Search. Approximate nearest neighbours search

Stanislav Protasov

Problem statement

1. Given 10^6 — 10^{11} documents in natural language
2. Given a query in natural language
3. Find a subset of documents which are *relevant** to the query
    a. **Boolean retrieval**. “relevance“ is 0/1 — document should include all words from query
    b. **Ranking**. “relevance“ is float 0..1

Applications: general and specific search engines, topic modelling, style analysis, ...

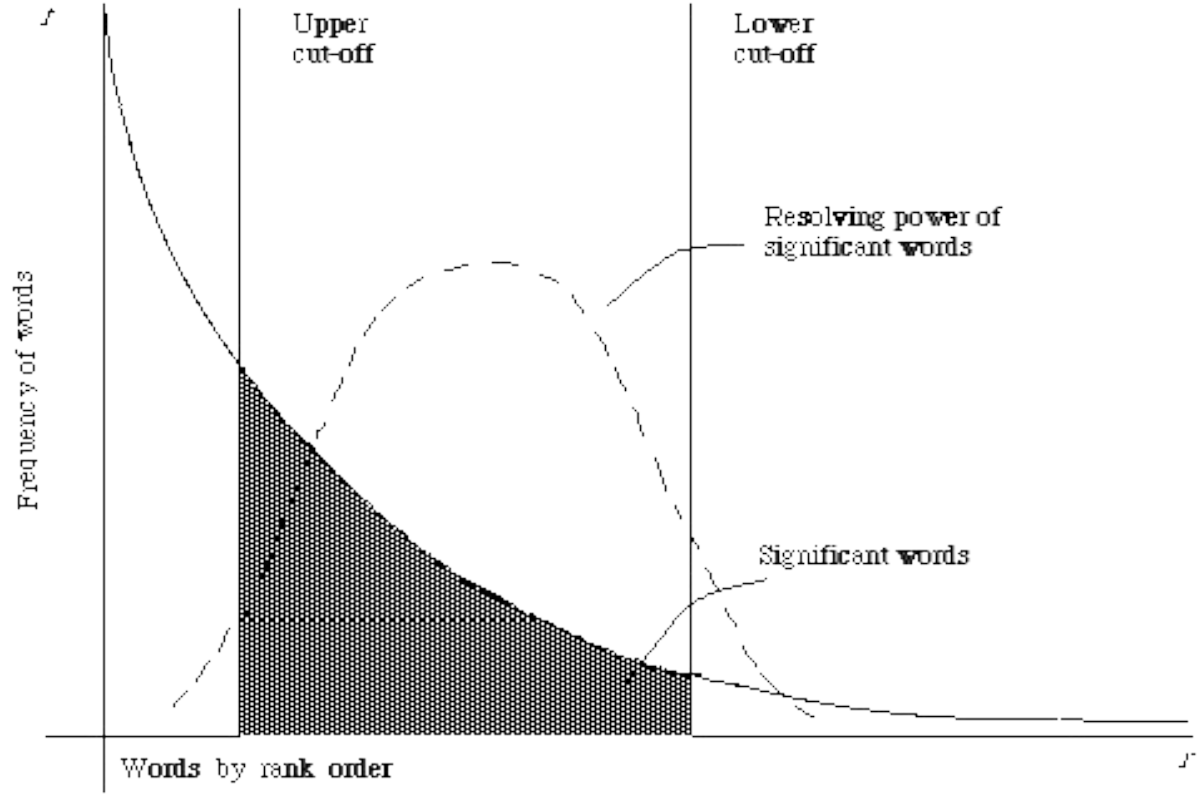
Inverted index

Techniques used across methods

- **case folding**: London = london; Лев = лев
- **Stemmer vs lemmer (lemmatizer)**:
  stemming: *compress* = *compress*ion = un*compress*ed
  бегу = бег
  vs
  lemmatization: better = good
  бегу = бегать
  Porter, Shawball RU, Natasha, SpaCy; pymystem3 RU
- ignore **stop words**: to, the, it, be, or, ...
  - Problems arise when search on “To be or not to be” or “the month of May” (stopword lists: RU, EN)
- **Thesaurus**: fast = rapid; лев = лёвушка
  - handbuilt clustering

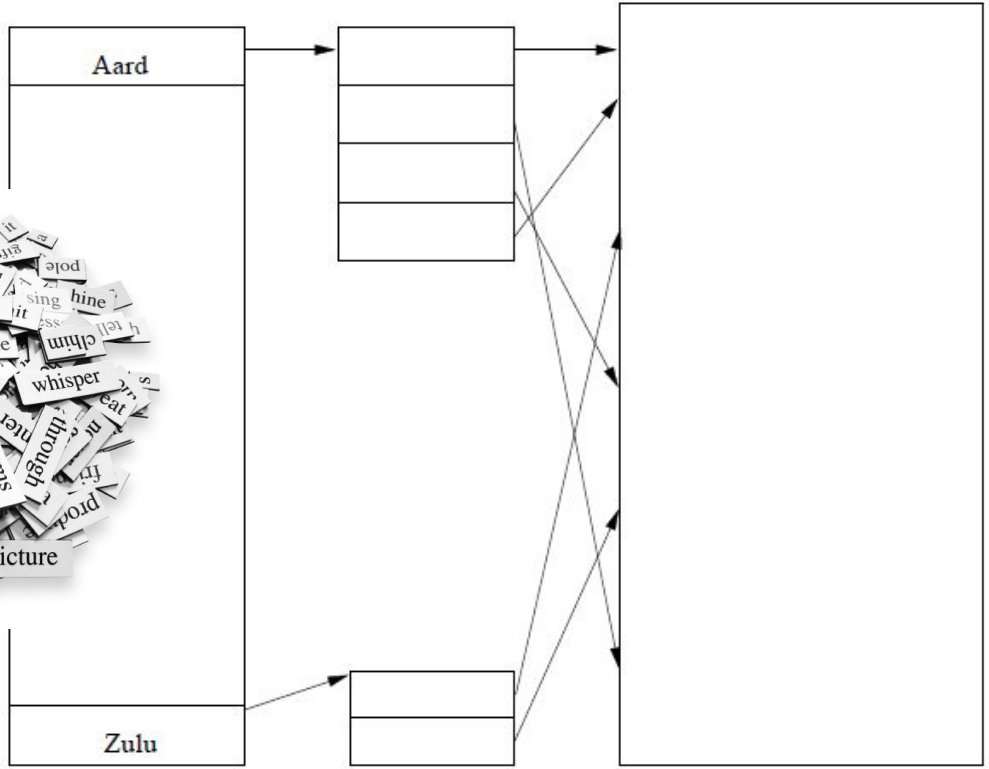
Document frequency

Idea: a term is more **discriminative** if it occurs only in **fewer** documents

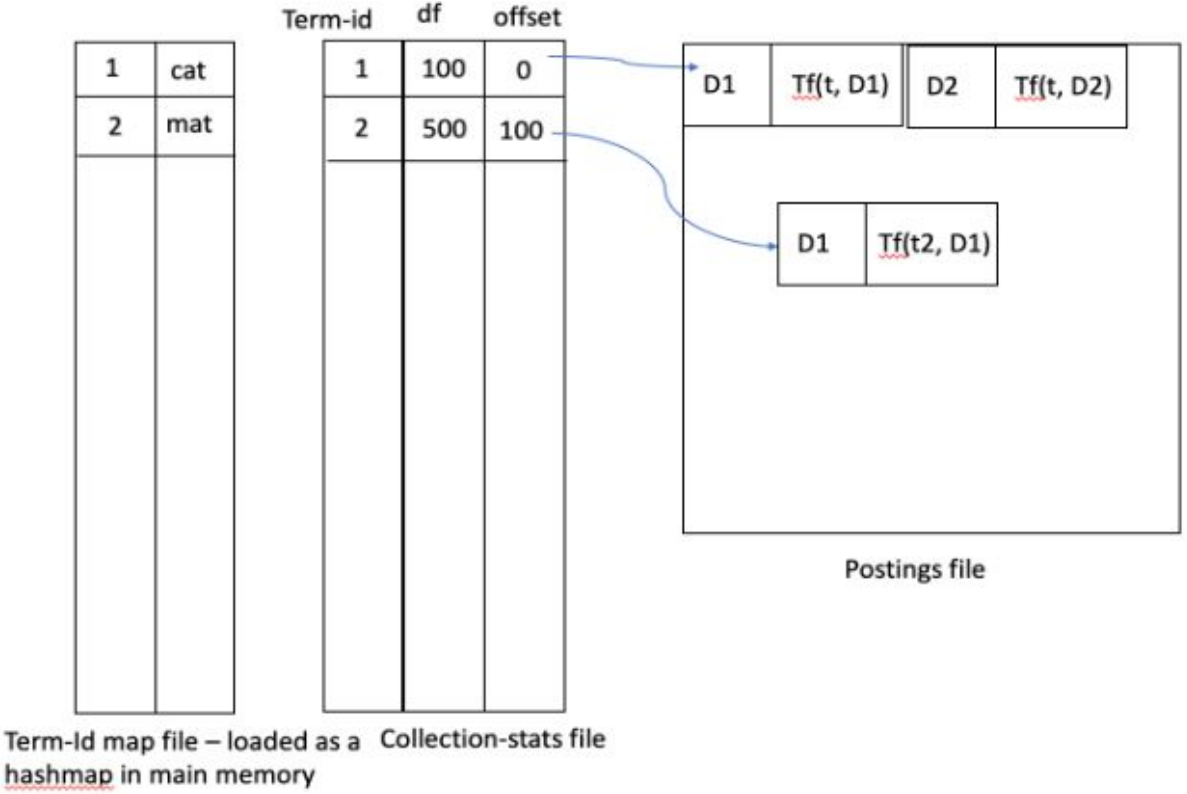


Lexicon and inverted index

```
33391  девочек 562 3811 3995 6172 3933 495
33392  кубического 4127
33393  славно 2288 2864 802 5634 2665
33394  бархатной 2562 616 5674
33395  бархатного 6260
33396  бархатном 6259 71 5070
33397  avertir 1121
33398  пустым 357 1224
```



Storing IVF (sparse matrix)



Agenda

- ANNS (not ANNs)
  - Clustering and IVF
  - Proximity graphs (NSW, HNSW)
  - Trees (second lecture)

Before we start...

What’s wrong with inverted index in terms of data structure?

Do you know the difference: O(N), $O_A(N)$, E(N)?

Approximate Nearest Neighbours Search

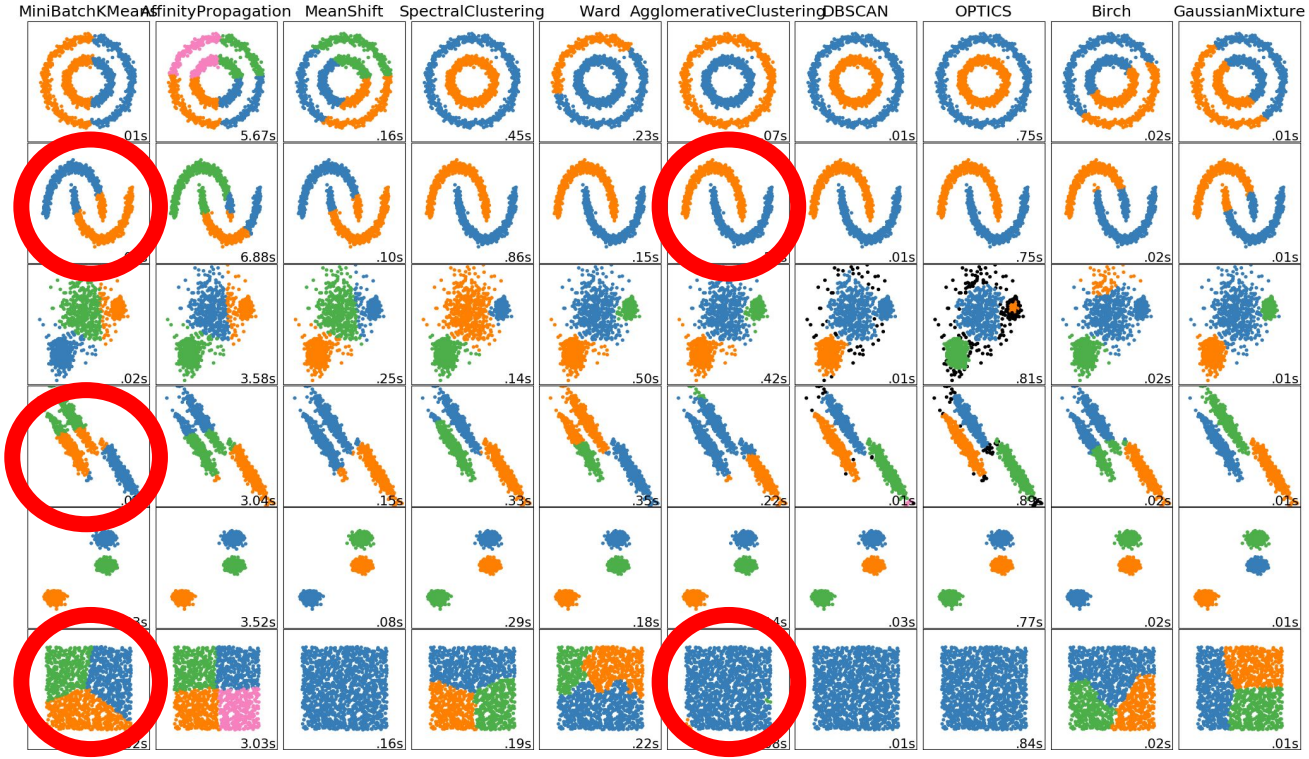
Approximation for k-NN search

1. Pre-select **k*c** elements from approximate neighbourhood (pre-ranking set).
2. Then select and re-rank relevant ones.

- Locality sensitive hashing
- **Search trees and supporting data structures**
- Vector compression, clustering, inverted indexing
- **Proximity graphs**

Hierarchical clustering and Inverted index revised

How clustering differ?



https://scikit-learn.org/stable/modules/clustering.html

Linkage criteria

- Single linkage (smallest distance) ~ DBSCAN
- Complete linkage (maximum distance)
- Minimum energy (variance grows slowly in we merge)
- Average distance and centroid-based approaches — kMeans

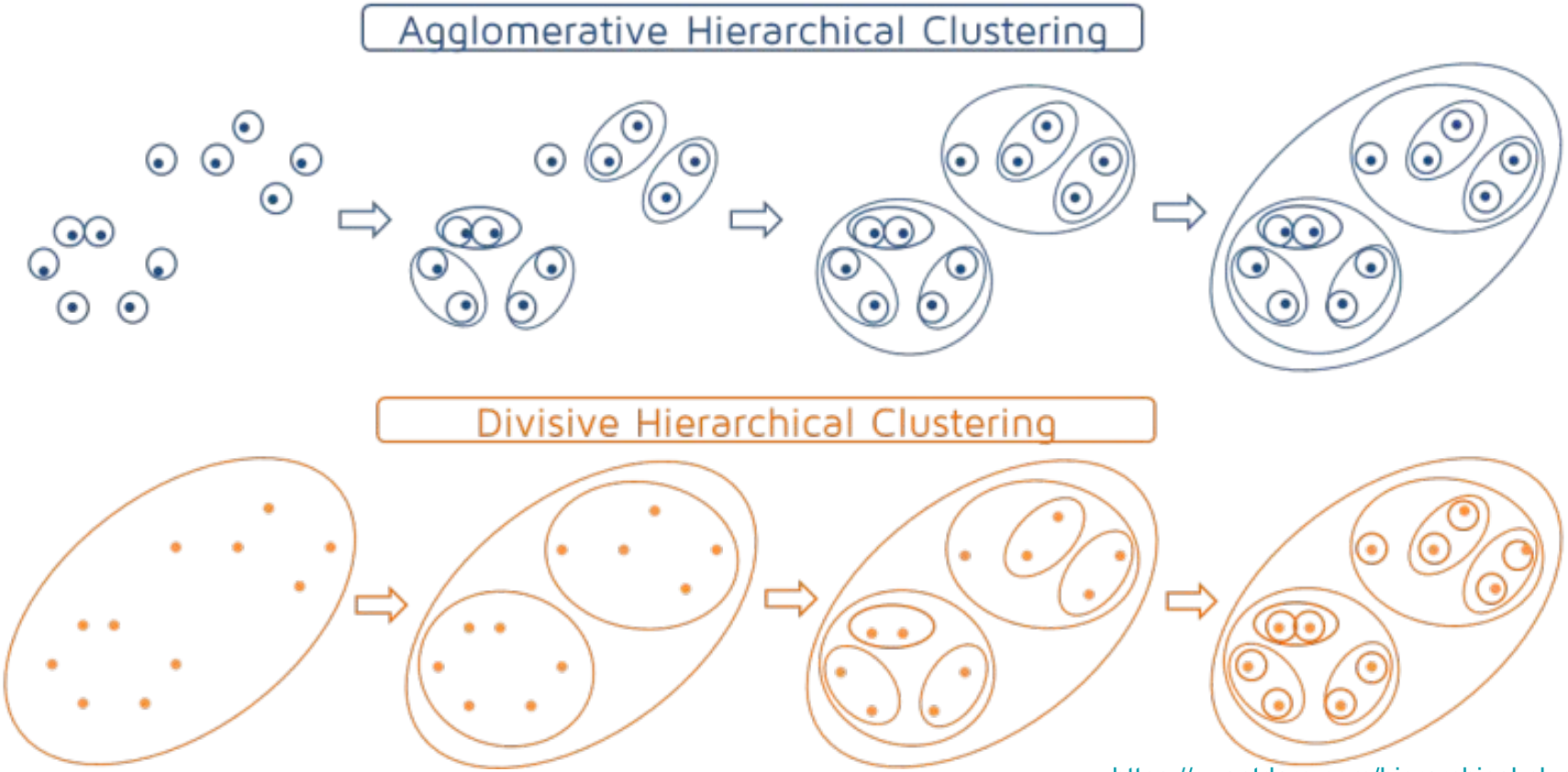
Why do we cluster?

For a flat list we run O(N) comparisons to find kNN

For $\sqrt{N}$ similar* clusters we can pick one closest** for $O(\sqrt{N})$ and find ***k*** NNs*** in $O(\sqrt{N})$.

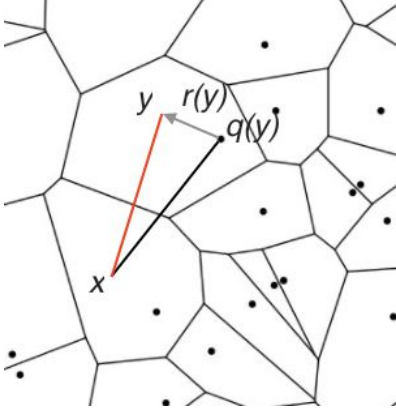
For two layers of $\sqrt[3]{N}$ …

How do we cluster?

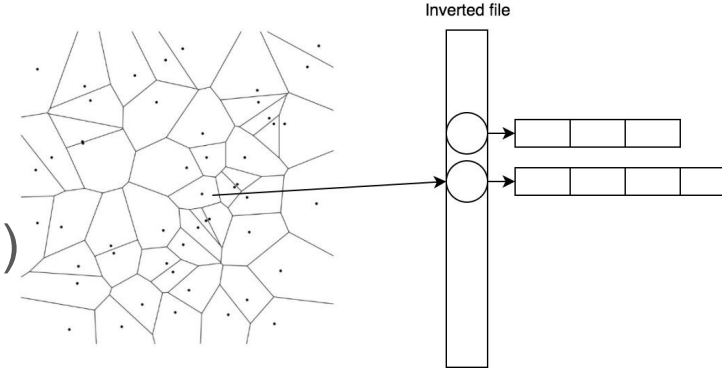


https://quantdare.com/hierarchical-clustering/

Revised IVF. FAISS (Facebook AI similarity search)

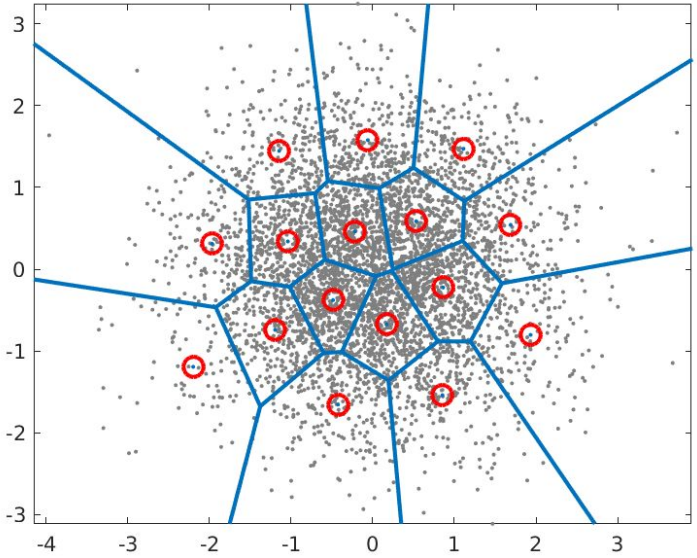


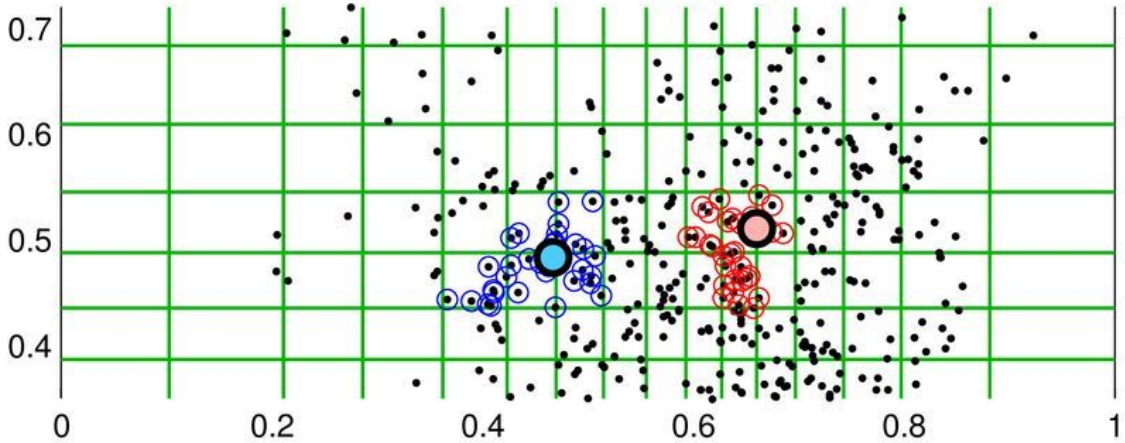
- Uses Voronoi diagram clusters (kMeans).
  Vectors are approximated with **centroids (VQ)**
- Build **inverted index** for points in clusters



- Vector compression: product quantizer (**PQ**)
  - Split R^{128} into 8 groups of 16 floats
  - Perform 256-means clustering of these “sub-vectors” and encode with 1 byte each

Vector Quantization and Product Quantization



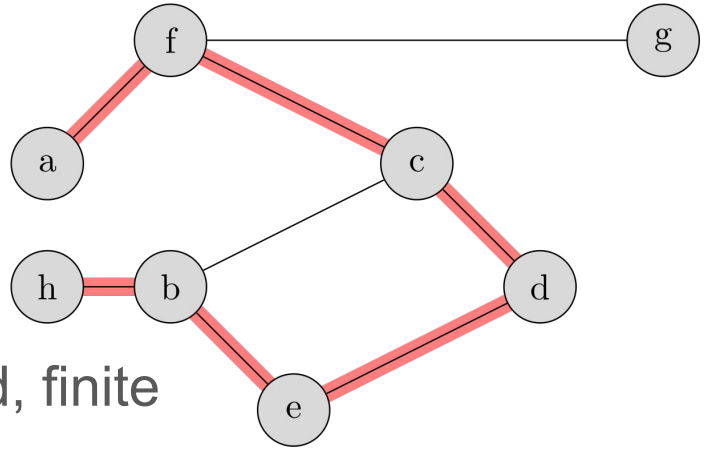


https://wiki.aalto.fi/pages/viewpage.action?pageId=149883153

https://arbabenko.github.io/MultiIndex/

Approach #2. Proximity graphs

Graphs cheat sheet



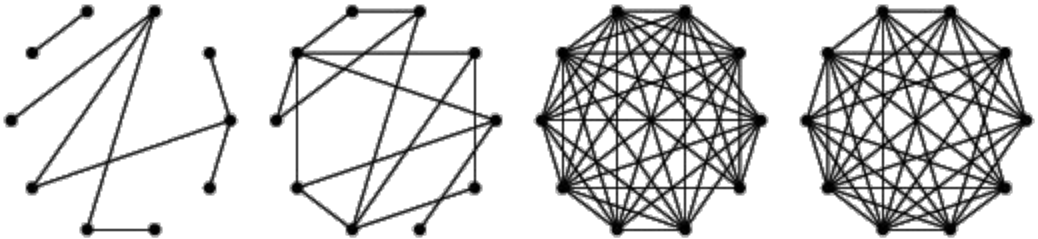
Graph - G = (V, E), can be weighted, directed, finite

[Simple] **path** - sequence of vertices and edges

Degree of vertex - number of incident edges

Graph diameter - longest shortest path between a pair of vertices

Random graph



Some random process (uniform, Gaussian, …) generates edges.

Almost every graph in the world. *Previously* considered as a model for social networks.

Small average shortest path - which is **good** for **search.**

Small clustering coefficient (defines how close are neighborhoods to cliques) - which is **bad** for **NN search.**

$$C(v) = \frac{e(v)}{deg(v)\,(deg(v)-1)\,/2}$$

$$\widetilde{C} = \frac{1}{N}\sum_{i=1}^{N} C(i)$$

Regular graphs

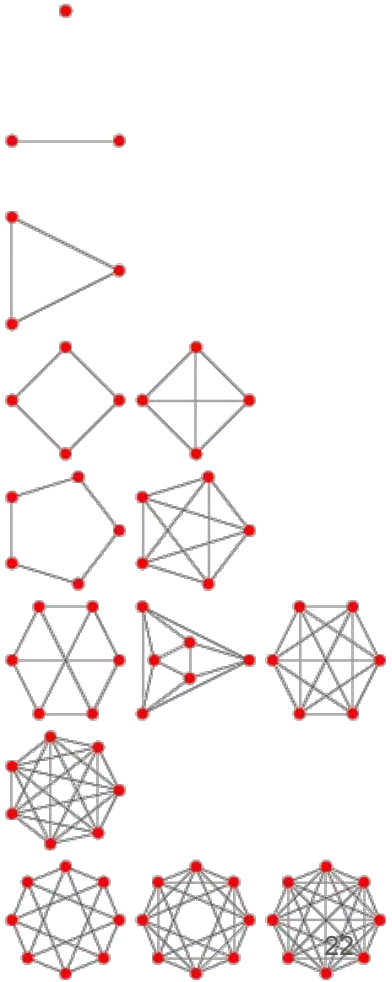
K-regular graph is a graph with **`deg(v) = K`** for any **`v`**.

Used to model big homogeneous networks.

Can also be random (as there are multiple K-regular graphs on the same size)

Big diameter - which is **bad** for **search**

Big clustering coefficient - which is **good** for **NN search**



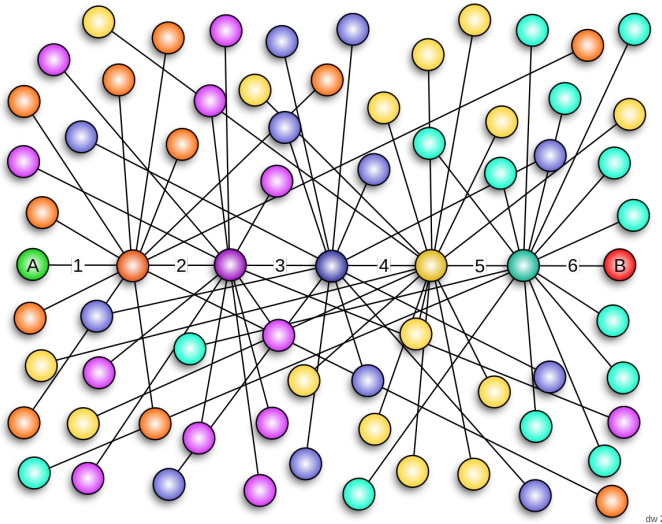
Small World experiment by Stanley Milgram, 1967

Initially it was considered, that social graph is kind of regular.

Experiment discovered (even with some questions to method) that even graph is **highly clustered**, **average path length is small.**

Was a basis for 6 handshakes rule.

New type of graphs was suggests: small world networks.



Small world network

Most vertices are not neighbours (small degree means *sparse* graph).

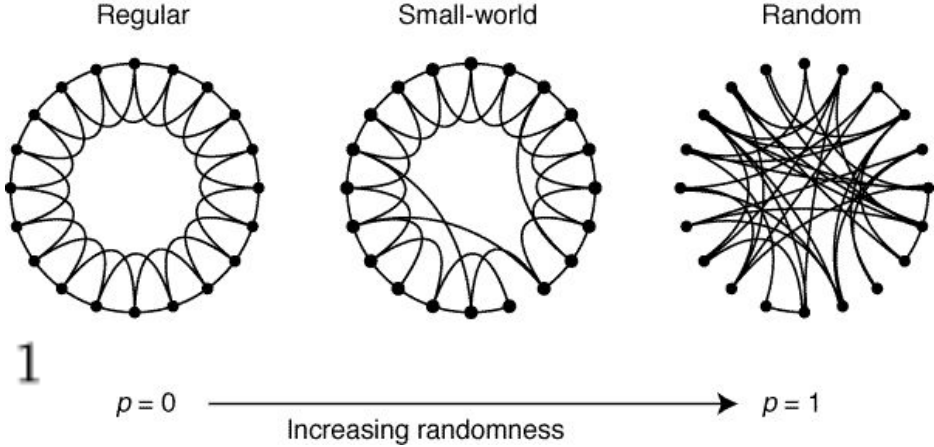
Nevertheless, small number of hops needed to reach any other node.

Typical path length ***L*** between 2 random nodes (of ***N***): $L \propto \log N$

Many real world networks are like this: internet, wiki, social graphs, power grids, brain cells. Although not all real networks like SW: many-generation networks, classmate graphs.

Watts–Strogatz model and **Kleinberg model** are how we describe and build SW networks

Watts–Strogatz model



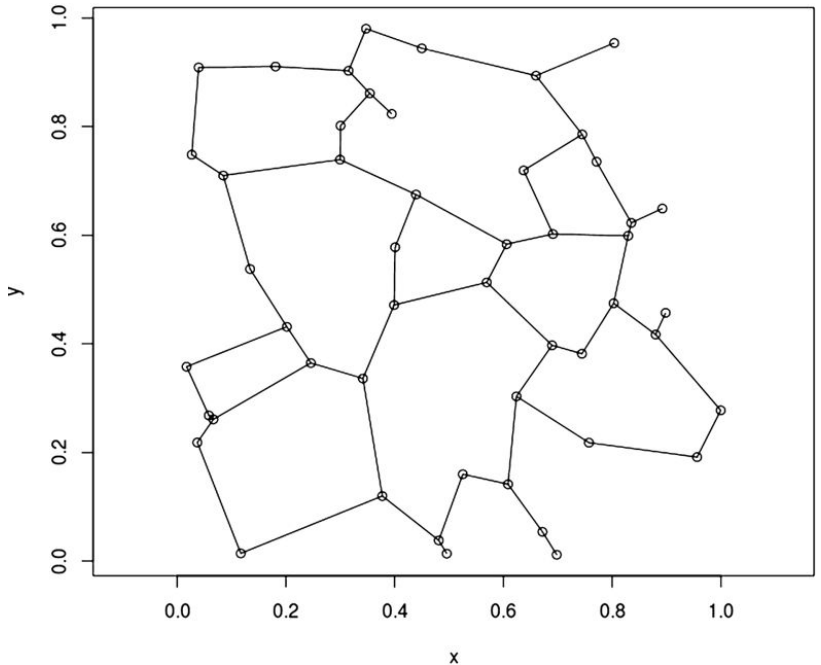
Given ***N*** nodes and ***K***-”regularity” (average degree ***K***) $N \gg K \gg \ln N \gg 1$

Given parameter ***p*** from [0, 1].

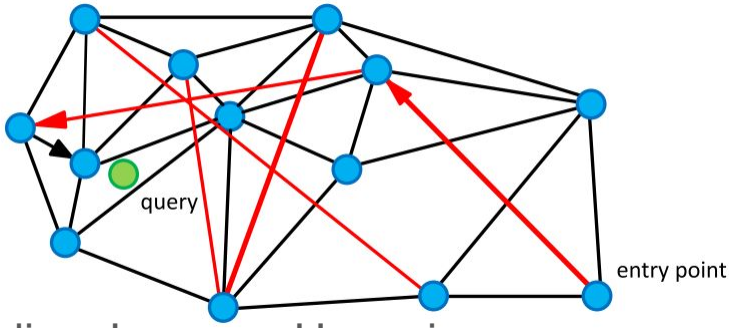
1) Construct a regular ring lattice.
2) take every edge connecting **vertex** to its **K/2 rightmost neighbors**, and rewire it with probability ***p***. Rewiring is done by **replacing destination** with vertex ***k*** (chosen **uniformly** at random from all possible nodes while avoiding self-loops and duplication).

Proximity graphs

A proximity graph is a simply a graph in which two vertices are connected by an edge if and only if the vertices satisfy particular geometric requirements.



Navigable small world networks



Idea is similar to skip-lists.

We can also measure **distance** (e.g. dot product, Euclidian, L_k-norm, Humming, Levenstein, …) between *query* and *current vertex*. Originally *Delaunay graph* needed to converge for exact search, but ANNS allows other small-world graphs.

Building:

1. One-by-one insertion via kNN search. Distant edges are created in the beginning.

Search:

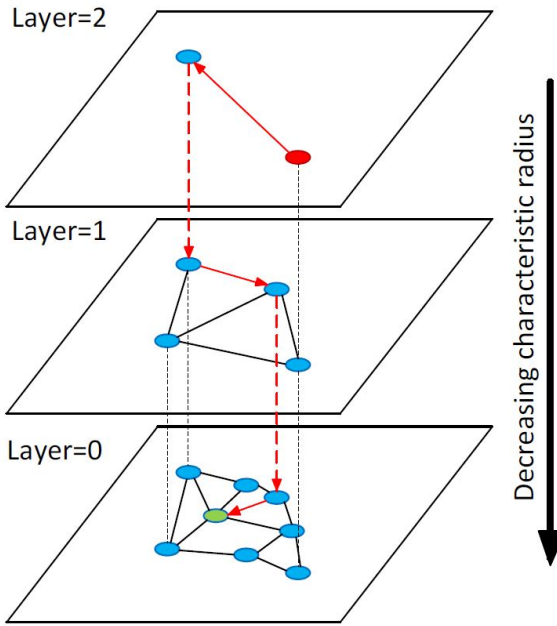
1. Perform greedy search. Move to the neighbour vertex **closest to query**
2. Update NN set on each step until it converges

Hierarchical navigable small world (github)

Layer 0 holds complete NSW network

Ideas:

- Better **start search** from a node with **high degree**
- Higher layer has longer links (skip-list!)
- Decrease layer size exponentially



Highlight:

1) **search procedure requires only `dist(u, v)` function**
2) **No embedding, hyperplanes, centroids of whatever needed**

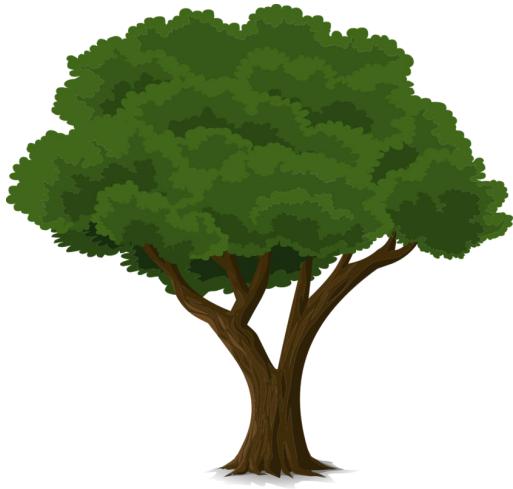
To read

An Introduction to Proximity Graphs

Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs

Forests of search trees

Stanislav Protasov



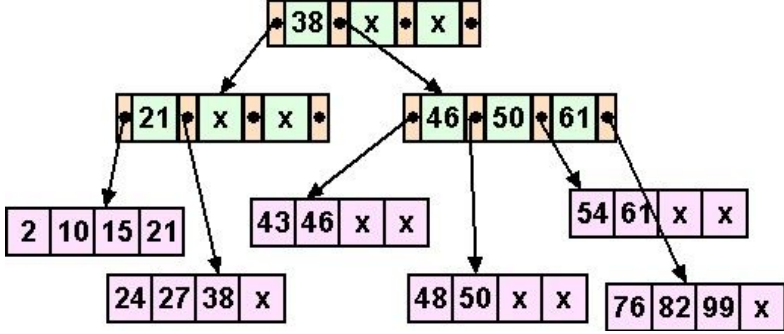
Agenda

ANNS with trees:

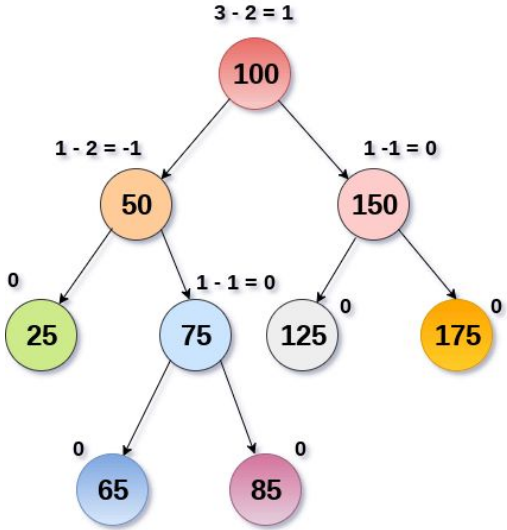
- Search trees
- Quad trees
- KD-trees and Ball Trees
- Annoy
- And some others

Search Trees

Refresher for [B]ST



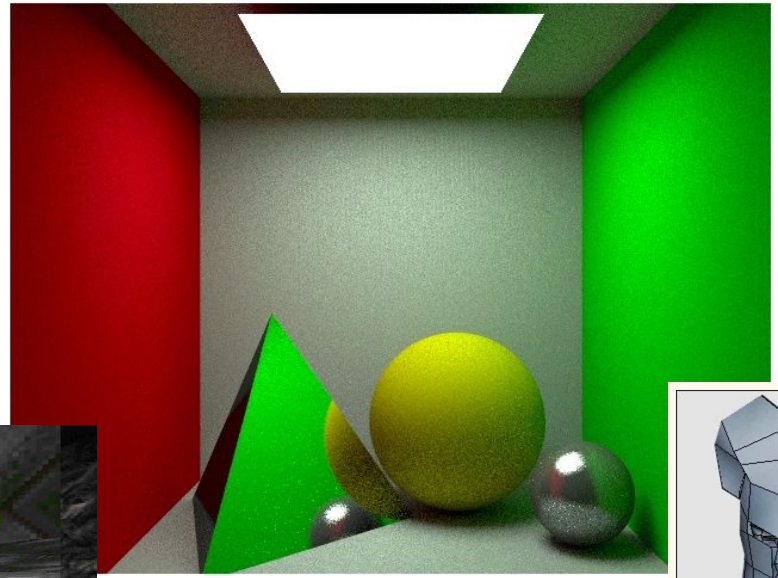
- K-ary (usually binary) trees
- Built upon comparable keys (scalars)
- Similar search procedure
- Preserved balance property,
  ensures O(log(N)) max path length
- Can be *homogeneous* (AVL)
  and not (*B+ tree*)

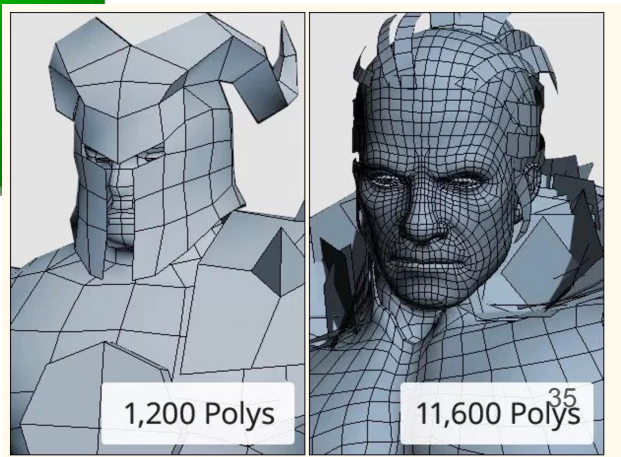


AVL Tree

But what if we have vectors?

Originated from Computer Graphics



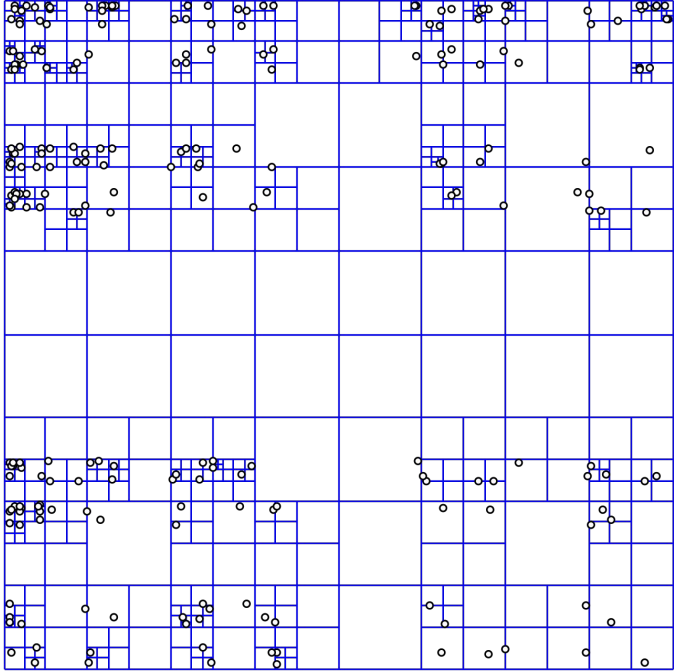


Trivial case: vector is a scalar

- Binary search trees:
  - Splay, RB, AVL trees are best for RAM
- N-ary search trees:
  - B-trees, LSM-trees are used with hard drives
- **Search:**
  - **Exact search is O(log(N))**
  - **K nearest neighbour search O(log(N) + K)**
  - **Range search O(log(N) + K)**

QuadTree (1974)

- Forms of Quad Trees:
  - Region
  - Point
  - Edge
  - Polygon



- All forms of quadtrees share some common features:
  - decompose space into **adaptable** cells
  - Each cell (or bucket) has a **maximum capacity**.
    When maximum capacity is reached, the bucket **splits**

QuadTree search

```
function queryRange(range) {
  pointsInRange = [];
  if (!this.boundary.intersects(range))
    return pointsInRange;

  for (int p = 0; p < this.points.size; p++) {
    if (range.containsPoint(this.points[p]))
      pointsInRange.append(this.points[p]);
  }
  if (this.northWest == null) // no children
    return pointsInRange;

  pointsInRange.appendArray(this.northWest->queryRange(range));
  pointsInRange.appendArray(this.northEast->queryRange(range));
  pointsInRange.appendArray(this.southWest->queryRange(range));
  pointsInRange.appendArray(this.southEast->queryRange(range));
  return pointsInRange;
}
```

QuadTree insertion #1

```
function insert(p) {
   if (!this.boundary.containsPoint(p))
     return false; // object cannot be added
   if (this.points.size < QT_NODE_CAPACITY && northWest == null) {
     this.points.append(p);
     return true;
   }
   if (this.northWest == null) this.subdivide();

   if (this.northWest->insert(p)) return true;
   if (this.northEast->insert(p)) return true;
   if (this.southWest->insert(p)) return true;
   if (this.southEast->insert(p)) return true;
}
```

QuadTree insertion #2

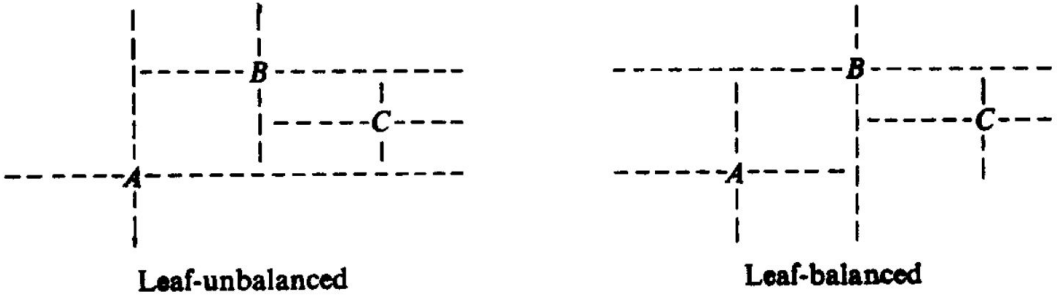


Fig. 2. Single balance

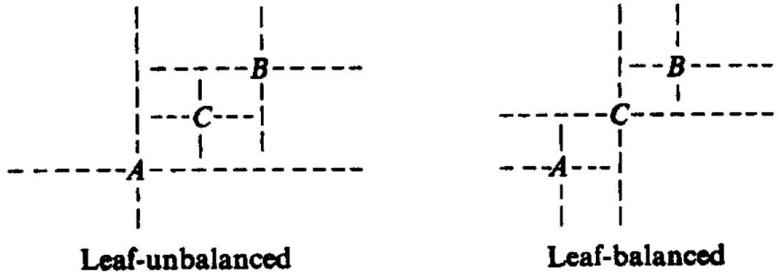


Fig. 3. Double balance

Table 2. Data on leaf-balanced insertion

QuadTree deletion

<<… In fact, it seems that ***one cannot do better than to reinsert all of the stranded nodes****, one by one, into the new tree. This answer is not very satisfactory, and it is a matter of some interest whether there exists any merging algorithm that works faster than* `n log n`*, where* `n` *is the total number of nodes in the two trees to be merged…>>*

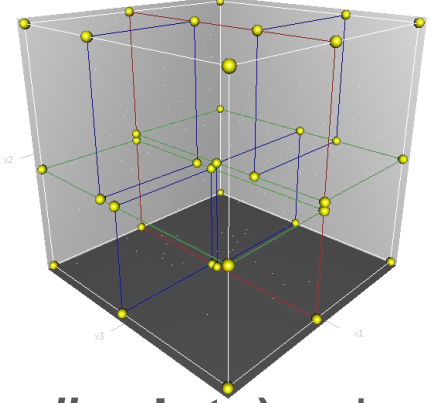
QuadTree optimization

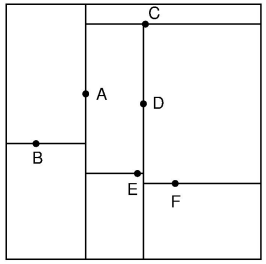
By an **optimized tree** we will mean a quad tree such that every node K has this property: **No subtree of K accounts for more than ½** of the nodes in the tree whose root is K.

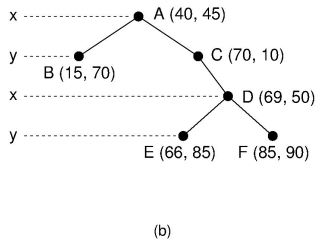
A simple recursive algorithm to complete optimization is this: Given a collection of **lexicographically ordered records**, we will first find one, **R**, which is to serve as the root of the collection, and then we will regroup the nodes into 4 subcollections which will be the four subtrees of R. The process will be called recursively on each subcollection… No subtree can possibly contain more than half the total number of nodes

Can you see any suboptimality?

K-d trees (1975)



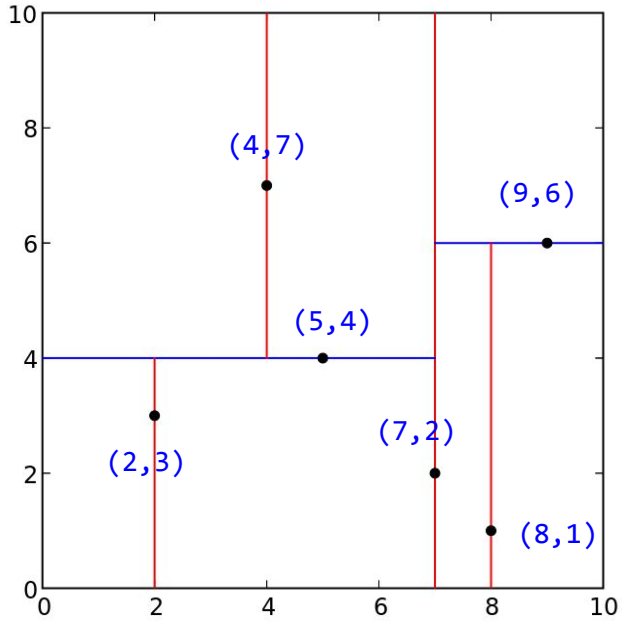


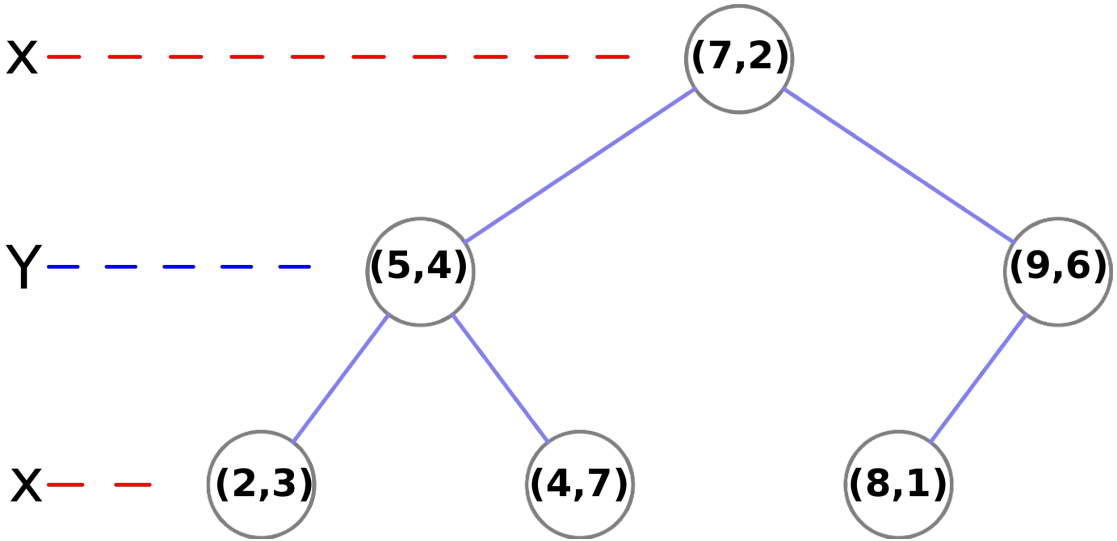


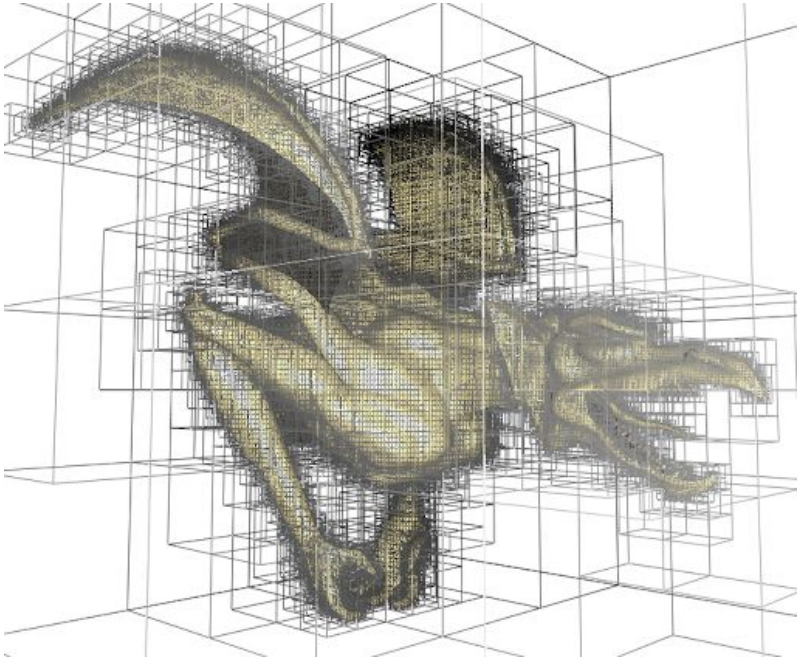
Ideas:

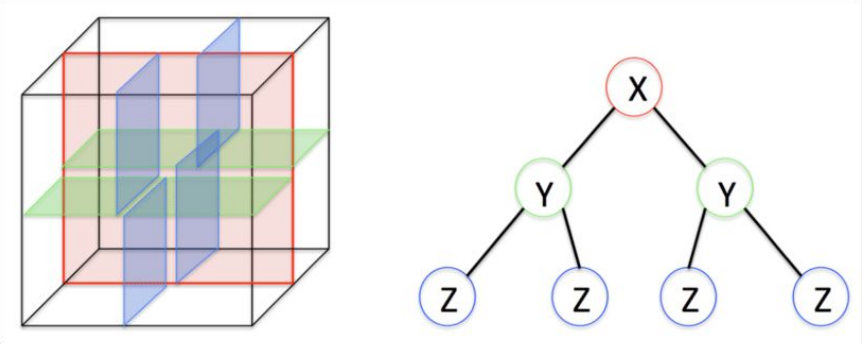
- Split points in 2 **equal (by #points)** subspaces, not 4
- Use **alternating coordinates** at each level
  (`x, y, z, x, y, z, ...`)
  - Thus, we need 2 levels to encode quadrants, but they are **equal**
  - And yes, this allows us to have **more than 2 dimensions**
- **Cool demo**

K-d trees: building example








X
Y
Y
Z
Z
Z
Z

K-d trees

Construction (“homogeneous”):

```
def buildKDTree(vectors, dim=0):
    if not vectors:
        return None      # stop condition, e.g.
    if len(vector) == 1:
        return Node(vector[0])
    vectors.sort(key = lambda x: x[dim])  # or Selection alg for O(N)
    med = len(vectors) // 2               # this will work only for no dups!
    left, med, right = vectors[:med], vectors[med], vectors[med+1:]
    node = Node(med)
    node.left = buildKDTree(left, (dim + 1) % K)
    node.right = buildKDTree(right, (dim + 1) % K)
    return node
```

K-d trees characteristics

Is built in $O(n(k + log(n)))$ time

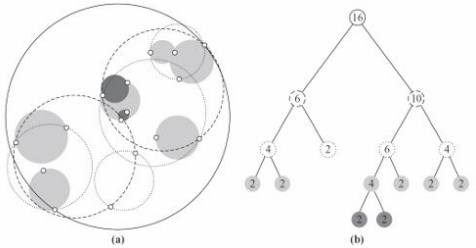
Requires O(*kn*) memory (at most node for a point)

Runs range search for $O(n^{1-\frac{1}{k}} + a)$ where *a* — result size

Runs 1-NN search in O(log(*n*)) time

To build hyperplanes it requires vector representation of keys

Ball Trees



Ball Trees and KD-trees are used sklearn implementations of all nearest neighbour tools.

Construction of Ball Tree is almost the same as for KD-tree, with one addition: at each step for pivot element we **compute a radius.**

Thus, we can utilize radius value in kNN search:

```
if distance(t, B.pivot) - B.radius ≥ distance(t, Q.first) then continue; [wiki]
```

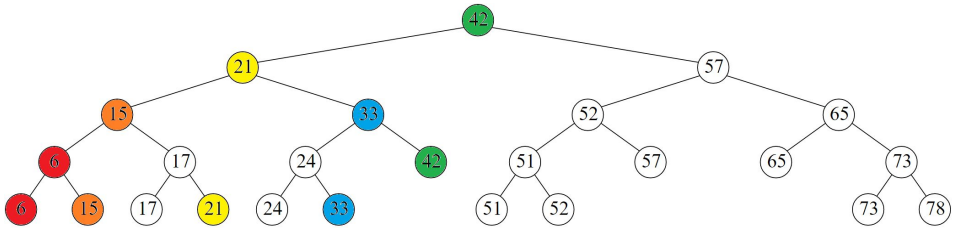
Here you can read about sophisticated construction algorithms.

PS In original algorithm you consequent dimensions, but dimension of biggest variance.

Faster range queries - range trees (1979)

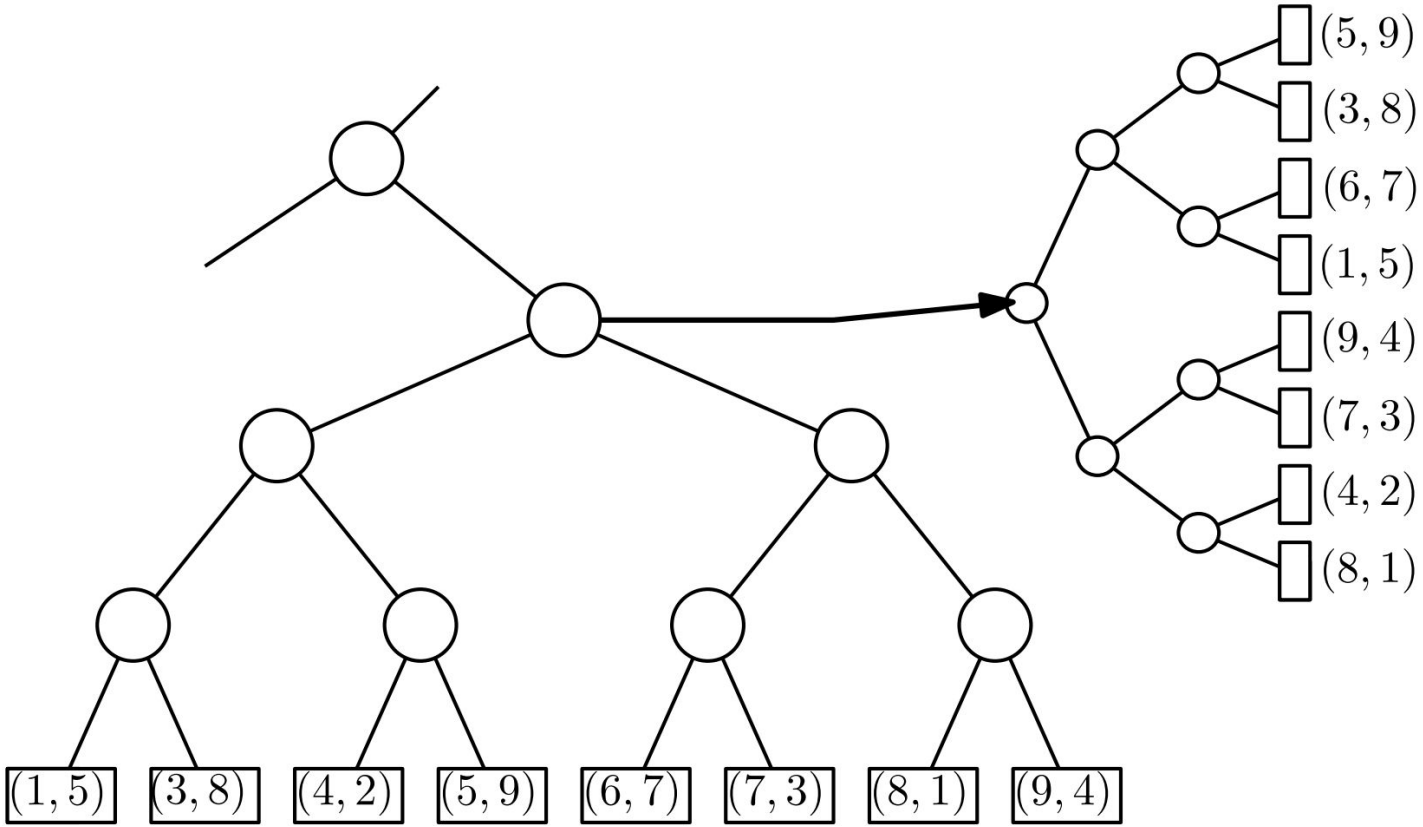
Ponts are in the leaves.

For **1-dimensional** case: **balanced non-homogeneous binary search tree** on those points. Internal nodes store predecessors (largest to the left)



Range trees in **higher dimensions** are constructed recursively by constructing a balanced binary **search tree on the first coordinate** of the points, and then, for **each vertex v** in this tree, constructing a **(d−1)-dimensional range tree** on the points contained in the **subtree of v**

Image source link



Sorting and looking for median is soooo boring...

Johnson-Lindenstrauss lemma

… **low-distortion embeddings** of points from high-dimensional into low-dimensional Euclidean space. The lemma states that a set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that **distances between the points are nearly preserved**. (Random projections).

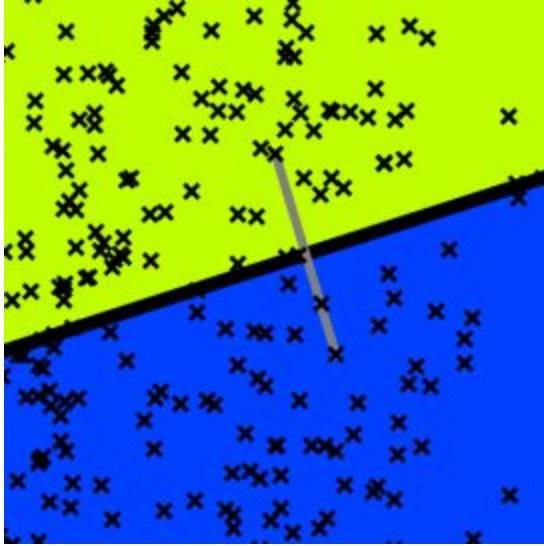
Given $0 < \varepsilon < 1$, a set X of m points in $\mathbb{R}^N$, and a number $n > 8\ln(m)/\varepsilon^2$, there is a linear map $f : \mathbb{R}^N \to \mathbb{R}^n$ such that

$$(1-\varepsilon)\|u-v\|^2 \leq \|f(u)-f(v)\|^2 \leq (1+\varepsilon)\|u-v\|^2$$

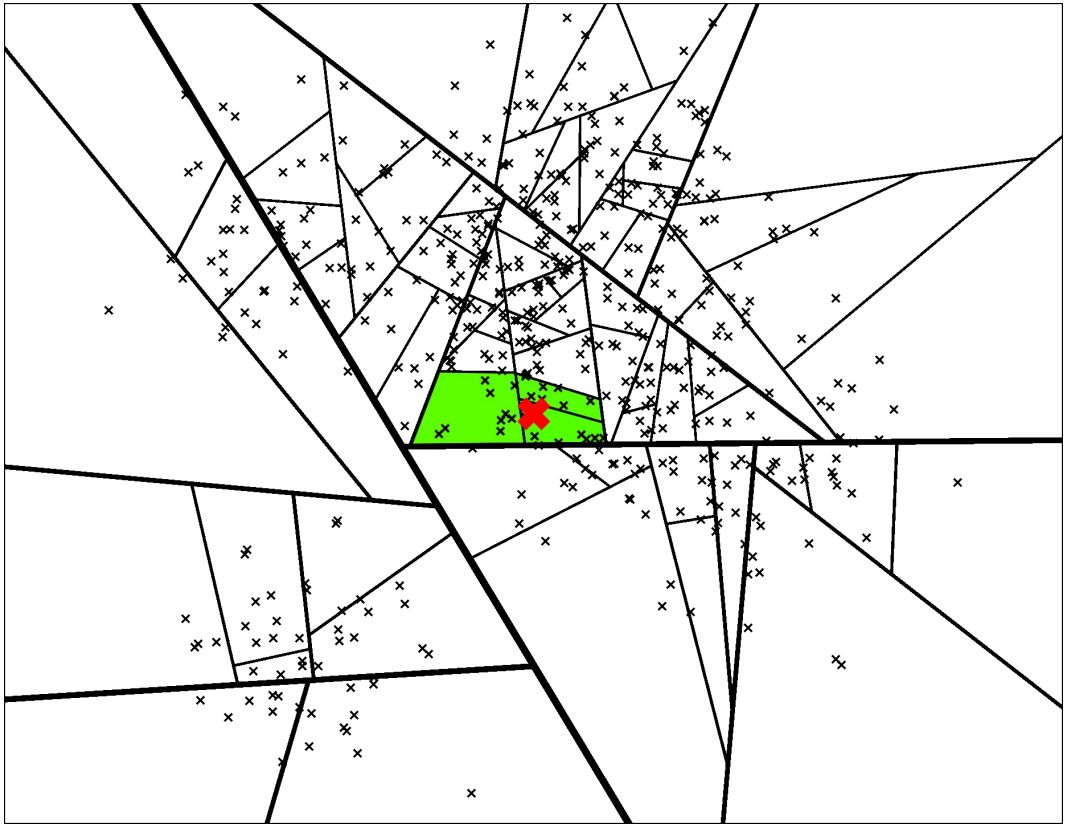
for all $u, v \in X$.

Annoy from Spotify (2015)

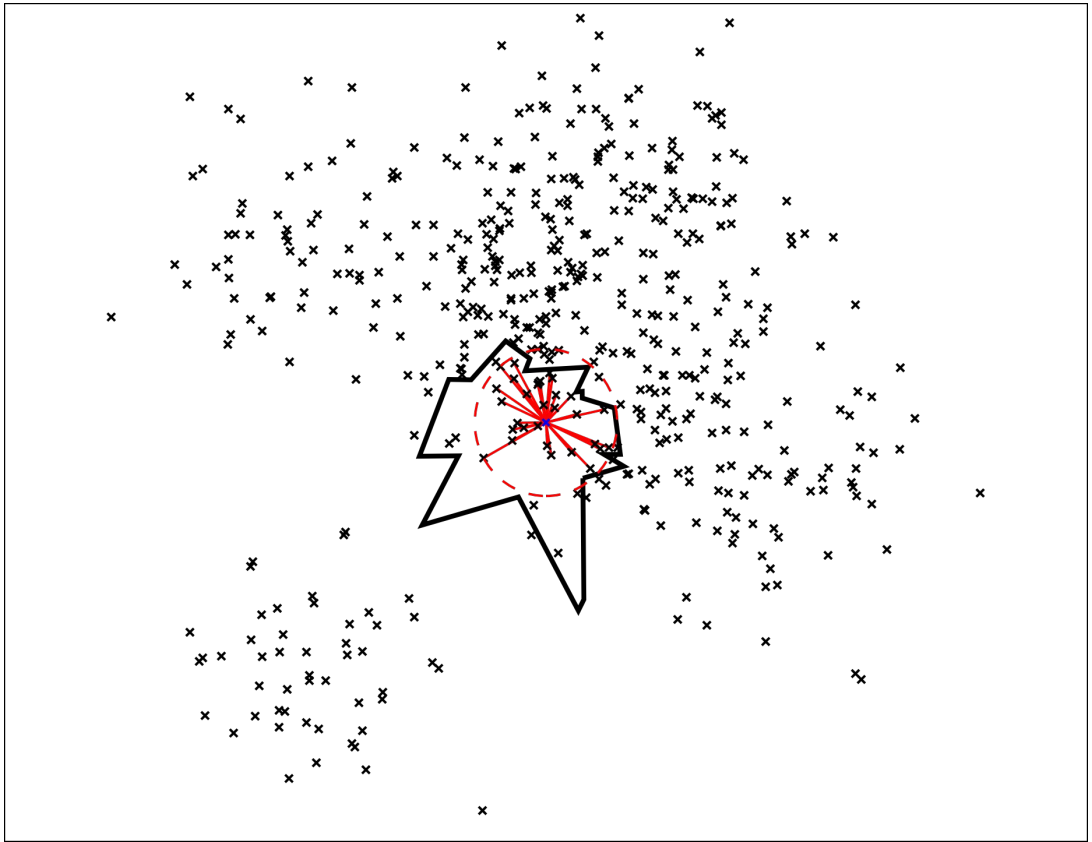
1. Instead of looking for a median, **select equidistant hyperplane for 2 random points** - then split is done in linear time (random projection)
2. Use "**soft threshold**" that allows traversing "wrong" branches for ANNS
3. Build **multiple search trees** over the same dataset (*compare to multiple searches in NSW*)
4. Generalization of binary space partitioning (BSP-tree) used in CG (Doom, Quake, ...) for visibility sorting.



Multiple trees (animation)



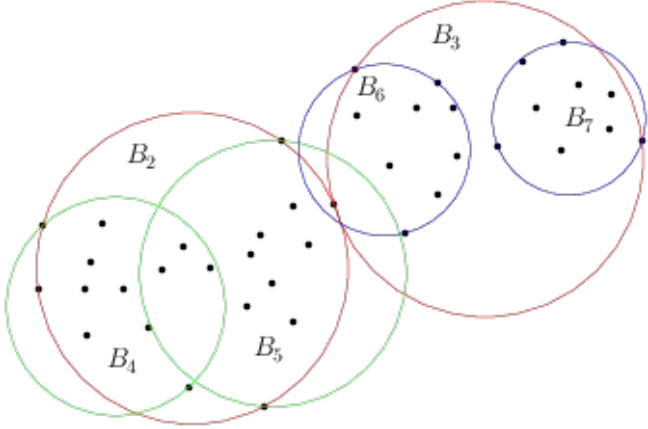
ANNS results with Annoy

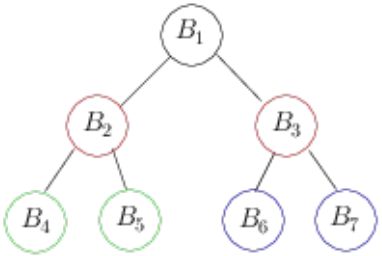


Oh no, I don’t have vectors!
Metric space

Vantage-point (VP) trees (1991)

Instead of dividing space by a plane, we can divide if by a **sphere** (or nested spheres, recursively). **Sphere** requires only **center** (one of dataset points) and **radius** (which can be estimated in any **metric space**). Radius is selected to split points into equal parts.






```
function Select_vp(S)
    P := Random sample of S;
    best_spread := 0;
    for p ∈ P
        D := Random sample of S;
        mu := Median_{d∈D} d(p, d);
        spread := 2nd-Moment_{d∈D} (d(p, d) − mu);
        if spread > best_spread
            best_spread := spread; best_p := p;
    return best_p;
```

```
function Make_vp_tree(S)
    if S = ∅ then return ∅
    new(node);
    node↑.p := Select_vp(S);
    node↑.mu := Median_{s∈S} d(p, s);
    L := {s ∈ S − {p} | d(p, s) < mu};
    R := {s ∈ S − {p} | d(p, s) ≥ mu};
    node↑.left := Make_vp_tree(L);
    node↑.right := Make_vp_tree(R)
    return node;
```

SEARCH

Ok, you must be lost...

All those trees recursively split the space into similar size parts

Quad Tree - works in R^2 only. Each node splits space into 4 non equal quadrants.

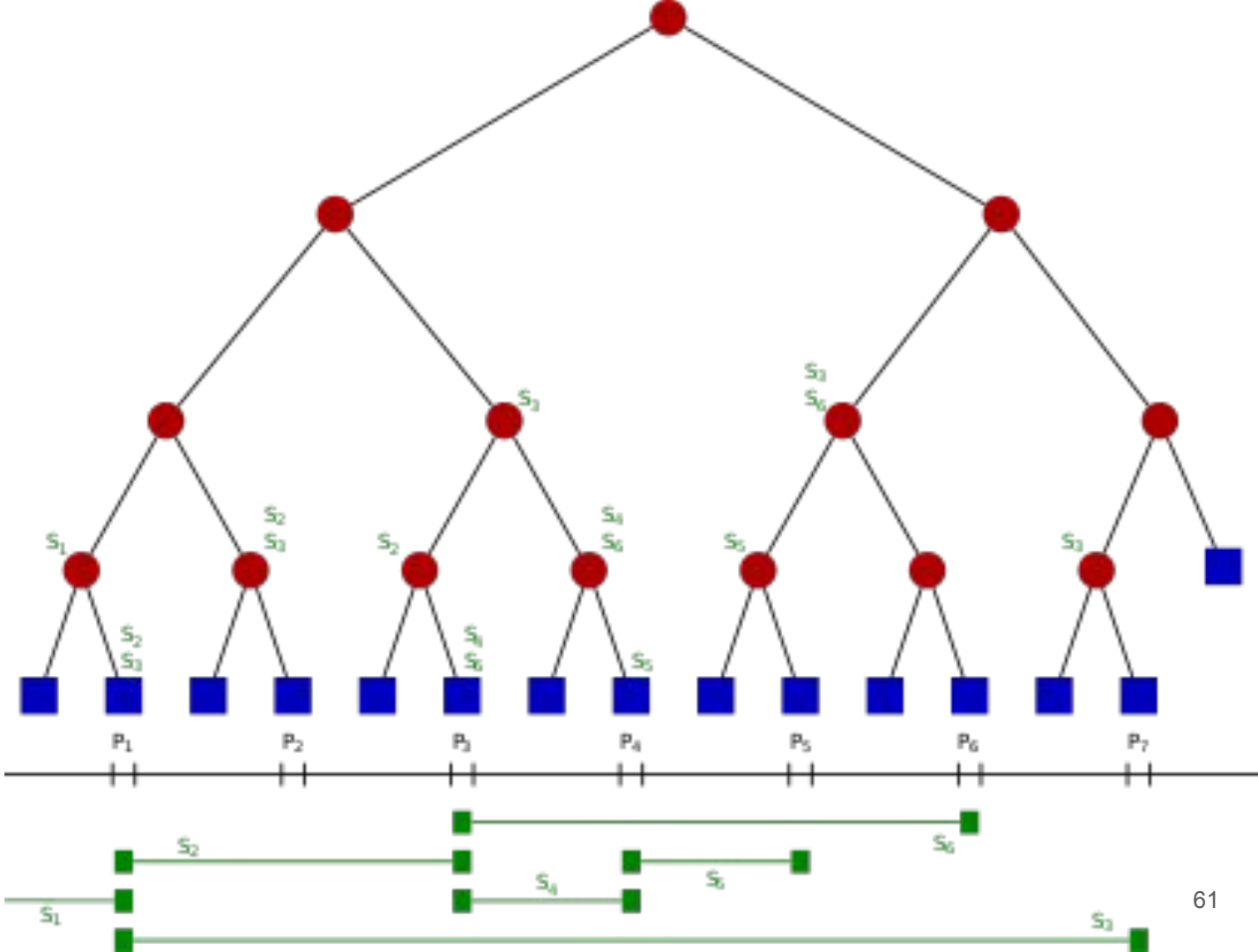
K-d Tree - works in R^K. Each node splits space into 2 equal parts.

Annoy - works in R^K. Instead of sorting and finding median - uses random separating hyperplanes. But compensate with multiple trees

Vantage-point tree - works for any metric space. Instead of hyperplanes uses spheres.

Offtopic: interval and BSP trees. When object is not a point

Interval tree



Interval tree

Tree that **holds intervals** and allows to search fastly which of them overlap the query (point or interval).

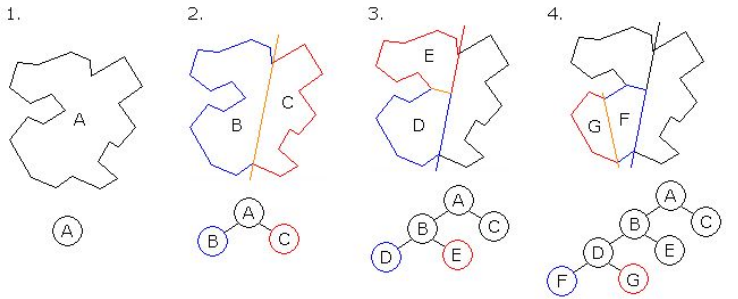
`Construction(L):`

1. You have a list of intervals ***L***.
2. By X_{center} split all intervals into “left”, “intersecting”, “right” lists.
3. Store “intersecting” in current node in 2 lists (sorted by start and by end).
4. Run `Construction(“left”)` and `Construction(“right”)` intervals.

`Search(p, node):`

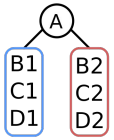
1. Compare p to node.X_{center}
2. Use sorted list in node to find intersecting
3. Go `Search(p, node.[left|right])` with respect to [1]

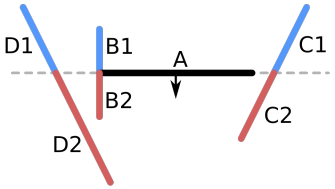
BSP-tree

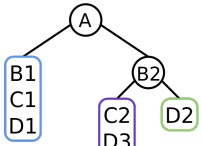


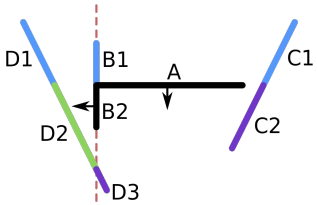
To store polygons in a list:

- Choose a polygon *P* from a list *L*.
- Make a node *N*, and add *P* to the list of *N*.
- For each other polygon Q in the list:
  - If *Q* is <u>in front of</u> *P* plane, move Q to the list L_F "***in front of P***".
  - If *Q* is <u>behind</u> *P* plane, move Q to the list L_B "***behind P***".
  - If *Q* <u>intersects</u> *P* plane, **split** it into two polygons and move them to the respective lists.
  - If that polygon lies in the plane containing *P*, add it to the **list of *N***.
- Apply this algorithm to L_F and L_B.









See also

M-trees

R-trees and R*-trees

Octree

…