Vision par Ordinateur :
Indexation et recherche d'images

Database indexing

Thanks to Laurent Amsaleg and Hervé Jégou for some of this material

Context: Similarity of Images

- Comparing description data (feature vectors) instead of images directly
  - Typically high-dimensional vectors
  - Vectors define points in high-dim spaces
- The similarity between images is in proportion to the similarity of their feature vectors
- Two images are said to be similar if their descriptors are close in the high-dim space
- Everything relies on a metrics between vectors
  - often a distance or a similarity
  - all dimensions involved

Context: Image retrieval

- Goal:
  - Given a feature vectors database,
  - Given a query image described by a set of query vectors

  ⇒ retrieve the closest feature vectors of the database:
  k-ppv or *ε-sphere*

  ⇒ return a **ranked** list of images:
  Voting mecanism

Two types of Searches

- Type 1 : K-nn search
  - searching for K nearest neighbors
  - result set of fixed size
  - near does not means close
- Type 2 : ε search
  - searching within a "ball":

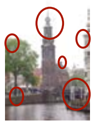
    $$\|p - p_i\| \le \varepsilon$$

  - bounds dissimilarity
  - result set of unpredictable size

Toy Example

- 1 query image

- Database: 2 images I_1 et I_2



- Description:
  - local descriptors of dimension 3
  - Distance L_1

Toy Example

1. Number of distances to compute?
2. Ranked list of similar images?

- Query descriptors:

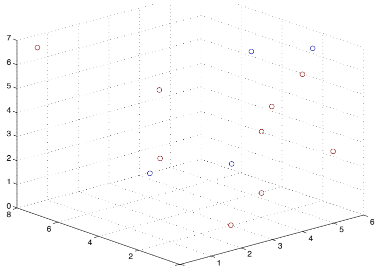
	x1	x2	x3
q1	5	5	1
q2	3	6	1
q3	5	4	6
q4	5	1	7

- DB descriptors:

	x1	x2	x3
d11	4	2	1
d12	0	7	7
d13	6	3	5

	x1	x2	x3
d21	5	0	3
d22	5	3	4
d23	6	5	2
d24	0	1	7
d25	4	7	1
d26	3	2	0

Toy Example



Toy Example

Computation of 36 distances L_1 between descriptors:

	q1	q2	q3	q4
d11	4	5	8	8
d12	13	10	9	11
d13	7	10	3	5
d21	7	10	7	5
d22	5	8	3	5
d23	2	5	6	10
d24	15	14	9	5
d25	3	2	9	13
d26	6	5	10	10

For each query vector:

⇒retrieve the closest feature vectors of the database:
k-ppv or *ε-sphere*

⇒ return then a **ranked** list of images:
Voting mecanism

Naïve, straightforward approach

- [Compute all the descriptors]
- Descriptors are all stored in a file
  - one after the other, sequentially
- Search Process:
  - read the file (large chunks at once)
  - compute distance between query and descriptors coming from disk
  - keep track of the K nearest neighbors (for ex)
- Exhaustive search, sequential

Naïve, straightforward approach

- Piece of cake!
- But very costly:
  - fetch data from disks (lots of I/Os)
  - compute distances over ALL data
- Not very realistic when dealing with:
  - very large volume of data
  - high dimensionality
  - complex metrics (EMD)

Python time: exhaustive search

- Generate a random matrix of 10 descriptors of dimension 5
- Given a query vector:
  - Use FLANN to retrieve the 3 k-nn
- Given several query descriptors:
  - Use FLANN to retrieve the 3 k-nn
- Print execution time to retrieve the 50-nn for d=500, and:
  - 1 query, N=10,000
  - 1 query, N=100,000
  - 1 query, N=1,000,000
  - 100 queries, N=10,000
  - 100 queries, N=1,000,000

How can we accelerate the search?

We need…

- To reduce the volume of data to analyze during searches
- To enclose searches
- To efficiently access data and return answer
- How? By structuring the descriptors

Doing this fast requires multidimensional indexing

How Can We Index Data?

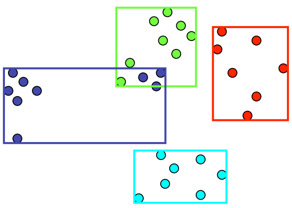
- Enclose the search
  - descriptors live in a high-dimensional space
  - analyze only interesting regions of space
- Key Idea:
  - group descriptors into **cells**
    - much fewer cells than descriptors
  - detect useless cells and ignore them
    - the ones containing descriptors that can not be part of the final result
  - fine-grain analysis of remaining cells
    - compute distances using the descriptors they contain

Table of Contents

Why Relying on Cells?

- First, compute distances from query to cells
- Many cells can be ignored without accessing any vector they contain
  - much fewer cells than descriptors
  - low complexity
  - simply need geometrical information on cells
  - great response time gains

Geometry Helps

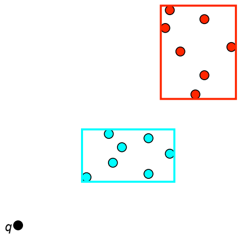


- Here is a data collection
- Searching for the 4 NN of a query point

Exercise Time…

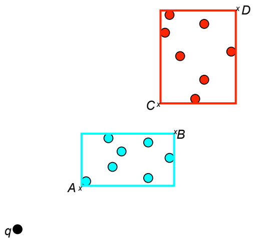
- Goal:
  - find simple geometric rules allowing to decide whether the contents of a particular cell might or might not be useful for answering a particular query
- Hints:
  - it's based on minimum and maximum distances from the query point to cell boundaries

Ready?



- Searching for the 4 NN of q
- blue cell obviously sufficient
- How can we eliminate **red cell**?

Go!



4 simple distances:

d(q,A) = min dist to blue cell

d(q,B) = max dist to blue cell

*d(q,C) = min dist to **red cell***

*d(q,D) = max dist to **red cell***

Question 1

Which pair of distances is useful?

qA & qB

qA & qC

qA & qD

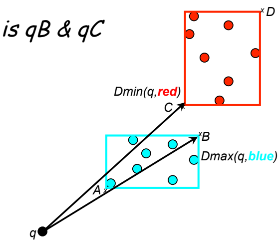
qB & qC

qB & qD

qC & qD

Ta daaa…

answer is qB & qC

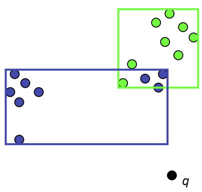


- *can reject **red** without analyzing its contents*

Your turn: more difficult (?)

- What happens in this case?
  - still searching for the 4 NN of q

Question 2



Can we prune green?

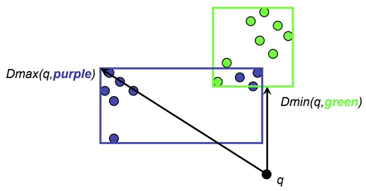
*Can we prune **purple**?*

*Can we prune **both**?*

Can we prune none?

Ta da daaa…

- Searching for the 4 NN of q



***Purple** is really close, so analysis needed*

cells overlap

Can not reject green. Its contents needs analysis

Answer= can prune none

Geometry Helps

- Compute distance to all cells
- if $dmin(q,C_i) \geq dmax(q,C_j)$ then reject C_i
  - few cells: few distance calculations
  - very few candidates: few distance calculations
  - great gain in performance!
  - make sure you have k nn

Geometry Helps Again!

- Previous rule computes a list of candidate cells
- List ordered w.r.t. increasing distance to q
  - fetch first cell
- Process all vectors in the current cell
  - update list of nearest neighbors
- if $dmin(q,C_i) \geq d(q,nn_k)$ then stop search
  - else fetch next cell; loop

Indexing =

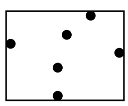
- Design algorithms for building cells
- Design data structures to organize cells
- Design algorithms traversing data structures efficiently

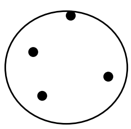
Table of Contents

1. Data indexing: why we need cells?
2. How to construct cells?
3. High-dimensional weirdness
4. Approximate searches

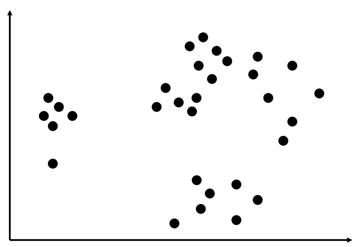
Grouping Descriptors into Cells?

- Two broad approaches for cell construction
  - based on distances between descriptors
  - based on high-dim space partitioning
- Information stored with cells
  - position in space, size, centre, population, …
  - Usually: minimum bounding rectangle/sphere

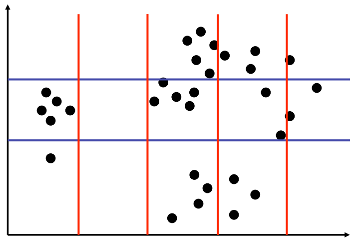




example : 2d feature space



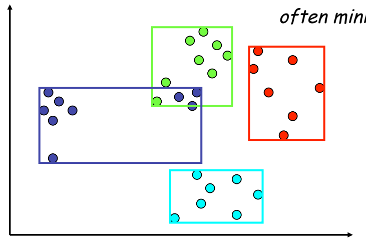
Grouping according to the partitioning of the feature space



Grouping according to data proximity

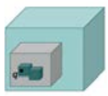
often rectangle or spheres

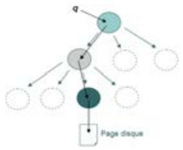
often minimum bounding



Multidimensional Indices

- Traditional indices for DBMS: trees

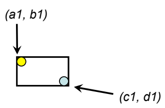




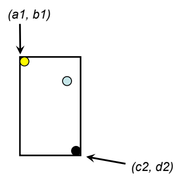
- Many indexing schemes have been designed to handle multidimensional data
- Overview of the 2 seminal approaches
  - R-Tree: data proximity
  - KD-Tree: space partitioning

R-Tree

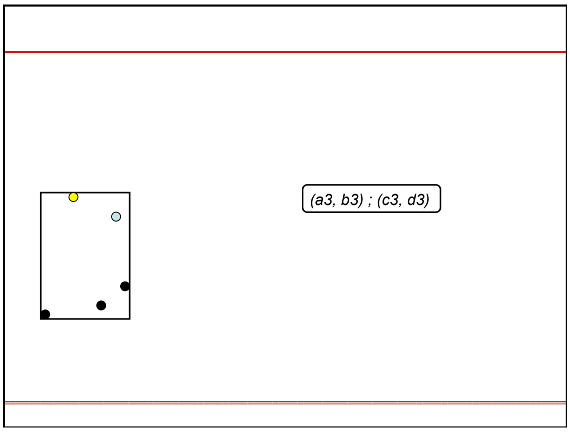
- Guttman, 1984, ACM SIGMOD
- B-Tree extended to high-dim spaces
  - nodes & leaves = hyper-rectangles
- Dynamic insertion of vectors
- Leaves (fixed size) group vectors
- Leaves split on overflow
  - update parent
- Balanced tree

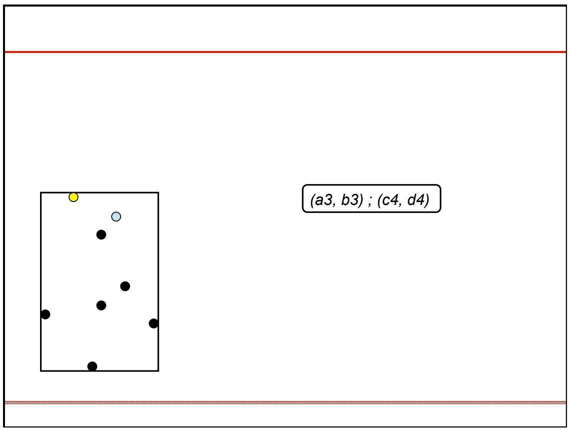


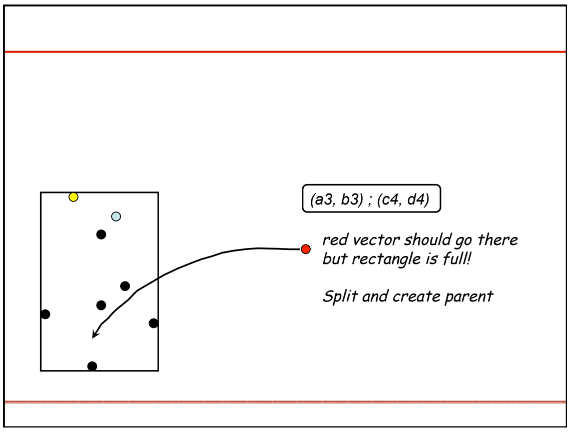
(a1, b1) ; (c1, d1)

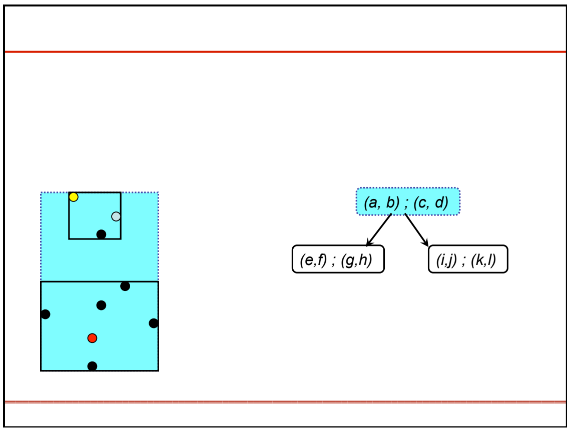


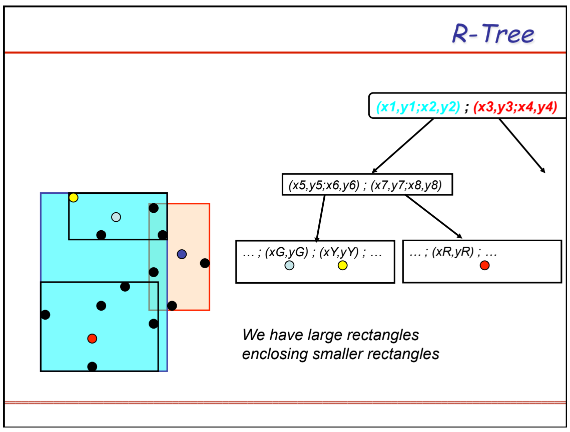
(a1, b1) ; (c2, d2)

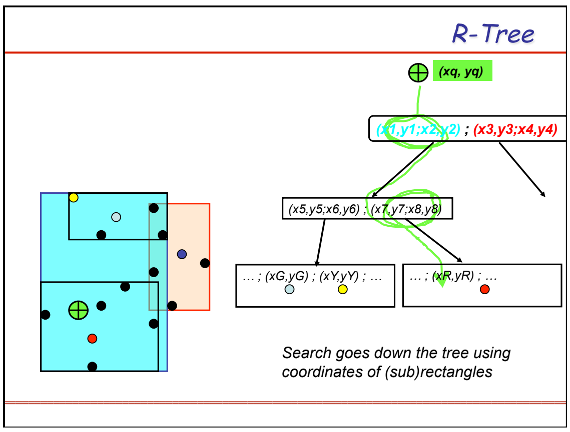

(a3, b3) ; (c3, d3)


(a3, b3) ; (c4, d4)


(a3, b3) ; (c4, d4)
red vector should go there
but rectangle is full!
Split and create parent

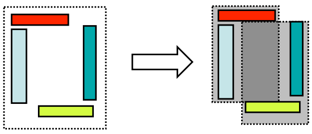

(a, b) ; (c, d)
(e,f) ; (g,h)
(i,j) ; (k,l)


R-Tree
(x1,y1;x2,y2) ; (x3,y3;x4,y4)
(x5,y5;x6,y6) ; (x7,y7;x8,y8)
… ; (xG,yG) ; (xY,yY) ; …
… ; (xR,yR) ; …
We have large rectangles
enclosing smaller rectangles


R-Tree
(xq, yq)
(x1,y1;x2,y2) ; (x3,y3;x4,y4)
(x5,y5;x6,y6) ; (x7,y7;x8,y8)
… ; (xG,yG) ; (xY,yY) ; …
… ; (xR,yR) ; …
Search goes down the tree using
coordinates of (sub)rectangles

Building One R-Tree

- Nodes and leaves have a fixed size
  - leaves are on disk
  - often a multiple of I/O granule (128kb)
- Nodes and leaves must split on overflow
  - many options for splitting
  - always problematic: causes overlap



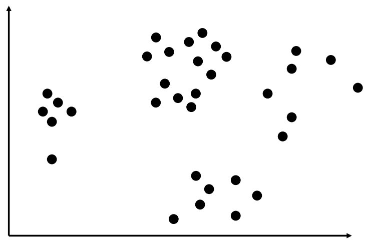
R-Tree

- Minimum Bounding Rectangles
- Overlapping exists
- Tree structure makes traversal fast (log)
- At the roots of almost all indices having cells based on the proximity of their points
  - SS-Tree: hyperspheres
  - SR-Tree: hyperrectangles ∩ hyperspheres
  - …

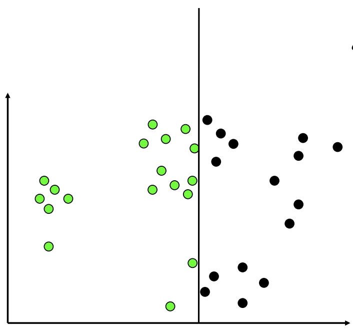
Kd-Tree

- Bentley, 1975. Communication of the ACM.
- This is a space partitioning strategy
- Space split using hyperplanes
  - perpendicular to one axis
  - splitting point = median of points on that axis
  - pick next axis

Kd-Tree

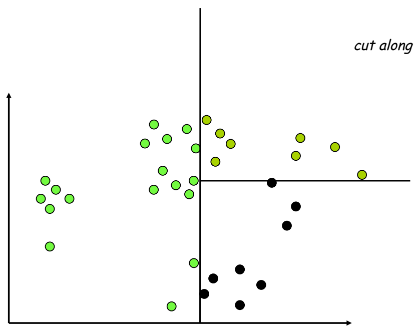


Kd-Tree



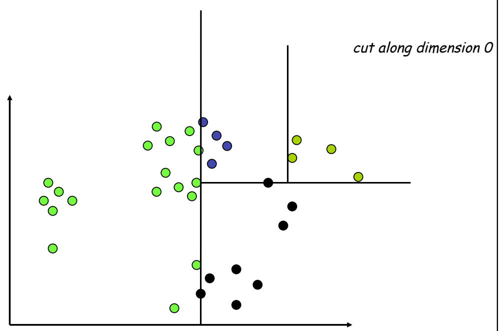
cut along dimension 0

Kd-Tree

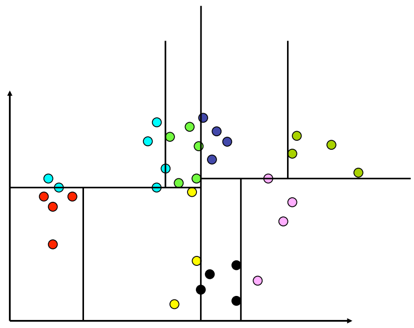


cut along dimension 1

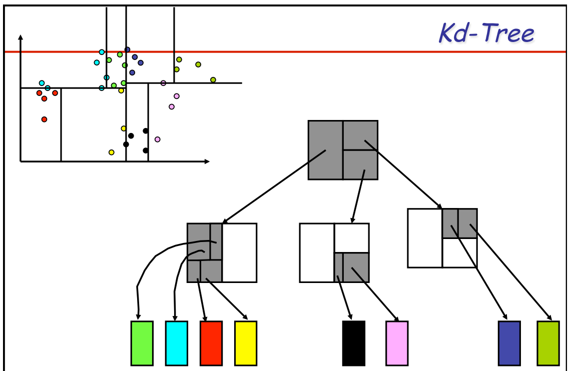
Kd-Tree



Kd-Tree



Kd-Tree



Kd-Tree

- All cells are disjoints
  - no overlap between cells
  - super-cool for pruning rules!
- Nodes in memory
- Leaves on disk
- At the roots of many space-partitioning approaches
  - BSP Tree (Binary Space Partitioning)
    - subdivision lines are not parallel to coordinate system
    - this leads to Voronoï cells
  - Variations
    - where to split (often median)
    - what dimension to use next
    - balanced tree, or not…

Python time: indexing the DB

- Compare the execution times to retrieve the 50-nn for d=500, and:
  - 1000 queries, N=1,000,000

When using a brute-force approach (linear search) and a KD-Tree

53

Are these good approaches?

- What about their:
  - pruning power
  - speedup w.r.t. sequential scan
  - construction complexity
  - …
- Do experiments and/or study high-dimensional spaces

Table of Contents

1. Data indexing: why we need cells?
2. How to construct cells?
3. High-dimensional weirdness
4. Approximate searches

Properties of High-Dimensional Spaces

- Descriptors are typically high-dimensional
- Some weird things happen in such spaces
  - algo fine at low-dim die at high-dim
- Watch your intuition. Things in 2d or 3d are way different
- This is called **curse of dimensionality**

Python time: high-dimensional spaces

- Study a uniform **IID** distribution, when *d* grows
  - Generate a DB containing N vectors of dimension d (N, d=parameters), uniformly distributed
  - Generate one or several query vector(s) of dimension d
  - For d=2, for each query, compute:
    - The distance its nearest neighbor (1-nn)
    - The distance to the farthest neighbor (N-nn)
    - The ratio between the 1-nn and 2-nn (Lowe matching criteria)
    - The ratio between the 1-nn and N-nn
  - Repeat for increasing values of d: 3, 5, 10, 50, 100, 200, 500, 1000
- What happens?

57

1) Vanishing Variance

- Shaft [ACM TODS 2006] have observed that, with an **IID** distribution, when *d* grows:
  - the distance between all pairs of points tend to be similar
  - both dist(q, nn) and dist(q, farthest neighbor) grow
  - dist(q, nn) grows faster than dist(q, fn)
  - the closest point to a given one is as far as its farthest
  - NN becomes very unstable
  - this is called "Vanishing Variance"
- Some data sets can hardly be indexed

2) High-dim Space Partitioning

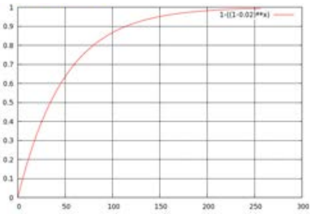
- The number of cells for space partitioning grows exponentially with *d*
- Example:
  - d = 30; split each dimension in 2
  - 2^{30} = 1 073 741 824 cells
  - much more cells than data points
  - many empty cells, costly to keep track of them
  - unlikely to have more than 1 point per cell
- Partitioning for cells is useless

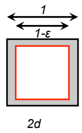
2) High-dim Space Partitioning

- If a point is near a frontier, then must analyze the neighboring cell
- How many cells touch any particular cell when d=30 and each split in 2 ?
- How likely is it that a random point gets near a frontier?
- Example: In $[0,1]^d$, what is the probability for a point (uniformly sampled) to be at less than δ from the frontier?
- Python: compute this proba for d=1, 2, 3, 5, 10, 50, 100, 500

2) High-dim Space Partitioning

- In $[0,1]^d$, the probability to be at less than δ from the frontier is:
  $P = 1-(1-2.\delta)^d$





$\delta = 0.01$

- *δ is small here (1%)*

2) High-dim Space Partitioning

- All data points are near frontiers
  - must check on the other side
- One cell has very many neighbors
  - must check many neighboring cells
- Many cells are empty
- Vectors are lonely in cells
  - cells are useless
- Nice, isn't it?

3) Rectangles and Spheres

- Simple shapes
  - easy to encode, cheap for geometrical rules
- Let's compare them in high-dim, d=30
- Compactness
  - volume = 1 ⇒ diagonal = 5.47, radius = 1.43
  - hyperspheres are much more compact than hyperrectangles are (corners take up space)

3) Rectangles and Spheres

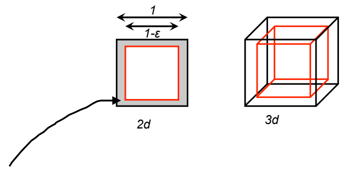
- Enlargement
  - to absorb a new data point, shape must expand
  - diagonal+=0.5; radius+=0.5
    ⇒volume R=35; volume S=126
  - hyperrectangles grow much slower than hyperspheres do

3) Rectangles and Spheres

- Consequences
  - in high-dim, rectangles are huge
  - in high-dim, spheres get fat
  - there is a lot of overlap
  - the more overlap, the less geometrical rules can be effective
  - so the more cells you need to analyze
- Cells are useless
  - isn't this cool?

4) Rectangles and Spheres

- Let's zoom on the volume of R or S



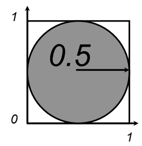
- this is the shell of the shape

4) Rectangles and Spheres

- the volume of the shell grows exponentially with *d*
- when *d* large, all the volume is in the shell
  - eating high-dim eggs will not stuff you
- it is unlikely anything can be in the "center"
- so again, frontier and enlargement problems
- Isn't creating cells a stupid idea?

5) Empty Space Phenomenon

- Whatever you do, when *d* grows, the space becomes more and more empty
- Example:
  - S is an hypersphere with radius of 0.5 in $[0,1]^d$
  - compute N to have at least 1 vector in S
  - d=2



$$V_d(r) = \frac{\pi^{d/2} r^d}{\Gamma(1+d/2)}$$

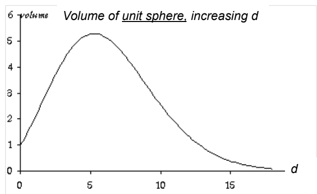
$$\text{when d even}: \Gamma(1+d/2) = (d/2)!$$

$$\text{when d odd}: \Gamma(1+d/2) = \sqrt{\pi}\frac{d!!}{2^{(d+1)/2}}$$

$$with \quad d!! = \begin{cases} 1, & d=-1,0,1 \\ d(d-2)!!, & d \geq 2 \end{cases}$$

5) Empty Space Phenomenon

- Let's increase *d*
  - d=10, N≈400   d=40, N=3000.10^{20}
  - d=20, N=40.10^6 d=100, N=5000.10^{69}
- Why is that?



- "Corners" are consuming all the space

Indexing High-dim Data

- If cells are overlapping
- If most of cells are empty
- If there is only one point per cell
- If you need to analyzed all neighboring cells
- Then:

Read all cells, randomly
=> Lot of (random) I/O

=> Much more costly than a sequential scan

Indexing High-dim Data

- This is very true for uniform data
  - theoretically proven – what can you do…
  - no indexing method will beat seq scan
  - particularly when searching exact neighbors
- Less true when data is not uniform
  - real data is more contrasted
  - vanishing variance not kicking so strongly ☺
  - but high dimensionality still causes trouble

Summary

- Consensus: only the sequential scan has a fair behavior
- All other approaches eventually fail, and degenerate to worse.
- What the hell is going on **during** a search?

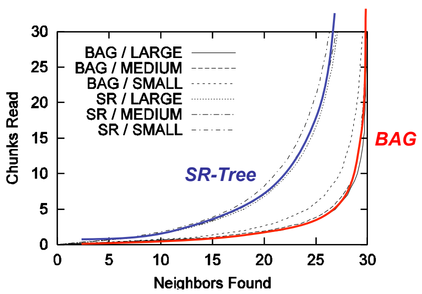
High-Dim. Data - What is going on?

- What are the effects of all this?
- What is happening to search processes?
- Elements for understanding:
  - years of experiments and observations
  - results extracted from a paper: "*A Case Study of the Quality vs. Time Trade-off for Approximate Image Descriptor Search*", Sigurðardóttir, Hauksson, Jónsson, Amsaleg, EMMA'05 (with ICDE)

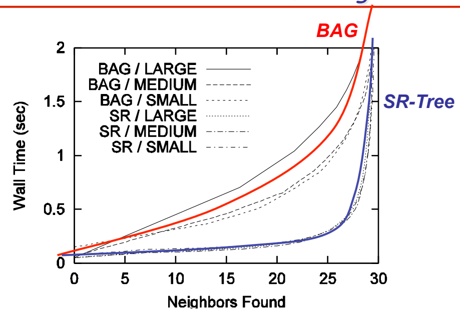
Experiments

- 5.017.298 descriptors, 24 dimensions
- 52.273 real-life images (news, photo agency)
- 2 cell forming approaches
  - SR-Tree
  - BAG

of chunks needed to get NN



Time to get NN



Overall

- You very quickly get the first NN
  - first cell(s) very profitable
- You get others more slowly
  - next cells less and less profitable
  - many NN ⇒ many cells
- Termination problem
  - overlap ⇒ many cells candidate (all!)
  - no way to say STOP

The Coolest Idea

- Why waiting 10h to get a perfect result if in 10 sec we can get a pretty good one?
- Approximate Searches
  - trade response time against result quality
  - this is where the action is!