

KD Trees and their Applications

Maria Despoina Siampou (sdi1600151), George Michas (sdi1400109)



HELLENIC REPUBLIC
National and Kapodistrian
University of Athens

April 12, 2019

Contents

- 1 The Problem
- 2 Introduction
- 3 2D Orthogonal Range Search
- 4 K-d Trees and Nearest-Neighbor-Search in R^d
- 5 Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs
- 6 Efficient Delaunay Tessellation through K-D Tree Decomposition
- 7 References

Contents

- 1 The Problem
- 2 Introduction
- 3 2D Orthogonal Range Search
- 4 K-d Trees and Nearest-Neighbor-Search in R^d
- 5 Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs
- 6 Efficient Delaunay Tessellation through K-D Tree Decomposition
- 7 References


Range Searching

Databases store records or objects.

In a data warehousing environment, the relational databases need to be optimized for data retrieval and tuned to support the analysis of business trends and projections.

This type of informational processing is known as online analytical processing (OLAP) or decision support system (DSS) processing.

A dimensional database needs to be designed to support queries that retrieve a large number of records and that summarize data in different ways. A dimensional database tends to be subject oriented.

 Finding **all** records in a database that satisfy specified range restrictions on a specified set of attributes (dimensions) is called range searching.

Examples

Examples of **range queries**:

- Search all watches with price [100,200]
- Search for all undergraduate students graduated between year 2000 to 2009 with GPA between 3.0 to 4.0
- Select a group of files by dragging a rectangle [P1(x1,y1), P2(x2,y2)] using mouse on your computer.
- “What products are selling well?”
- “At what time of year do certain products sell best?”
- “In what regions are sales weakest?”

Contents

- 1 The Problem
- 2 Introduction
- 3 2D Orthogonal Range Search
- 4 K-d Trees and Nearest-Neighbor-Search in R^d
- 5 Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs
- 6 Efficient Delaunay Tessellation through K-D Tree Decomposition
- 7 References

Introduction

1D range query problem

Preprocess a set of n points on the real line such that the ones inside a 1D query range (interval) can be reported fast.

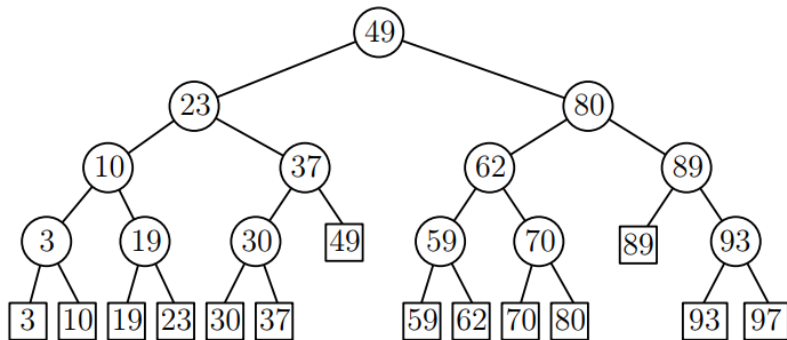
The points p_1, \dots, p_n are known beforehand, the query $[x, x']$ only later

A solution to a query problem is
a data structure description, a query algorithm, and a construction algorithm



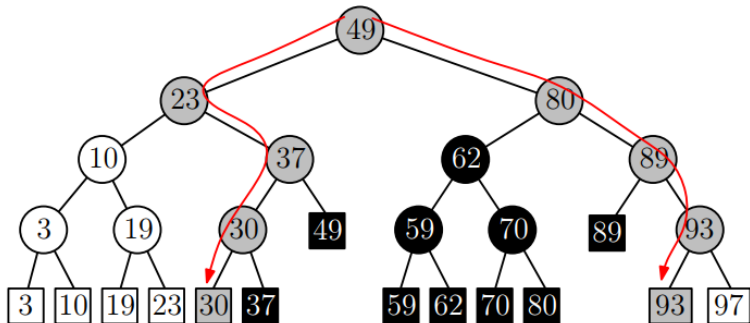
AVL Tree

A balanced binary search tree with the points in the leaves



AVL Tree

A 1-dimensional range query with $[25, 90]$



1D Range Query - Algorithm

Algorithm 1DRANGEQUERY($\mathcal{T}, [x : x']$)

1. $v_{\text{split}} \leftarrow \text{FINDSPLITNODE}(\mathcal{T}, x, x')$
2. **if** v_{split} is a leaf
3. **then** Check if the point in v_{split} must be reported.
4. **else** $v \leftarrow lc(v_{\text{split}})$
5. **while** v is not a leaf
6. **do if** $x \leq x_v$
7. **then** REPORTSUBTREE($rc(v)$)
8. $v \leftarrow lc(v)$
9. **else** $v \leftarrow rc(v)$
10. Check if the point stored in v must be reported.
11. $v \leftarrow rc(v_{\text{split}})$
12. Similarly, follow the path to x' , and ...

Query time analysis

The efficiency analysis is based on counting the numbers of nodes visited for each type :

- White nodes: Never visited by the query;
 - No time spent
- Grey nodes: visited by the query, unclear if they lead to output;
 - Since the tree is balanced, its depth is $O(\log n)$
- Black nodes: visited by the query, whole subtree is output;
 - A (sub)tree with m leaves has $m-1$ internal nodes; Traversal visits $O(m)$ nodes and finds m points for the output

Theorem

A set of n points on the real line can be preprocessed in $O(n \log n)$ time into a data structure of $O(n)$ size so that any 1D range query can be answered in $O(\log n)$ time, where k is the number of answers reported.

Contents

- 1 The Problem
- 2 Introduction
- 3 2D Orthogonal Range Search
- 4 K-d Trees and Nearest-Neighbor-Search in R^d
- 5 Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs
- 6 Efficient Delaunay Tessellation through K-D Tree Decomposition
- 7 References

Introduction

2D range query problem

Preprocess points $P = \{p_1, p_1, \dots, p_n\} \subset \mathbb{R}^2$, to answer queries efficiently: Which points lie inside a query rectangle $[x : x_0] [y : y_0]$?

- $p = (p_x, p_y)$ lies in the rectangle iff $p_x \in [x, x_0]$ $p_y \in [y, y_0]$.

Question

Why can't we simply use a balanced binary tree in x-coordinate?

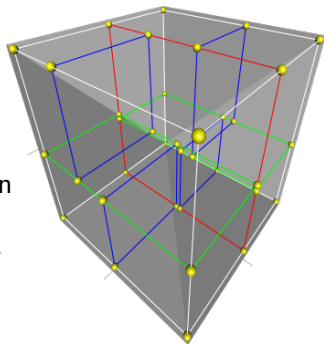
Or, use one tree on x-coordinate and one on y-coordinate, and query the one where we think querying is more efficient?

KD Trees - Introduction

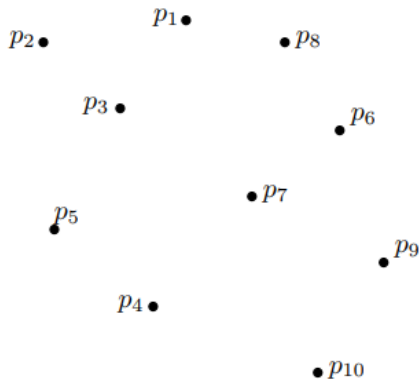
A **k-d tree** is a space-partitioning data structure for organizing points in a k-dimensional space.

The k-d tree is a **binary tree** in which every leaf node is a k-dimensional point : Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces.

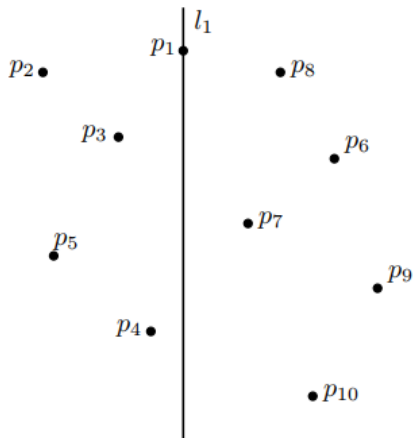
Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. Every node in the tree is associated with one of the k dimensions, with the hyperplane perpendicular to that dimension's axis.



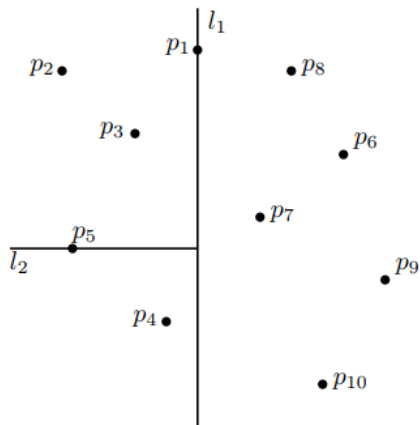
Constructing a kd tree



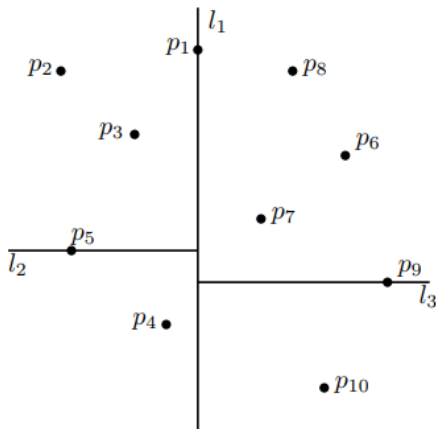
Constructing a kd tree



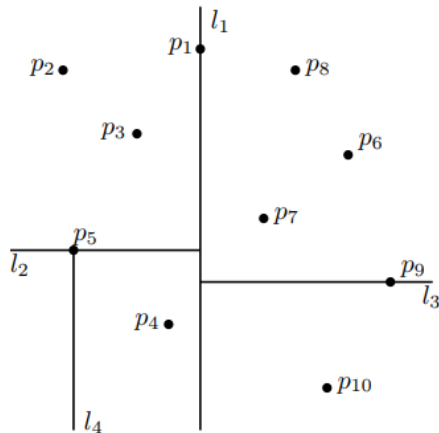
Constructing a kd tree



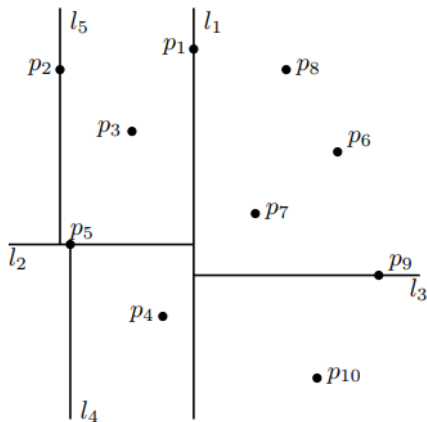
Constructing a kd tree



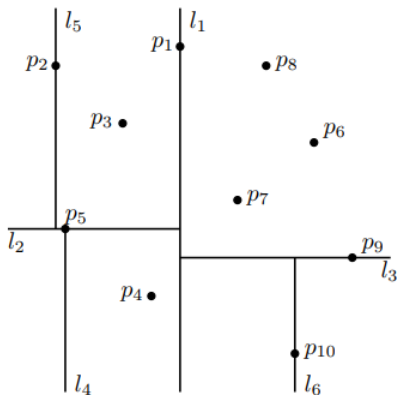
Constructing a kd tree



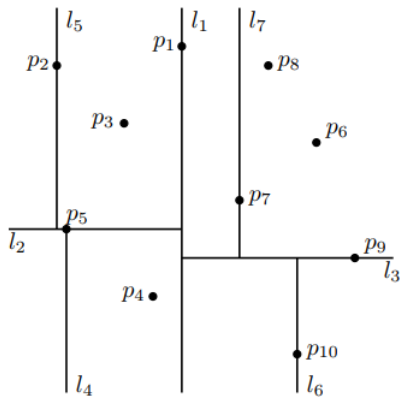
Constructing a kd tree



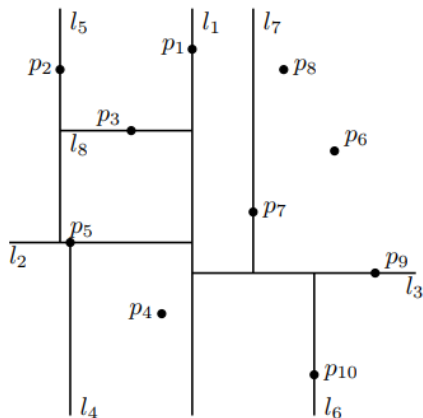
Constructing a kd tree



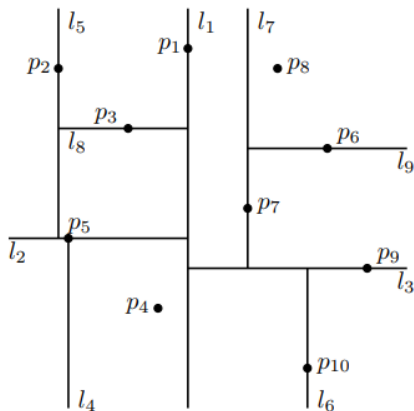
Constructing a kd tree



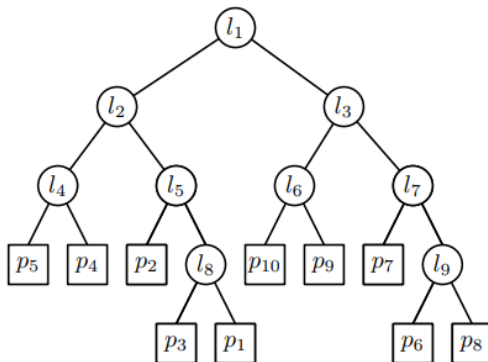
Constructing a kd tree



Constructing a kd tree



Constructing a kd tree



Constructing a kd tree

Algorithm BUILDKDTREE($P, depth$)

1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** $depth$ is even
4. **then** Split P with a vertical line ℓ through the median x -coordinate into P_1 (left of or on ℓ) and P_2 (right of ℓ)
5. **else** Split P with a horizontal line ℓ through the median y -coordinate into P_1 (below or on ℓ) and P_2 (above ℓ)
6. $v_{\text{left}} \leftarrow \text{BUILDKDTREE}(P_1, depth + 1)$
7. $v_{\text{right}} \leftarrow \text{BUILDKDTREE}(P_2, depth + 1)$
8. Create a node v storing ℓ , make v_{left} the left child of v , and make v_{right} the right child of v .
9. **return** v

The *median* of a set of n values can be computed in **$O(n)$** time
(randomized: easy; worst case: much harder)

Theorem

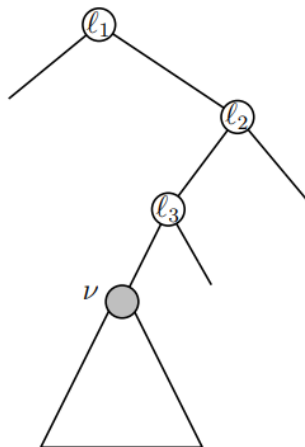
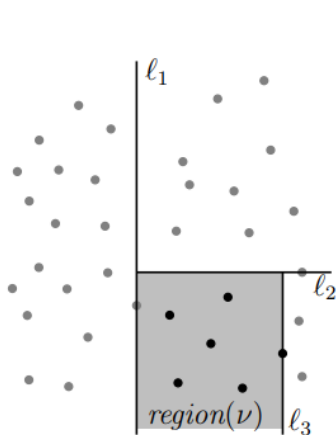
Let $T(n)$ be the time needed to build a kd-tree on n points:

- $T(1) = O(1)$
- $T(n) = 2 \cdot T(n/2) + O(n)$

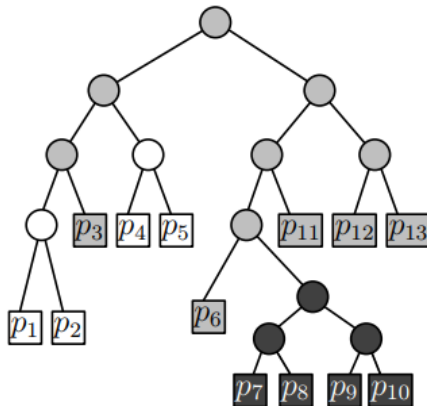
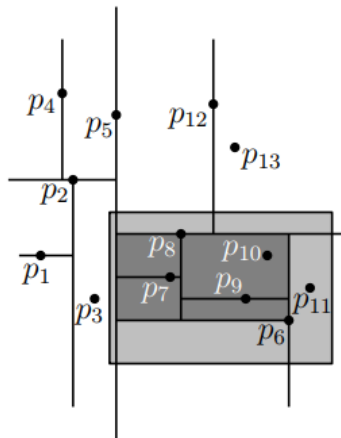
A kd-tree can be built in **$O(n \log n)$** time and with $O(n)$ storage (points stored at leaves)

Constructing a kd tree

How do we know $\text{region}()$ when we are at a node ?



Constructing a kd tree



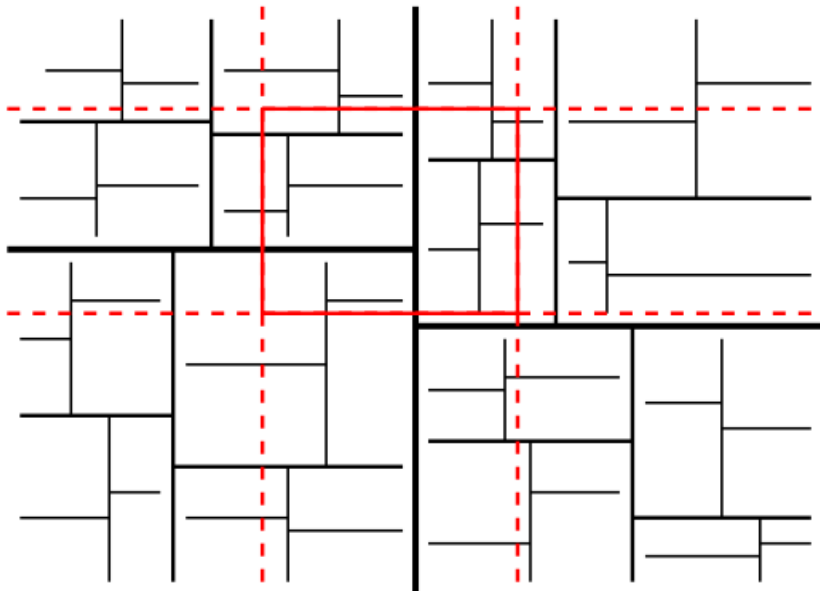
Algorithm SEARCHKDTREE(v, R)

Input. The root of (a subtree of) a kd-tree, and a range R

Output. All points at leaves below v that lie in the range.

1. **if** v is a leaf
2. **then** Report the point stored at v if it lies in R
3. **else if** $region(lc(v))$ is fully contained in R
4. **then** REPORTSUBTREE($lc(v)$)
5. **else if** $region(lc(v))$ intersects R
6. **then** SEARCHKDTREE($lc(v), R$)
7. **if** $region(rc(v))$ is fully contained in R
8. **then** REPORTSUBTREE($rc(v)$)
9. **else if** $region(rc(v))$ intersects R
10. **then** SEARCHKDTREE($rc(v), R$)

KD - Tree Query time analysis



Query time analysis

- Binary depth : $(1/2) \log n$
- Binary nodes : $2^{(1/2) \log n} = \sqrt{n}$.
- Number of grey nodes : $4 O(\sqrt{n}) = O(\sqrt{n})$ (at most the number of grey nodes for two vertical and two horizontal lines)
- Black nodes : $O(k)$ (reporting a whole subtree with k leaves)

Theorem

A set of n points in the plane can be preprocessed in $O(n \log n)$ time into a data structure of $O(n)$ size so that any 2D range query can be answered in $O(\sqrt{n} + k)$ time, where k is the number of answers reported. For range counting queries, we need $O(\sqrt{n})$ time.

Contents

- 1 The Problem
- 2 Introduction
- 3 2D Orthogonal Range Search
- 4 K-d Trees and Nearest-Neighbor-Search in R^d**
- 5 Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs
- 6 Efficient Delaunay Tessellation through K-D Tree Decomposition
- 7 References

Kd-Trees and Nearest-Neighbor-Search in R^d

NNS in R^d

Given a finite data set $X \subset R^d$, $x^* \in X$ is a nearest neighbor of query $q \in R^d$, iff :

$$\text{dist}(q, x^*) \leq \text{dist}(q, x) , \forall x \in X$$

ANNS in R^d

Given a finite data set $X \subset R^d$ and $\epsilon > 0$ with $\epsilon \in R$, $x^* \in X$ is an ϵ -approximate nearest neighbor of query $q \in R^d$, iff :

$$\text{dist}(q, x^*) \leq (1 + \epsilon)\text{dist}(q, x) , \forall x \in X$$

If $\epsilon = 0$, this reduces to exact NNS.

Kd-Trees in High Dimensions

- In high dimensional space, tree-based data structures are affected by the **curse of dimensionality**, either the running time or the space requirement grows exponentially in d .
- Do we have other techniques for big dimensions ?
- If a tree is fast in small dimensions, can we gather a lot of trees to get the speed up we want in big dimensions? Maybe something like a **forest**?

Techniques for A/NNS in High Dimensions

- **Locality Sensitive Hashing (LSH)** : The basic idea of this technique is to hash the points of the data set so as to ensure that the probability of collision is much higher for objects that are close to each other than for those that are far apart. **It reduces the dimensionality of high-dimensional data.**
Space : $O(dn^{1+\rho})$, Query(NNS/ANNS) : $O(dn^\rho)$ for some $\rho \in (0, 1)$.
- A different hashing approach is to represent points by **short binary codes** to approximate and accelerate distance computations. Recent research on learning such codes from data distributions is very active.
- **Product Quantization (PQ)** : A more general technique is to use any discrete representation of points, again learned from data points. A popular approach is product quantization (PQ), which both compresses data points and provides for fast asymmetric distance computations, where points remain compressed but queries are not.

k-d Generalized Randomized Forest (k-d GeRaF)

- The solution for big dimensions is to construct a **group of kd-trees**.
- The key idea is to generate **m k-d trees** based on **Randomization**.
- **Randomization** amounts to either generating a different randomly transformed pointset per tree (e.g., rotation or shuffling), or choosing splits at random at each node (e.g., split dimension or value).

- **Rotation** : Every tree uses a randomly rotated pointset, thus using a different set of dimensions/coordinates.
- **Split dimension** : We find the **t dimensions of highest variance** for the input set and then choose uniformly at random one of these t dimensions at each node.
- **Split value** : We find the median of current dimension and then perturb it by a number δ uniform distributed in $[-\frac{3\Delta}{\sqrt{d}}, \frac{3\Delta}{\sqrt{d}}]$, where Δ is the diameter of the current pointset and d the dimension.
- **Shuffling** : The split value may be witnessed in several points, instead of picking always the same point, shuffle them to break ties.

Parameters

- **m** : Number of trees in forest. A small number yields fast building and search, but may reduce accuracy, a large m covers space better and enhances accuracy, but slows down building and search.
- **t** : Number of dimensions used for splits. As d increases, a larger t is better, until accuracy begins to drop. The optimum t depends on the input.
- **p** : Maximum number of points per leaf. A large p means short trees, and saves space, a small p accelerates search, but may reduce accuracy.
- **c** : Maximum number of leaves to be checked during search. The higher this number, the higher the accuracy and search time.
- ϵ : Determines search accuracy, more accurate search comes at the expense of slower query.
- **k** : Number of neighbors to be returned for a query.

k-d GeRaF Search and Distance

Search

- Descend every tree to leaf, store unvisited branch nodes in min-priority queue Q.
- Examine nodes in Q, until $\frac{c}{1+\epsilon}$ leaves are checked.
- While descending the tree:
 - at leaf: update currently best distance.
 - at node: if query in the left half-space: insert right child to Q, descend to left child or vice versa.

Distance Computation

$$\text{dist}(q, x) = \|q - x\|^2 = \|q\|^2 + \|x\|^2 - 2q^t x$$

where $\|q\|$ is constant and also $\|x\|$ can be computed and stored at once (q represents the query for NNS and ANNS).

k-d GeRaF Build Algorithm

Algorithm 1: *k*-d GeRaF: building

```
input  : pointset  $X$ , #trees  $m$ , #split-dimensions  $t$ , max #points per leaf  $p$ 
output: randomized  $k$ -d forest  $F$ 
1 begin
2    $V \leftarrow \langle \text{VARIANCE of } X \text{ in every dimension} \rangle$ 
3    $D \leftarrow \langle t \text{ dimensions of maximum variance } V \rangle$ 
4    $F \leftarrow \emptyset$  ▷ forest
5   for  $i \leftarrow 1$  to  $m$  do
6      $f \leftarrow \langle \text{random transformation} \rangle$  ▷ isometry, shuffling
7      $F \leftarrow F \cup (f, \text{BUILD}(f(X)))$  ▷ build on transformed  $X$ , store  $f$ 
8   return  $F$ 

9 function BUILD( $X$ ) ▷ recursively build tree (node/leaf)
10  if  $|X| \leq p$  then ▷ termination reached
11    return leaf( $X$ )
12  else ▷ split points and recurse
13     $s \leftarrow \langle \text{one of dimensions } D \text{ at random} \rangle$ 
14     $v \leftarrow \langle \text{MEDIAN of } X \text{ in dimension } s \rangle$ 
15     $(L, R) \leftarrow \langle \text{SPLIT of } X \text{ in dimension } s \text{ at value } v \rangle$ 
16    return node( $c, v, \text{BUILD}(L), \text{BUILD}(R)$ ) ▷ build children on  $L, R$ 
```

k-d GeRaF Search Algorithm

Algorithm 2: *k*-d GeRaF: searching.

input : query point q , forest F , #neighbors k , max #leaf-checks c
output: k nearest points

```
1 begin
2    $Q.$ INIT()                                ▷ min-priority queue, initially empty
3   for  $i \leftarrow 1$  to  $m$  do
4      $\text{DESCEND}(q, F[i], \text{FALSE})$               ▷ descend  $i$ -th tree, store path in  $Q$ , no checks
5    $\ell \leftarrow 0$                              ▷ # of leaves checked
6    $H.$ INIT( $k$ )                                ▷ min-heap of size  $k$ 
7   while  $\neg Q.$ EMPTY()  $\wedge \ell < c/(1 + \epsilon)$  do
8      $(N, d) \leftarrow Q.$ EXTRACT-MIN()          ▷ (node, distance)
9      $\text{DESCEND}(q, N, \text{TRUE})$                   ▷ descend again, but check leaves now
10     $\ell \leftarrow \ell + 1$                     ▷ increase leaves checked
11  return  $H$ 

12 function  $\text{DESCEND}(q, \text{node } N, \text{check})$       ▷ descend node  $N$  for query  $q$ 
13    $d \leftarrow N.$ DIST( $q$ )                     ▷ signed distance to boundary
14   if  $d < 0$  then                             ▷  $q$  is in negative half-space
15      $Q.$ INSERT( $N.$ right,  $|d|$ )                 ▷ remember right child
16      $\text{DESCEND}(q, N.$ left,  $\text{check}$ )             ▷ descend left child
17   else
18      $Q.$ INSERT( $N.$ left,  $|d|$ )                   ▷ and vice versa
19      $\text{DESCEND}(q, N.$ right,  $\text{check}$ )

20 function  $\text{DESCEND}(q, \text{leaf } N, \text{check})$       ▷ test query  $q$  on leaf  $N$ 
21   if  $\neg \text{check}$  then return for  $i \in N.$ POINTS do
22      $H.$ INSERT( $i, \|q - X_i\|^2$ )              ▷ distances to points  $X_i$  in leaf  $N$ 
```

k-d GeRaF Benchmark

n	d	miss %				search (μsec)			
		BBD	LSH	FLANN	GeRaF	BBD	LSH	FLANN	GeRaF
10^3	100	100	0	16	0	1	212	12	199
	1000	100	50	100	50	5	1850	34	14
	5000	100	0	100	0	39	8675	149	122
	10000	100	37	100	2	276	17000	289	520
1000	10^3	100	50	100	50	5	1850	34	14
10000		100	0	100	0	5	1780	–	390
100000		100	8	100	0	276	–	–	10900

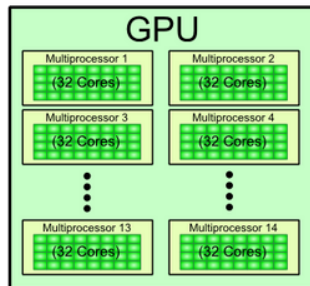
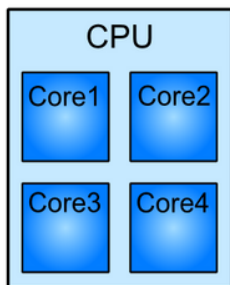
Table 3. Klein bottle search for $\epsilon = 0.1$, for varying n or d , where the other parameter is fixed. Search times in gray represent failure cases where miss rate is 100%. Queries are nearly equidistant from the points, which explains high miss rates. ‘–’ indicates preprocessing does not finish after 2hr.

Contents

- 1 The Problem
- 2 Introduction
- 3 2D Orthogonal Range Search
- 4 K-d Trees and Nearest-Neighbor-Search in R^d
- 5 Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs
- 6 Efficient Delaunay Tessellation through K-D Tree Decomposition
- 7 References

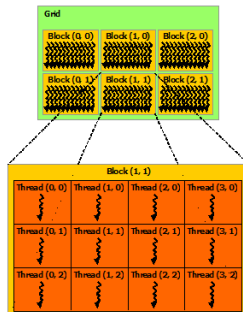
GPU Computing Introduction

CPU/GPU Architecture Comparison



GPU CUDA Example

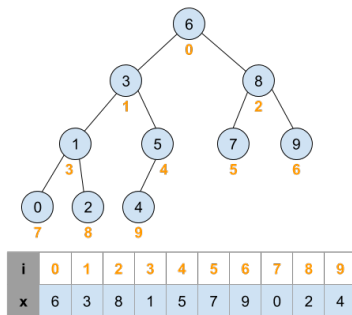
- **CUDA** is a parallel computing platform and application programming interface (API) model created by **Nvidia** and example how threads are arranged is the following:



- In **GPU-programming** you have to remember that the optimal way to do computations is to create **1-D array**, so next we will show you how a kd-tree(BST tree in general) can be constructed with an array.

Tree Implementation with Array

- **Root** is stored at **index 0**.
- **Children of a node v** with **index i** are stored at indices $\{2i, 2i + 1\}$, respectively.
- Lets see the following example :



Buffer k-d Tree Composition

Composition

Buffer k-d tree is composed of **four** parts:

- **Top Tree** : Consists of the first h levels of a standard k-d tree (tree in the form of an array).
- **Leaf Structure** : Consists of Blocks, every Block has one-to-one correspondence with the leaves of the top tree. (leaf i is stored at index $i - (2^h - 1)$).
- **Set of Buffers** (one buffer per leaf of the top tree) : Every query indices and can accommodate a predefined number $B \geq 2$ of integers each. In addition, a variable that determines the filling status is stored for each buffer.
- **Two Input Queues** : Two (first-in-first-out) queues of size m . The height of the buffer k-d tree is given by the height of its top tree $h = \{0, 1, \dots\}$ and B denotes its buffer size.

Buffer k-d Tree Composition

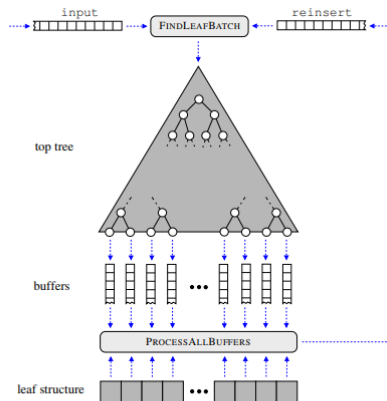


Figure 1. A buffer k -d tree: In each iteration, the procedure FINDLEAFBATCH removes query indices from both queues and distributes them to the buffers (or removes them if no further processing is needed). In case enough work has been gathered, the procedure PROCESSALLBUFFERS is invoked, which updates the nearest neighbors and reinserts all query indices into *reinsert*. The process stops as soon as both queues and all buffers are empty.

Buffer k-d Tree Parallel NNS

- The main idea is to “**delay**” the querying process by performing several iterations. In each iteration, query indices are propagated through the **top tree and are stored in the corresponding buffers**. As soon as the buffers get full, all collected nearest neighbor queries are processed at a single blow.
- As soon as the buffers get full, all collected nearest neighbor queries are processed at a **single blow**. This essentially leads to a separation of the two main phases of the classical k-d tree-based search:
 - Finding the leaf that needs to be processed next. (cannot be parallelized easily on GPUS)
 - Updating the nearest neighbors. (By far the most significant part of the overall runtime)
- The method is designed for processing huge amounts of queries in R^d with d larger than 4 and up to ≈ 25 .

Buffer k-d Tree Parallel NNS Algorithm

Algorithm 1 LAZYPARALLELNN

Require: A chunk $Q = \{\mathbf{q}_1, \dots, \mathbf{q}_m\} \subset \mathbb{R}^d$ of query points.

Ensure: The $k \geq 1$ nearest neighbors for each query point.

- 1: Construct buffer k-d tree \mathcal{T} for $P = \{\mathbf{x}, \dots, \mathbf{x}_n\} \subset \mathbb{R}^d$.
 - 2: Initialize queue input with all m query indices.
 - 3: **while** either input or reinsert is non-empty **do**
 - 4: Fetch M indices i_1, \dots, i_M from reinsert and input.
 - 5: $r_1, \dots, r_M = \text{FINDLEAFBATCH}(i_1, \dots, i_M)$
 - 6: **for** $j = 1, \dots, M$ **do**
 - 7: **if** $r_j \neq -1$ **then**
 - 8: Insert index i_j in buffer associated with leaf r_j .
 - 9: **end if**
 - 10: **end for**
 - 11: **if** at least one buffer is half-full (or queues empty) **then**
 - 12: $l_1, \dots, l_N = \text{PROCESSALLBUFFERS}()$
 - 13: Insert l_1, \dots, l_N into reinsert.
 - 14: **end if**
 - 15: **end while**
 - 16: **return** list of k nearest neighbors for each query point.
-

Buffer k-d Tree Parallel NNS (Sub Routines)

Algorithm 2 FINDLEAFBATCH

Require: A sequence $i_1, \dots, i_M \in \{1, \dots, m\}$ of query indices.

Ensure: A sequence r_1, \dots, r_M of leaf indices.

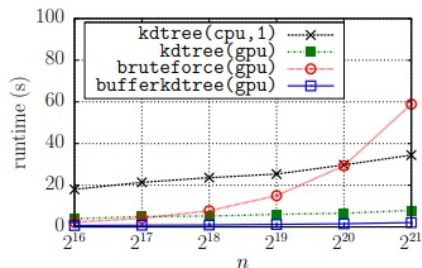
- 1: **for all** i_1, \dots, i_M **process k-d tree in parallel**
 - 2: Initialize stack for i_j (that stems from the previous call)
 - 3: Find next leaf index r_j for i_j (-1 if root is reached twice).
 - 4: **end for**
 - 5: **return** r_1, \dots, r_M
-

Algorithm 3 PROCESSALLBUFFERS

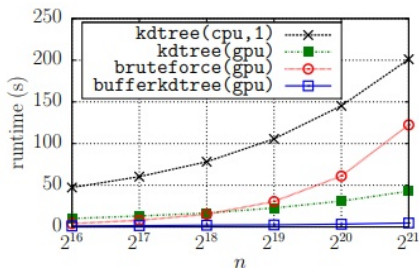
Ensure: A sequence $i_1, \dots, i_N \in \{1, \dots, m\}$ of query indices.

- 1: $I = \emptyset$
 - 2: **for** $j = 1, \dots, 2^h$ **do**
 - 3: Remove all query indices $i_1, \dots, i_{N(b_j)}$ from buffer b_j .
 - 4: **for all** $i_1, \dots, i_{N(b_j)}$ **do in parallel**
 - 5: Update nearest neighbors w.r.t. all points in the leaf associated with the buffer b_j .
 - 6: **end for**
 - 7: $I = I \oplus i_1, \dots, i_{N(b_j)}$ (concatenate indices)
 - 8: **end for**
 - 9: **return** I
-

Buffer k-d Tree Parallel NNS Benchmark



(a)



(b)

Increasing the number n of training patterns for (a) `psf_colors` and (b) `psf_model_mag` ($m = 10^6$, $k = 10$).

Contents

- 1 The Problem
- 2 Introduction
- 3 2D Orthogonal Range Search
- 4 K-d Trees and Nearest-Neighbor-Search in R^d
- 5 Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs
- 6 Efficient Delaunay Tessellation through K-D Tree Decomposition**
- 7 References

Delaunay Triangulation

A Delaunay tessellation :

- is fundamental geometric structures for converting a point set into a mesh. It's important in data analysis, and represents the geometry of a point set or approximate its density.
- has the added property that the mesh automatically adapts to the distribution of the points: cells are smaller where points are closer together and larger where points are farther apart. This property can be used for accurate density estimation or for deriving geometric statistics about cells that further inform features such as clusters and boundaries in the data.
- computing algorithm performs poorly when the input data is unbalanced.

Can we achieve better performance ?

Delaunay Triangulation

The time and space requirements to compute and store a mesh representation of those points, demand a parallel tessellation algorithm designed for distributed memory high-performance computing (HPC).

The parallel efficiency of such algorithms depends primarily on load balance:

- Perfectly balanced point distribution with the same number of points per processor can achieve near perfect speedup.
- Worst possible distribution is essentially serial, and can easily cause an algorithm to exceed available memory.

Efficient Delaunay Tessellation through K-D Tree Decomposition

We cannot control the nature of the input data, but we can control its distribution to processors.

Data distribution involves two steps:

- the global domain is first decomposed into 3D regions.
- the blocks are assigned to processors.

We implement a distributed **k-d tree** to decompose a 3D domain into irregular size blocks by recursively splitting alternating dimensions of the domain with approximately equal number of input points in each child block.

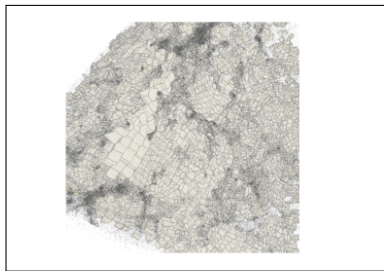
Efficient Delaunay Tessellation through K-D Tree Decomposition

Let's say, we have a late-stage cosmological simulations, where dark matter clumps into dense clusters called halos and vacates regions called voids.

In this case, the number of points that fall inside a block of a regular decomposition varies by orders of magnitude based on the number of clusters in the block.

The situation is exacerbated with increasing number of smaller blocks.

To cope with this problem, we can decompose the domain into a k-d tree.



Efficient Delaunay Tessellation through K-D Tree Decomposition

A k-d tree is built from a point set by finding the median of the points along the first coordinate, splitting the data into two halves based on the median, and recursively repeating the procedure on the two halves, cycling through the coordinates used for splitting.

By construction, the result is a balanced tree, where every block at a given level has the same number of points.

We decompose the domain in parallel into a prescribed number of blocks (assumed to be a power of two). Our blocks are 3D rectangular boxes containing a subset of the global input points. Blocks are processed by an MPI rank or a thread.

Contents

- 1 The Problem
- 2 Introduction
- 3 2D Orthogonal Range Search
- 4 K-d Trees and Nearest-Neighbor-Search in R^d
- 5 Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs
- 6 Efficient Delaunay Tessellation through K-D Tree Decomposition
- 7 References



Dmitriy Morozov, Tom Peterka, *Efficient Delaunay Tessellation through K-D Tree Decomposition*, 2016.



k - d trees: https://en.wikipedia.org/wiki/K-d_tree



Fabian Gieseke, Justin Heinermann, Cosmin Oancea, Christian Igel
Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs, 2014.



Yannis Avrithis, Ioannis Z. Emiris, and Georgios Samaras
High-dimensional approximate nearest neighbor: k-d Generalized Randomized Forests, 2016