A Replacement for Voronoi Diagrams of Near Linear Size

Raj Varma

May 7, 2009

Abstract: A compressed quad tree based replacement for $\epsilon$ approximate voronoi diagrams with near linear complexity using hierarchial clustering and prioritized point location among balls and with applications for improved approximate nearest neighbour search using point location among equal balls, fat triangulations of proximity diagrams in two and higher dimensions and for fast approximate proximity search.
A Voronoi diagram is a decomposition of metric space determined by distances to a specified discrete set of objects in the space. It is a fundamental structure in geometric computing. It is induced by the distance function of a point-set \( P \subseteq \mathbb{R}^d \). The NN (nearest neighbor) distance function \( V_P(q) \) returns the distance between \( q \) and its nearest point in \( P \). The Voronoi diagram is the decomposition of \( \mathbb{R}^d \) into cells, so that inside each cell \( c \) there is a single point \( p_c \in P \) such that for any point \( q \in c \) the nearest-neighbor of \( q \) in \( P \) is \( p_c \). It has a worst case complexity of \( \Theta(n^{\lceil\frac{d}{2}\rceil}) \).

1 Related previous work

1.1 Approximate Nearest Neighbors: PIOTR INDYK and RAJEEV MOTWANI

**Problem:** Approximate Nearest Neighbour: Given a point set \( P \) in some metric space \( X \), find a point \( p \in P \) that is an \( \epsilon \)-approximate nearest neighbour of the query \( q \) in that for all \( p' \in P \), \( d(p, q) \leq (1 + \epsilon)d(p', q) \)

**Solution:** They do this by reducing the \( \epsilon - \text{NNS} \) to the following problem of point location in equal balls (PLEB).

**Point Location in Equal Balls (PLEB):** Given \( n \) radius-\( r \) balls centered at \( G = (c_1, ..., c_n) \) in \( M = (X, d) \), PLEB is a data structure which for any query point \( q \in X \), if there exists \( c_i \in G \) such that \( q \in B(c_i, r) \) then return \( c_i \), else return NO.

They do the reduction from \( \epsilon - \text{NNS} \) to PLEB a data structure called a ring-cover tree. Using this structure we can either find a ring-separator or a cover for any point set \( P \). It also allows us to decompose \( P \) into smaller sets \( S_1, ..., S_t \) such that for all \( i \), \( |S_i| \leq |\ell c|P| \) for some \( c < 1 \), and \( \sum_i |S_i| \leq b|P| \) for \( b < 1 + \frac{1}{\log^2 n} \).

Also while searching in \( P \) it restricts the search to one of the sets \( S_t \). Also since they reduce an optimization problem to its decision version it can be viewed as a variant of parametric search. Then they give two solutions to the point location problem. In the first method decompose each ball into a bounded number of cells and store them in a dictionary. By using this technique we have a query time of \( O(d) \), and preprocessing time is exponential in \( d \). In the second method, they use the technique called locality-sensitive hashing. In this they use the hash functions such that the probability of collision is much higher for objects that are close to each other than for those that are far apart. The existence of such functions for any domain implies the existence of fast \( \epsilon - \text{NNS} \) algorithms for that domain, with preprocessing cost only linear in \( d \) and sublinear in \( n \) (for \( \epsilon > 1 \)).

**Results:** The bounds of their algorithm are

- Preprocessing Time: polynomial in \( n \) and \( d \), and a truly sublinear query time (for \( \epsilon > 1 \))
- Query time: polynomial in \( \log n \) and \( d \)

1.2 Approximate Nearest Neighbors in High Dimensional Spaces: Kushilevitz, Ostrovsky and Rabani

**Problem:** Let \( V \) be a vector space of dimension \( d \), and let \( ||.|\| \) be Minkowsky distance function for \( V \). Given a database consisting of \( n \) vectors in \( V \), a slackness parameter \( \epsilon > 0 \), and a query vector \( q \), a \((1+\epsilon)\)-approximate nearest neighbor of \( q \) is a database vector \( a \) such that for any other database vector \( b \), \( ||q-a|| \leq (1+\epsilon)||q-b|| \). Given such a database and \( \epsilon > 0 \), design a data structure \( S \) and a \((1+\epsilon)\)-approximate nearest neighbor search algorithm (using \( S \)).

**Solution:** They assume that all the vectors are inside a hypercube. For Euclidean spaces, they reduce the problem to a search in several hypercubes by using projections onto random lines through the origin. They
also take into fact that in expectation the projection of any vector preserves its length. This property underlies methods of distance preserving embeddings into low-dimensional spaces, like the Johnson-Lindenstrauss Lemma. These techniques when applied directly, guarantee correct answers to most queries, but not to all possible queries. So in order to overcome this difficulty, they make use of the theory of Vapnik-Chervonenkis (or VC-) dimension to show the existence of a small finite sample of lines that closely imitate the entire distribution for any vector. A clustering argument, and another sampling argument allows us to use this sample to reduce our problem to the cube.

**Results:** The bounds of their algorithm are

- Data structures are of size \((dn)^O(1)\)
- Query time: For \(d\)-dimensional Euclidean spaces, our search algorithm runs in time \(O(d^2 \text{poly log}(dn))\).

## 2 A Replacement for Voronoi Diagrams of Near Linear Size with applications to NN search

The above mentioned algorithms for NN search have data-structures of polynomial size and query time with polynomial dependency on the dimension. Also they do not have a space decomposition associated with them. So we need an algorithm that overcomes these problems.

**Problem:** For a given point set \(P\), find a decomposition of \(\mathbb{R}^d\) into cells \(c\) such that for any point in \(c\) the point \(p_c \in P\) is an approximate NN in \(P\). Here each cell \(c\) is associated with a point \(p_c \in P\). We would also like to minimize the number of cells and the complexity of each cell in the decomposition.

**Algorithm:** In this algorithm first we hierarchically cluster the input points into a tree. Then we use this clustering to reduce the NN search problem to a sequence of point-location queries among equal balls and build a datastructure using this clustering to answer the NN queries. Then we use this construction to answer NN queries by doing point-location among balls (PLEB). So the steps of the algorithm can be outlined as

**Steps:**
- We do a hierarchial clustering of the points and then using this clustering we reduce the problem of NN search to point location among equal balls.
- We build an NN Tree that which provides us with a search structure.
- Then we extract the relevant set of balls using the NN tree.
- From this set of prioritized balls we generate an approximate voronoi diagram

### 2.1 Reduction from NN search to PLEB queries

Given a point set \(P\) with \(n\) points and a parameter \(r\), we define the following terms.

- \(UB_P(r)\): It is the union of equal balls of radius \(r\) centered at the points of \(P\) \((\bigcup_{p \in P} B(p; r))\).

- **\(\gamma\)-approximate nearest neighbour:** For a query point \(q \in \mathbb{R}^d\), let \(NN_P(q)\) be the point of \(P\) which is closest to \(q\) and the closest distance is given by \(d_P(q) = ||qNN_P(q)||\). For a parameter \(\gamma > 0\), \(y \in P\) is an \(\gamma\)-approximate nearest neighbor of \(q\) if \(d_P(q) \leq ||qy|| \leq (1 + \gamma)d_P(q)\).

- **PLEB:** A \(PLEB(P; r)\) (point-location in equal balls) is a data-structure so that given a query point \(q \in \mathbb{R}^d\), it decides whether \(q \in UB_P(r)\) and returns a point \(u\) of \(P\) such that \(q \in B(u; r)\) if \(q \in UB_P(r)\).
Let $r_{median}$ be the radius such that $UB_{med} = UB_P(r_{median})$ has exactly $\frac{n}{2}$ connected components and let $P_{med} = PLEB(P,r_{median})$ be the PLEB. Let $q$ be the given query point.

**case 1:** Let $q \in UB_{med}$. Since $UB_{med}$ has $n/2$ connected components we can search for the nearest neighbour recursively in the connected component of $UB_{med}$. Also since it has $n/2$ connected components, the search continues recursively into a subset of $P$ of cardinality at most $1 + n/2$.

**case 2:** If $d_P(q) \geq r_{top} = (4nr_{median} \log n)/\epsilon$. This means that the query point $q$ is far away from the points of $P$. So now from each connected component of $UB_{med}$, we extract one point of $P$ that lies inside it and since we have $n/2$ connected components we get a subset $P^+ \subset P$ that contains $n/2$ points. Now we continue the recursive search into $P^+$ and this introduces accumulative error into the search results, which is smaller than $1 + \epsilon/3$.

**case 3:** If $r_{median} \leq d_P(q) \leq r_{top}$. Observe, that $r_{top}/r_{median} = O((n \log n)/\epsilon)$. Divide the interval $[r_{median}, r_{top}]$ by $M = \log_{1+\epsilon/3} r_{top}/r_{median} = O((1/\epsilon) \log(n/\epsilon))$ PLEBs: $\rho_1, ..., \rho_M$, where

$$P_i = PLEB(P, r_{median}(1 + \epsilon/3)^i),$$

for $i = 1, ..., M$.

Now we do a binary search on $\rho_1, ..., \rho_M$ and find the one can find $\epsilon$-NN to $q$ in $P$ such that $q \notin UB(P, r_{median}(1 + \epsilon/3)^i)$ and $q \in UB(P, r_{median}(1 + \epsilon/3)^{i+1})$, using $O(\log M)$ PLEB queries. So, by the above three cases we have that we can find an $\epsilon$-NN to a point by performing $O(\log(n/\epsilon))$ PLEB queries.

### 2.2 Hierarchial Clustering

Now as the first step of algorithm we do the Hierarchial Clustering of points. For this we need to compute the approximate Minimum Spanning Tree of the points.

#### 2.2.1 $\lambda$-MST of $P$

For the construction of point set at each stage we split the point-set into two subsets and recursively compute a nd-MST for each point-set. For this we try to separate the original set into two subsets such that the two subsets are furthest away from each other. Then the two nd-MST of the two subsets are merged by adding an edge, and assigning it appropriate weight.

Let $R = R(P)$ be the minimum axis parallel box containing $P$, and let $l = l(P) = \sum_{i=1} l_i(R)$, where $l_i(R)$ is the projection of $R$ to the $i$-th dimension. We can find an axis parallel strip $H$ of width $\geq l/((n1)d)$, such that there is at least one point of $P$ on each of its sides, and there is no points of $P$ inside $H$. To find this strip, project the point-set into the $i$-th dimension, and find the longest interval between two consecutive points. Repeat this process for $i = 1, ..., d$, and use the longest interval encountered.

Let $P^+, P^-$ is the splitting of $P$ into two sets by $H$. Now apply the MST algorithm recursively on these two point sets which results in the construction of two trees $T^+, T^-$. Now merge the two trees by picking up any two points $p \in P^+, q \in P^-$, such that $T = T^+ \cup T^- \cup \{e\}$ and $e = pq$. Now set $len(e) = l/((n1)d)$. The construction can be performed in $O(nd\log^2 n)$

#### 2.2.2 $\lambda$-stretch hierarchial clustering of $P$

$F \rightarrow nd$-stretch hierarchial clustering of $P$ : A directed tree storing the points of $P$ in its nodes, so that for any point $p \in P$ there is as an associated value $rloss(p)$. This value is also associated with the out-edge emanating from $p$ in $F$. 

4
First we compute the nd-MST of \( P \) using the above algorithm and sort the edges of \( T \) by their \( \text{len} \) value. Now starting from a forest that corresponds to all the points of \( P \), compute \( F \) by performing a sequence of merges.

In the \( i \)-th iteration, we take the \( i \)-th edge \( x_iy_i \) and merge the two trees \( T_i^x, T_i^y \) that contains \( x_i \) and \( y_i \) by creating a new edge \( e_i \) connecting the two roots of \( T_i^x \) and \( T_i^y \). This can be done by hanging \( T_i^x \) on \( T_i^y \).

Then, we set \( r_{\text{loss}}(e_i) = \text{len}(x_iy_i)/2 \), where \( \text{len}(x_iy_i) \) is the weight associated with the edge \( x_iy_i \) in MST. The construction can be performed in \( O(nd \log^2 n) \).

**Interval PLEB(IPLEB)**: It is a data structure which is used to answer PLEB queries over a range of distances.

**Lemma**: For \( 1/2 > \gamma > 0 \), given an interval \([a, b]\), and a point-set \( P \), we can construct \( O((\log b/a)/\gamma) \) PLEBs, so that given a query-point \( q \), we can decide if:

1. \( d_P(q) \leq a \).
2. \( d_P(q) \leq b \).
3. Find a point \( y \in P \), so that \( ||qy|| \leq (1 + \gamma/3)d_P(q) \).

If (1) or (2) happens only two PLEB queries are being carried out, if (iii) happens then \( O(\log((\log(b/a)/\gamma)) \) PLEB queries are being performed.

Now we use \( F \)-the \( \lambda \)-stretch hierarchial clustering of \( P \) for constructing approximate NN search tree using IPLEBs. BuildNNTree. For a given point set and its hierarchial clustering \( F \), the BuildNNTree constructs the search structure (tree) by breaking \( F \) into several subtrees. The BuildNNTree will have a maximum of three children on each node and it has a depth of atmost \( |\log n| + 1 \). The NNTree is constructed based on the idea that we discussed for the reduction of NN search to PLEBS. So it will have three children which are the three conditions of the reduction. The FindApproxNN returns the nearest neighbour to the query point \( q \) and is a \( 2\gamma \)-NN to \( q \) for \( 1/2 \geq \gamma \geq 0 \). Below are the algorithms for BuildNNTree and approximate NN search tree.

**BuildNNTree**

**Input**: \( M \subseteq P \) - set of points

\( F \)- nd- hierarchial clustering of \( M \)

\( \gamma \)- approx. parameter

\([r_{\text{min}}, r_{\text{max}}]\) - range of distances that should be considered for NN queries

**Output**: Tree \( TM \) for NN-search on \( M \)

Create a node \( v \)
Set \( r_v^- \leftarrow \max(r_{\text{median}}(M), r_{\text{min}}) \)
Set \( r_v^+ \leftarrow \min(r_{\text{top}}(M), r_{\text{max}}) \)
Set \( P_v \leftarrow M \)
if \( r_v^- \geq r_v^+ \) then nil
Compute \( I_v \leftarrow I(M, r_v^-, r_v^+, \gamma/3) \)
\( M^+ \leftarrow M(r_v^+) \)
\( T \leftarrow \text{BuildNNTree}(M^+, \gamma, r_v^+, r_{\text{max}}) \)
Set \( T \) outer child of \( v \) : \( v_{\text{outer}} \leftarrow T \)
for \( X \in CC_M(r_v^-) \) do
if \( |X| > 1 \) then
\( v_X \leftarrow \text{BuildNNTree}(X, \gamma, r_{\text{min}}, r_v^-) \)
return \( v \)
end BuildNNTree
such that

P

balls

covers space. For a ball

b

\lceil
\gamma

point which is

so that for a query point

q

Let

u

be the point returned by

I

if

r_u^c \leq d_p(q) \leq r_u^c +

return

u

(*d_p(q) \leq r_u^c +*)

Let

X \in CC_P(r_u^-) \text{ s.t. } u \in X

if

|X| = 1

return

u

return

FindApproxNN(v_X,q)
end FindApproxNN

3 NN Search via PLEBs and cubes

In the above algorithm, FindApproxNN, our IPLEB structure returns the point which is approximate nearest neighbour to our query point

q

given that it belongs to a particular IPLEB. So the next step of our algorithm is to find a smallest ball

B

of IPLEB that contains

q

canonical ball of

q

in

B

: For a set of balls

B

such that

\bigcup_{b \in B} = \mathbb{R}^d

and a query point

q \in \mathbb{R}^d

, the canonical ball of

q

in

B

is the smallest ball of

B

that contains

q

Lemma

: For an interval PLEB

I = I(P,a,b,\gamma)

, one can compute a set

B

of

O(|P|/\gamma) \log(b/a)

balls, so that for a query point

q

if

a \leq d_p(q) \leq b

then the canonical ball containing

q

in

B

corresponds to a point which is

\gamma

-NN to

q

in

P

.

Now partition

\mathbb{R}^d

into a uniform axis-parallel grid

\tilde{\mathcal{G}}(u)

centered at the origin with the side-length of the grid as

2^{\lceil \log u \rceil}

where

u

is a real number. Varying the value of

u

gives us a multi-resolution grid that covers space. For a ball

b = b(p,r)

, let

GC(b,\epsilon)

be the set of cubes of the grid

\tilde{\mathcal{G}}(r\epsilon/3d)

that intersects

b

. Let

b_x = \bigcup_{c \in GC(b,\epsilon)}

. Clearly,

b_x

is an

\epsilon

-approximation to

b

. Note that

|GC(b,\epsilon) |

= \mathcal{O}(1/\epsilon^d)

. For a set of balls

B

, let

GC(B,\epsilon) = \bigcup_{b \in B} GC(b,\epsilon)

. For

c \in GC(b,\epsilon)

, let

r(c)

be the radius of the smallest ball

b \in B

such that

c \in GC(b,\epsilon)

. Similarly, let

p(c)

be the center of the ball that realizes

r(c)

.

Theorem

: Let

P

be a set of

n

points in

\mathbb{R}^d

. Given a parameter

\epsilon > 0

, one can compute a set

C(P)

of

O(n \frac{\log n}{\epsilon^d} \log \frac{2}{\epsilon})

cubes, all of them taken from the hierarchical grid

\tilde{\mathcal{G}}(.)

. For any query point

q

, let

c(q)

be the cube of

C

that contains it and has the smallest value of

r(c)

associated with it. Then,

p(c)

is an

\epsilon

-NN to

q

in

P

. The running time needed to compute this set of cubes is

O(n \frac{\log n}{\epsilon^d} \log \frac{2}{\epsilon})

.

Theorem

: Let

P

be a set of

n

points in

\mathbb{R}^d

, and a parameter

\epsilon > 0

, then one can compute a set

C(P)

of

O(u \frac{\log u}{\epsilon^d} \log \frac{2}{\epsilon})

regions. Each region is either a cube or an annulus (i.e., the difference between two cubes, one contained inside the other). The regions of

C(P)

are disjoint and provide a covering of space. Furthermore, each such region

c \in C

has an associated point

p \in P

, so that for any point

q \in c

, the point

p

is a

\epsilon

-NN of

q

in

P

. Thus, given a query point

q \in \mathbb{R}^d

, one can compute in

O(\log(n/\epsilon))

time a

\epsilon

-NN to

q

in

P

. The overall time to perform this construction is

O(n \frac{\log n}{\epsilon^d} \log \frac{2}{\epsilon})

.

From the above theorem it is clear that the nearest neighbour can be found in

O(\log(n/\epsilon))

and doesn’t depend on

d

. exponentially. We implement the NN search in the cubes using compressed quadtrees. The following diagrams illustrate how the balls are streamed to a compressed quad tree to get the required approximate voronoi diagram.
4 Implementation:

I have used the code present in Sariel’s website for the implementation part. The code he has uses quicktime(qt) to demonstrate how the space decomposition is done and nearest neighbours are found. The above shown figures are the one that you get using a qt for implementation. I tried to implement a normal c++ code without any qt by using parts of his code. I am trying to plug in the qt part so that the nearest neighbour can be visualized.

5 References

- A Replacement for Voronoi Diagrams of Near Linear Size by Sariel Har-peled
- Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality by Piotr Indyk and Rajeev Motwani
- Efficient Search for Approximate Nearest Neighbor in High Dimensional Spaces by Eyal Kushilevitz, Rafail Ostrovskyt and Yuval Rabad