, the conditional probability of failure is

$$P^k_{(miss)}\{\text{Fail} \mid P^k_{(sep)}\} = (1 - \text{Prob}[\text{Capture} \mid P^k_{(sep)}]) \leq 1/\alpha.$$

So the probability of failure through all the recursions is

$$\text{Prob}[\text{Fail}] = \sum_{k=1}^{K} P^k_{(sep)} P^k_{(miss)} \leq \frac{1}{\alpha} \sum_{k=1}^{\log N/T} \frac{1}{2^k} \leq \frac{1}{\alpha}.$$

Thus we arrive at the following Lemma:

Lemma 1. *Assuming a uniform distribution of points, if we set the search range as* $s = \sqrt{\alpha}\frac{\delta^{(\mathcal{D})}}{\sqrt{m}}$, $\alpha > 1$, *where* $\delta^{(\mathcal{D})}$ *is minimum of the closest distances returned by the left and the right halves, ACP-D will find the closest pair with a probability of* $\geq 1 - 1/\alpha$. *Q.E.D.*

Assuming a uniform distribution and letting the spread length of points be $r = \max_i p_i[c^{(\mathcal{H})}] - \min_j p_j[c^{(\mathcal{H})}]$ on the chosen coordinate, within search range s, the expected number of points residing in will be sN/r. Thus the expected number of pairs that we need to compute distances for would be $(sN/r)^2$. Therefore, the expected running time will be

$$T(N) = 2T(\frac{N}{2}) + \tilde{O}(\frac{s^2N^2}{r^2}m) = 2T(\frac{N}{2}) + \tilde{O}(\frac{\alpha(\delta^{(\mathcal{D})})^2N^2}{r^2})$$

assuming that each distance computation takes $O(m)$ time. As long as $r^2 = \Omega(N)$ ACP-D algorithm's expected run time will be:
$T(N) = 2T(N/2) + \tilde{O}(\alpha(\delta^{(\mathcal{D})})^2N) = \tilde{O}(N \log N)$ as α and $\delta^{(\mathcal{D})}$ are some constants. □

4.2 Proof of Corollary 1

Proof. Applying the probabilistic recurrence relationship [13] and the revised version [2], under the same assumptions of Theorem 1, for a positive β, we have

$$\text{Prob}\{T(N) \geq (\beta + 1)\alpha(\delta^{(\mathcal{D})})^2N \log N\} \leq e^{-\beta}.$$

□

4.3 Proof of Theorem 2

Proof. In the proof for the JL lemma [4], the following lemma is also given:

Lemma 2. *For any fixed vector $v \in \Re^m$, projection matrix $\Phi : \Re^m \rightarrow \Re^d$ with i.i.d. Gaussian entries, i.e., $\Phi_{ij} = \frac{1}{\sqrt{d}}\mathcal{N}(0,1)$, the following statements are true: $E[||\Phi v||^2] = ||v||^2$ and $Prob[||\Phi v||^2 \geq (1+\epsilon)||v||^2] \leq e^{-(\epsilon^2-\epsilon^3)d/4}$, as well as $Prob[||\Phi v||^2 \leq (1-\epsilon)||v||^2] \leq e^{-(\epsilon^2-\epsilon^3)d/4}$.*

For simplicity in both analysis and implementation, we use Gaussian projections. Let the closest pair (in $\Re^m$) be (p_a, p_b) with a distance of δ^*. Using the second equation of the above theorem, we obtain:

$$\text{Prob}\{||\Phi(p_b - p_a)||^2 \geq (1+\epsilon)||(p_b - p_a)||^2\} \leq e^{-(\epsilon^2-\epsilon^3)d/4}.$$

Let the distance between p_a and p_b in $\Re^d$ be $\delta^{(\mathcal{P})}$. The above equation becomes:

$$\text{Prob}\{(\delta^{(\mathcal{P})})^2 \geq (1+\epsilon)\delta^{*2}\} \leq e^{-(\epsilon^2-\epsilon^3)d/4}.$$

The probability that the above event happens at least once in the k iterations is:

$$\text{Prob}\{(\delta^{(\mathcal{P})})^2 < (1+\epsilon)\delta^{*2}]\} \geq 1 - e^{-(\epsilon^2-\epsilon^3)dk/4}.$$

We want this to be a high probability. By high probability we mean a probability of $\geq (1 - N^{-\alpha})$, α being a probability parameter (normally assumed to be a constant ≥ 1). We want $1 - e^{-(\epsilon^2-\epsilon^3)dk/4} \geq 1 - N^{-\alpha}$. This happens when:

$$e^{-(\epsilon^2-\epsilon^3)dk/4} \leq N^{-\alpha} \quad \Rightarrow \quad (\epsilon^2 - \epsilon^3)dk/4 \geq \alpha \log N \quad \Rightarrow \quad dk \geq \frac{4\alpha}{\epsilon^2 - \epsilon^3} \log N.$$

□

4.4 Proof of Corollary 2

Proof. Projecting the input points into real numbers, i.e., $d = 1$, has a great computational advantage. To identify pairs within a specific distance in $O(N)$ time, we use quantization technique. Let the minimum and maximum projected values be m_1 and m_2, respectively. We partition the range $[m_1, m_2]$ into intervals of length $L = 2(1+\epsilon)\delta_u$ each. Each such interval has an integer index (starting from 1). We also extend the range $[m_1, m_2]$ by a random number r in the range $[0, (1+\epsilon)\delta_u]$. Specifically, we use the range: $[m_1 - r, m_2]$. Then for each point p_i we identify the interval that it falls into as: $\text{ID}(p_i) = \lceil (p_i/L) \rceil$. This can be done in a total of $O(N)$ time for all the points. For each interval, we generate all possible pairs from out of the points that belong to this interval. From out of all of these candidate pairs we pick the one with the least distance (in $\Re^m$). Clearly, the probability that two points that are within a distance of $\leq (1+\epsilon)\delta_u$ (in $\Re^m$) will fall into the same interval is $\geq 1/2$. Thus the number of iterations k can be computed in the same manner as above.

□