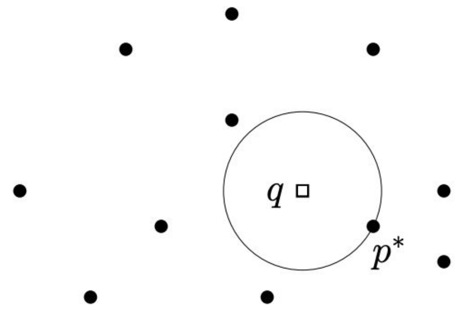
Parallel Nearest Neighbors in Low Dimensions with Batch Updates*

Guy E. Blelloch, Magdalen Dobson (CMU)
Presented by Vishaal Ram

Nearest Neighbors

k-nearest neighbors problem

- the problem: given a point set P, after some initial preprocessing, be able to compute for any point, the k-nearest neighbors in P
  - optionally be able to support efficient updates
- computing nearest neighbors of points is a fundamental problem in computer science
  - applications in graphics, AI, and particle physics
- research currently is focused on approximate high dimension nearest neighbors and exact lower dimension nearest neighbors
  - this paper is focused on the later category

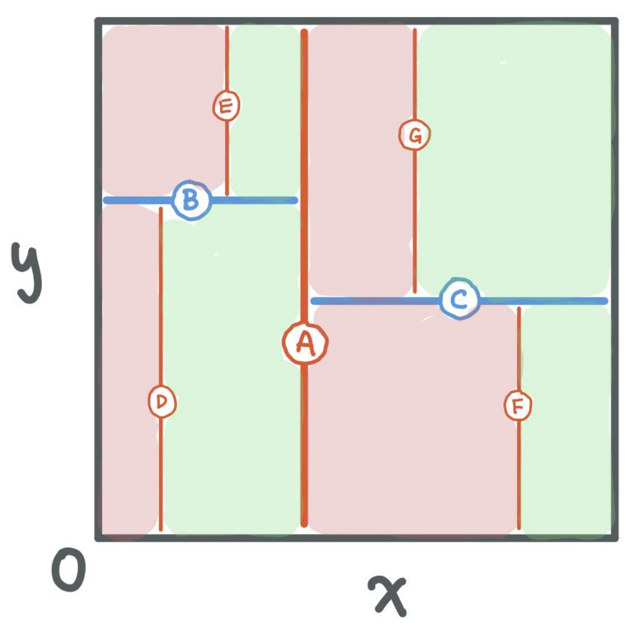


Current Algorithms

kd-tree

- The most common method for computing k-nearest neighbors in low dimensions in the kd-tree
- Each node represents a point in the point set
- The children of the node consist of all points on either side of the parent node in some dimension
- The subdividing dimension typically alternates through each of the n-dimensions

- Question: How should the split node be selected?

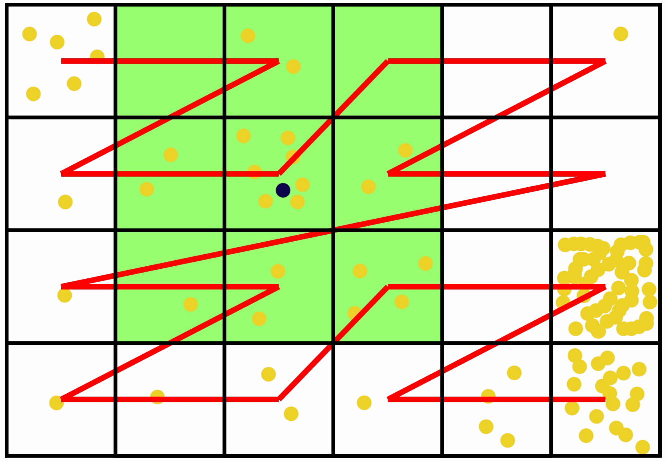


kd-tree complexity

- Construction:
    - At each level of the tree, the median is computed in O(n) time. As there are O(log n) levels, the total complexity becomes O(n log n). However on very unbalanced trees, this can approach O(n^2) in practice
- Insertion/Deletion:
    - Insertion and deleting is quick, involving a O(log n) binary search to find the desired point.
- Nearest Neighbors:
    - Nearest neighbors can also be computed in O(log n) time. The idea is to traverse the tree, keeping track of the intersection of the dividing hyperplane and a sphere centered at the search point.

Morton ordering

- The morton curve (or Z-order curve) describes a space filling curve on an arbitrary dimensional space.
- For a point, we take its integer coordinates in binary form and interleave the digits. We then sort the points based on this new value.
- We take advantage of the following property of such an ordering:
  - for any given points p and q in the ordering, all points in the rectangle between p and q lie between the points in the ordering



Morton ordering complexity

- Construction:
    - extremely quick, a simple sort can be performed in O(n log n) time worst case
- Insertion/Deletion:
    - set insertion and deletion can also be done in O(n log n)
- Nearest Neighbors:
    - we can simply scan neighboring points to find the nearest neighbors. The worst case complexity is still O(n) if the points are not evenly distributed, though in most practical cases it’s closer to O(k)

Previous Work

- Arya and Mount
  - balanced box decomp tree (variant of kd-tree with splitting rule that attempts to divide both space and point count). Their queries take O(k/e log n) work though their implementation doesn't support updates
- Chan
  - a minimalistic Morton ordering approach that uses random offset to reduce adversarial inputs. Has O(n log n) preprocessing time and O(1/e log n) expected time per query
- Conner and Kumar
  - An improvement on Chan that provides expected O(k log k) work for queries given constant bound expansion

zd-tree

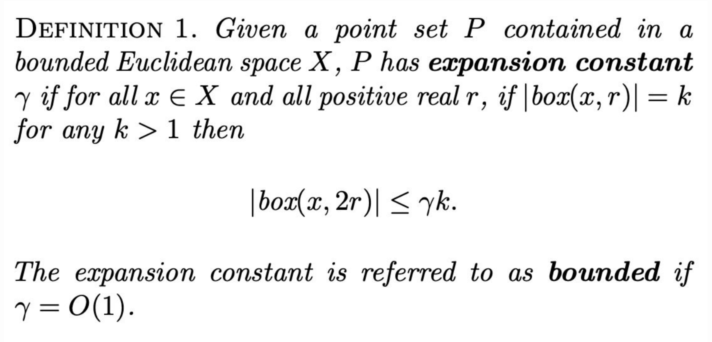
zd-tree

- Based on the previous two approaches we conclude that kd-trees have slow preprocessing but fast query times while morton ordering has fast preprocessing but slow query times
- The idea is to combine both approaches:
    - we construct a kd-tree with a splitting rule based on morton ordering
    - this aims to combine the advantages of both approaches
    - we call this new data structure the zd-tree
- We also have an efficient parallel batch update algorithm
- Assumptions:
    - Bounded expansion constant and bounded ratio

Assumptions

Bounded Expansion Constant

- requires that the density of points in metric space doesn't change rapidly



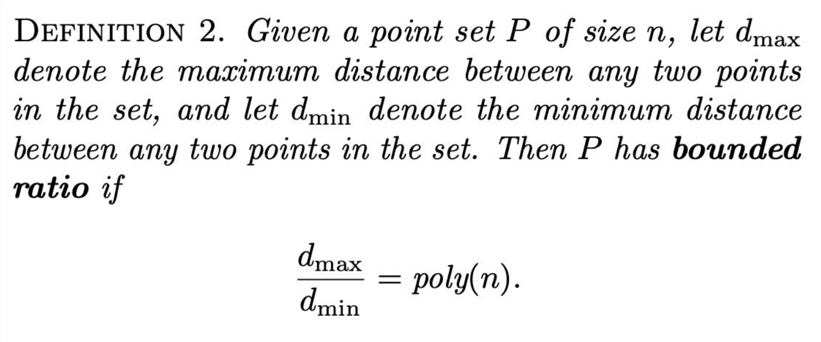
DEFINITION 1. *Given a point set P contained in a bounded Euclidean space X, P has* ***expansion constant*** *γ if for all $x \in X$ and all positive real r, if $|box(x,r)| = k$ for any $k > 1$ then*

$$|box(x, 2r)| \leq \gamma k.$$

The expansion constant is referred to as ***bounded*** *if $\gamma = O(1)$.*

Bounded Ratio

- requires that the distances between points are within some bounded ratio



DEFINITION 2. *Given a point set P of size n, let $d_{\max}$ denote the maximum distance between any two points in the set, and let $d_{\min}$ denote the minimum distance between any two points in the set. Then P has* ***bounded ratio*** *if*

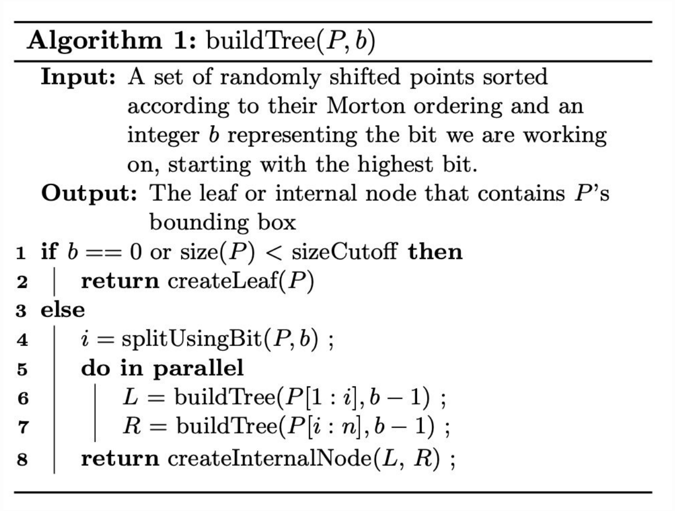
$$\frac{d_{\max}}{d_{\min}} = poly(n).$$

Data Structure

- We construct a kd-tree as follows:
    - the root note represents the entire bounding box
    - we then split the points into child nodes depending on whether their Morton value at bit i is a 0 or 1
    - each internal node of the tree stores the two opposite corners of the bounding box represented by the children, as well as the parent node
    - the leaf nodes additionally store the set of points it contains
        - every node is contained in exactly one leaf node
        - the number of nodes in the leaf is bounded by a constant

Construction

- To construct the tree we first preprocess the input
    - shift each coordinate by a random offset. this will not affect the queries and will allow us to achieve better average complexity
- Using a comparison function by Chan, we can compute the Morton sort in O(n log n) work, though with a linear time radix sort we can do O(n^e) span.
- Creating the tree is done with a divide and conquer algorithm
    - we parallel recurse on both sides of the tree, computing the split point with a binary search
    - even when completely unbalanced, the total work remains O(n log n)



Algorithm 1: buildTree(P, b)

Input: A set of randomly shifted points sorted according to their Morton ordering and an integer b representing the bit we are working on, starting with the highest bit.

Output: The leaf or internal node that contains P's bounding box

```
if $b == 0$ or size($P$) < sizeCutoff then
    return createLeaf($P$)
else
    $i$ = splitUsingBit($P, b$) ;
    do in parallel
        $L$ = buildTree($P[1 : i], b - 1$) ;
        $R$ = buildTree($P[i : n], b - 1$) ;
    return createInternalNode($L$, $R$) ;
```

Downward Search

- We first define a downward recursive algorithm for k-nearest neighbors to a point p
- Maintain a candidate set of k points that we update as we find closer points
- let r be the distance from p to the kth farthest point (r = inf if <k points)
- search vertex v only if bounding box intersects ball of radius r around p
- if node is a leaf, iterate through all points and update set if necessary
    - otherwise recurse on children, prioritizing those whose center is close to p

Upward Search

- A more efficient search that uses downward search as a subroutine
- The idea is that during a search, only a small proportion of the tree actually needs to be traversed
- We start at the leaf node containing p, once again maintaining a set of candidate points
- First add all the nodes in the leaf node
- Then we traverse up to the parent only if the ball around p with radius r extends outside the current node's bounding box
    - when searching the parent, we also search the parent's other children using the downward search algorithm

Algorithm 2: searchDown(T, p, N)

needed subroutines:

distance(p, N, k) returns infinity if N has fewer than k points and otherwise returns the distance from p to the furthest point in N; $k = 1$ if not specified.

insert(N, p, k) adds p to the set N keeping only the k closest points to p.

withinBox(T, p, r) returns true if p is within a distance r of the bounding box for T.

Input: A pointer to a tree node T, the query point p, and a current set of up to k nearest neighbors N.

Output: The k-nearest neighbors of p.

```
$r \leftarrow$ distance$(p, N, k)$ ;
if withinBox$(T, p, r)$ then
    if $T =$ Leaf then
        $Q \leftarrow$ set of points contained in $T$ ;
        for $q \in Q$ do
            if $q \neq p$ then
                if distance$(q, p) <$ distance$(p, N, k)$ then
                    insert$(N, p, k)$;
    else
        $R \leftarrow T$.Right() ;
        $L \leftarrow T$.Left() ;
        $\ell \leftarrow$ distance$(p, L$.center());
        $r \leftarrow$ distance$(p, R$.center()) ;
        if $\ell < r$ then
            N ' = searchDown$(L, p, N)$;
            return searchDown$(R, p, N')$;
        else
            N' = searchDown$(R, p, N)$;
            return searchDown$(L, p, N')$;
```

Algorithm 3: searchUp(C, p)

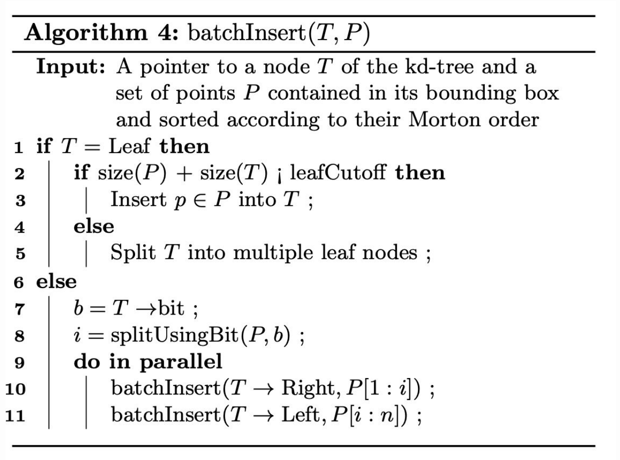
withinBox(T, p, r) with negative r returns true if p is in within the bounding box of T and at least r from the boundary.

Input: A leaf C of the kd-tree and a point p within the bounding box of C.

```
$N = \emptyset$
$Q \leftarrow$ set of points contained in $C$ ;
for $q \in Q$ do
    if $q \neq p$ then
        if $d(q, p) <$ distance$(p, N, k)$ then
            insert$(N, p, k)$;
$r \leftarrow$ distance$(p, N, k)$;
$P \leftarrow C$.Parent() ;
while not withinBox$(C, p, -r)$ and $P \neq \top$ do
    if $P$.Left() $= C$ then
        $N =$ searchDown$(P$.Right()$, p, N)$;
    else
        $N =$ searchDown$(P$.Left()$, p, N)$;
    $C = P$;
    $r \leftarrow$ distance$(p, N, k)$;
    $P \leftarrow C$.Parent() ;
return $N$
```

Batch-dynamic Updates

- Since all points are stored at leaf nodes, performing updates is extremely simple
- To insert a single point q, first locate the leaf node that would contain q
- If inserting q directly would increase the size of the leaf past the threshold, subdivide into two children and insert q accordingly
  - otherwise directly add q into the leaf node
- We extend to parallel batch update by recursion from the root, splitting up the batch along the way
- We avoid cases where the insertion would require rebuilding the entire tree (e.g. cases where the number of bits changes) by requiring a fixed bounding box for all the points beforehand

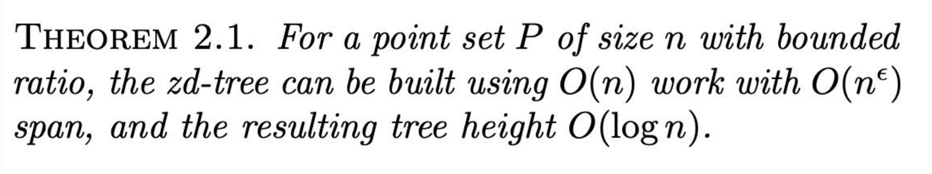


Algorithm 4: batchInsert(T, P)

```
Input: A pointer to a node $T$ of the kd-tree and a
       set of points $P$ contained in its bounding box
       and sorted according to their Morton order
if $T$ = Leaf then
    if size$(P)$ + size$(T)$ ¡ leafCutoff then
        Insert $p \in P$ into $T$ ;
    else
        Split $T$ into multiple leaf nodes ;
else
    $b = T \rightarrow$bit ;
    $i =$ splitUsingBit$(P, b)$ ;
    do in parallel
        batchInsert$(T \rightarrow \text{Right}, P[1:i])$ ;
        batchInsert$(T \rightarrow \text{Left}, P[i:n])$ ;
```

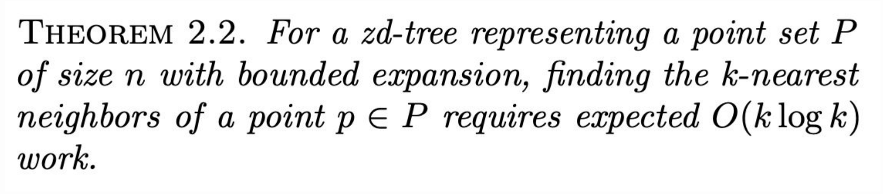
Complexity Analysis

Build Time


THEOREM 2.1. *For a point set P of size n with bounded ratio, the zd-tree can be built using $O(n)$ work with $O(n^{\epsilon})$ span, and the resulting tree height $O(\log n)$.*

- From the bounded ratio assumption, the bounding box has max length d_max, which must be divided until the space between points is at most d_min.
- Since d_max/d_min = poly(n), d_max is halved O(log n) times so the tree has depth O(log n).
- For sorting, radix sort takes O(n) as we only require O(log n) bits, proving the work bound.

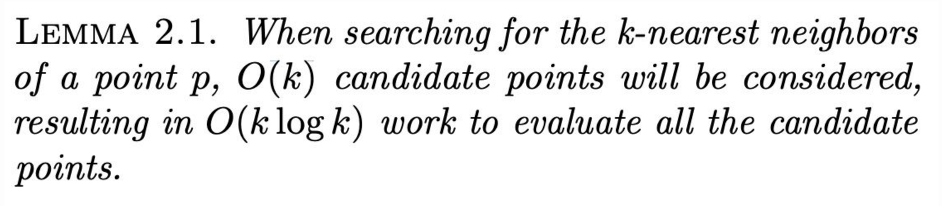
Search Time



THEOREM 2.2. *For a zd-tree representing a point set P of size n with bounded expansion, finding the k-nearest neighbors of a point $p \in P$ requires expected $O(k \log k)$ work.*

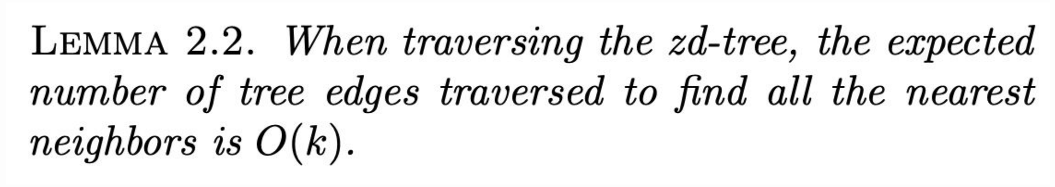
- We apply the upward search algorithm we presented before
- For the proof, we divide into two parts: the work from searching through points in the leaf nodes and the work from traversing the zd-tree to find those leaves

Lemma 1


LEMMA 2.1. *When searching for the k-nearest neighbors of a point p, $O(k)$ candidate points will be considered, resulting in $O(k \log k)$ work to evaluate all the candidate points.*

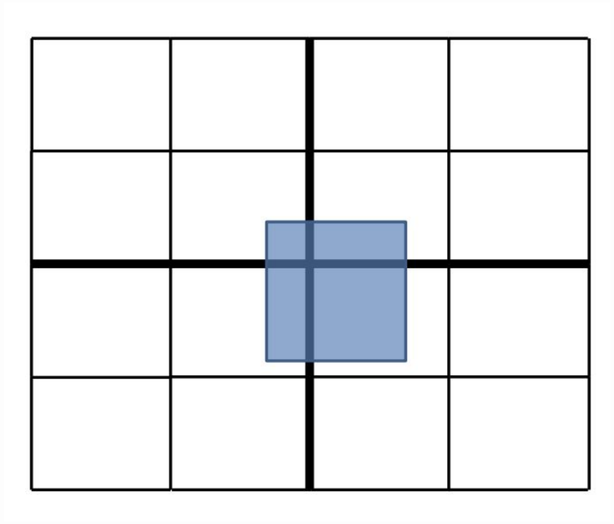
- Firstly, the log k factor arises from the O(log k) cost of each point insertion in the set
- The initial approximation is found by traversing up the tree from the starting leaf, stopping once we read a node with at least k descendants
  - we know the parent B's bounding box has O(k) points by the expansion constant
  - Let r be the side length of B, then we know the desired points must be within box(p, r)
  - If we expand the box twice, the resulting box Q must contain all the desired points and by expansion ratio, we still encompass O(k) points as desired.

Lemma 2



LEMMA 2.2. *When traversing the zd-tree, the expected number of tree edges traversed to find all the nearest neighbors is $O(k)$.*

- let B be the search area box defined in the previous lemma
- The largest cut of B, divides it into 2^d boxes, which each adds at most O(k) cost from traversing their leaf nodes
- Thus it suffices to show that the expected max distance between two leaf nodes in B is O(1)



Lemma 2 cont.

- Suppose without loss of generality, the search space is a box with side length 2^h.
- The probability that it's contained within a search box of side length 2^(h+j) is precisely ((2^j-1)/2^j)^d due to random offsets
- This the the probability that the length between two leaf nodes is j
- Therefore the expectation of the length is

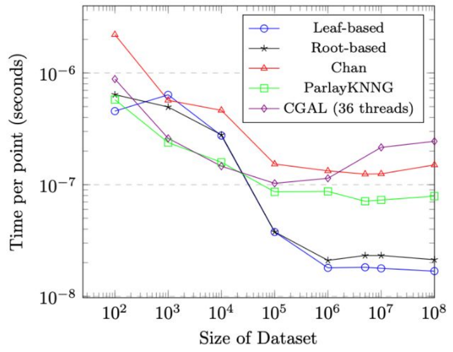
$$\sum_{j=1}^{\infty}\left(1-\left(1-\frac{1}{2^j}\right)^d\right) = O(1)$$

- and the result follows.

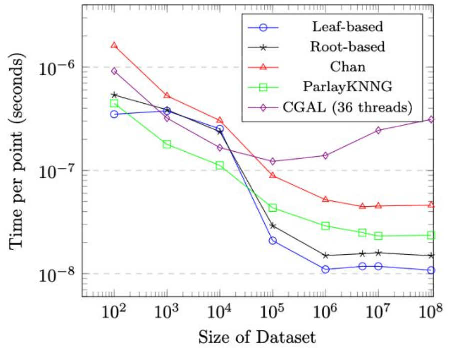
Experiments

Results

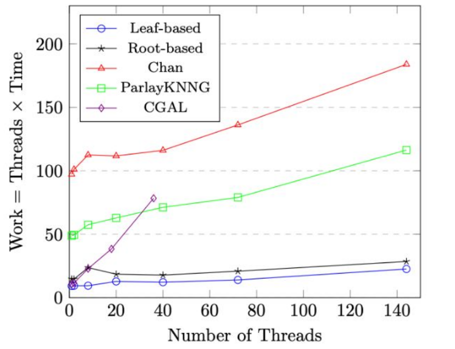
- tests against other state-of-the-art implementations were done on a 72-core machine



(a) Time required to calculate nearest neighbors as the size of the dataset increases. Calculated by dividing the total time by the number of points queried.



(b) The same as (a) but with a 2D dataset drawn randomly from a square instead of a 3D dataset.



(c) Total work (threads × time) required to build a tree of 10 million points, then build the nearest neighbor graph of the point set. Shown as the number of threads vary.

Conclusion

- Strengths
  - based on empirical results, the algorithm scales well with large datasets, both in build time and query execution
  - the algorithm is extremely work efficient, best utilizing thread capacity
  - the tests support the strong theoretical results
- Weaknesses
  - analysis and testing has only considers small dimensions
  - needs more thorough testing of span and overall parallelism