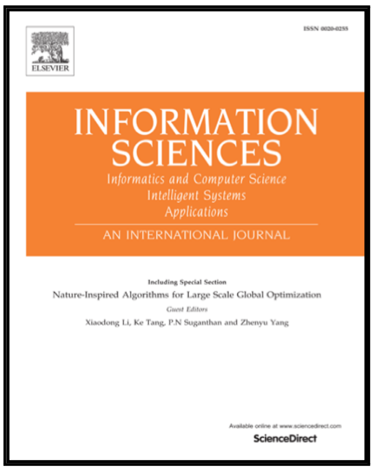


Fast Neighbor Search By Using Revised K-D Tree

Yewang Chen, Lida Zhou, Yi Tang, Jai Puneet Singh,
Nizar Bouguila, Cheng Wang, Huazhen Wang, Jixiang Du

PII: S0020-0255(18)30712-6
DOI: https://doi.org/10.1016/j.ins.2018.09.012
Reference: INS 13923

To appear in: *Information Sciences*

Received date: 19 May 2018
Revised date: 4 September 2018
Accepted date: 5 September 2018

Fast Neighbor Search By Using Revised K-D Tree

Yewang Chen

College of Computer, Science and Technology Huaqiao University, Xiamen, China, 361021, email: ywchen@hqu.edu.cn

Lida Zhou

College of Computer, Science and Technology Huaqiao University, Xiamen, 361021, China

Yi Tang[1]

School of Mathematics and Information Science, Guangzhou University, China, 510006 email: ytang@gzhu.edu.cn

Jai Puneet Singh

Concordia Engineering and Computer Science Concordia University Montreal, Quebec, Canada, H3G 2W1.

Nizar Bouguila

Concordia Institute for Information Systems Engineering, Faculty of Engineering and Computer Science, Concordia University, Montreal, Quebec, Canada, H3G 2W1. email: nizar.bouguila@concordia.ca

Cheng Wang

College of Computer, Science and Technology Huaqiao University, Xiamen, China, 361021

Huazhen Wang

College of Computer, Science and Technology Huaqiao University, Xiamen, China, 361021

Jixiang Du

College of Computer, Science and Technology Huaqiao University, Xiamen, China, 361021

Abstract

We present two new neighbor query algorithms, including range query (RNN) and nearest neighbor (NN) query, based on revised k-d tree by using two techniques. The first technique is proposed for decreasing unnecessary distance computations by checking whether the cell of a node is inside or outside the specified neighborhood of query

[1]corresponding author: Yi Tang, ytang@gzhu.edu.cn

point, and the other is used to reduce redundant visiting nodes by saving the indices of descendant points. We also implement the proposed algorithms in Matlab and *C*. The Matlab version is to improve original RNN and NN which are based on k-d tree, *C* version is to improve k-Nearest neighbor query (kNN) which is based on buffer k-d tree. Theoretical and experimental analysis have shown that the proposed algorithms significantly improve the original RNN, NN and kNN in low dimension, respectively. The tradeoff is that the additional space cost of the revised k-d tree is approximately $O(\alpha n \log(n))$.

Keywords: `k-d tree`, NN, kNN, RNN

2010 MSC: 00-01, 99-00

1. Introduction

Neighbor query, as a form of proximity search, is the optimization problem of finding the point in a given set that is closest (or most similar) to a given point. Closeness is typically expressed in terms of a dissimilarity function: the less similar the objects, the larger the function values. Formally, the nearest-neighbor (NN) query problem is defined as follows: given a set S of points in a space M and a query point $q \in M$, find the closest point in S to q. k-Nearest Neighbor query (kNN) is a direct generalization of this problem, the task of kNN is to find the first k closest points. Range query, known as range nearest-neighbor (RNN) query, is another type of neighbor query, which is to find neighbors within a specified neighborhood.

Neighbor query is a fundamental problem in computational geometry and machine learning, computer vision, pattern recognition, computational geometry, data compression, coding theory etc, and has been widely used in various applications. For example, content-based image and video retrieval Li et al. (2018); Cao et al. (2015) are nearest neighbor problems where the main goal is to find examples that are most relevant to the query in a large database; Some clustering algorithms, such as DBSCAN Chen et al. (2018 (in press), DPeak Rodriguez and Laio (2014), DCore Chen et al. (2018) perform the task of clustering based on density, where the density of an arbitrary point p is defined as the total number of points within a given range of p, which is in fact a RNN problem; Finding the best match for local image features in large data sets Philbin et al. (2007); Clustering local features into visual words by using k-means or similar algorithms Zhang et al. (2017). Besides, neighbor query also can be widely used in other fields, such as network security Cai et al. (2017); Zhu et al. (2018); Gao et al. (2018), cloud computing Li et al. (2015); Zhou et al. (2016), secure transmission Fan et al. (2017), model analysis He et al. (2017), and water data analysis Wang et al. (2018) etc.

However, the naive version of NN, RNN and KNN algorithms are easy to implement by computing the distances from the test example to all stored examples, but it is computationally intensive for large training sets. Many nearest neighbor search algorithms have been proposed over the years, which generally seek to reduce the number of distance evaluations actually performed. The goal of this paper is to improve these algorithms, and the main contributions of this paper are the followings: (1) The drawbacks of current k-d tree algorithms is discussed, and two techniques are invented to prune redundant distance computations and node visiting. (2) We implement our idea in both Matlab and *C*, the Matlab version is to improve the original RNN and NN, and *C* version is to improve kNN based on buffer k-d tree. (3) We conduct a series of experiments on real application and synthetic data sets, and the experimental results demonstrate significant improvement of the proposed algorithm.

The rest of this paper is organized as follow: Section 2 lists related works; Section 3 describes the notations used in this manuscript; Section 4 presents the drawbacks of current k-d tree based range search; Section 5 introduces the proposed methods in detail; Section 6 shows the experimental results of the proposed algorithms on various data sets, and Section 7 gives conclusion and our future works.

2. Related Works

There are some techniques that are used in neighbor query, such as partition trees, graph methods, hashing techniques and probabilistic approaches.

(1) Partition trees are one of the most popular techniques for RNN and NN, they are used to recursively split the space into subspaces, and organize the subspaces via a tree structure. Most approaches of this kind select hyper-planes or hyper-spheres to

partition the space and divide the data points into subsets, according to the distribution of data points.

K-d tree Bentley (1975) is a typical partition tree, which is widely used in many applications Zhang et al. (2016), and has various variants, such as optimized k-d trees Silpa-Anan and Hartley (2008), FRS Chen et al. (2017), and buffer k-d trees Gieseke et al. (2014) which is currently the fastest algorithm for NN and kNN query, as far as we know. However, k-d tree is not suitable for high-dimensional spaces. As a general rule, if the dimensionality is d, the number of data points, n, should satisfy $n >> 2^d$. A query with an axis-parallel rectangle in a k-d tree storing n points can be performed in $O(n^{1-1/d} + m)$ time, and in $O(log(n))$ time if ϵ is small De Berg et al. (2000), where m is the number of the reported points.

In addition to k-d tree, there are various other partitioning trees that can be used in RNN and NN. Leibe et al. Leibe et al. (2006) proposed a ball-tree data structure constructed using a mixed partitional-agglomerative clustering algorithm. Schindler et al. Schindler et al. (2007) proposed a new way of searching the hierarchical k-means tree. Philbin et al. Philbin et al. (2007) conducted experiments showing that an approximate flat vocabulary outperforms a vocabulary tree in a recognition task. Marius et al. Marius Muja (2014) described a modified k-means tree algorithm that gives the best results for some data sets, while randomized k-d trees are the best for others. Tao Tao et al. (2002) developed a new index structure called the U-tree for minimizing the range query overhead in uncertain database. Besides, there are also other techniques, such as R* tree Hjaltason and Samet (1999), PCA tree, k-means tree Muja and Lowe (2014), Exo-tree Hu and Lee (2006), anchors hierarchy Moore (2000), vptree Yianilos (1993), cover tree Beygelzimer et al. (2006) and spill-tree Liu et al. (2004) [23].

(2) Many hashing techniques are also proposed to approximately solve the problems of RNN and NN, such as ANN based on trinary-project tree Wang et al. (2014), Product quantization for nearest neighbor search Jegou et al. (2011), LSH (Locality-Sensitive Hashing) Andoni and Indyk (2008), FLANN Marius Muja (2014). For example, to solve the approximate nearest neighbor search problem (NNS) on the sphere, Becker et al. Becker et al. (2016) proposed a method using locality-sensitive filters

(LSF), with the property that nearby vectors have a higher probability of surviving the same filter than vectors which are far apart. Bawa et al. Bawa et al. (2005) showed that the performance of the standard LSH algorithm is critically dependent on the length of the hashing key and proposed the LSH Forest, a self-tuning algorithm that eliminates this data dependent parameter. Muja et al. Muja and Lowe (2009) proposed an automatic nearest neighbor algorithm configuration method by combining grid search with a finer grained NelderMead downhill simplex optimization process. They also invented new algorithms Muja and Lowe (2014) for approximate nearest neighbor matching and evaluate and compare them with previous algorithms. The authors showed that the optimal nearest neighbor algorithm and its parameters depend on the data set characteristics, and then describe an automated configuration procedure for finding the best algorithm to search a particular data set. Huang et al. Huang et al. (2016) discussed location sensitive hash functions and their applications such as biometric encryption Wu et al. (2016), keyword search in security Yang et al. (2018).

(3) Computing the quantification probabilities also has attracted much attention in the database community. Cheng et al. Cheng et al. (2004) used numerical integration, which is quite expensive. Cheng et al. Cheng et al. (2008) and Bernecker et al. Bernecker et al. (2011) proposed some filter refinement methods to give upper and lower bounds on the quantification probabilities. Agarwal et al. Agarwal et al. (2016) presented an efficient NN algorithms for (i) computing all points that are nearest neighbors of a query point with nonzero probability and (ii) estimating the probability of a point being the nearest neighbor of a query point, either exactly or within a specified additive error.

3. Notations

Before starting, we introduce some notations. Let $P \subset \mathbb{R}^d$ be data points set, where d is dimension; n be the cardinality of P; p_i be the i^{th} point in P; $dist(p, q) = \|p-q\|_2$ be the distance from q to p; $Range(p, \epsilon)$ be the ϵ-neighborhood of p, which is defined as below:

$$Range(p, \epsilon) = \{o | \|p, o\|_2 \leq \epsilon, o \in \mathbb{R}^d\} \tag{1}$$

We also define $OutRange(p, \epsilon)$ as a L_∞ norm ball centered by p with radius ϵ as below:

$$OutRange(p, \epsilon) = \{o | \|p, o\|_\infty \leq \epsilon, o \in \mathbb{R}^d\} \tag{2}$$

Then the RNN query is defined as follows:

Definition 1. *Given a distance ϵ, the Range Nearest Neighbor of q is defined as*

$$RNN(q, \epsilon) = \{p | p \in Range(q, \epsilon) \cap P\} \tag{3}$$

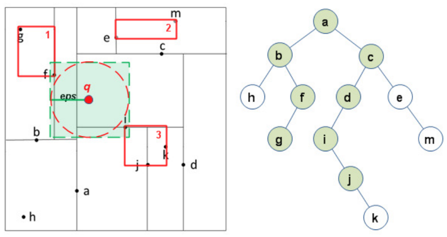
4. The drawbacks of current k-d tree based range search

Fig. 1 shows an example of the subdivision and structure of a k-d tree. A k-d tree for a set of n points uses $O(n)$ storage and can be constructed in $O(n \log(n))$ time. Each point is a node in k-d tree, and there exists a minimum hyper-rectangle, which is called as a cell, that covers the point and all its descendants. For example, as shown in Fig. 1, *cell* 1 is the cell of node f, which is a hyper-rectangle that covers f and g. The cell of node i is *cell* 3 that covers i, j and k.

In the task of performing range query based on k-d tree, we have to query within an axis-parallel rectangle first. For example, to retrieve $RNN(q, \epsilon)$ in Fig. 1 (a), there are two steps as following:

- Recursively visit possible nodes $A = \{a, b, c, d, f, i, g, j\}$, according to splitted dimension, then report the result: $B = OutRange(q, \epsilon) \cap A = \{f, i\}$.
- Check distances from q to all nodes in B, and then report $RNN(q, eps) = \{\phi\}$.

The complexity of the first step, i.e. rectangular range query is $O(n^{1-1/d} + m)$ De Berg et al. (2000) where m is the number of reported points. Obviously, the worst complexity is $O(n)$ if $OutRange(q, \epsilon)$ covers all points. However, we notice that the complexity depends on the total number of visiting nodes and distance computations, and many of them are redundant as shown below:



(a) subdivision (b) structure

Figure 1: The subdivision and structure of a k-d tree. In (a), the dashed red circle is $Range(q, eps)$, the shaded green square is $OutRange(q, eps)$, and each red rectangle represents the cell of a node. In (b) all shaded nodes should be visited.

(1) Redundant visiting nodes: For a query point q, it has to traverse the sub-tree of a node, if $Range(q, \epsilon)$ covers the whole cell of the node. For example, $Range(q, \epsilon)$ covers the cell of the root node, then $RNN(q, \epsilon) = P$, which means traversing the whole tree is inevitable. In fact, it is unnecessary.

(2) Redundant distance computations: Although, k-d tree filters some distance calculations, there are still some redundant distance computations, e.g. $dist(q, f)$ and $dist(i, q)$, as the cells f and q don't intersect with $Range(q, eps)$. Also, there is no need to visit their descendant nodes g and j, respectively. Similarly, it is unnecessary to compute distances from q to all points in those cells that do not intersect with $Range(q, eps)$.

Thus, the main process of querying nearest neighbor in a k-d tree is listed as below [2]:

1. Start from root node, the algorithm recursively moves down the tree, in the same way that it would if the search point were being inserted. Once the algorithm reaches a leaf node, it saves this leaf node as the "current best" point c.
2. Unwind the recursion of the tree, and compares each visiting node. If $dist(q, c) > dist(q, v)$ then $c = v$, where v is the current visiting node.
3. Check whether there could be any points that can become current best on the

[2]https://en.wikipedia.org/wiki/K-d_tree

other side of the splitting plane, by the way of judging whether the splitting hyperplane intersect with $Range(q, \epsilon)$ where $\epsilon = dist(q, c)$. Since the hyperplanes are all axis-aligned, the algorithm simply makes a comparison to check whether the distance between the splitting coordinate of the search point and current node is lesser than the distance from the search point to the current best.

(a) If the hypersphere intersects with the plane, there could be nearer points on the other side of the plane, so the algorithm has to check the other branch to find them, following the same way as the entire search.

(b) Otherwise, ignore the whole branch on the other side of the current node.

In Friedman et al. (1977), Friedman et al. claimed the above algorithm runs in $O(log(n))$ time. However, in high-dimensional data, the algorithm has to visit many more branches than in lower-dimensional spaces. In particular, in the case of the data is high-dimensional and sparse, it runs in $O(n)$.

Take Fig. 2 for example, suppose node f is the current nearest point to q, we can see that $Range(q, dist(q, f))$, the hypersphere of q that has a radius equal to $dist(q, f)$ as shown by the green circle, intersects with the hyperplane of g, while doesn't intersect with that of h. Therefore, the algorithm ignores h, but has to visit g. Similarly, on the other side, the algorithm has to visit node c, d, i, j, and ignores k, e, m.

Actually, there are also many redundant visiting nodes in this algorithm, the reason is that the nearest distance from a cell of current searching node c to query point q is always larger than the distance from q to the splitting coordinate of c. For example, *Range(q,dist(q,f))* doesn't intersect with the cells of i and j, but intersects with the hyperplanes of the two nodes. Therefore, node i and j are unnecessary to visit.

Therefore, our goal is to improve the original k-d tree based *RNN* and *NN* by decreasing the number of visiting nodes and distance computations.

5. The proposed methods

Let ξ be a cell of node s, and t be a point, there are three cases between ξ and $Range(t, \epsilon)$, as follows:

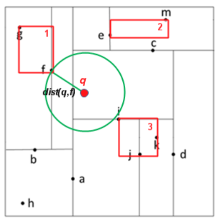


Figure 2: An example of searching nearest point to q. Node f is current best node, $Range(q, dist(q, f))$ intersect with the hyperplanes of i and j, but doesn't intersect with the cells of i and j.

- Case (1) non-intersect: all points within cell ξ are all far from t, there is no need to visit these points.
- Case (2) $Range(t, \epsilon)$ covers (includes) ξ: all points within cell ξ are all neighbors of t. Also, it is unnecessary to visit these points.
- Case (3) intersect: it is necessary to visit its children nodes.

Thus, we can directly filter visiting nodes and distance computations for case (1) and case (2). Therefore, the key is to judge the case between ξ and $Range(t, \epsilon)$, as well as to retrieve all points within ξ directly instead of traversing the subtree rooted at node s.

5.1. Determine relationship between a cell and searching range

If the closest and the farthest distance of ξ from q are known, then t is easy to determine the relationship between a cell and a query point: case (1) holds if the closest distance is larger than ϵ, case (2) holds if the farthest distance is less than ϵ, otherwise case (3) holds.

As cell ξ is a hyper-rectangle, we can determine the closest and the farthest distance as below.

Let $rect \subset \mathbb{R}^d$ be a hyper-rectangle in d dimension, and $lvex$ and $uvex$ be two key vertexes of $rect$, such that $\forall q \in rect\ s.t.\ \forall j\ lvex_j \le q_j \le uvex_j$, where $j = 1, 2, ..., d$. Let $cent$ be the center point of $rect$, i.e. $cent = (lvex + uvex)/2$, and $dvec = uvex - cent = cent - lvex$ be a vector that indicates the distances from $cent$ to each face (hyper-plane) of $rect$. Obviously, $\forall i$ we have $dvec_i \ge 0$.

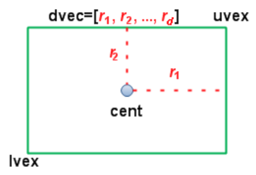


Figure 3: An example of the $lvex$, $uvex$, $cent$ and $dvec$ of a cell.

Given an arbitrary point p, we say p is in the range of $rect$ in the i^{th} dimension if $lvex_i \leq p_i \leq uvex_i$, otherwise it is out of the range of $rect$ in this dimension. Also, we have $|cent_i - p_i| \leq dvec_i$ if p is in the range of $rect$ in the i^{th} dimension, otherwise $|cent_i - p_i| > dvec_i$.

Fig. 3 shows the concepts of $lvex$, $uvex$, $cent$ and $dvec$. For example, let $lvex = [1, -1, 1]'$, $uvex = [3, 7, 5]'$, then $cent = [2, 3, 3]'$, and $dvec = [1, 4, 2]'$.

For point p, we define the closest and the farthest distance of $rect$ from p as below:

Definition 2. *$ldist(rect, p)$ is the closest distance of rect from p, i.e. $ldist(rect, p) = \min\limits_{q}(dist(p, q)), q \in rect$.*

Obviously, $ldist(rect, p) = 0$ if $p \in rect$.

Definition 3. *$udist(rect, p)$ is the farthest distance of rect from p, i.e. $udist(rect, p) = \max\limits_{q}(dist(p, q)), q \in rect$.*

Let x be a real number, $I(x)$ and $G(x)$ are two discriminant functions, as follows:

$$I(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases} \qquad G(x) = \begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$

Theorem 1. *Let $u = cent - p$, the farthest point in rect from p is v, where $v_i = cent_i + dvec_i * G(u_i)$, i.e. $udist(rect, p) = \|v - p\|_2$.*

Proof. First, $\because$ $dvec = uvex - cent = cent - lvex$ and $G(u_i) = \pm 1$, $\therefore$ $\forall i = 1, 2, ..., d$ we have:

$$v_i = \begin{cases} uvex_i, & G(u_i) = 1 \\ lvex_i, & G(u_i) = -1 \end{cases}$$

This means $v \in rect$, and it is just one vertex of $rect$.

Second, $\because v_i - p_i = cent_i + dvec_i * G(u_i) - p_i = (cent_i - p_i) + dvec_i * G(cent_i - p_i)$, $\therefore$ we have:

$$v_i - p_i = \begin{cases} cent_i - p_i + dvec_i, \ cent_i - p_i \geq 0 \\ cent_i - p_i - dvec_i, \ cent_i - p_i < 0 \end{cases} \tag{4}$$

$\forall q \in rect$, we have $lvex_i \leq q_i \leq uvex_i$, $\therefore (cent_i - dvec_i) \leq q_i \leq (cent_i + dvec_i)$, thus we have:

$$cent_i - p_i - dvec_i \leq q_i - p_i \leq cent_i - p_i + dvec_i \tag{5}$$

- Case (1) $cent_i - p_i \geq 0$: $\because dvec_i \geq 0$, $\therefore |cent_i - p_i - dvec_i| \leq |cent_i - p_i| + |dvec_i| = cent_i - p_i + dvec_i$, then $|q_i - p_i| \leq cent_i - p_i + dvec_i$. According to Equation (4) $|v_i - p_i| = cent_i - p_i + dvec_i$, yields:

$$|v_i - p_i| \geq |q_i - p_i|$$

- Case (2) $cent_i - p_i < 0$: $\because dvec_i \geq 0$, $\therefore |cent_i - p_i - dvec_i| \geq |cent_i - p_i + dvec_i|$, thus $|q_i - p_i| \leq |cent_i - p_i - dvec_i|$. According to Equation (4) $|v_i - p_i| = |cent_i - p_i - dvec_i|$, yields:

$$|v_i - p_i| \geq |q_i - p_i|$$

From the above two cases, we have $dist(v, p) = \|v - p\|_2 \geq \|q - p\|_2 = dist(q, p)$, i.e v is the farthest point in $rect$ from p. $\square$

Theorem 1 tells us that for any point p, the farthest point in $rect$ from p is always one of the vertexes of $rect$. Also, it shows the way to find this vertex which is nontrivial to understand mathematically.

However, it is much easy to explain it geometrically as shown in Fig. 4. For point p in this figure, we first go forward from p to $cent$, and then turn the direction according to the rule of $dvec_i * G(cent_i - p_i)$ in i^{th} dimension, which makes it always go along with the direction that is the farthest away from p in each dimension. For example, p is on the left of $cent$ in horizontal axis, then $cent_1 - p_1 > 0$ which makes

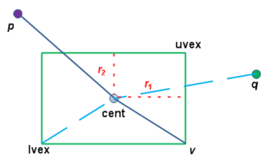


Figure 4: An example of finding the farthest point in $rect$. v and $lvex$ is the farthest point in $rect$ from p and q, respectively.

$G(cent_1 - p_1) = 1$, and yields $v_1 = cent_1 + r_1$. Vertically, p is on the top of $cent$, then $cent_2 - p_2 < 0$ which makes $G(cent_2 - p_2) = -1$. Hence, $v_2 = cent_2 - r_2$, and finally, we determine the farthest vertex is: $v = [cent_1 - r_1, cent_2 - r_2, ...]'$. Similar to point q, $lvex$ is found as the farthest point from q in the same way.

Theorem 2. *Let $u = |cent - p| - dvec$ where $|cent - p|$ is element wise absolute terms, then $ldist(rect, p) = \|v\|_2$, where $v_i = u_i * I(u_i)$, and the closest point in $rect$ from p is w, where $w_i = p_i + v_i * G(cent_i - p_i)$.*

Proof. First, $\forall q \in rect$, we have $(cent_i - dvec_i) \leq q_i \leq (cent_i + dvec_i)$, thus $|q_i - cent_i| \leq dvec_i$, then we have:

$$\begin{aligned} |q_i - p_i| &= |(q_i - cent_i) + (cent_i - p_i)| \\ &\geq |\,|cent_i - p_i| - |q_i - cent_i|\,| \\ &\geq |\,|cent_i - p_i| - dev_i\,| \end{aligned}$$

$\because u_i = |cent_i - p_i| - dvec_i$, then we have:

$$v_i = \begin{cases} u_i, & |cent_i - p_i| - dvec_i \geq 0 \\ 0\ , & |cent_i - p_i| - dvec_i < 0 \end{cases}$$

Therefore, we have $|q_i - p_i| \geq |v_i|$, and then $\|q - p\|_2 \geq \|v\|_2$, i.e. $ldist(rect, p) \geq \|v\|_2$.

Second, we are to prove that $w \in rect$, as follows:

- Case (1) $cent_i - p_i \geq 0$ and $|cent_i - p_i| - dvec_i \geq 0$: we have $w_i = p_i + v_i = p_i + cent_i - p_i - dvec_i = cent_i - dvec_i = lvex_i$;

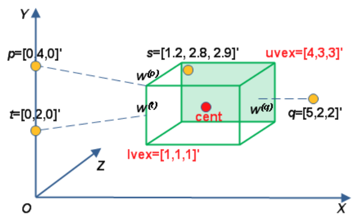


Figure 5: An example of finding the closest point in $rect$. $w^{(p)}$, $w^{(q)}$ and $w^{(t)}$ are the closest points in $rect$ from p, q and t, respectively. While the closest point from s in $rect$ is s itself, because $s \in rect$.

- Case (2) $cent_i - p_i < 0$ and $|cent_i - p_i| - dvec_i \geq 0$: we have $w_i = p_i - v_i = p_i - (-(cent_i - p_i) + dvec_i = cent_i + dvec_i = uvex_i$;
- Case (3) $cent_i - p_i \geq\ 0$ and $|cent_i - p_i| - dvec_i < 0$: we have (a) $w_i = p_i + v_i = p_i + 0 = p_i$, and (b) $lvex_i \leq p_i \leq uvex_i$ otherwise $|cent_i - p_i| > dvec_i$ which conflicts with $|cent_i - p_i| - dvec_i < 0$;
- Case (4) $cent_i - p_i <\ 0$ and $|cent_i - p_i| - dvec_i < 0$: we have (a) $w_i = p_i - v_i = p_i - 0 = p_i$, and (b) $lvex_i \leq p_i \leq uvex_i$ otherwise $|cent_i - p_i| > dvec_i$ which conflicts with $|cent_i - p_i| - dvec_i < 0$.

From Case (1) - Case (4), we conclude that $w \in rect$, and $dist(w, p) = \sqrt{(w_i - p_i)^2} = \sqrt{v_i^2} = \|v\|_2$.

Therefore, $ldist(rect, p) = \|v\|_2$ and w is the closest point from p in $rect$. $\square$

Theorem 2 tells a fact that for any point p, the closest point in $rect$ from p is always on the boundary of $rect$, and it presents the way to find this point which is also nontrivial to understand mathematically.

It is also much easy to geometrically explain as shown in Fig. 5, $lvex = [1, 1, 1]'$ and $uvex = [4, 3, 3]'$ are the two key vertexes of the rectangle, and then $cent = [2.5, 2, 2]'$, $dvec = [1.5, 1, 1]'$.

For point $p = [0, 4, 4]'$, it is out of the range of $rect$ in each dimension. In the first dimension, $\because p_1 < cent_1 \therefore I(u_1) = 1$ which means p is out of the range of $rect$ in the first dimension, and $G(cent_1 - p_1) = 1$ which means p is on the left of $cent$, ($G(cent_i - p_i) = -1$ means p is on the right of $cent$ in i^{th} dimension), then

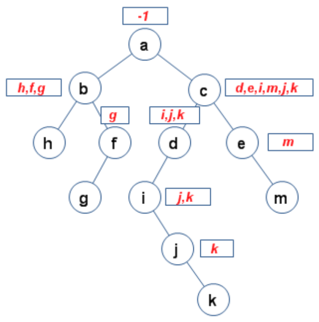


Figure 6: An example of revised k-d tree. Each node holds its all descendant points' indices.

we have $w_1^{(p)} = cent_1 - dvec_1 = 1$ according to Theorem 2. Similarly, $w_2^{(p)} = cent_2 + dvec_2 = 3$ and $w_3^{(p)} = cent_3 - dvec_3 = 1$, thus $w^{(p)} = [1, 3, 1]'$ is a vertex of $rect$.

For point q, $w_1^{(q)} = 4$ is determined by the same way as finding $w_1^{(p)}$ because q_1 is out of the range of $rect$. However, in the second dimension, $\because lvex_2 < q_2 < uvex_2$, then we have $w_2^{(q)} = q_2 = 2$ directly, and so does $w_3^{(q)} = q_3 = 2$. Thus, we find that $w^{(q)} = [4, 2, 2]'$ which is on a face of $rect$. Similarly, $w^{(t)} = [1, 2, 1]'$ which is on an edge of $rect$, while s is totally inside $rect$, thus the closest point in $rect$ from s is itself.

5.2. Retrieve all descendant nodes directly

We modify the original k-d tree by simply adding an additional array to save indices of all descendants for each node, as shown in Fig. 6. Each rectangle with a red label represents an additional array for a node besides it. For the root node a, all points except a itself are its own descendants, then we just save '-1'; for a non-leaf node, e.g. node b, its descendants are h, f, g, they are all saved in its additional array; for leaf node, such as k, nothing is saved.

Given an arbitrary node p and a query point q, if $Range(q, \epsilon)$ covers the whole cell of p, we can return all descendants of p directly in $O(1)$ time, instead of traversing its sub-tree which runs in $O(m)$ time, where m is the total number of descendants of p. Thus, m times of visiting nodes are saved.

Theorem 3. *Suppose n is sufficient large, the space complexity of the additional cost* *in revised k-d tree is $O(\alpha\, n\log(n))$, where $\alpha \in (\frac{\lfloor \log(n)\rfloor - 3}{\log(n)}, \frac{H}{\log(n)})$, H is the depth of the revised k-d tree.*

Proof. Let the root node be level 1, and its children be level 2 etc, then the revised k-d tree has H levels. Let n_i be the total number of nodes in i^{th} level. Obviously, $1 \le n_i \le 2^{i-1}$, $n = \sum_{i=1}^{H} n_i$, and $n - \sum_{j=1}^{i} n_j$ points saved in the i^{th} level.

(1) Obviously, the additional space cost $T(n)$ is:

$$T(n) = 1 + \sum_{i=2}^{H}(n - \sum_{j=1}^{i} n_j) < nH$$

Let $T(n) = \alpha n\log(n)$, and then $\alpha < \frac{H}{\log(n)}$.

(2) $\because$ k-d tree is binary, $\therefore$ $H \ge \lfloor \log(n)\rfloor + 1$, then:

$$\begin{aligned} T(n) &= 1 + \sum_{i=2}^{H}(n - \sum_{j=1}^{i} n_j) \\ &\ge 1 + \sum_{i=2}^{\lfloor \log(n)\rfloor}(n - \sum_{j=1}^{i} 2^{j-1}) \end{aligned}$$

While in full binary tree, $H = \lfloor \log(n)\rfloor + 1$, the first $\lfloor \log(n)\rfloor$ levels are all full, then $n_i = 2^{i-1}$ $s.t.$ $i = 1, 2, ..., \lfloor \log(n)\rfloor$. At the last level, the additional space cost is zero, because all nodes in this level are leaves. Then we have $2^{\lfloor \log(n)\rfloor} < n$ and the total space cost yields:

$$\begin{aligned} T(n) &= 1 + \sum_{i=2}^{\lfloor \log(n)\rfloor}(n - \sum_{j=1}^{i} 2^{j-1}) \\ &= n(\lfloor \log(n)\rfloor - 1) - \sum_{i=1}^{\lfloor \log(n)\rfloor}\sum_{j=1}^{i} 2^{j-1} \\ &= n(\lfloor \log(n)\rfloor - 1) - \sum_{i=1}^{\lfloor \log(n)\rfloor}(2^i - 1) \\ &= (n+1)(\lfloor \log(n)\rfloor - 1) - 2^{\lfloor \log(n)\rfloor + 1} + 3 \\ &\ge (n+1)(\lfloor \log(n)\rfloor - 1) - 2n + 3 \\ &> (n+1)(\lfloor \log(n)\rfloor - 3) \end{aligned}$$

Let $T(n) = \alpha n\log(n)$, then we have:

$$\alpha > \frac{(n+1)(\lfloor \log(n) \rfloor - 3)}{n \log(n)} > \frac{\lfloor \log(n) \rfloor - 3}{\log(n)}$$

□

The theorem above tells that the additional space cost is minimized if k-d tree is full. A balanced k-d tree is very close to full binary tree, because the first $H-2$ levels of any balanced tree must be full. However, in most cases k-d tree is not balanced and full. Fortunately, in most cases $log(n) < H << n$ and Brown Brown (2015) invented an algorithm to build a balanced k-d tree in $O(kn \log(n))$ time. Thus, the average cost of building such a revised k-d tree is about $O(\alpha n \log(n))$ which is acceptable.

5.3. Range Query algorithm

Algorithm 1, which is also named as FSR, presents the detail of retrieving neighbors for a query point q within $Range(q, \epsilon)$. In each subroutine, we filter some nodes according to Theorem 1 and 2.

Simply, the complexity is $O(n - \Psi_1 - \Psi_2)$, where Ψ_1 is the total number of filtered nodes whose cells are outside $Range(q, \epsilon)$, while Ψ_2 is the total number of filtered nodes whose cells are inside $Range(q, \epsilon)$. Both of Ψ_1 and Ψ_2 depend on dimension, data distribution and the size of ϵ.

In fact, any recursion path will stop at some nodes, called as *stop nodes*, whose cells are either outside or inside $Range(q, \epsilon)$. While their ancestral cells all intersect with $Range(q, \epsilon)$, which implies that these stop nodes as well as their parents should distribute around nearby the border of $Range(q, \epsilon)$. Fig. 7 shows an example, the yellow point in the center is query point, q, with $\epsilon = 50,000$. Other colored points are all visited points, where green points are stop nodes outside $Range(q, \epsilon)$, and blue points are stop nodes inside $Range(q, \epsilon)$. Black points are all filtered. We can clearly see that most visiting nodes and stop nodes distribute around the border of $Range(q, \epsilon)$. In section 6.5, more experiments about the distribution of visiting and stop nodes on different data sets will be shown. Hence, the larger surface area of $Range(q, \epsilon)$, the more points will distribute around the border region, and the more points should be visited.

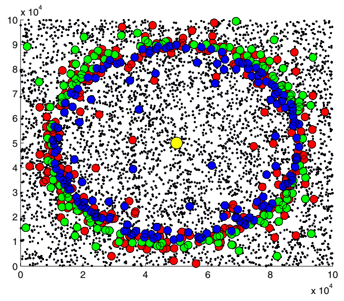


Figure 7: A distribution of visiting and stop points in 2 dimension. The yellow point in the center is query point, q, with $\epsilon = 50,000$. Other colored points are all visited points, where green points are stop nodes outside $Range(q, \epsilon)$, and blue points are stop nodes inside $Range(q, \epsilon)$. Black points are all filtered.

Suppose $\epsilon \leq \min(dvec)$ and $Range(q, \epsilon)$ is totally inside the cell of root node, i.e. inside the whole space of data set P. Let Y be the total number of visiting points being distributed around the surface region, and S be the surface area of $Range(q, \epsilon)$. Because $S \propto \epsilon^{d-1}$, then $Y \propto \epsilon^{d-1}$. Therefore, we can roughly determine Algorithm 1 averagely runs in about $O(\min(\beta\epsilon^{d-1}\log(n), n))$ time, where β is a coefficient of data distribution, and then in low dimension it runs in $O(\log(n))$ time because $n >> \epsilon^{d-1}$.

Comprehensively, we have:

- The best complexity is $O(1)$, if $Range(p, \epsilon)$ covers the whole cell of root node or non-intersects with it, regardless of dimension and data distribution.
- The complexity is $O(\log(n))$, if $Range(p, \epsilon)$ is small that intersects few cells or covers few nodes.
- The average complexity is $O(\min(\beta\epsilon^{d-1}\log(n), n))$.
- In low dimension, the complexity is about $O(\log(n))$.

5.4. Nearest Neighbor Query Algorithm

Algorithm 2, named as $FNNS$, shows the detail of retrieving the nearest neighbor for a query point q. In each subroutine, we also use Theorem 2 to filter redundant nodes.

Algorithm 1 Fast Range Search: FRS

```
1: Input: data $P$, revised $k$-d tree $kdt$, query point $q$, scanning radius $\epsilon$, current node $c$
2: Output: $RNN(q, \epsilon)$
3: if $c$ is not leaf then
4:     $fDist$=$udist(rect_c, q)$ according to Theorem 1.
5:     if $fDist \le \epsilon$ then
6:         if $c$ is root node then
7:             $RNN(q, \epsilon) = P$;
8:         else
9:             $RNN(q, \epsilon) = c$ + all descendants of $c$;
10:        end if
11:        Return;
12:    end if
13:    $cDist$=$ldist(rect_c, q)$ according to Theorem 2.
14:    if $cDist \ge \epsilon$ then
15:        $RNN(q, \epsilon) = NULL$; Return;
16:    end if
17: end if
18: if $c \in OutRange(q, \epsilon)$&$dist(q, c) \le \epsilon$ then
19:     add $c$ into $RNN(q, \epsilon)$;
20: end if
21: %Searching left child and right child recursively;
22: if $c$ is not leaf then
23:     $lRNN = FRS(P, kdt, q, eps, c.left)$;
24:     $rRNN = FRS(P, kdt, q, eps, c.right)$;
25:     $RNN(q, \epsilon) = lRNN + rRNN$;
26: end if
```

Table 1: The details of data sets and the revised k-d tree. PCA real dim is the real dimensionality got by PCA; n is the cardinality of each data set, $T(n)$ is the total number of additional points saved in revised k-d tree.

	PAM	*House*	*Rand4*	*Blog*
dim	4	7	4	59
PCA real dim	4	**5**	4	**5**
n	3,850,505	2,049,280	100,000	52,397
H (depth)	21	21	17	16
$T(n)$	72,815,821	37,171,606	1,380,436	660,902
$T(n)/n$	**18.91**	**18.14**	**13.80**	**12.61**
$\frac{\lfloor \log(n) \rfloor - 3}{\log(n)}$	0.823	0.811	0.783	0.765
$\alpha = \frac{T(n)/n}{\log(n)}$	**0.864**	**0.865**	**0.831**	**0.805**
$\frac{H}{\log(n)}$	0.960	1.001	1.024	1.021

As discussed in Section 4, the original NN algorithm runs in $O(log(n))$ time in low dimension, but in high-dimensional sparse space, it is $O(n)$. Although the complexity of $FNNS$ is still at the same order of magnitude as original NN, in fact it performs much better, because $FNNS$ optimizes the strategy for filtering redundant visiting nodes, as shown in line 11-14, which greatly improves the original NN. In Section 6.6, we will presents the improvements of $FNNS$ in various data sets.

5.5. *kNN Algorithm*

Currently, as stated above, as far as we know, buffer k-d tree Gieseke et al. (2014) is the fastest kNN algorithm, which uses buffer and modern many-core devices such as GPU to accelerate in parallel. The k-d tree build in this algorithm is also a little different from the original k-d tree, as shown in Fig.8, there are four parts: (1) a top tree, (2) a leaf structure which contains more than one point, (3) a set of buffers (one buffer per leaf of the top tree), and (4) two input queues. Each leave node contains a set of points, which consists of blocks and stores all rearranged patterns. The blocks are in a one-to-one correspondence with the leaves of the top tree. The buffer component consists of buffers, one buffer for each leaf. These buffers will be used to store query

Algorithm 2 Fast Nearest Neighbor Searching: $FNNS$

```
1: Input: data $P$, revised $k$-d tree $kdt$, query point $q$, current node $c$, searching radius $cur\_radius$
2: Output: the index of the nearest point, $bestNode$, to $q$, minimum distance $min\_dist$
3: global $bestNode$
4: global $min\_dist$
5: if $c$ is root node then
6:     $min\_dist$=$dist(c, q)$
7:     $curDist$=$dist(c, q)$
8:     $bestNode = c$
9: else
10:    $cur\_dist = cur\_radius$
11:    $cDist$=$ldist(rect_c, q)$ according to Theorem 2
12:    if $cDist \geq min\_dist$ then
13:        return ;
14:    end if
15:    if $min\_dist > dist(c, q)$ then
16:        $min\_dist = dist(c, q)$
17:        $bestNode = c$
18:        $cur\_dist = min\_dist$;
19:    end if
20: end if
21: if c.left is not empty then
22:    $FNS(P, kdt, q, c.left, curDist)$
23: end if
24: if c.right is not empty then
25:    $FNS(P, kdt, q, c.right, curDist)$
26: end if
27: if $c$ is root node then
28:    return [$bestNode$, $min\_dist$];
29: end if
```

Algorithm 3 Revised PROCESSALLBUFFERS

```
1:  Ensure: A sequence $i_1, ..., i_N \in \{1, ..., m\}$ of query indices
2:  $I = NULL$
3:  for $j = 1, ..., 2^h$ do do
4:      Remove all query indices $i_1, ..., i_{N(b_j)}$ from buffer $b_j$
5:      for all $i_1, ..., i_{N(b_j)}$ do in parallel  do
6:          if the nearest distance from the $rect$ of current leaf associated with the buffer
            $b_j <$ the $k^{th}$ nearest distance from current query point then
7:              Update nearest neighbors w.r.t. all points in the leaf
8:          end if
9:      end for
10:     $I = I \bigoplus i_1, ..., i_{N(b_j)}$ (concatenate indices)
11: end for
12: Return $I$
```

indices and can accommodate a predefined number $B > 1$ of integers each.

For all query points, it uses FINDLEAFBATCH to find all candidate leaves for each query point in parallel, and then invokes PROCESSALLBUFFERS to use brute force algorithm to find all k-nearest neighbors from candidate leaves for each query point in parallel.

Here, we only apply our first technic in PROCESSALLBUFFERS to filter unnecessary distance computation from those points contained in candidate leaves, i.e, if the nearest distance from the $rect$ of a leaf to the query point q is larger than the current k^{th} nearest distance from q, then skip this leaf. Line 6 in Algorithm 3 presents filtering process.

6. Experiments

In this section, we conduct experiments to evaluate the proposed algorithms, in order to make comparisons with original original k-d tree based algorithm and exhaustive algorithm on different data sets, as well as kNN based on buffer k-d tree. All experiments are conducted on a machine equipped with 3.3GHz CPU and 8 GB memory,

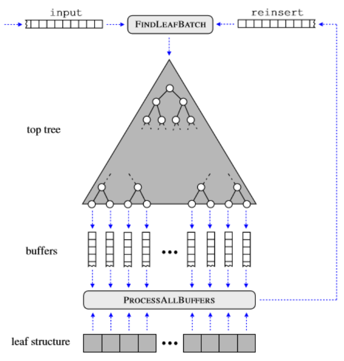
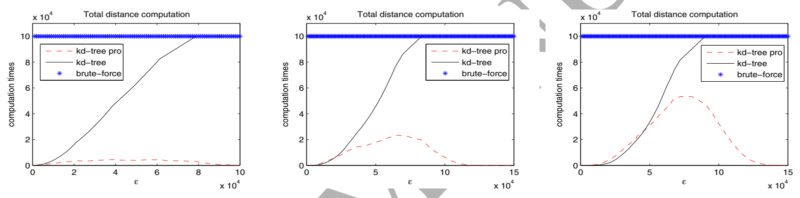


Figure 8: The data structure of buffer k-d tree Gieseke et al. (2014). A buffer k-d tree is composed of (1) a top tree, (2) a leaf structure (a leaf may contains more than one point), (3) a set of buffers (one buffer per leaf of the top tree), and (4) two input queues.



(a) dim=2 (b) dim=3 (c) dim=4

Figure 9: Comparison of total distance computations on synthetic 2-dim, 3-dim and 4-dim *Random Data*, respectively. The total distance computations of our RNN algorithm is #NormalComputation + #ExtraComputation.

and Windows 10 64-bit OS. The RNN and NN algorithms are coded in MATLAB, and compared to original k-d tree based algorithm. The proposed kNN is coded under the framework of buffer k-d tree in *C* .

6.1. Experimental data sets

Several real and synthetic data sets are employed in our experiments, we clear all same data which makes all rows in each data set unique, and all data are normalized such that the domain of each dimension is $[0, 10^5]$. They are as follows:

- Synthetic data: *Rand2* is a 2-dimensional random data, *Rand3* is a 3-dimensional random data, and *Rand4* is a 4-dimensional random data, all of them have the

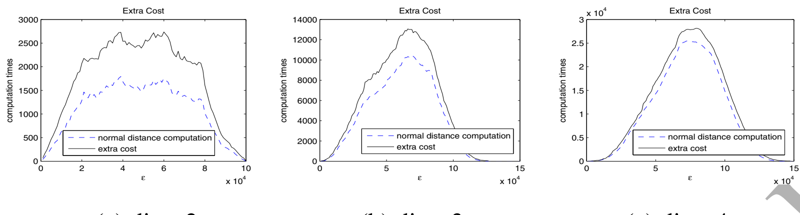

(a) dim=2 (b) dim=3 (c) dim=4

Figure 10: Comparison between normal distance computations and extra computations of our RNN algorithm on synthetic 2-dim, 3-dim and 4-dim *Random Data*, respectively.

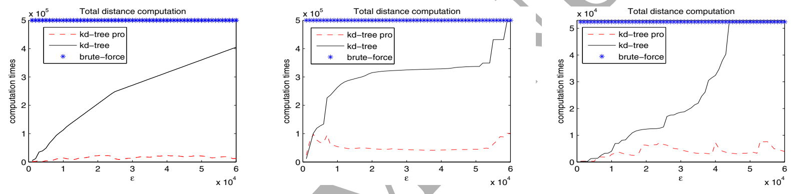

(a) PAM real dim=4 (b) House real dim=5 (c) Blog real dim=5

Figure 11: Comparison of total distance computations on *PAM, Household*, and *BlogFeedback*. The total distance computations of our RNN algorithm is #NormalComputation + #ExtraComputation.

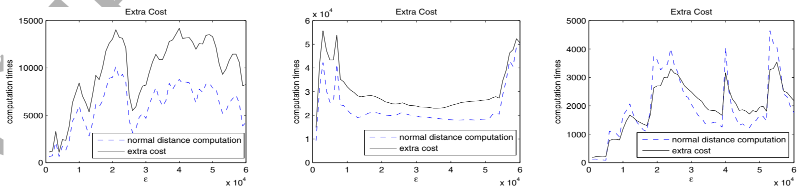
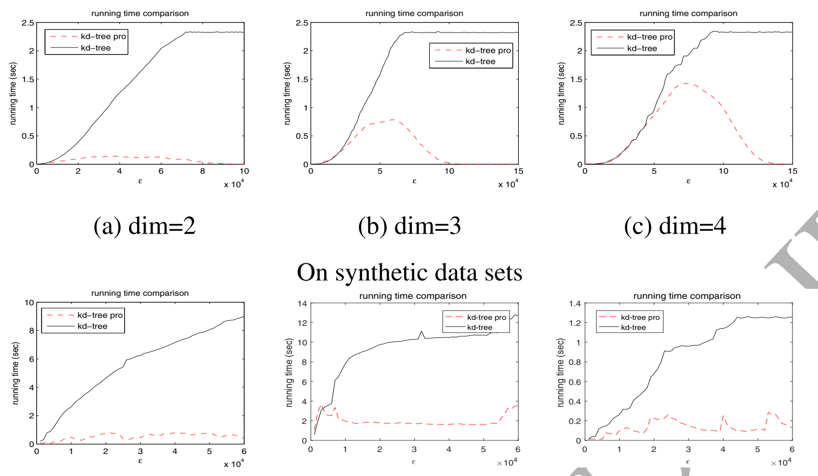

(a) PAM real dim=4 (b) House real dim=5 (c) Blog real dim=5

Figure 12: Comparison between normal distance computations and extra computations of our RNN algorithm on *PAM, Household*, and *BlogFeedback*.



(a) dim=2 (b) dim=3 (c) dim=4

On synthetic data sets

(d) PAM real dim=4 (e) House real dim=5 (f) Blog real dim=5

On realtime data sets

Figure 13: Comparison of running time of RNN on synthetic data and realtime data sets.

same cardinality of 100,000, while *Rand5* is a 5-dimensional random data with cardinality 2,000,000.

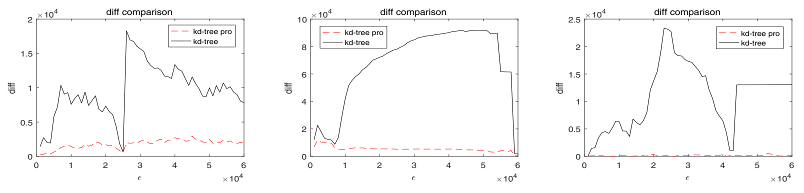
- Realtime application data: *PAMPA (PAM)* [3], *Household (House)*, *Reaction Network (Reaction)*, *BlogFeedback (Blog)* [4], *kdd04* (74 dim) and *Tom Hardware Information (Tom)* (97 dim) all come from UCI archive [5].

The first three rows of Table. 1 show more details. For each data set, we use PCA to find the real dimension: Let $|eig_1| \geq |eig_2| \geq, ..., \geq |eig_d|$ be the ordered eigenvalues of a data set, set $w = 99\%$, the real dimension is $RealDim\ s.t.\ [\sum_{i=1}^{RealDim} eig_i]/\sum_{i=1}^{d} eig_i \geq w$.

[3] *PAMPA* a real data set of 4 dimension with cardinality 3,850,505, obtained by taking the first 4 principle components of a PCA on a database Reiss and Stricker (2012)

[4] $BlogFeedback$ is a 59-dimensional data set with cardinality 52,397 obtained by taking the first 59 numeric attributes and the 60th-280th attributes are omitted, because most values in the 60th-280th attributes are zero.

[5] http://archive.ics.uci.edu/ml/index.php



(a) PAM real dim=4 (b) House real dim=5 (c) Blog real dim=5

Figure 14: The comparison of $diff$ on 3 realtime application data sets.

6.2. *Experiment 1: space cost of the revised k-d tree*

The last 6 rows in Table. 1 show the detail of the revised k-d trees for each data set, because $Rand2$, $Rand5$ and $Rand10$ are similar, we only list the detail of $Rand10$.

We can clearly see that the depths are all far less than cardinalities n, i.e. $H << n$, and $\frac{\lfloor \log(n) \rfloor - 3}{\log(n)} < \alpha < \frac{H}{\log(n)}$ which is consistent with Theorem 3. Furthermore, all α are much closer to $\frac{\lfloor \log(n) \rfloor - 3}{\log(n)}$ than to $\frac{H}{\log(n)}$.

6.3. *Experiment 2: the comparisons of distance computations and running time for RNN*

In this part, we conduct experiments to compare the proposed algorithm with *kd-tree ori* and brute-force by presenting the distance computations and running time. (Here, we only use 500,000 data points of PAM and $House$ in the following experiments.)

Distance computations comparison: According to Algorithm 1, there are two types of distance computation for a query point: (1) ***Extra Distance Computation*** is to compute the farthest and closest distance from a cell, which is the tradeoff of our algorithm. (2) ***Normal Distance Computation*** is to compute a distance from another point. Therefore, the total distance computations of the proposed algorithm is:

#totalComputation=#NormalComputation + #ExtraComputation

We randomly select some points as query points, and increase ϵ from a small value to a large one which makes all points as neighbors. Then, use the mean distance computations to make comparison for each ϵ.

The results on synthetic data set are shown in Fig. 9, we can see that on 2-dim and 3-dim data sets, the superiority of the proposed algorithm is significant. On 4-dim data set, the distance computation times also grows fast with ϵ, but it is still less than that in original algorithm. When the number of total distance computations reaches its peak, it decreases quickly to 0. Fig.10 compares the normal distance computations and extra distance computations on the same 3 data sets, we can see the extra distance computation is closed to the normal distance computations, but both of them increase quickly with the dimension. That's the reason that the proposed algorithm still not suitable for high-dimensional data.

On realtime applications, the proposed algorithm still has significant advantages to the original algorithm in low real dimension, as Fig. 11 (a) (b) and (c) show. We also present the comparison between the normal distance computations and extra distance computations on the same 3 data sets as shown in Fig.12, the results are similar to those in Fig.10.

Running time comparison: Because Matlab is inefficient for loops and recursion which are heavily used in our algorithm, therefore we only compare running time between the proposed method and *kd-tree ori*. Fig. 13 (a), (b) and (c) illustrate the comparison of running time on synthetic 2-dim, 3-dim and 4-dim data set, respectively, and Fig. 13 (d), (e) and (f) present the comparison of running time on *PAM*, *House* and *Blog*, respectively. They are all consistent with distance computations showed above.

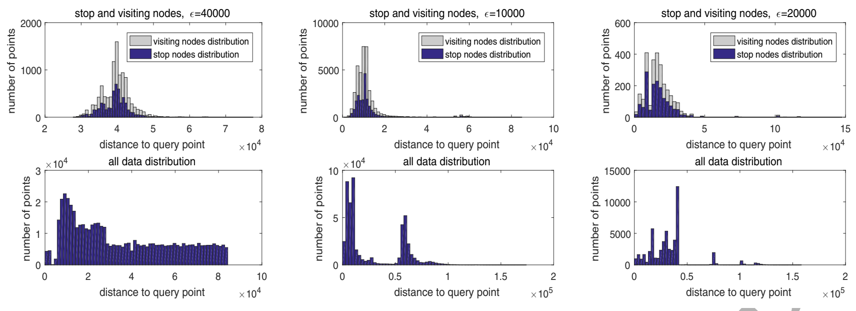
6.4. Experiment 3: the comparisons of visiting nodes

We show experiments in this section to compare the difference between visiting nodes and distance computations, as follows:

$$diff = abs(|visitingNodes| - totalComputation) \tag{6}$$

where $|visitingNodes|$ is the total number of visiting nodes, and $totalComputation$ is the total number of distance computations which including normal distance computations and extra distance computations.

From Fig. 14, we can see that the proposed algorithm is much stable, and combined with Fig. 11, it is inferred that many unnecessary visiting nodes in the proposed



(a) PAM (b) House (c) Blog

Figure 15: The first row shows three distributions of stop nodes, visiting nodes and all points on *PAM*, *Household* and *BlogFeedBack*, respectively. The second row is the distributions of all data points on the same three data sets, respectively. Query points $q =$ the 1000^{th}, 2000^{th} and 4000^{th} point in *PAM*, *Household* and *BlogFeedBack*, respectively.

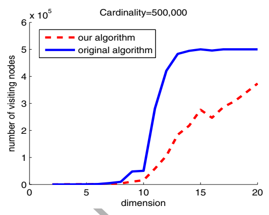


Figure 16: The comparison of visiting nodes between our algorithm $FNNS$ and original NN algorithm on synthetic random data sets $SYNDS$, and for each data set in $SYNDS$, the query points q is the geographical center.

algorithm are saved.

6.5. Experiment 4: the distribution of stop and visiting nodes

In order to examine the distribution of visiting and stop nodes, we conduct series of 450 experiments as Fig. 15 presents. Because short of pages, we only list three distribution examples, each of which has different searching range ϵ. As we can see, most of the distances from these visiting and stop nodes to query point concentrate in the vicinity of ϵ. This means most of these points distribute around the border of $Range(q, \epsilon)$, which is consistent with the analysis in Section 5.3.

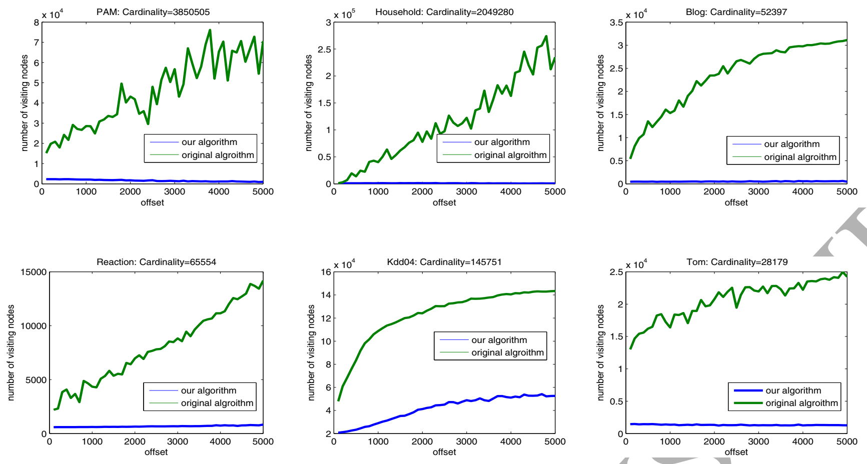


Figure 17: The comparison on mean number of visiting nodes between our algorithm $FNNS$ and original NN algorithm on different realtime data sets.

6.6. *Experiment 5: comparison between $FNNS$ and original NN*

In this section, we only compare visiting nodes of our fast nearest neighbor searching algorithm $FNNS$ with original NN on both synthetic $SYNDS$ and realtime data sets, as shown in Fig. 16 and Fig. 17, respectively.

On $SYNDS$, for each data set $ds \in SYNDS$, we choose the geographical center of ds as query point q. We can see that both the numbers of visiting nodes increase with dimension in these random data sets, but our algorithm $FNNS$ is clearly much better than the original NN. It is also observed that our algorithm still runs in $O(n)$ time in high dimension because of the "cures of dimensionality".

On realtime data sets, we randomly choose 30 points as seeds. For each seed point s, we shift it in each dimension and yield query point q as follows:

$$q_i = s_i + (-1)^{RI} \times offset \tag{7}$$

where $offset = 100 \times step$, RI is a random integer.

In the following experiments, $step$ varies from 1,2..., to 50, which makes $offset$ changes from 100,200,... to 5000. For each $offset$, we calculate mean number of visiting nodes for all query points:

$$|visitingnodes_{mean}|=\frac{1}{30}\times\sum_{i=1}^{30}(number\ of\ visiting\ nodes\ of\ q_i)$$

The results conducted on different data sets are shown in Fig. 17. We can see that $FNNS$ outperforms original NN evidently, on all data sets except kdd04 it runs in $log(n)$ expected time for all different query points regardless of their position. While the complexity of original NN depends on the location of query point q, the farther of q from its nearest point, the higher complexity of the algorithm.

6.7. Experiment 6: comparison between the revised kNN and buffer k-d tree

In this section, we benchmark buffer k-d tree and our kNN, and make comparisons on different data sets. The number of query points are 2000, they are all generated randomly.

The basic configuration of buffer k-d tree is: tree_depth=10, num_threads=1, and num_nXtrain_chunks=10.

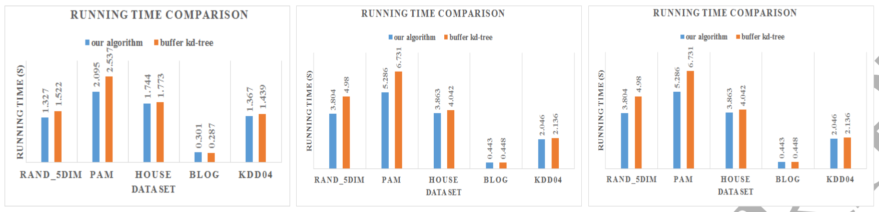
The device of our machine for running OpenCL Stone et al. (2010) (Buffer k-d tree works via OpenCL) is: platform 0- Intel(R) OpenCL; device 0- Intel(R) HD Graphics 4400; Number of compute units:20; Size of memory (GB) 1.45; Maximum memory allocation (GB) 0.3634; OpenCL version 1.2.

Fig.18 shows the running time of both algorithms on 5 data sets, the value of k is 1, 41 and 121, respectively. Table 2 presents the total distance computations on the same data sets with the same value of k.

We can see that on low dimension our kNN has great superiority to buffer k-d tree, e.g., on RAND_5dim and PAM. It is worth noting that improvement of our kNN on running time is not as remarkable as on total distance computations, the reason is the framework of buffer k-d tree relies on OpenCL which has basic overhead. However, with dimension grows the superiority vanishes, e.g., on BLOG (59 dim), both algorithms perform similarly, which means our algorithm no longer has any effect to filter unnecessary distance computation in such high dimension.

6.8. Comprehensive analysis

From the above experiments, we can see that the number of redundant visiting nodes and distance computation are greatly reduced in the proposed algorithms in low



(a) $k = 1$ (b) $k = 41$ (c) $k = 121$

Figure 18: The running time comparisons of kNN for 2000 random query points on different data sets with different k. Our kNN algorithm improves buffer k-d tree greatly in low dimension, while the effectiveness vanish with dimension grows.

Table 2: Comparison of total distance computations of kNN for 2000 random query points on different data sets.

	data set	rand_5dim	pam	house	blog
k=1	our kNN	605,203,501	2,893,962,806	1,740,878,290	23,456,390
	buffer kd	2,003,902,872	3,854,265,000	1,910,181,802	23,922,000
	speedup	**3.31**	**1.33**	**1.10**	**1.02**
k=41	our kNN	995,721,799	3,064,925,450	1,817,370,755	23,477,900
	buffer kd	2,003,906,245	3,854,265,000	1,910,215,318	23,922,000
	speedup	**2.01**	**1.26**	**1.05**	**1.02**
k=121	our kNN	1,037,119,337	3,129,228,967	1,843,986,073	23,470,730
	buffer kd	2,003,903,906	3,854,265,000	1,910,137,114	23,922,000
	speedup	**1.93**	**1.23**	**1.03**	**1.02**

dimension. The tradeoff is that the additional space cost of the revised k-d tree is averagely about $O(\alpha n \log(n))$.

In high dimension, the superiority is not so remarkable. But for RNN, in the case of ϵ is either very small or large, our algorithm is much better than original and brute-force algorithm, regardless of dimension. All experiments are consistent with the complexity analysis in section 5.3.

7. Conclusion

RNN (Range Query), NN (Nearest Neighbor Query) and kNN (k-Nearest Neighbor Query) are fundamental problems in computational geometry, data mining and machine learning, and are widely used in many applications. There exists great number of unnecessary visiting nodes and distance computations in current RNN, NN and kNN algorithm based on k-d tree.

In this paper, we propose new RNN, NN and kNN algorithm, which greatly reduces unnecessary visiting nodes and distance computations, based on two techniques as follows:

- The first one is to check whether the cell of a node is inside or outside $Range(q, \epsilon)$, if it holds then the distance computations from q to all points inside the cell are all filtered.
- The second one is to reduce unnecessary visiting nodes based on a revised k-d tree, whose space cost is about $O(\alpha n \log(n))$. With the help of the revised k-d tree, we can retrieve all descendants of any node in $O(1)$ time, instead of traversing the subtree of the node.

There are two main disadvantages of the proposed algorithms, the first one is the extra distance computations increase rapidly with dimension, which makes it currently not suitable for high-dimensional data if ϵ is not large enough, and the other is the space cost is still relatively high.

Averagely, the proposed RNN runs in $O(\min(\beta\epsilon^{d-1}\log(n), n))$ time. In the case of low dimension or small ϵ, the algorithm performs in $O(\log(n))$ time. Best of all, the complexity is only $O(1)$, if $Range(p, \epsilon)$ covers the whole cell of the data space or non-intersects with it. The worst case is still $O(n)$ which happens in high dimension and $Range(p, \epsilon)$ just intersects most nodes' cell.

Although the complexity of the proposed NN query algorithm $FNNS$ is at the same order of magnitude as original NN algorithm, in fact it significantly improves the later. Our new kNN algorithm based on the framework of buffer k-d tree also greatly improve buffer k-d tree in low dimension.

Our future works are: (1) take the advantages of other techniques such as cover tree, PCA tree and convex hull Barber et al. (1996) etc, to filter more unnecessary distance computations; (2) decrease the space cost.

8. acknowledgements

This work was supported by the National Science Foundation of China (No.61673186); Science and technology project of Quanzhou City 2018Z008; the Open Project Program of the National Laboratory of Pattern Recognition (NLPR) (NO. 201700002); the Natural Science Foundation of Fujian Province (No.2016J01303); Project of science and technology plan of Fujian Province of China (No.2017H01010065).

References

References

Agarwal PK, Aronov B, Har-Peled S, Phillips JM, Yi K, Zhang W. Nearest-neighbor searching under uncertainty ii. ACM Transactions on Algorithms (TALG) 2016;13(1):3.

Andoni A, Indyk P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Communications of the ACM 2008;51(1):117–22.

Barber CB, Dobkin DP, Huhdanpaa H. The quickhull algorithm for convex hulls. ACM Transactions on Mathematical Software (TOMS) 1996;22(4):469–83.

Bawa M, Condie T, Ganesan P. Lsh forest: self-tuning indexes for similarity search. In: Proceedings of the 14th international conference on World Wide Web. ACM; 2005. p. 651–60.

Becker A, Ducas L, Gama N, Laarhoven T. New directions in nearest neighbor searching with applications to lattice sieving. In: Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics; 2016. p. 10–24.

Bentley JL. Multidimensional binary search trees used for associative searching. Communications of the ACM 1975;18(9):509–17.

Bernecker T, Emrich T, Kriegel HP, Mamoulis N, Renz M, Züfle A. A novel probabilistic pruning approach to speed up similarity queries in uncertain databases. In: IEEE International Conference on Data Engineering. 2011. p. 339–50.

Beygelzimer A, Kakade S, Langford J. Cover trees for nearest neighbor. In: Proceedings of the 23rd international conference on Machine learning. ACM; 2006. p. 97–104.

Brown RA. Building kd tree in o (knlog n) time. Journal of Computer Graphics Techniques 2015;4(1):50–68.

Cai J, Wang Y, Liu Y, Luo JZ, Wei W, Xu X. Enhancing network capacity by weakening community structure in scale-free network. Future Generation Computer Systems DOI: 101016/jfuture201708014 2017;.

Cao Y, Zhou Z, Sun X, Gao C. Coverless information hiding based on the molecular structure images of material. Computers Materials & Continua 2015;54(2):197–207.

Chen Y, Singh JP, Zhou L, Bouguila N. Frs: Fast range search by pruning unnecessary distance computations based on k-d tree. In: IEEE International Conference on Data Mining Workshops. 2017. p. 1160–5.

Chen Y, Tang S, Bouguila N, Wang C, Du J, Li HL. A fast clustering algorithm based on pruning unnecessary distance computations in dbscan for high-dimensional data. Pattern Recognition 2018 (in press);doi:https://doi.org/10.1016/j.patcog.2018.05.030.

Chen Y, Tang S, Zhou L, Wang C, Du J, Wang T, Pei S. Decentralized clustering by finding loose and distributed density cores. Information Sciences 2018;433-434:649–60.

Cheng R, Chen J, Mokbel M, Chow CY. Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data. In: IEEE International Conference on Data Engineering. 2008. p. 973–82.

Cheng R, Kalashnikov DV, Prabhakar S. Querying imprecise data in moving object environments. IEEE Transactions on Knowledge and Data Engineering 2004;16(9):1112–27.

De Berg M, Van Kreveld M, Overmars M, Schwarzkopf OC. Computational geometry. In: Computational geometry. Springer; 2000. p. 1–17.

Fan L, Lei X, Yang N, Duong TQ, Karagiannidis GK. Secrecy cooperative networks with outdated relay selection over correlated fading channels. IEEE Transactions on Vehicular Technology 2017;66(8):7599–603.

Friedman JH, Bentley JL, Finkel RA. An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software (TOMS) 1977;3(3):209–26.

Gao C, Lv S, Wei Y, Wang Z, Zheli Liu XC. M-sse: An effective searchable symmetric encryption with enhanced security for mobile devices. IEEE ACCESS, 2018;doi:10.1109/ACCESS.2018.2852329.

Gieseke F, Heinermann J, Oancea CE, Igel C. Buffer kd trees: Processing massive nearest neighbor queries on gpus. In: ICML. 2014. p. 172–80.

He P, Deng Z, Gao C, Wang X, Li J. Model approach to grammatical evolution: deep-structured analyzing of model and representation. Soft Computing 2017;21(18):5413–23.

Hjaltason GR, Samet H. Distance browsing in spatial databases. ACM Transactions on Database Systems (TODS) 1999;24(2):265–318.

Hu H, Lee DL. Range nearest-neighbor query. IEEE Transactions on Knowledge and Data Engineering 2006;18(1):78–91.

Huang Y, Li W, Liang Z, Xue Y, Wang X. Efficient business process consolidation: combining topic features with structure matching. Soft Computing 2016;22(2):645–57.

Jegou H, Douze M, Schmid C. Product quantization for nearest neighbor search. IEEE Transactions on Pattern Analysis And Machine Intelligence 2011;33(1):117–28.

Leibe B, Mikolajczyk K, Schiele B. Efficient clustering and matching for object class recognition. In: BMVC. 2006. p. 789–98.

Li J, Liu Z, Chen X, Xhafa F, Tan X, Wong DS. L-encdb: A lightweight framework for privacy-preserving data queries in cloud computing. Knowledge-Based Systems 2015;79:18–26.

Li Y, Wang G, Nie L, Wang Q, Tan W. Distance metric optimization driven convolutional neural network for age invariant face recognition. Pattern Recognition 2018;75:51–62.

Liu T, Moore AW, Gray AG, Yang K. An investigation of practical approximate nearest neighbor algorithms. In: NIPS. volume 12; 2004. p. 2004.

Marius Muja DGL. Scalable nearest neighbor algorithms for high dimensional data. IEEE Transactions on Pattern Analysis And Machine Intelligence 2014;36(11):2227–40.

Moore AW. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In: Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence. Morgan Kaufmann Publishers Inc.; 2000. p. 397–405.

Muja M, Lowe DG. Fast approximate nearest neighbors with automatic algorithm configuration. VISAPP (1) 2009;2(331-340):2.

Muja M, Lowe DG. Scalable nearest neighbor algorithms for high dimensional data. IEEE Transactions on Pattern Analysis and Machine Intelligence 2014;36(11):2227–40.

Philbin J, Chum O, Isard M, Sivic J, Zisserman A. Object retrieval with large vocabularies and fast spatial matching. In: Computer Vision and Pattern Recognition, IEEE Conference on. IEEE; 2007. p. 1–8.

Reiss A, Stricker D. Introducing a new benchmarked dataset for activity monitoring. In: 2012 16th International Symposium on Wearable Computers. IEEE; 2012. p. 108–9.

Rodriguez A, Laio A. Clustering by fast search and find of density peaks. Science 2014;344(6191):1492–6.

Schindler G, Brown M, Szeliski R. City-scale location recognition. In: IEEE Conference on Computer Vision and Pattern Recognition. 2007. p. 1–7.

Silpa-Anan C, Hartley R. Optimised kd-trees for fast image descriptor matching. In: IEEE Conference on Computer Vision and Pattern Recognition. 2008. p. 1–8.

Stone JE, Gohara D, Shi G. Opencl: A parallel programming standard for heterogeneous computing systems. Computing in science &amp; engineering 2010;12(3):66–73.

Tao Y, Papadias D, Shen Q. Continuous nearest neighbor search. In: Proceedings of the 28th international conference on Very Large Data Bases. VLDB Endowment; 2002. p. 287–98.

Wang H, Wang W, Cui Z, Zhou X, Zhao J, Li Y. A new dynamic firefly algorithm for demand estimation of water resources. Information Sciences 2018;438:95–106.

Wang J, Wang N, Jia Y, Li J, Zeng G, Zha H, Hua XS. Trinary-projection trees for approximate nearest neighbor search. IEEE Transactions on Pattern Analysis And Machine Intelligence 2014;36(2):388–403.

Wu Z, Tian L, Li P, Wu T, Jiang M, Wu C. Generating stable biometric keys for flexible cloud computing authentication using finger vein. Information Sciences 2016;:431–47.

Yang L, Han Z, Huang Z, Ma J. A remotely keyed file encryption scheme under mobile cloud computing. Journal of Network & Computer Applications 2018;(106):90–9.

Yianilos PN. Data structures and algorithms for nearest neighbor search in general metric spaces. In: SODA. volume 93; 1993. p. 311–21.

Zhang S, Yang Z, Xing X, Gao Y, Xie D, Wong HS. Generalized pair-counting similarity measures for clustering and cluster ensembles. IEEE Access 2017;(5):16904–18.

Zhang Y, Wang N, Zhang S, Li J, Gao X. Fast face sketch synthesis via kd-tree search. In: European Conference on Computer Vision. Springer; 2016. p. 64–77.

Zhou Z, Dong M, Ota K, Wang G, Yang LT. Energy-efficient resource allocation for d2d communications underlaying cloud-ran-based lte-a networks. IEEE Internet of Things Journal 2016;3(3):428–38.

Zhu Y, Zhang Y, Li X, Hongyang Yan JL. Improved collusion-resisting secure nearest neighbor query over encrypted data in cloud. Concurrency and Computation: Practice and Experience, 2018;doi:10.1002/cpe.4681.