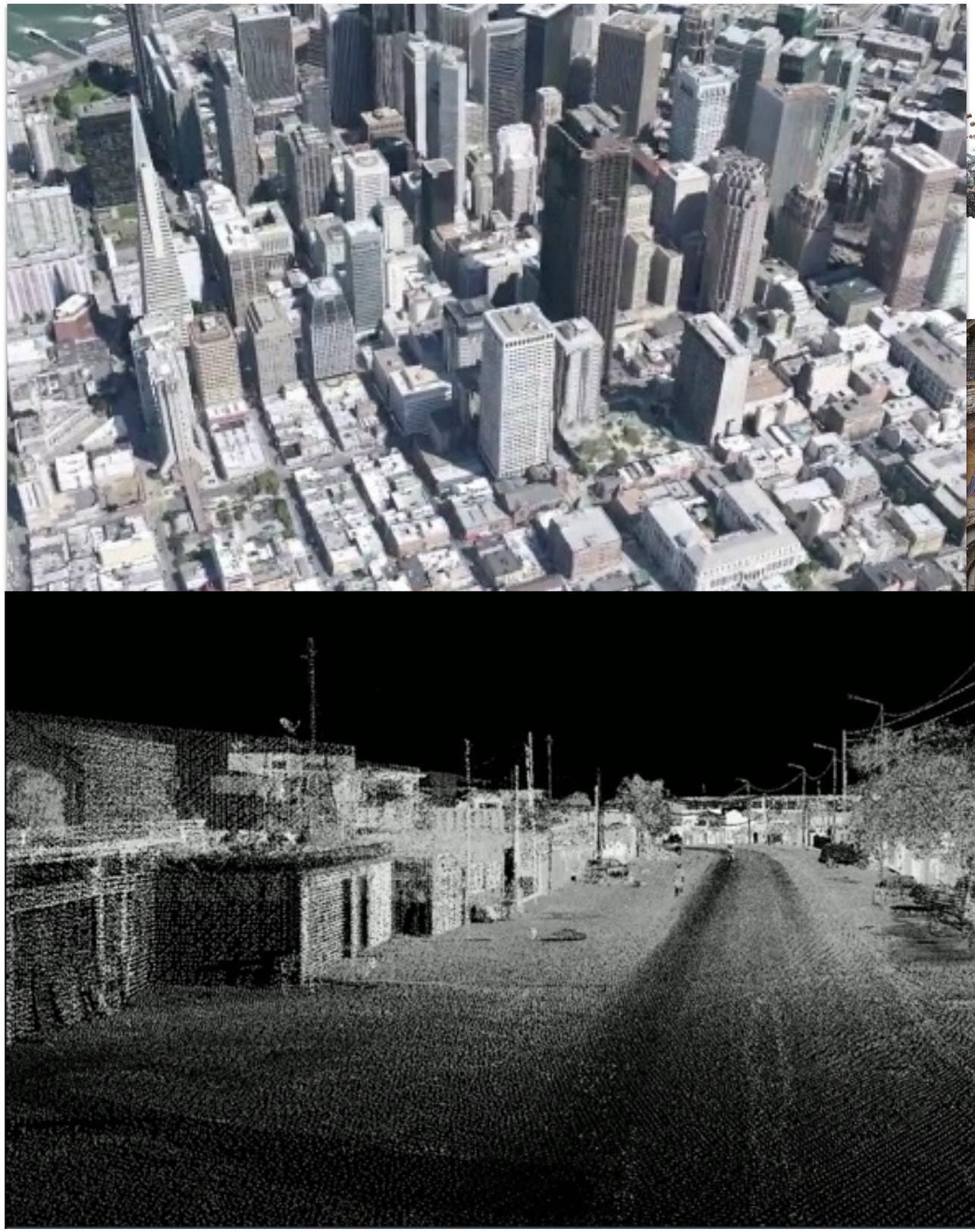
Lecture 14:

Large-scale 3D Reconstruction

Visual Computing Systems
CMU 15-769, Fall 2016

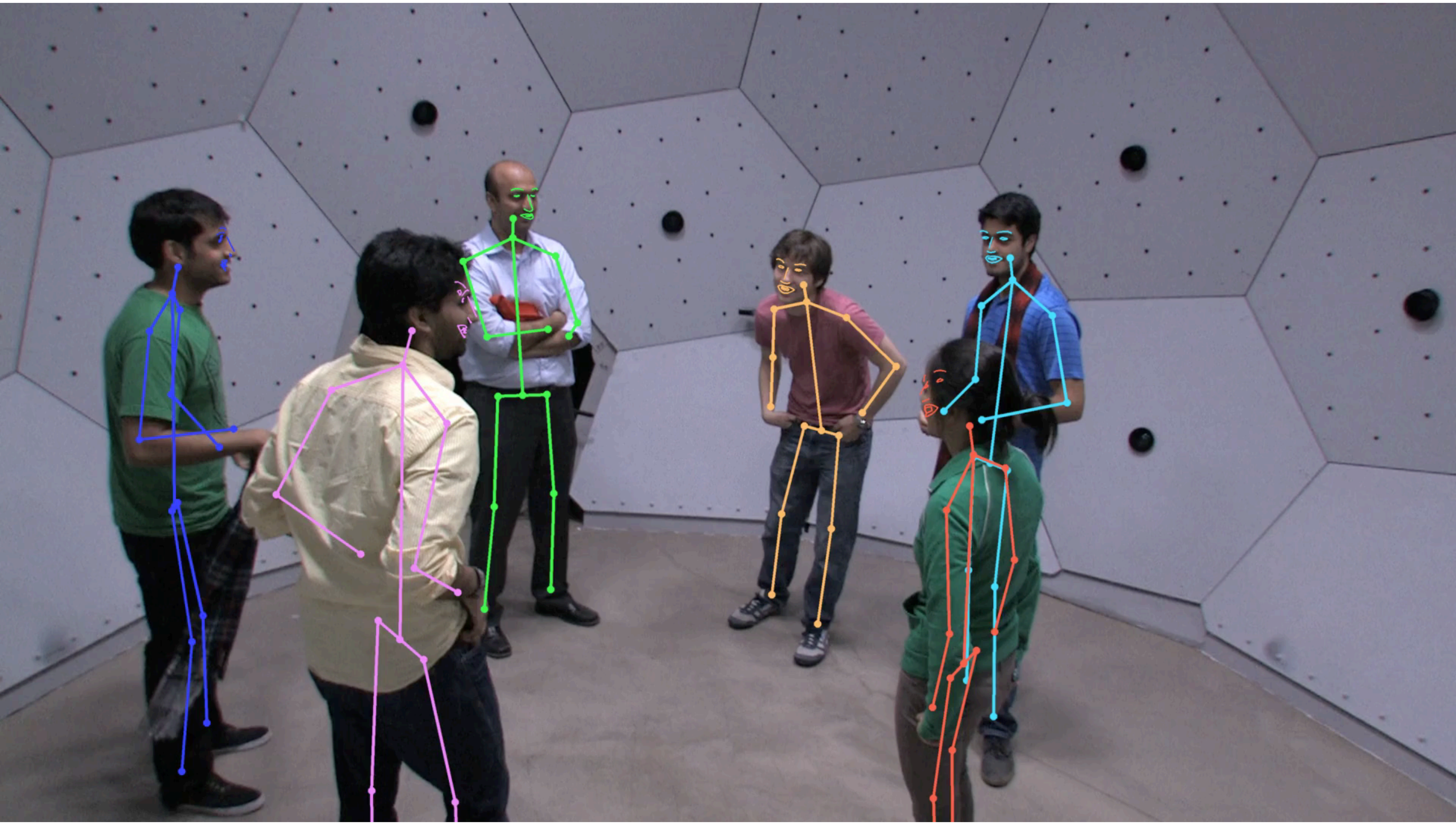
Understanding 3D structure of a scene



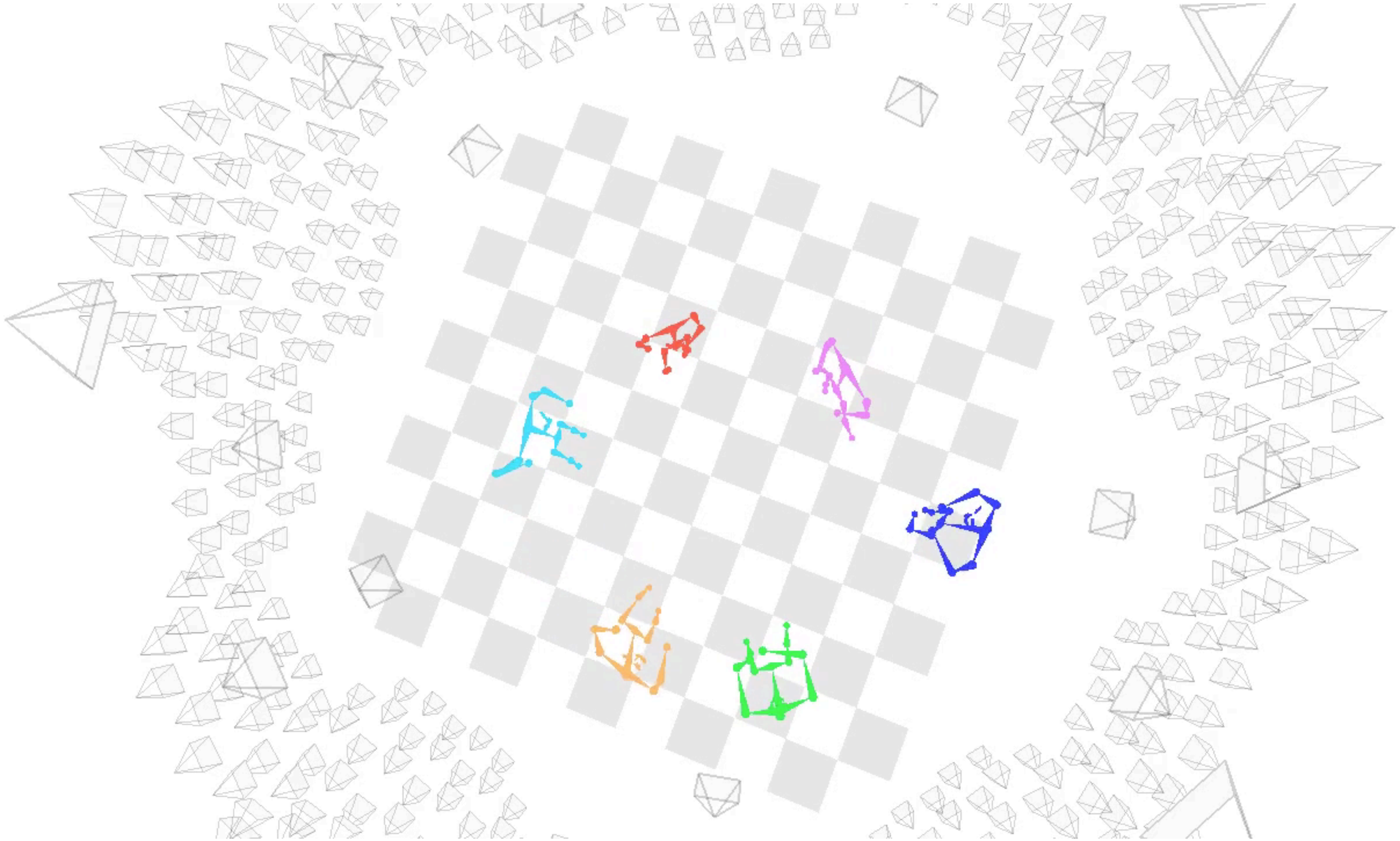
Understanding 3D structure of a scene

ImageCredit: DroneDeploy

Understanding 3D structure of a scene



Understanding 3D structure of a scene



Understanding 3D structure of a scene

Augmented Reality

To know what/where to draw on screen, must know where headset is, and where objects in the scene are

Two styles of approach:

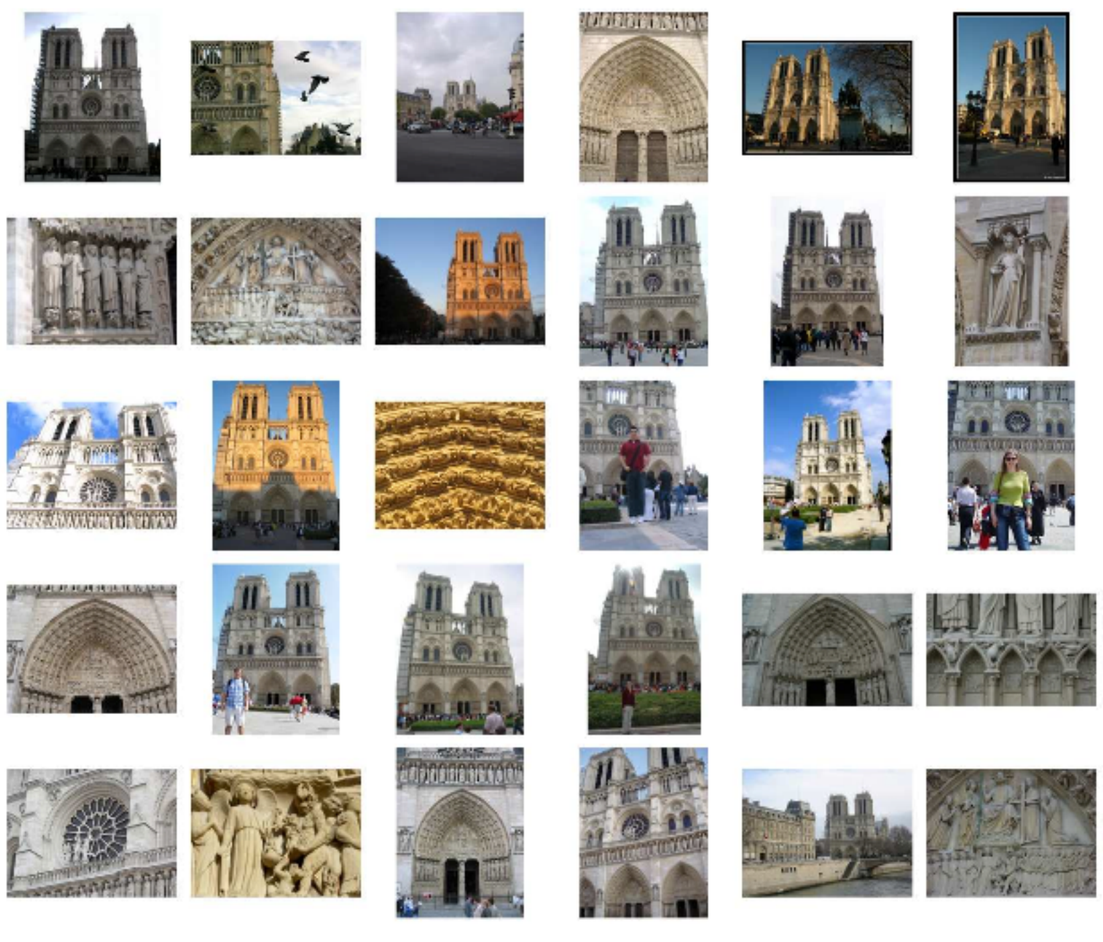
- Reconstruction from sparse feature correspondence
  - Based on structure from motion (SfM)
  - Will discuss today

- Dense reconstruction (tracking-less)
  - Will discuss next time

3D reconstruction from large photo collections

Reconstructing city-scale scenes

- Input:
  - Unstructured collection of millions of photos from same location (obtained from Flickr, Facebook, etc.)
- Output:
  - Sparse 3D representation of scene (point cloud)
  - Position of camera for each photo



Computing depth of scene point from two images

Assume two calibrated cameras are looking at the same scene

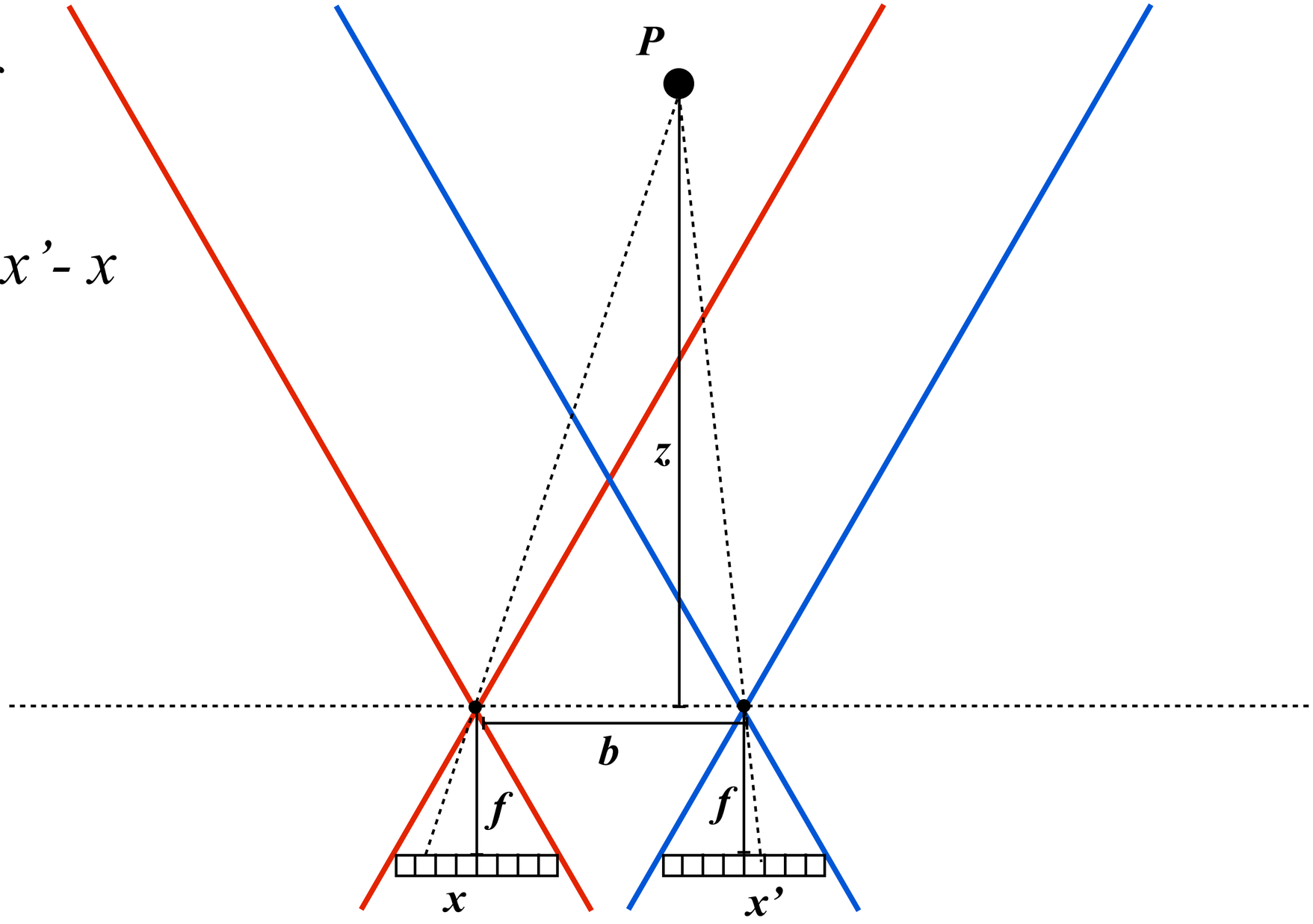
Binocular stereo 3D reconstruction of point *P*: depth from disparity

Focal length: f

Baseline: b

Disparity: $d = x' - x$

$$z = \frac{bf}{d}$$

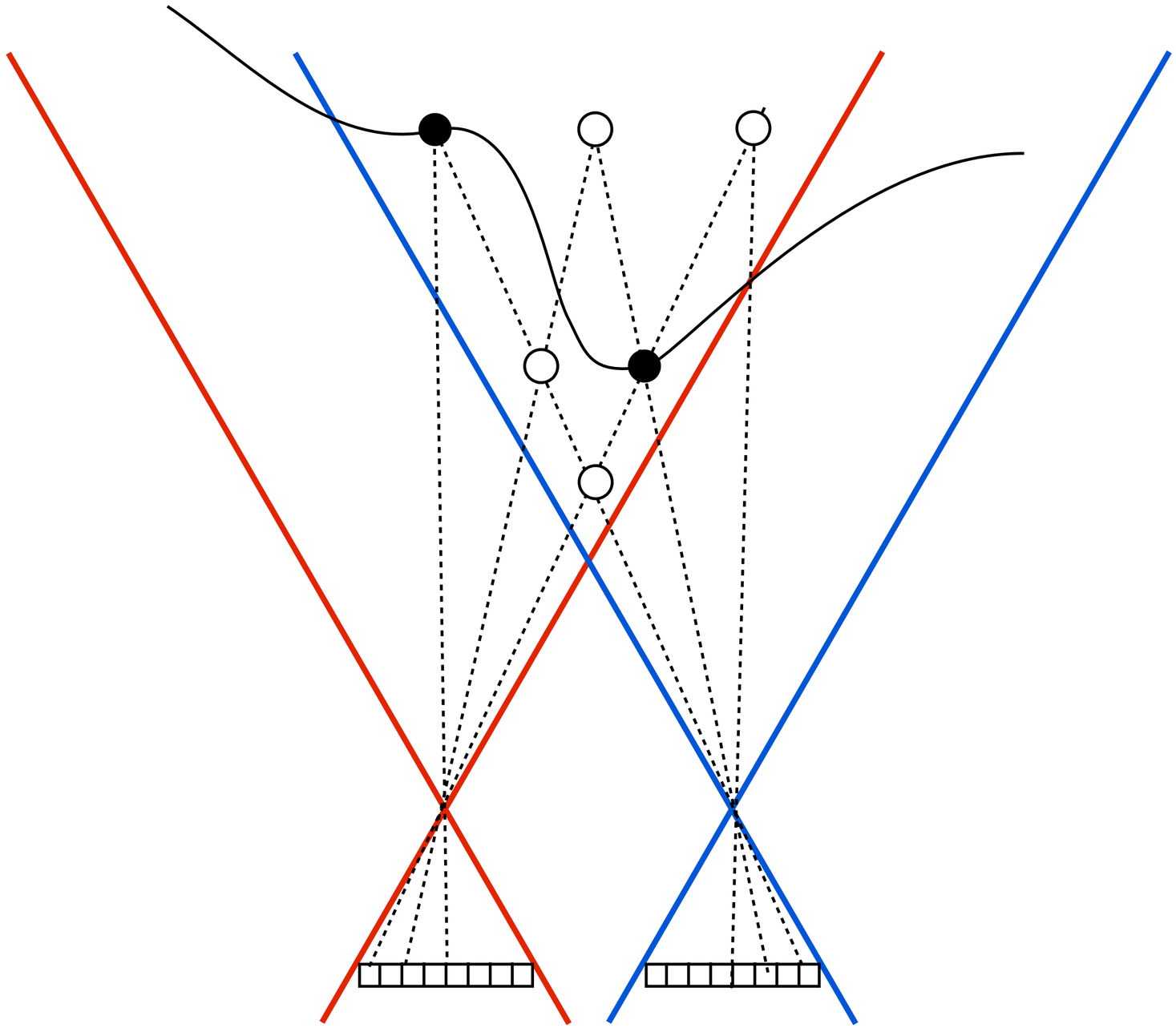


Simple reconstruction example: cameras aligned (coplanar sensors), separated by known distance, same focal length

"Disparity" is the distance between object's projected position in the two images: x - x'

Correspondence problem

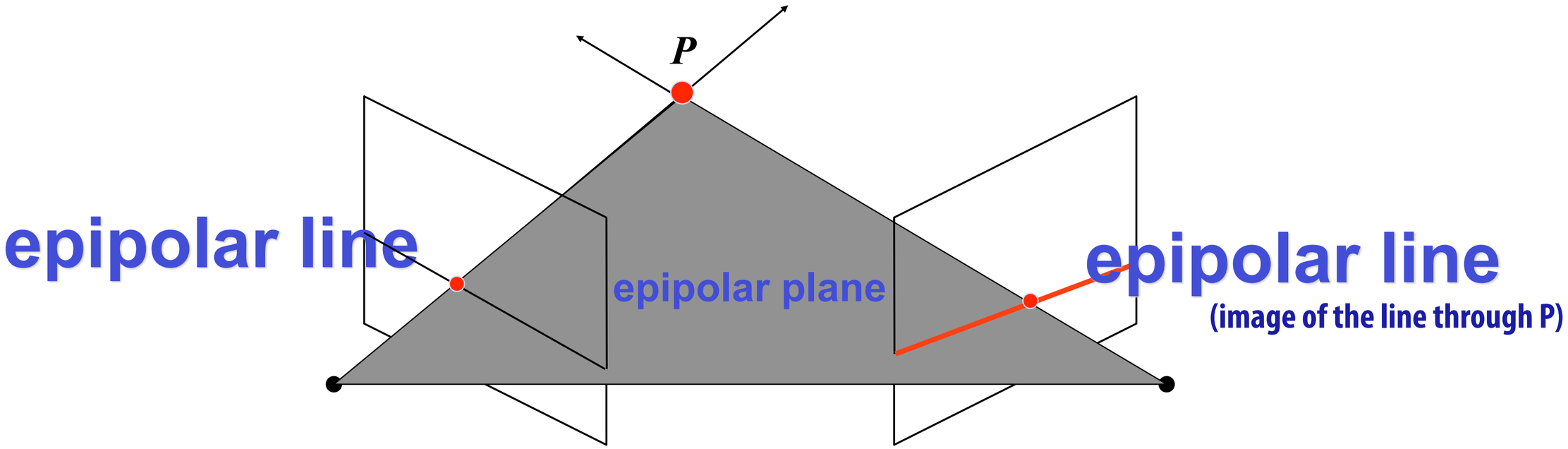
How to determine which pairs of pixels in image 1 and image 2 correspond to the same scene point?



Epipolar constraint

Goal: determine pixel correspondence from pixel values

- Corresponding pixels = pairs of pixels that contain the same scene point



Epipolar Constraint

- Reduces correspondence problem to 1D search along conjugate epipolar lines
- Point in left image will lie on line in right image (epipolar line)

Slide credit: S. Narasimhan

But in our 3D scene reconstruction problem…

- We don’t know the relative orientation of the cameras
- We don’t even know which images are observing the same scene
  - Correspondence may not even exist

Preliminaries and background

1. Image similarity / retrieval basics
   - Need to find image pairs that may potential be views of the same scene
2. Nearest neighbor search and approximate nearest neighbor search (ANN) using a KD-tree
   - Need to find image pairs that may potential be views of the same scene
   - Need to find matching keypoints given a pair of images
3. RANSAC algorithm overview
   - Used for estimating relative orientation of pair of cameras

Background part 1: image retrieval basics

Are these images similar?






Photographs of my backyard, over six-month period.

Similarity via pixel differences

Image 1

Similarity via pixel differences

Image 2

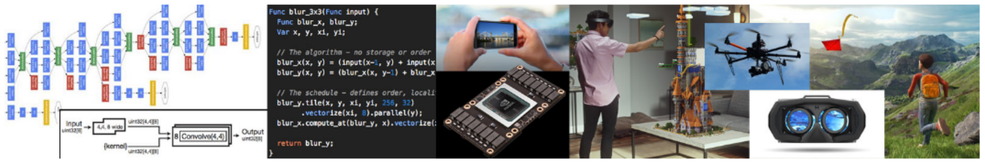
Similarity via pixel differences

diff(x,y) = image1(x,y) - image2(x,y)

Are these two web pages similar?

CMU 15-769

VISUAL COMPUTING SYSTEMS



Visual computing tasks such as 2D/3D graphics, image processing, and image understanding are important responsibilities of modern computer systems ranging from sensor-rich smart phones to large datacenters. These workloads demand exceptional system efficiency and this course examines the key ideas, techniques, and challenges associated with the design of parallel (and heterogeneous) systems that serve to accelerate visual computing applications. This course is intended for graduate and advanced undergraduate-level students interested in architecting efficient future graphics, image processing, and computer vision platforms and for students seeking to develop scalable algorithms for these platforms.

Basic Info

Mon/Wed 10:30-11:50am
GHC 4303
Instructor: **Kayvon Fatahalian**

See the **course info** page for more info on policies and logistics.

Fall 2016 Schedule

Aug 31 **Course Introduction + Parallel Hardware Architecture Review**
History of visual computing, review of multi-core, multi-threading, SIMD, heterogeneity via CPUs/GPUs/ASICs/FPGAs

Part 1: High-Efficiency Image Processing

Sep 7 **The Digital Camera Image Processing Pipeline: Part I**
From raw sensor measurements to an RGB image: demosaicing, correcting aberrations, color space conversions

Sep 9 **The Digital Camera Image Processing Pipeline: Part II**
JPG image compression, auto-focus/auto-exposure, high-dynamic range processing

Sep 12 **Efficiently Scheduling Image Processing Algorithms on Multi-Core Hardware**
Balancing parallelism/local/extra work, programming using Halide

Sep 14 **Image Processing Algorithm Grab Bag**
Fast bilateral filter and median filters, bilateral grid, optical flow

Sep 26 **Specializing Hardware for Image Processing**

Parallel Computer Architecture and Programming (CMU 15-418)

From smart phones, to multi-core CPUs and GPUs, to the world's largest supercomputers and web sites, parallel processing is ubiquitous in modern computing. The goal of this course is to provide a deep understanding of the fundamental principles and engineering trade-offs involved in designing modern parallel computing systems as well as to teach parallel programming techniques necessary to effectively utilize these machines. Because writing good parallel programs requires an understanding of key machine performance characteristics, this course will cover both parallel hardware and software design.

[Our Self-Made Online Reference]

[Policies, Logistics, and Details]

When We Meet

Tues/Thurs 9:00 - 10:20am
Baker Hall A51 (Giant Eagle Auditorium)
Instructor: **Kayvon Fatahalian**

Spring 2013 Schedule

Jan 15 **Why Parallelism?**

Jan 17 **A Modern Multi-Core Processor: Forms of Parallelism + Understanding Latency and BW**
Assignment 1 out

Jan 22 **Parallel Programming Models and Their Corresponding HW/SW Implementations**

Jan 24 **Parallel Programming Basics (the parallelization thought process)**
Assignment 1 due

Jan 29 **GPU Architecture and CUDA Programming**
Assignment 2 out

Jan 31 **Performance Optimization I: Work Distribution**

Feb 5 **Performance Optimization II: Locality, Communication, and Contention**

Feb 7 **Parallel Application Case Studies**

Feb 12 **Workload-Driven Performance Evaluation**
Assignment 2 due
Assignment 3 out

Another example: which web page is most similar to the search query...

Are these two web pages similar?

CMU 15-769

VISUAL COMPUTING SYSTEMS

Visual computing tasks such as 2D/3D graphics, image processing, and image understanding are important responsibilities of modern computer systems ranging from sensor-rich smart phones to large datacenters. These workloads demand exceptional system efficiency and this course examines the key ideas, techniques, and challenges associated with the design of parallel (and heterogeneous) systems that serve to accelerate visual computing applications. This course is intended for graduate and advanced undergraduate-level students interested in architecting efficient future graphics, image processing, and computer vision platforms and for students seeking to develop scalable algorithms for these platforms.

Basic Info

Mon/Wed 10:30-11:50am
GHC 4303
Instructor: **Kayvon Fatahalian**

See the **course info** page for more info on policies and logistics.

Fall 2016 Schedule

Aug 31 **Course Introduction + Parallel Hardware Architecture Review**
History of visual computing, review of multi-core, multi-threading, SIMD, heterogeneity via CPUs/GPUs/ASICs/FPGAs

Part 1: High-Efficiency Image Processing

Sep 7 **The Digital Camera Image Processing Pipeline: Part I**
From raw sensor measurements to an RGB image: demosaicing, correcting aberrations, color space conversions

Sep 9 **The Digital Camera Image Processing Pipeline: Part II**
JPG image compression, auto-focus/auto-exposure, high-dynamic range processing

Sep 12 **Efficiently Scheduling Image Processing Algorithms on Multi-Core Hardware**
Balancing parallelism/local/extra work, programming using Halide

Sep 14 **Image Processing Algorithm Grab Bag**
Fast bilateral filter and median filters, bilateral grid, optical flow

Sep 26 **Specializing Hardware for Image Processing**

Parallel Computer Architecture and Programming (CMU 15-418)

From smart phones, to multi-core CPUs and GPUs, to the world's largest supercomputers and web sites, parallel processing is ubiquitous in modern computing. The goal of this course is to provide a deep understanding of the fundamental principles and engineering trade-offs involved in designing modern parallel computing systems as well as to teach parallel programming techniques necessary to effectively utilize these machines. Because writing good parallel programs requires an understanding of key machine performance characteristics, this course will cover both parallel hardware and software design.

[Our Self-Made Online Reference]

[Policies, Logistics, and Details]

When We Meet

Tues/Thurs 9:00 - 10:20am
Baker Hall A51 (Giant Eagle Auditorium)
Instructor: **Kayvon Fatahalian**

Spring 2013 Schedule

Jan 15 **Why Parallelism?**

Jan 17 **A Modern Multi-Core Processor: Forms of Parallelism + Understanding Latency and BW**
Assignment 1 out

Jan 22 **Parallel Programming Models and Their Corresponding HW/SW Implementations**

Jan 24 **Parallel Programming Basics (the parallelization thought process)**
Assignment 1 due

Jan 29 **GPU Architecture and CUDA Programming**
Assignment 2 out

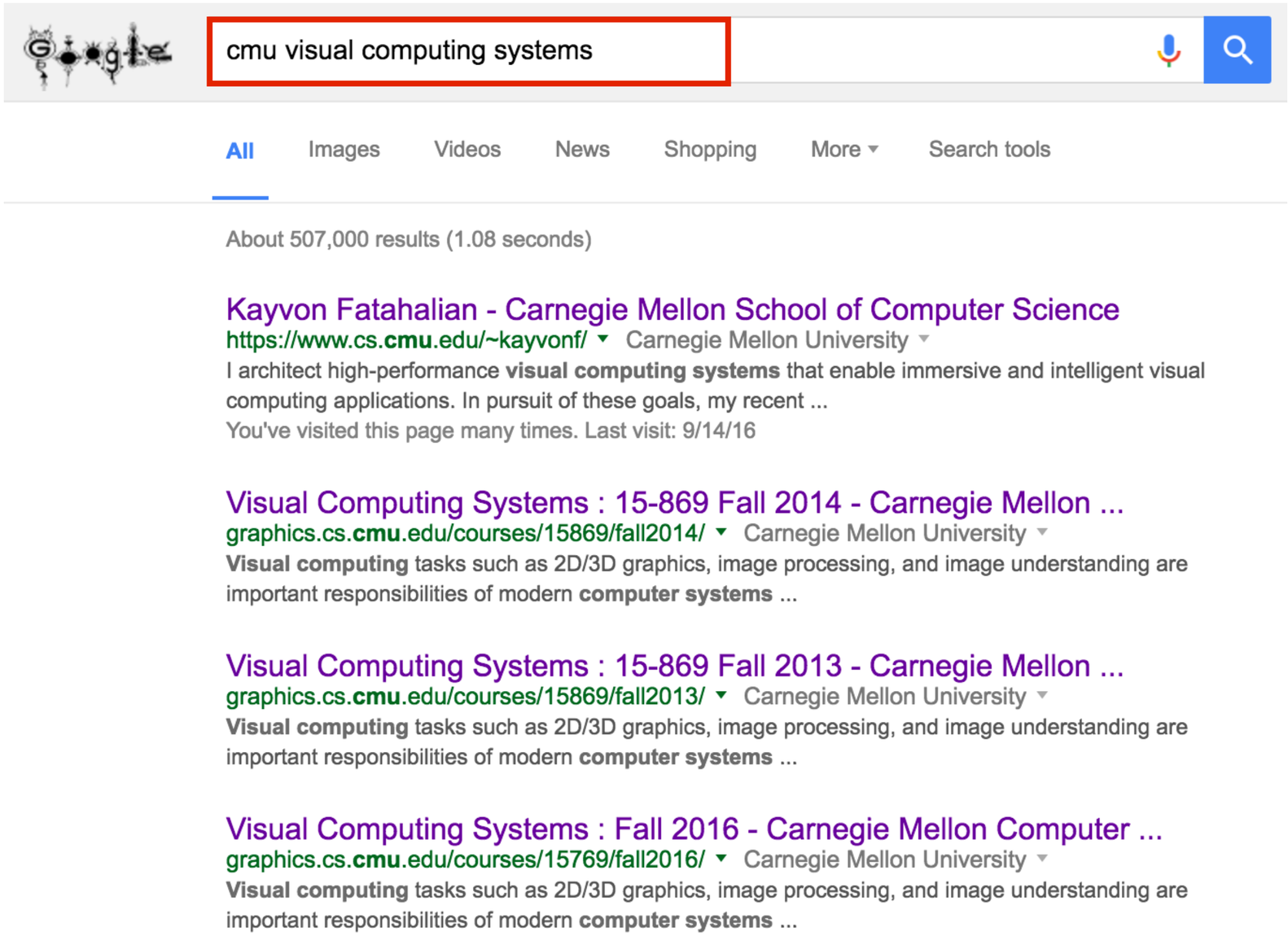
Jan 31 **Performance Optimization I: Work Distribution**

Feb 5 **Performance Optimization II: Locality, Communication, and Contention**

Feb 7 **Parallel Application Case Studies**

Feb 12 **Workload-Driven Performance Evaluation**
Assignment 2 due
Assignment 3 out

Another example: which web page is most similar to the search query...



One simple definition of similarity

Given text query with words: w_1 *and* w_2

for each document *d* in database:

score(d, w_1, w_2) = number of occurrences of w_1 *and* w_2 in *d*

Return top 20 results in sorted order based on score

Ways to improve the above approach:

- **Improve accuracy of document scoring function (return more meaningful "top documents" results)**
- **Improve query execution time: above solution is O(N) for database of N documents**

Accelerating document retrieval: use an index

To simplify, let:

`score(d,w1,w2) = 1 if d contains w1 and w2, 0 otherwise`

```
Document 0: Kayvon is teaching 15-769 today. Yay 15-769!
Document 1: 15-769 is awesome, Kayvon claims.
Document 2: Kayvon is occasionally awesome.
```

Index: maps words to database document ids

- Kayvon: 0, 1, 2
- is: 0, 1, 2
- teaching: 0
- 15-769: 0, 1
- yay: 0
- thinks: 1
- today: 0
- awesome: 1, 2
- occasionally: 2

Query: `kayvon awesome`

Partial result set:

`kayvon: {0, 1,,2}`
`awesome: {1, 2}`

Result:

$\{0,1,2\} \cap \{1, 2\} = \{1,2\}$

Full inverted index

Inverted index contains one entry per word occurrence:

```
score(d,w1,w2) =
     number of occurrences of w1 or w2   (if d contains w1 and w2)
     0                                   otherwise
```

```
Document 0: Kayvon is teaching 15-769 today. Yay 15-769!
Document 1: 15-769 is awesome, Kayvon claims.
Document 2: Kayvon is occasionally awesome.
```

Index: maps words to (document, position)

- Kayvon: (0,0), (1,3), (2,0)
- is: (0,1), (1,1), (2,1)
- teaching: (0, 2)
- 15-769: (0,3), (0,6), (1, 0)
- yay: (0, 5)
- claims: (1,4)
- today: (0, 4)
- awesome: (1,2), (2,3)
- occasionally: (2,2)

Query: `kayvon 15-769`

Partial result set:

```
kayvon: {(0,0), (1,3), (2,0)}
15-769: {(0,3), (0,6), (1,0)}
```

Result (per document counts shown):

```
{0 (1), 1 (1), 2 (1)} ∩
{0 (2), 1 (1)}
= {0 (3), 1 (2)}
```

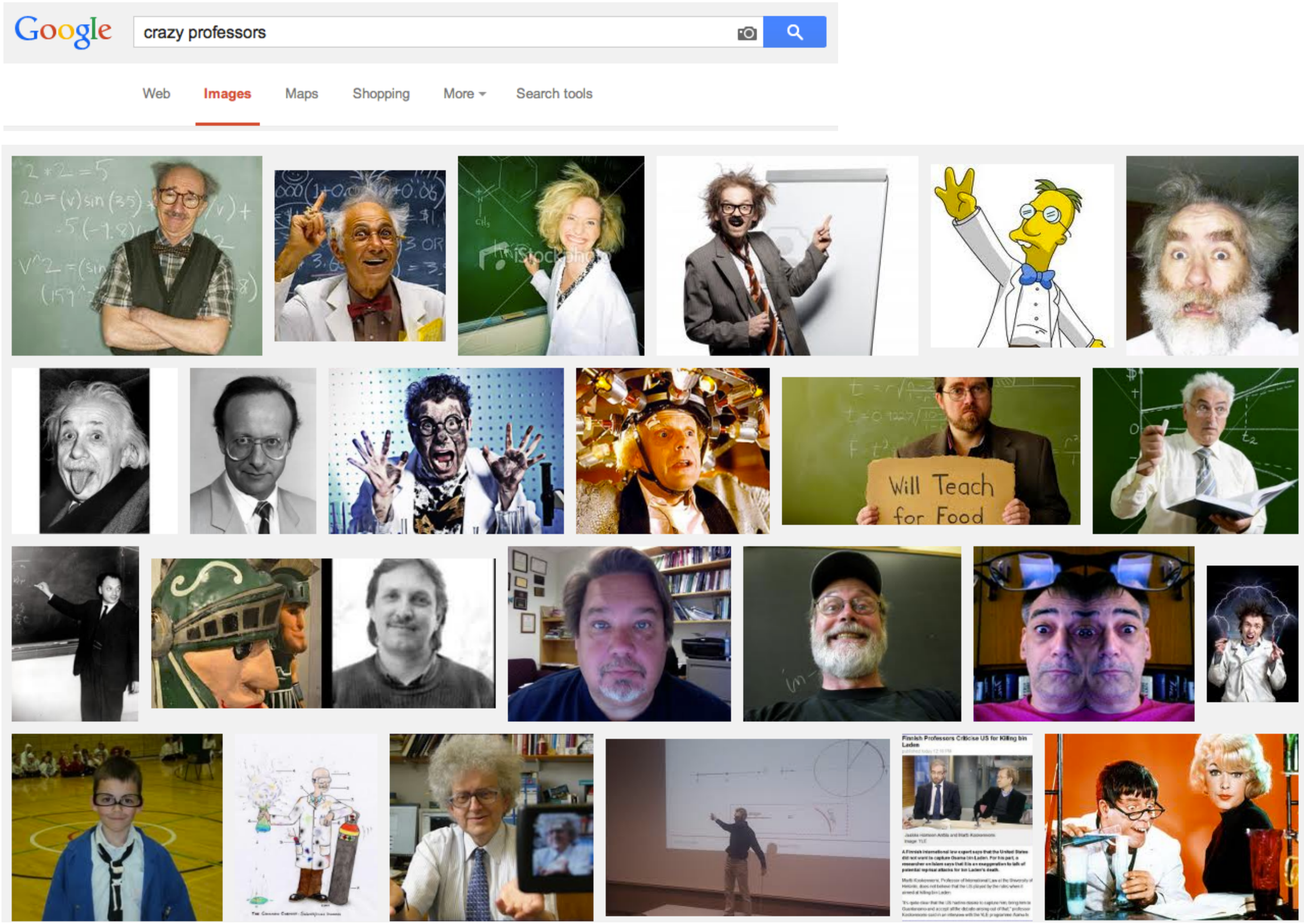
Ranking:

```
0, 1
```

TF-IDF weighting

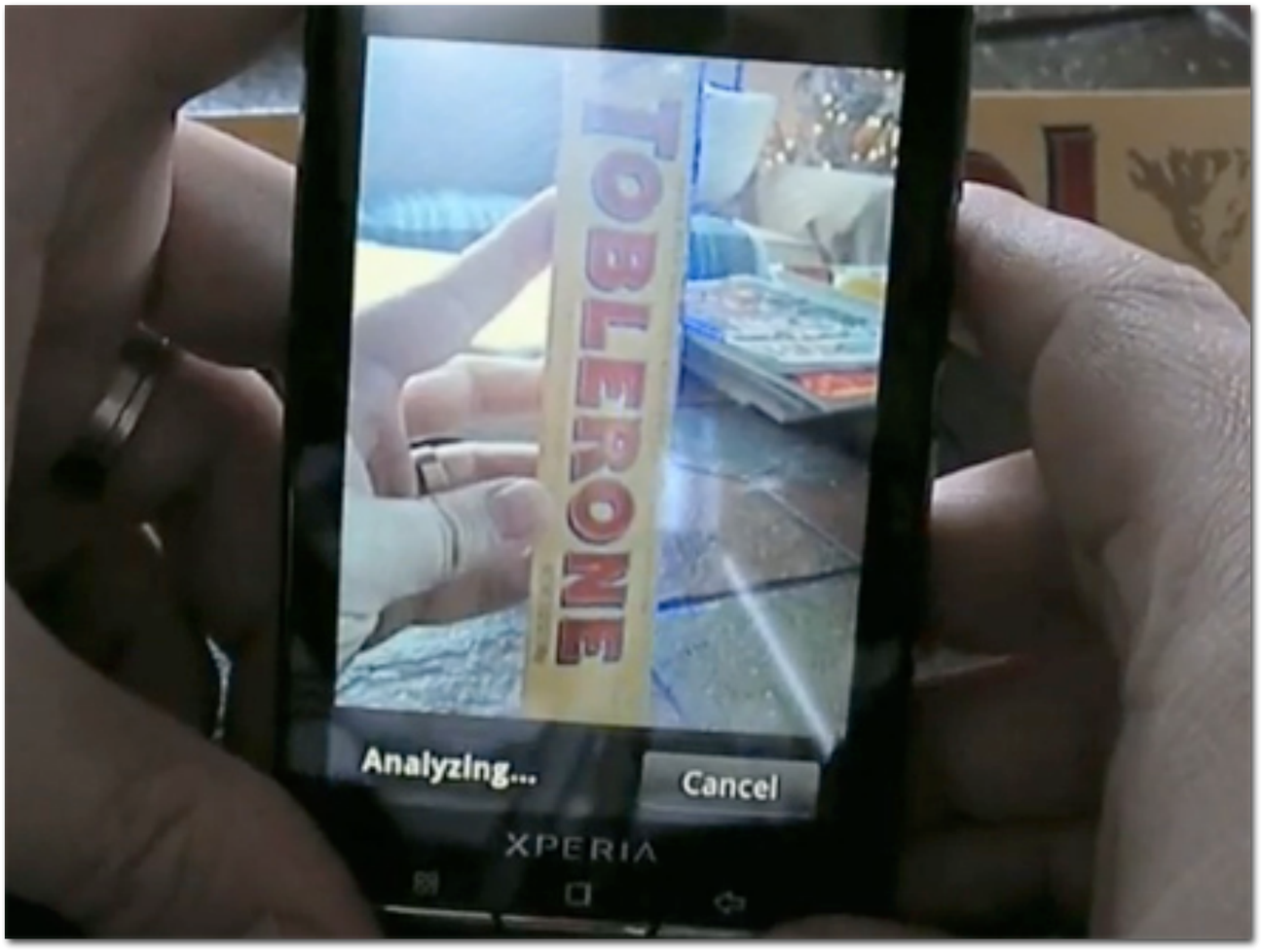
- Term frequency:
  - TF(w,d) = the number of occurrences of word w in document d
  - Measure of how relevant a document is for a given query word
- Inverse document frequency:
  - $\text{IDF}(w, D) = \log \frac{|D|}{|\{d \in D : w \in d\}|}$
    - $|D|$ ← Number of documents in database D
    - $|\{d \in D : w \in d\}|$ ← Number of documents containing w
  - Measure of how discriminative a word is (IDF is small for common words)
    - Depends on number of occurrences in entire document database
    - Idea: words that appear in most documents should influence score less
- $\text{tfidf\_score}(w, d, D) = \text{TF}(w,d) \times \text{IDF}(w, D)$
- Many variants on how to compute TF(w,d)
  - Binary: 1 of 0, depending on whether word is in document
  - Normalized frequency: number of occurrences normalized by document size

Searching for images (via text query)



Content-based image retrieval

- Search for images, based on a query image
  - Take a photo, find similar looking photos
  - Take a photo, find information about (objects, people, etc.) in photo



Text-based document retrieval

- Key idea in text-based document retrieval was the breakdown of document into words
  - Documents that have the same words are likely to be similar
  - Words are a meaningful granularity of text to latch on to

Content-based image retrieval

- If we wanted to follow the text analogy, what are the words?
  - Pixels?
  - Blocks of pixels?
  - Descriptors/features computed from images?

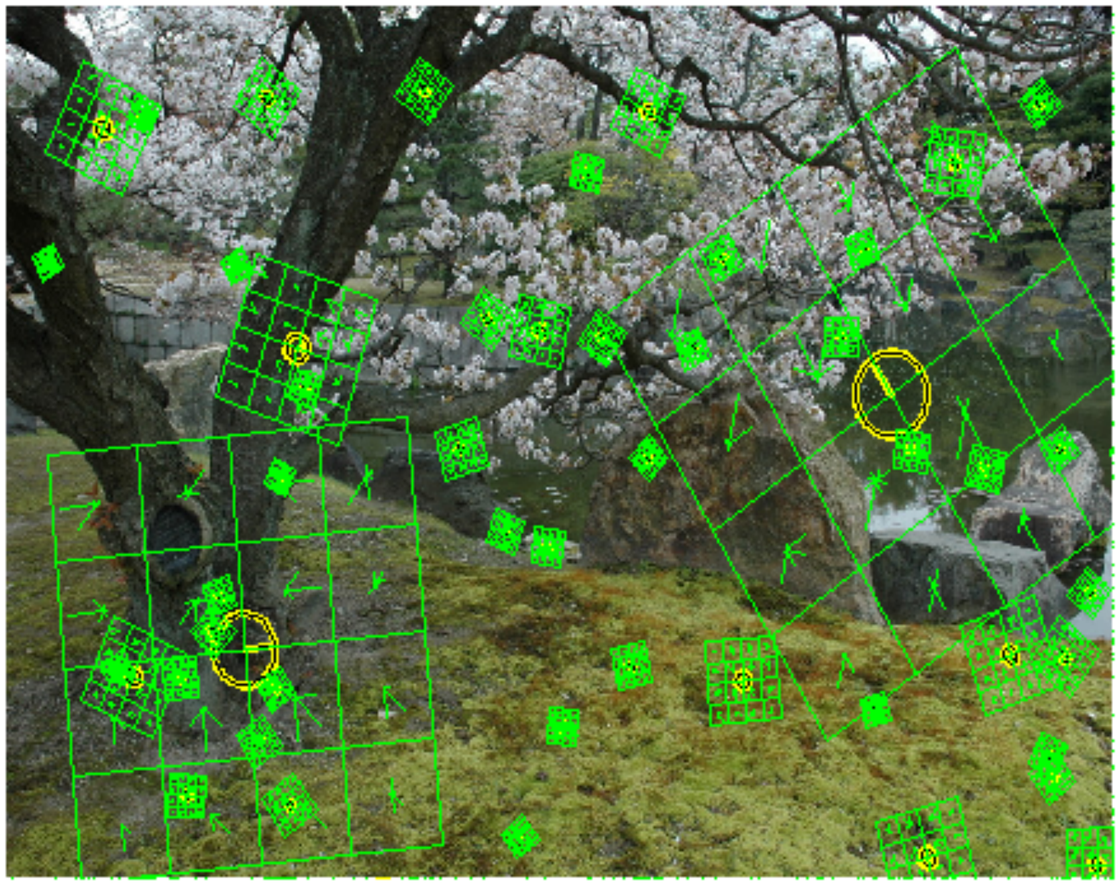
Correspondence

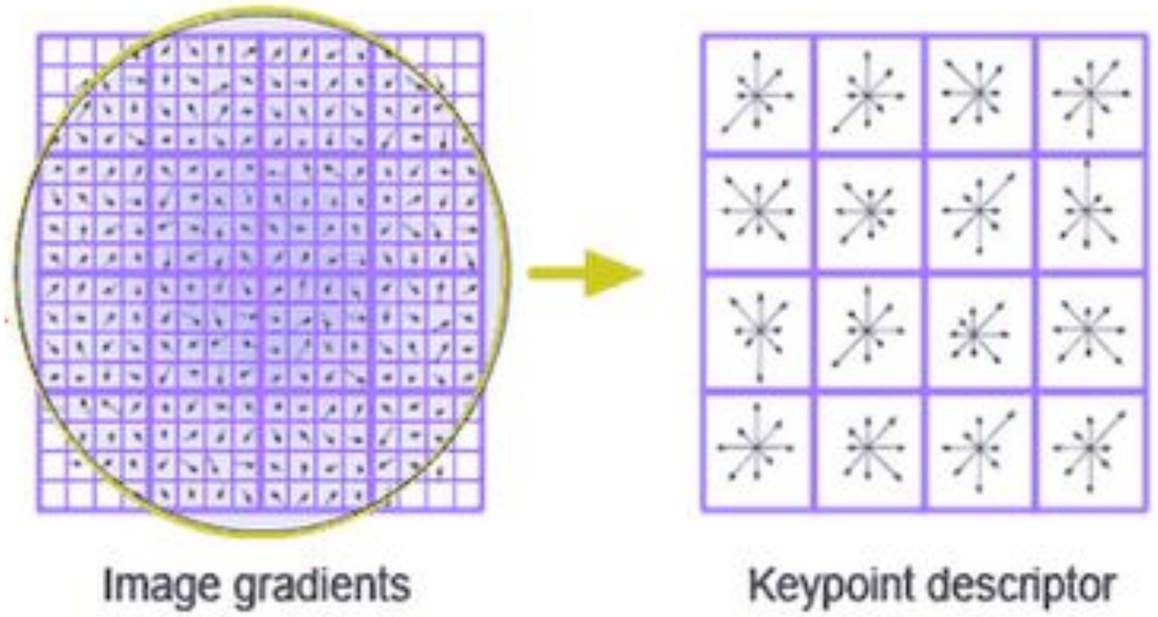
- **Defining similarity requires us to quantify the notion of correspondence**
  - Example: pictures of the same place are similar
  - Example: pictures containing the same/similar objects are similar

- **Seek image representations ("descriptors") such that numerically similar descriptors correspond to meaningful correspondences**
  - Example: similar descriptor value corresponds to same object in the scene: descriptor's value is invariant to noise, lighting, affine object transformation (rotation, translation, scale)
  - In the previous lectures we discussed deep learning based methods to learn good descriptors

Sparse SIFT descriptors

- Interest-point-based, orientation of gradients descriptor
- Find interest points (locations in image that are extrema of gradients)
- Compute 128-element descriptor for interest points





Pool gradient samples from 4x4 window into 8-bin histogram

Concatenate 4x4 grid of histograms to get full descriptor (8 x 16 = 128)

Figure credits:

R. Bandara, Codeproject

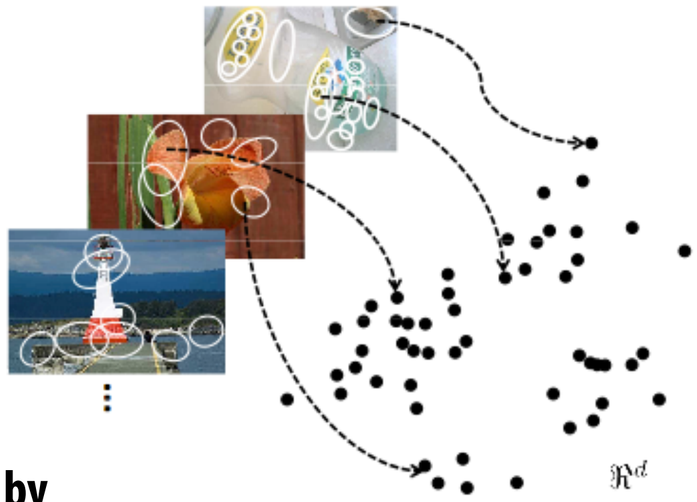
Chen, Kong, Oh, Sanan, Wohlberk 09

Visual words

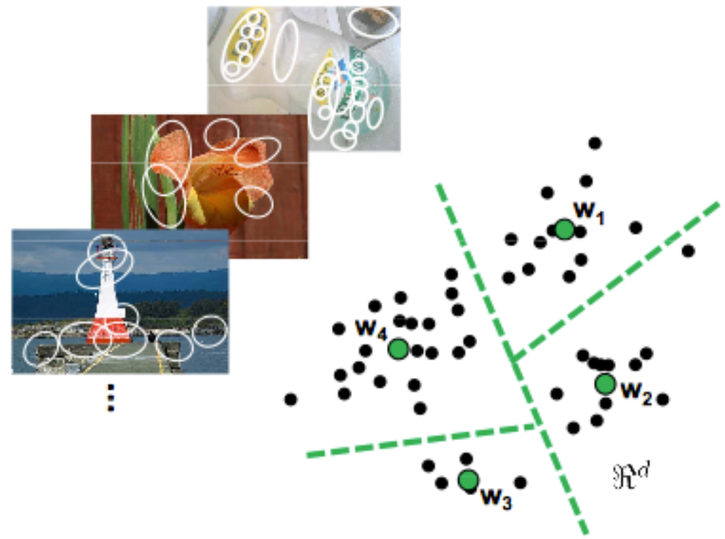
- Text document is made up of words (discrete values in a vocabulary)
- Descriptors are points in continuous high-dimensional descriptor space
- Idea: construct "visual words" from descriptors

(A) Features in images

(B) Compute "vocabulary" for dataset by clustering all features across all images: represent each cluster by its mean (or median) feature



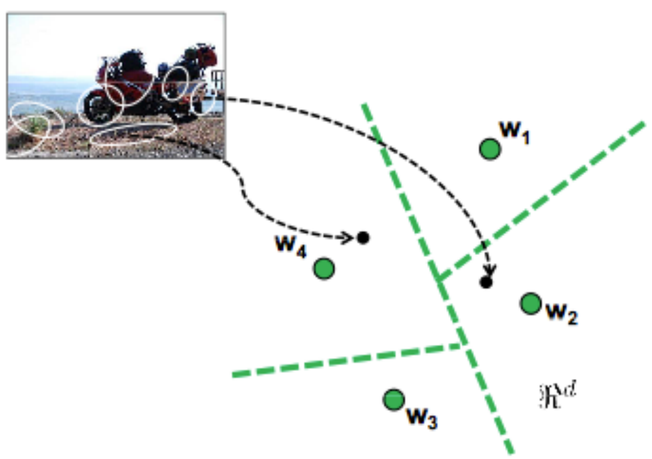
(a)



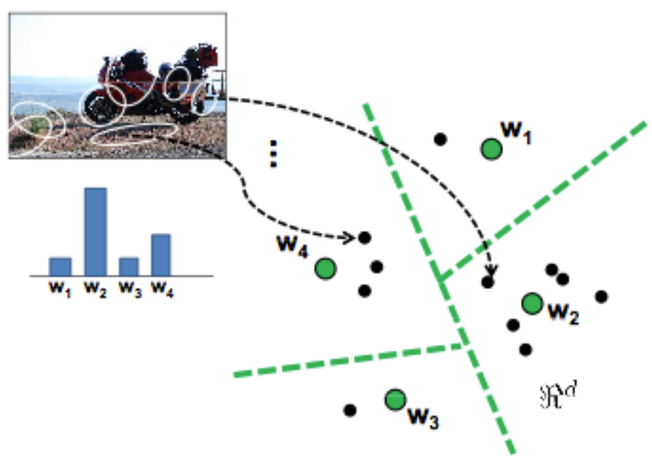
(b)

(C) Bin (discretize) image features by assigning feature to closest cluster in vocabulary

(D) Represent image by its histogram of visual word counts



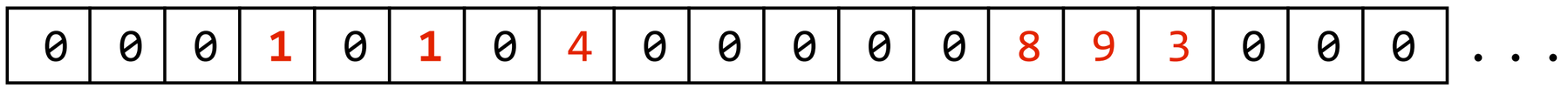
(c)



(d)

Bag of words (BOW) image descriptor:

- **Bag of words (BOW) descriptor:**
  - **Image descriptor is a histogram of word occurrences**
  - **Very sparse vector**



- **Given query image descriptor *q*, compute score for database image *d*:**
  - **Example: dot product of normalized query descriptor and DB image descriptor:**

$$\text{score}(q,d) = \frac{q \cdot d}{\|q\|\|d\|}$$

  - **Improvement: weight descriptor elements by visual word IDF values**
  - **Many alternative distance functions:**
    - **e.g., histogram intersection: $\min(q_i, d_i)$ rather than inner product**

Summary

Image search using bag of words descriptors and an inverted index acceleration structure:

1. Compute features for image collection
2. Build vocabulary (visual words) by clustering features in collection
3. Compute inverted index:
   - For each visual word, index stores list of images with word, plus the tf-idf weight for that word in that image: *tfidf_score(w, d, D) = tf(w,d) * idf(w, D)*
4. For each query image:
   - Compute bag of words descriptor
   - Use inverted index to find candidate set of similar images
   - Compute score between query and candidate images (e.g., dot product of descriptors)
   - Rank results by score

Background part 2: nearest neighbor search using a KD-tree

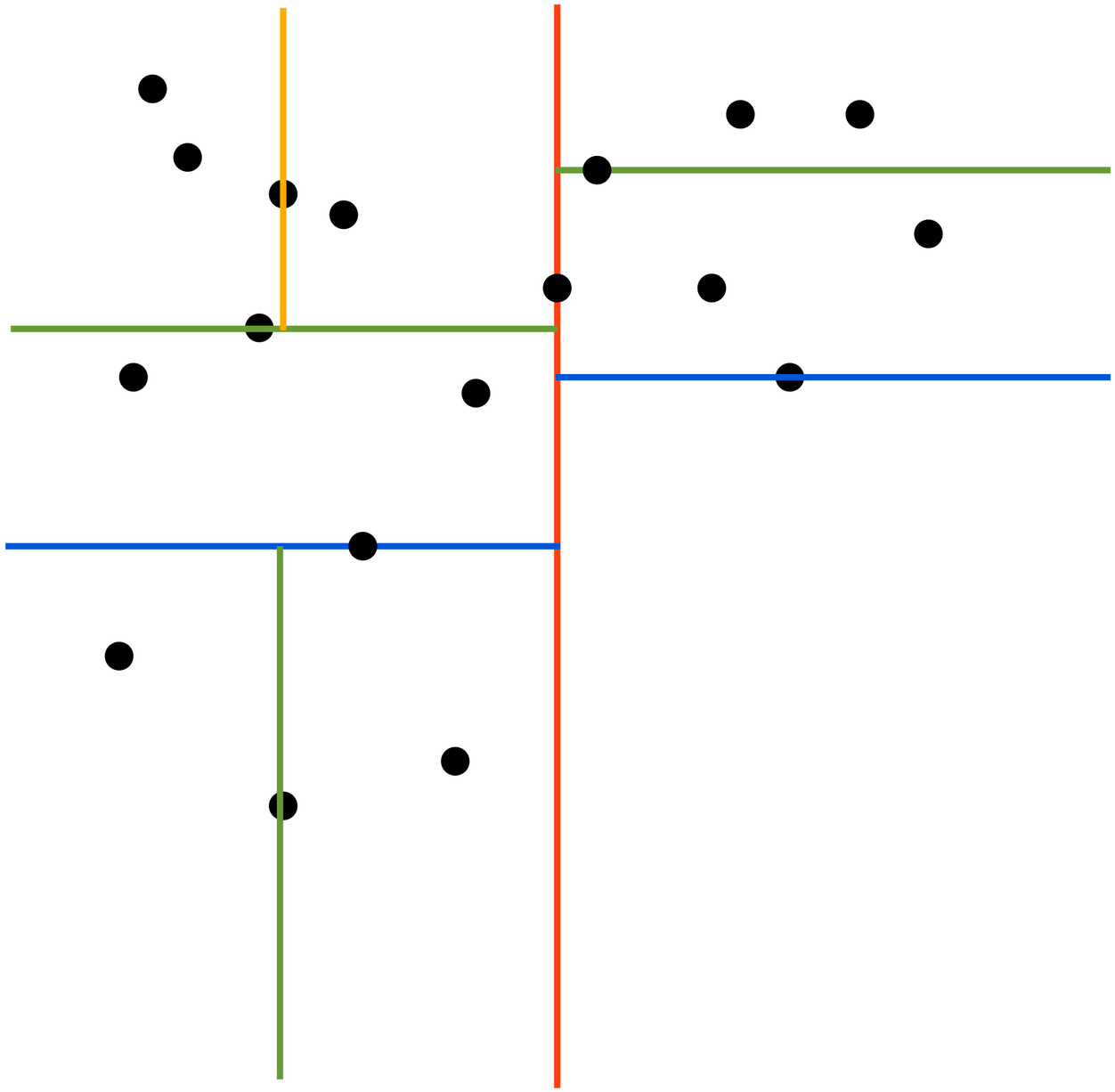
Search application: establish feature correspondence

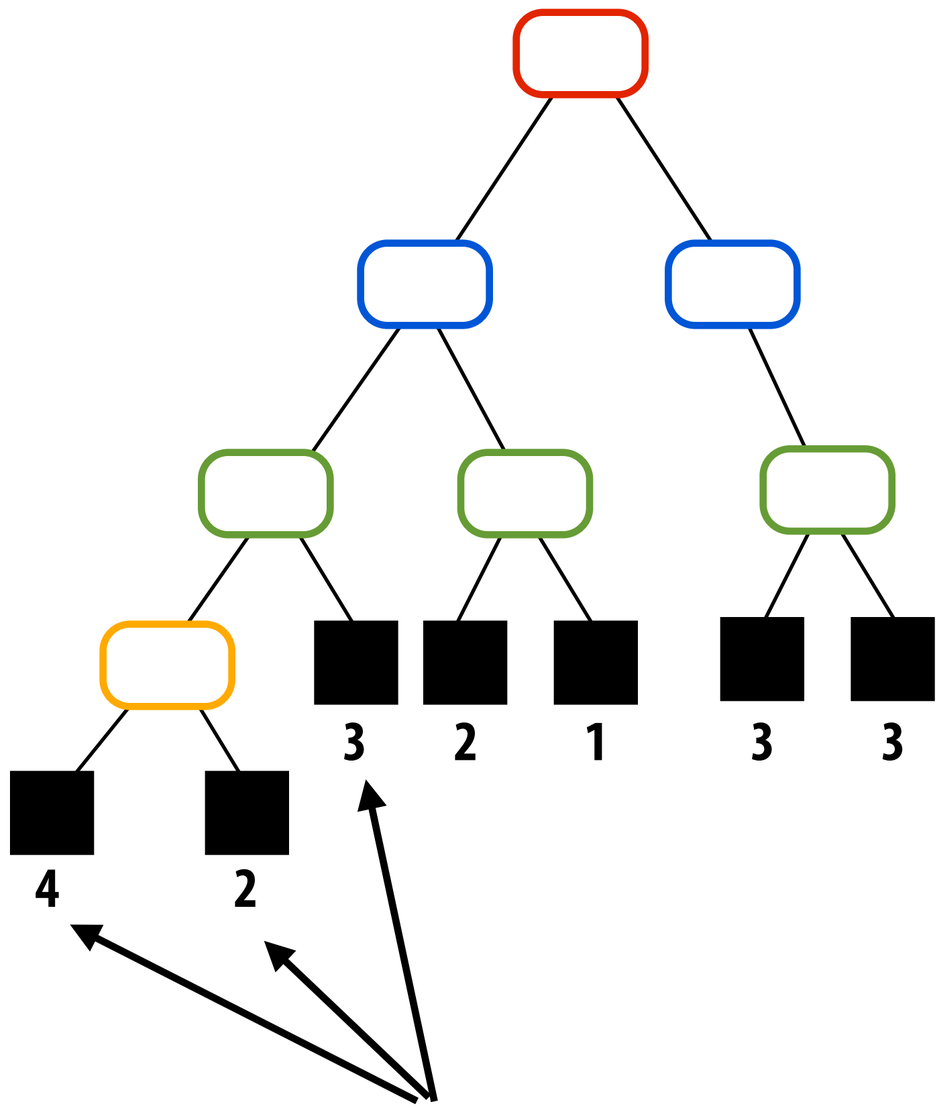
- For all descriptors in image 1, find nearest neighbor descriptor in image 2
  - Thousands of descriptors per image
  - Recall SIFT descriptor (length-128 vector at each keypoint)

Image credit:
http://stackoverflow.com/questions/14360215/correspondence-analysis-in-opencv

Review: K-D tree

- Spatial partitioning hierarchy
- K = dimensionality of space (below: K = 2)

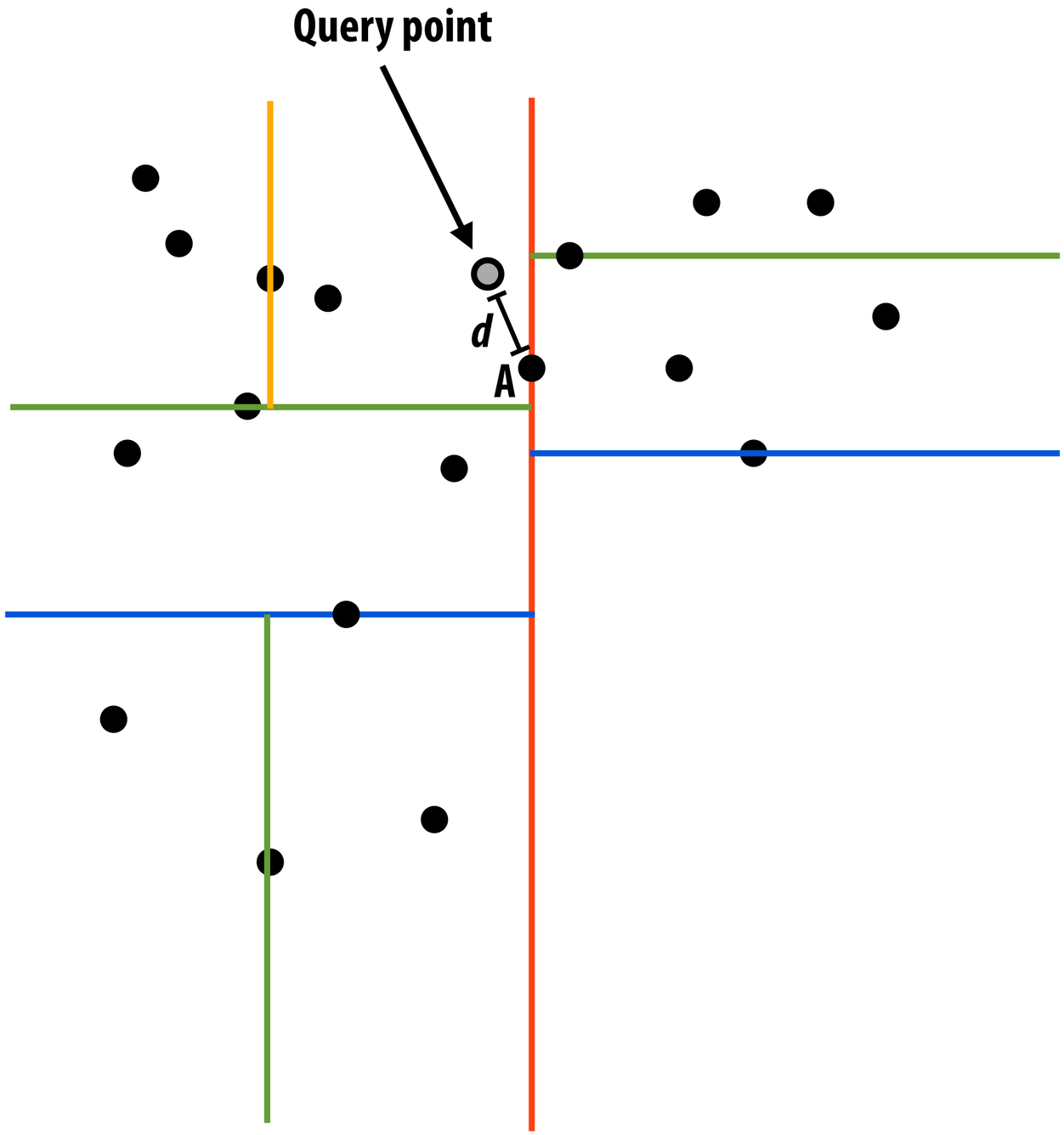


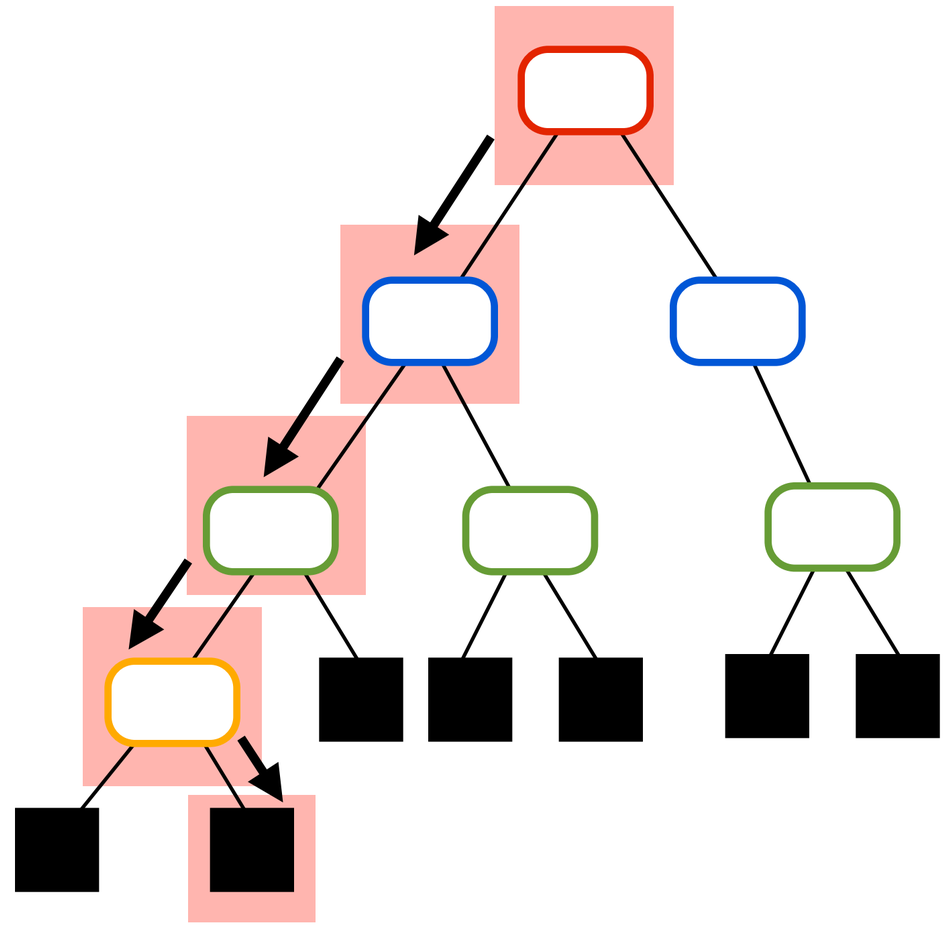


Counts of points in leaf nodes

Nearest neighbor search with K-D tree

Step 1: traverse to leaf cell containing query: compute closest point in this cell to the query.

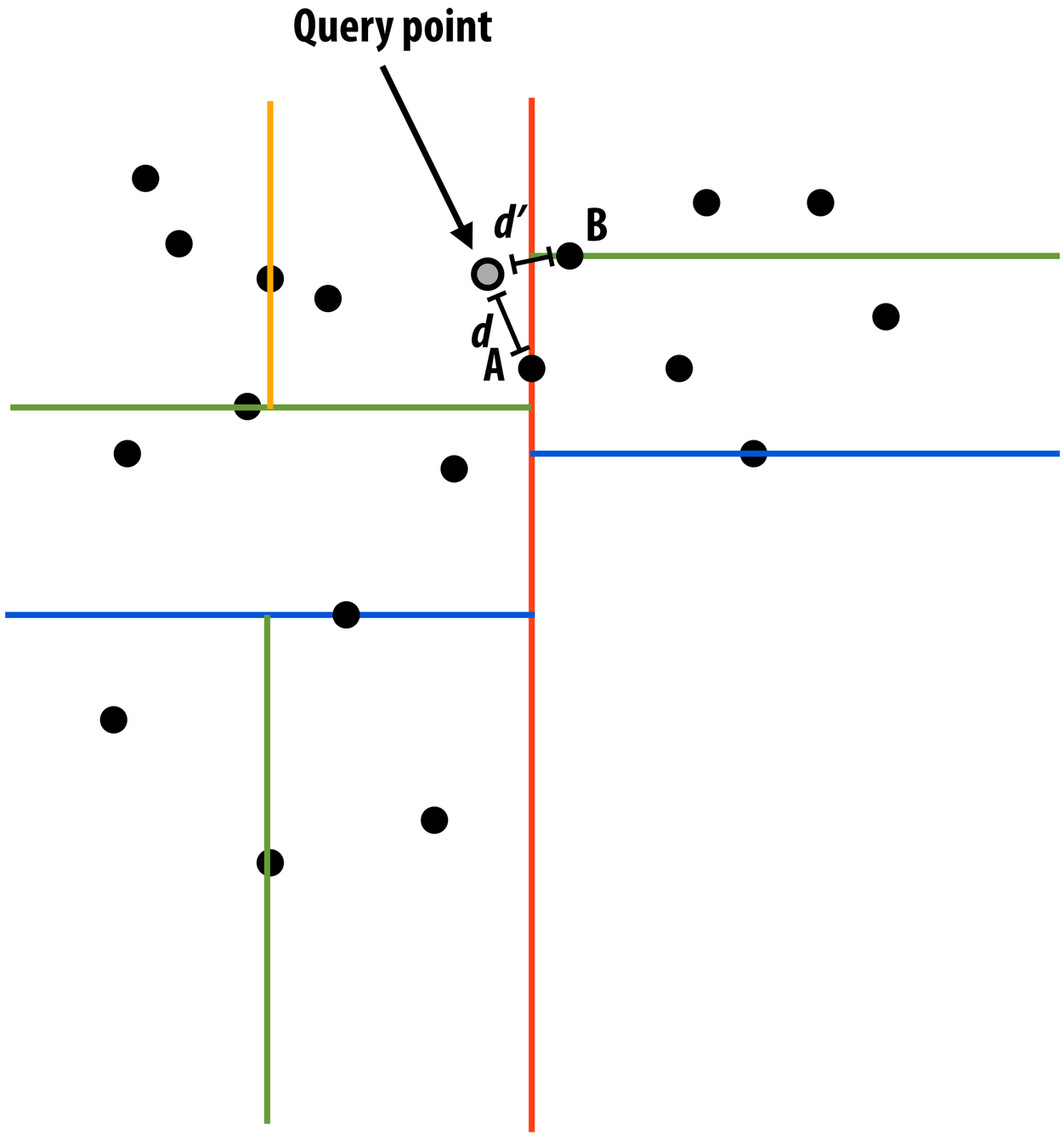


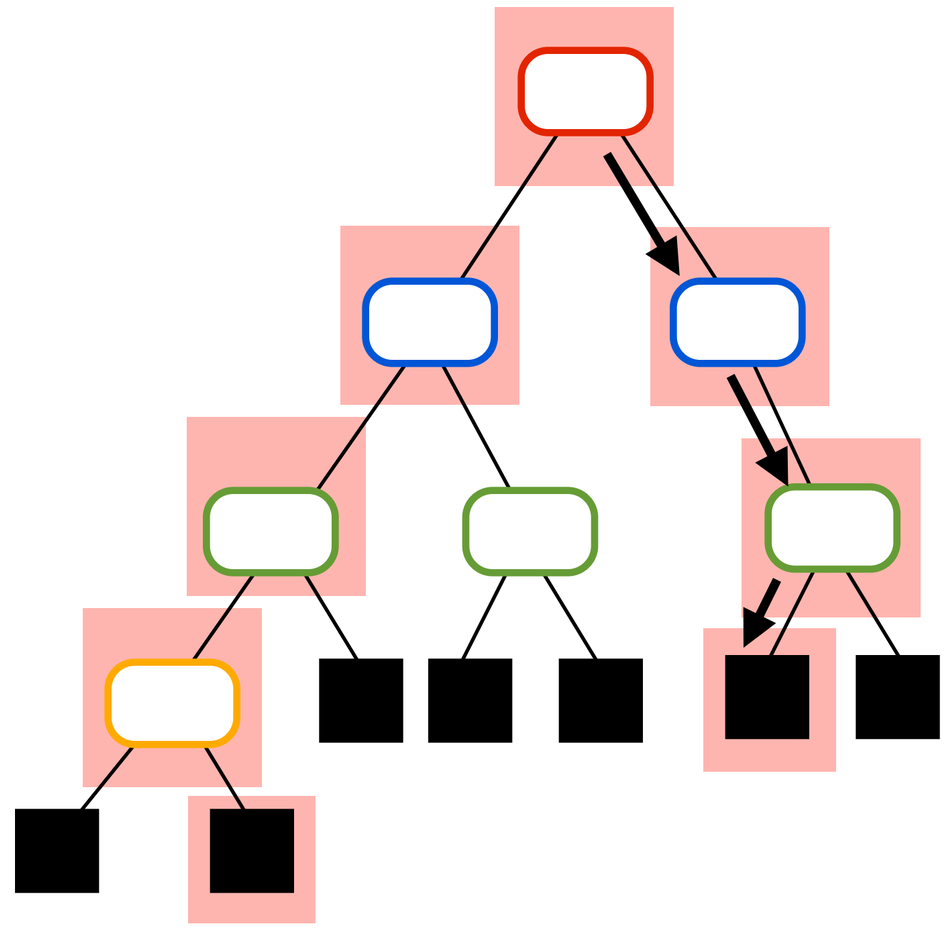


Closest so far: A (at distance *d*)

Nearest neighbor search with K-D tree

Step 2: backtrack: if distance to other cells is closer than distance to closest point found so far, must check points in this cell

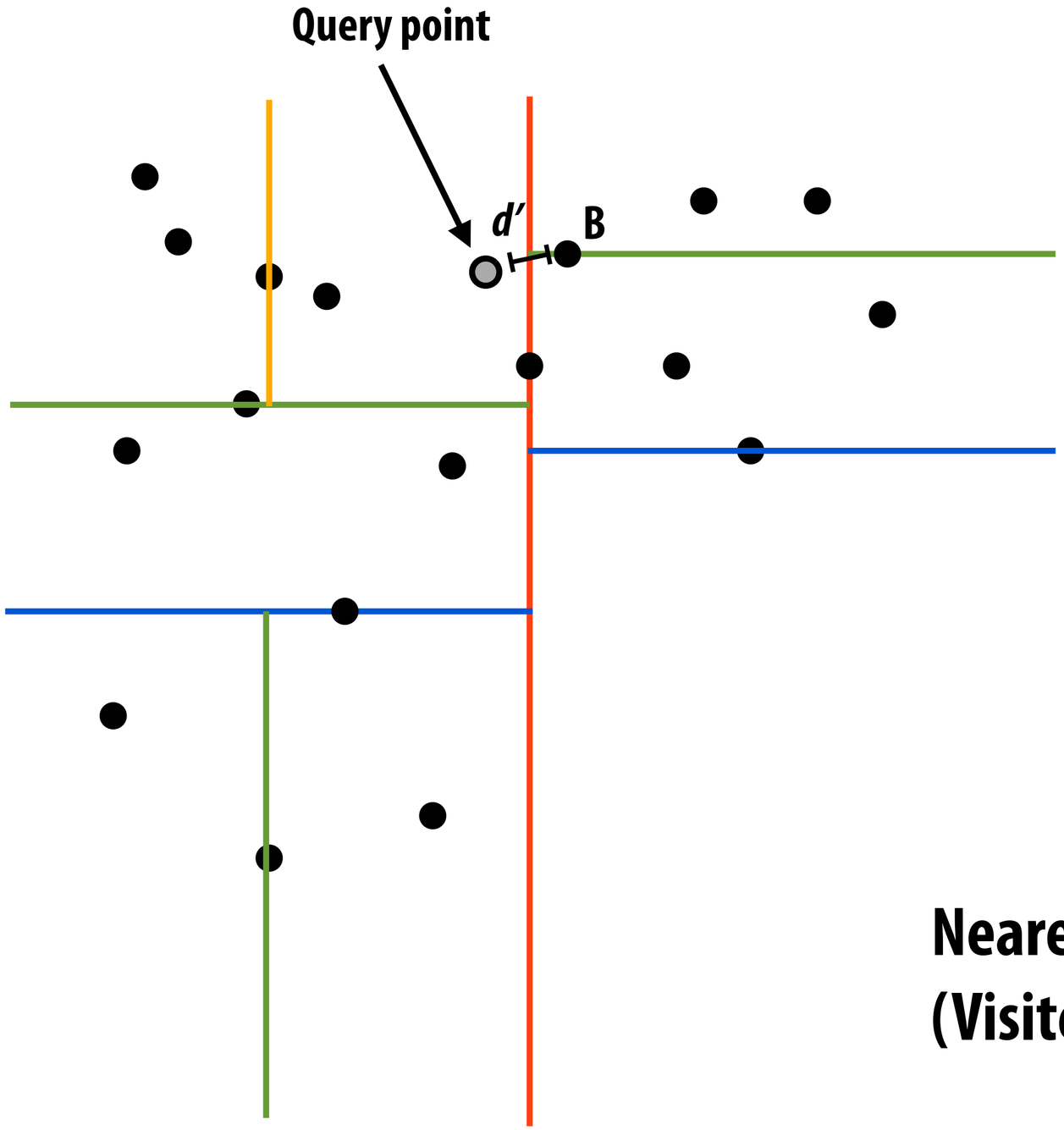


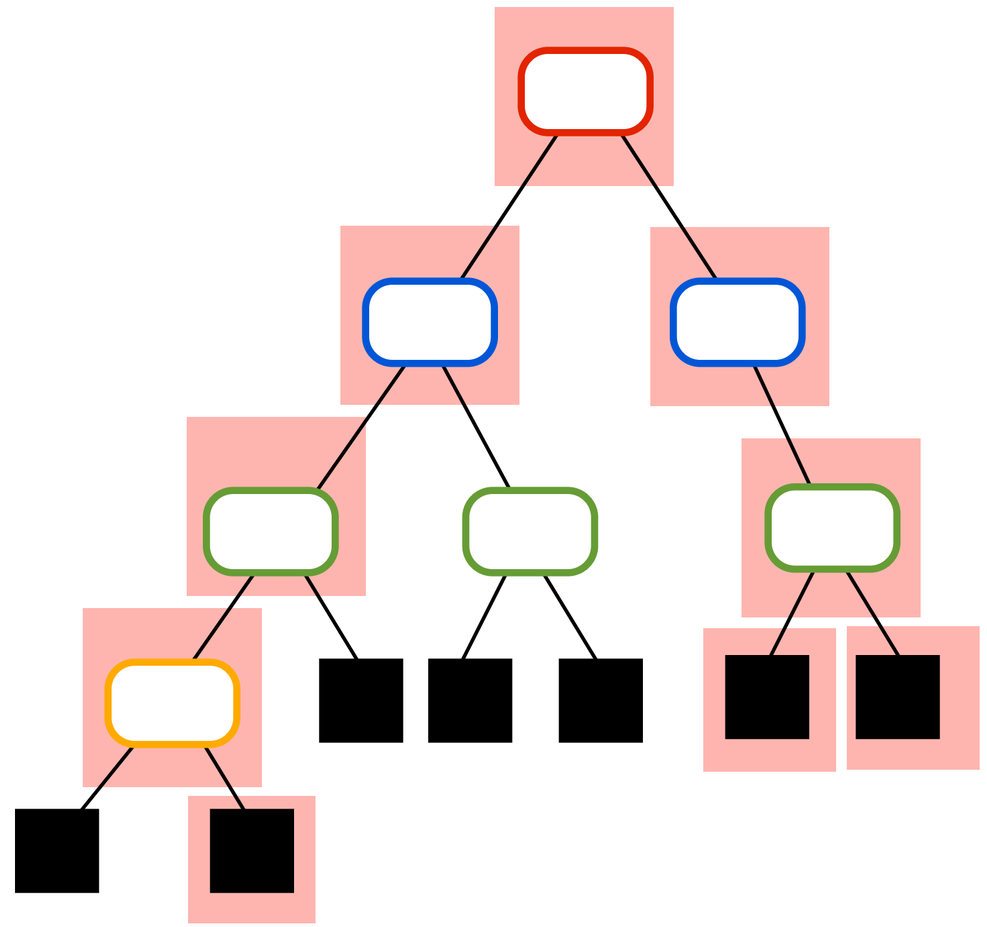


Closest so far: B (at distance *d′*)

Nearest neighbor search with K-D tree

Step 2: backtrack: if distance to other cells is closer than distance to closest point found so far, must check points in this cell





Nearest neighbor result: B (at distance *d'*)
(Visited nodes during query shown in pink)

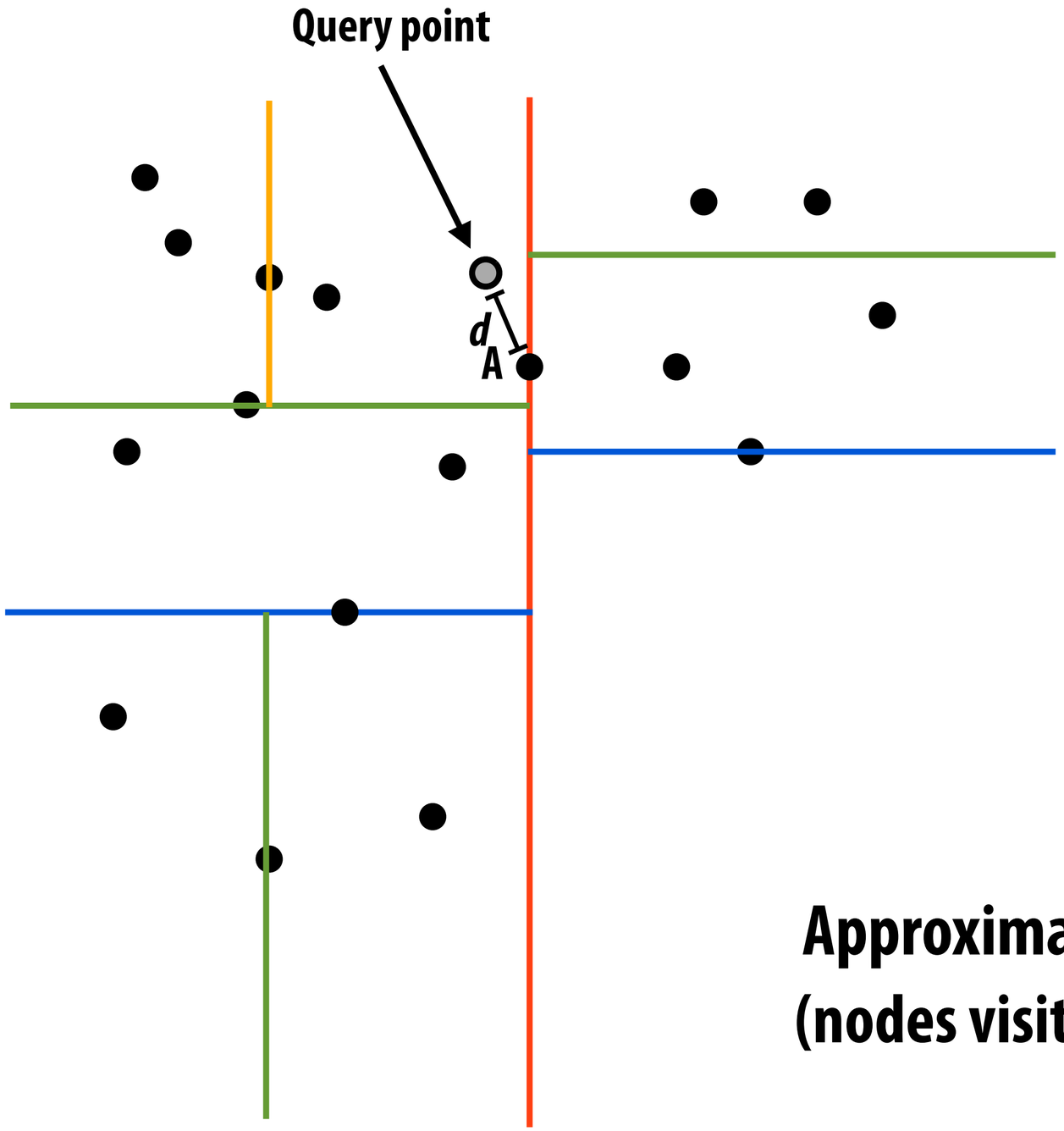
Nearest neighbor search with K-D tree

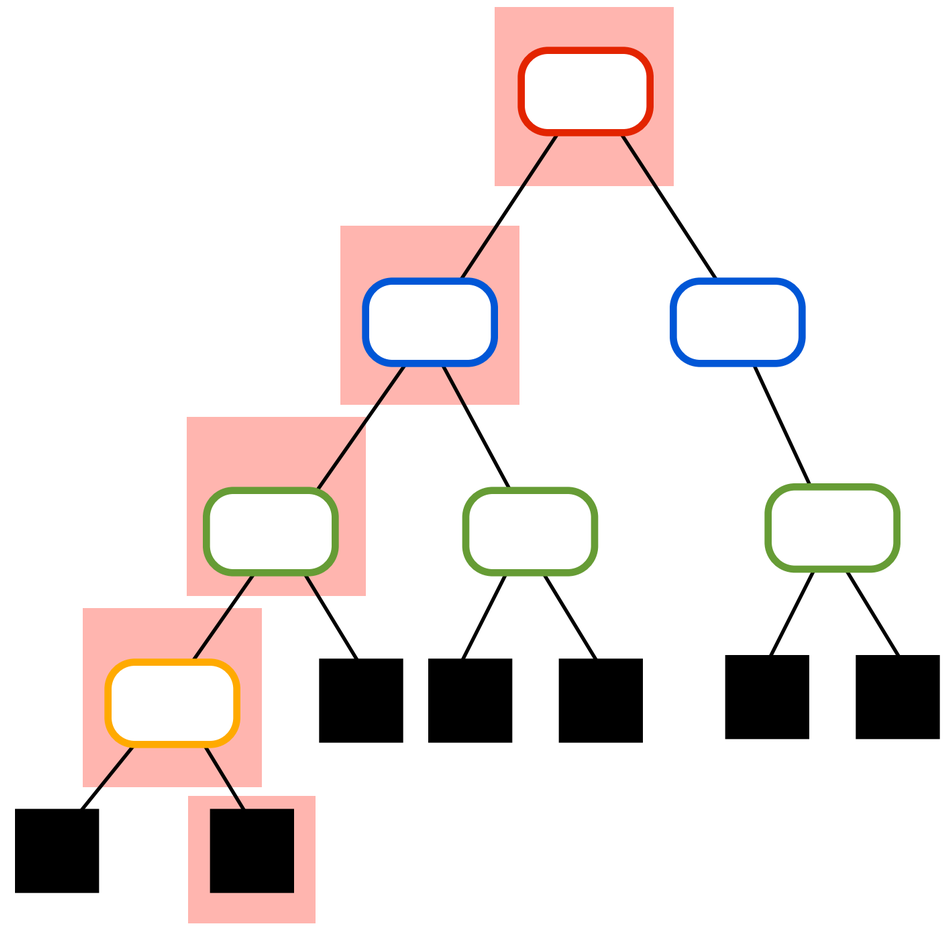
Problem: when K is large (high-dimensional space) backtracking results in visiting nearly all nodes in tree

Rule of thumb for good efficiency: $N > 2^K$

Approximate nearest neighbor (ANN) search

One simple answer: just take closest point in leaf node containing query



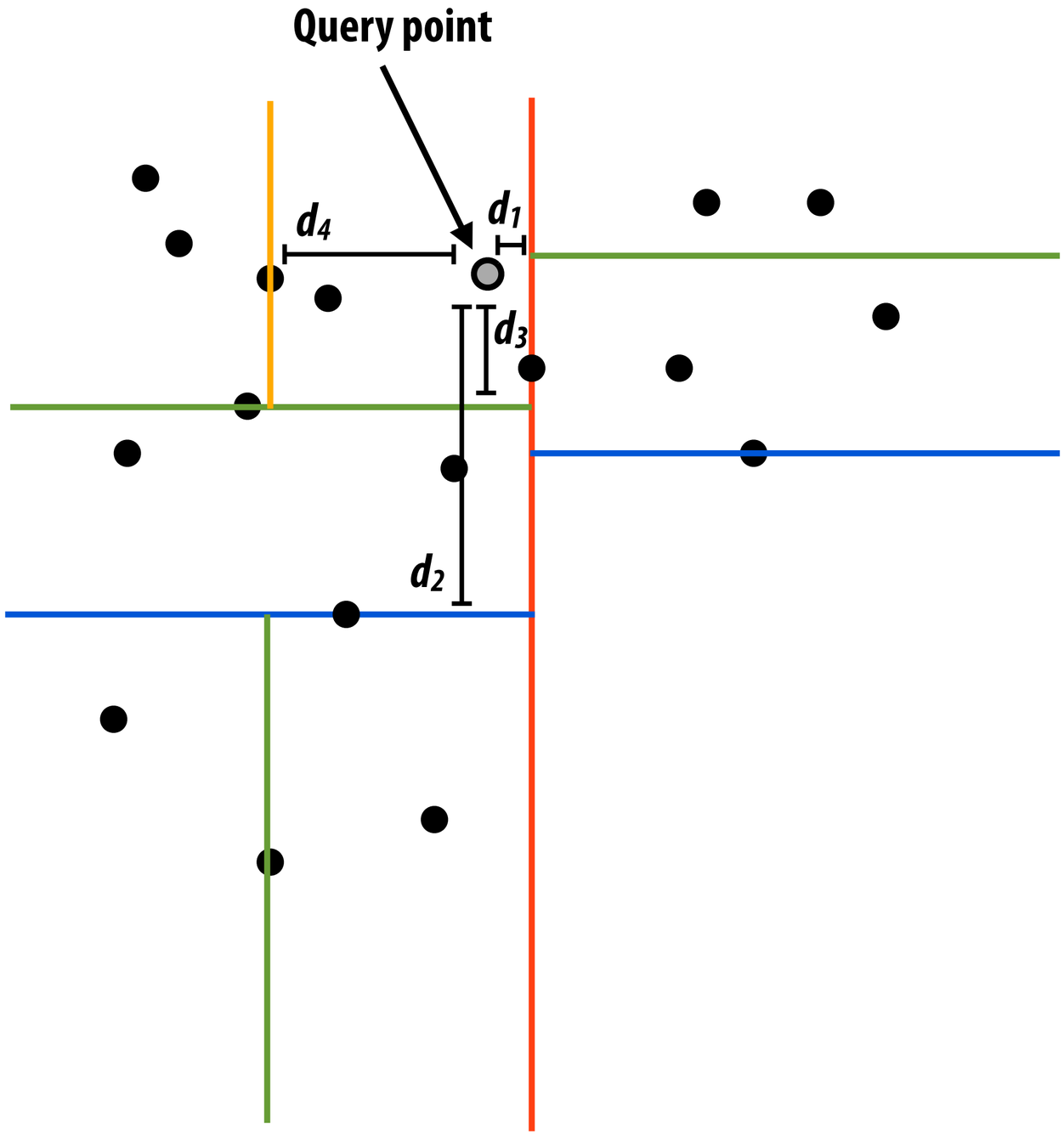


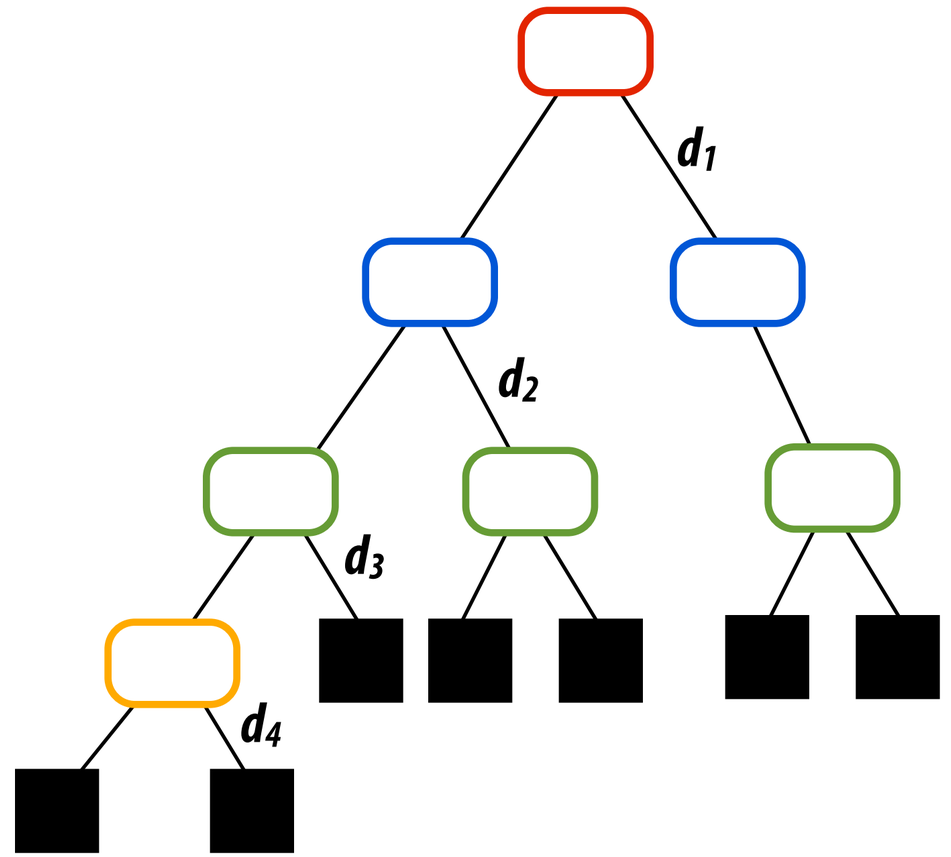
Approximate nearest neighbor: A (at distance *d*)
(nodes visited during query shown in pink)

Approximate nearest neighbor search

Improvement: place nodes in priority queue during downward traversal

Resume downward traversal from closest node to query





Simple K-D tree construction

- To find a partition for a node:
  - Partition axis for which the variance of current data points is the highest
  - Split at the median of the current data points *

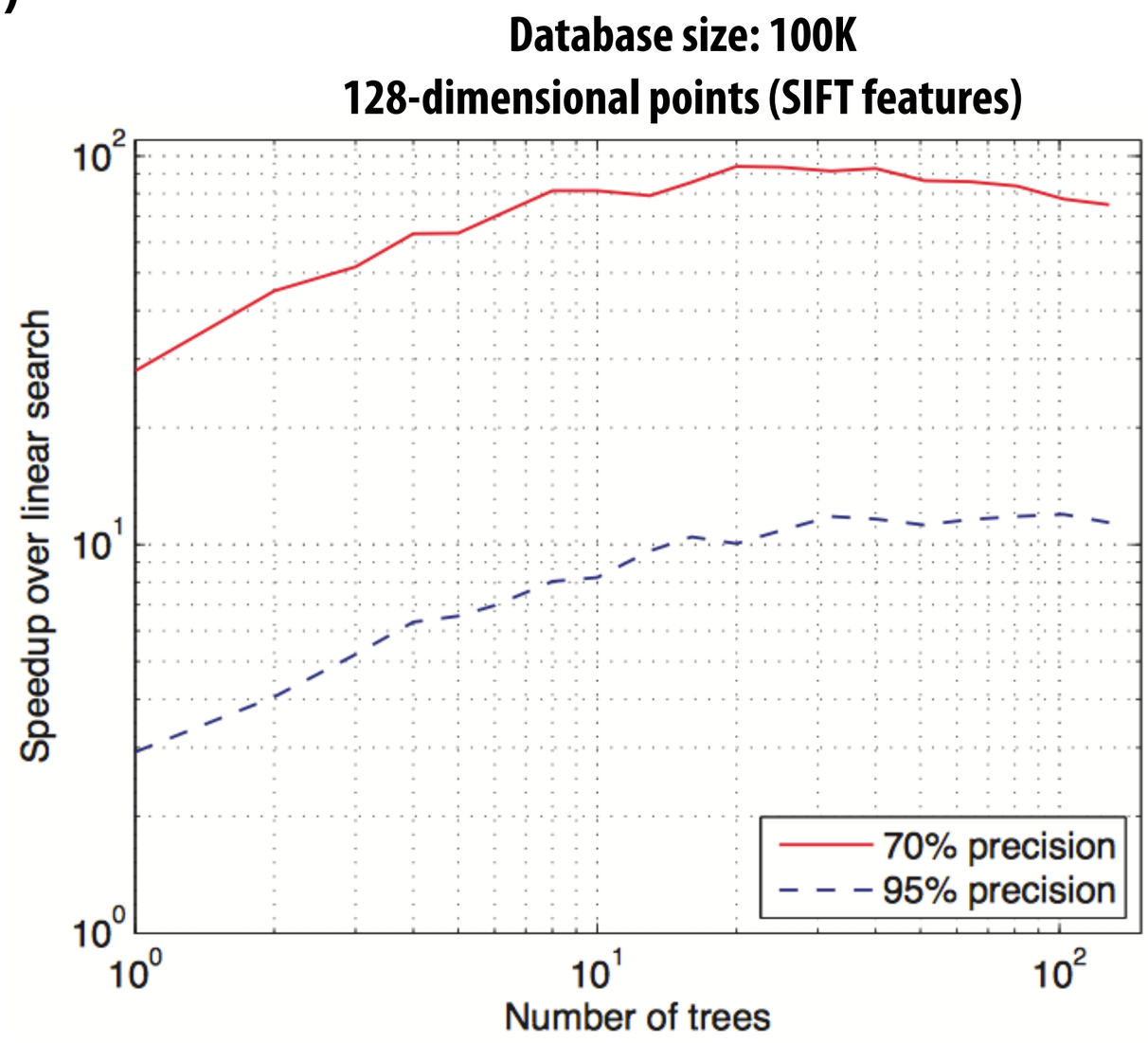
* Better metrics exists (e.g., in 3D for ray tracing the surface area heuristic is common)

Constructing a randomized K-D tree

- To find a partition for a node:
  - Randomly choose axis to partition
    - Draw from distribution weighted proportionally with variance of current data points is the highest
    - Even simpler: pick partition axis by uniformly sampling from top N axes with highest variance
  - Randomly choose partition point on chosen axis
    - Draw from distribution weighted toward at the median of the current data points (make it likely to split near the median of the data points)

Approximate nearest-neighbor (ANN) search using a forest of randomized K-D trees

- Construct a set of random K-D trees ("forest")
- For each tree, find NN in leaf cell containing query
  - Add all nodes (across all trees) traversed along the way to a priority queue (node priority = distance from query to node)
- Take closest of all answers across all trees as an initial ANN
- For top D nodes in queue, resume downward search from that node (D = 5 in figure [Muja et al. 2009])
- Use variant of this solution to get approximate k-NN

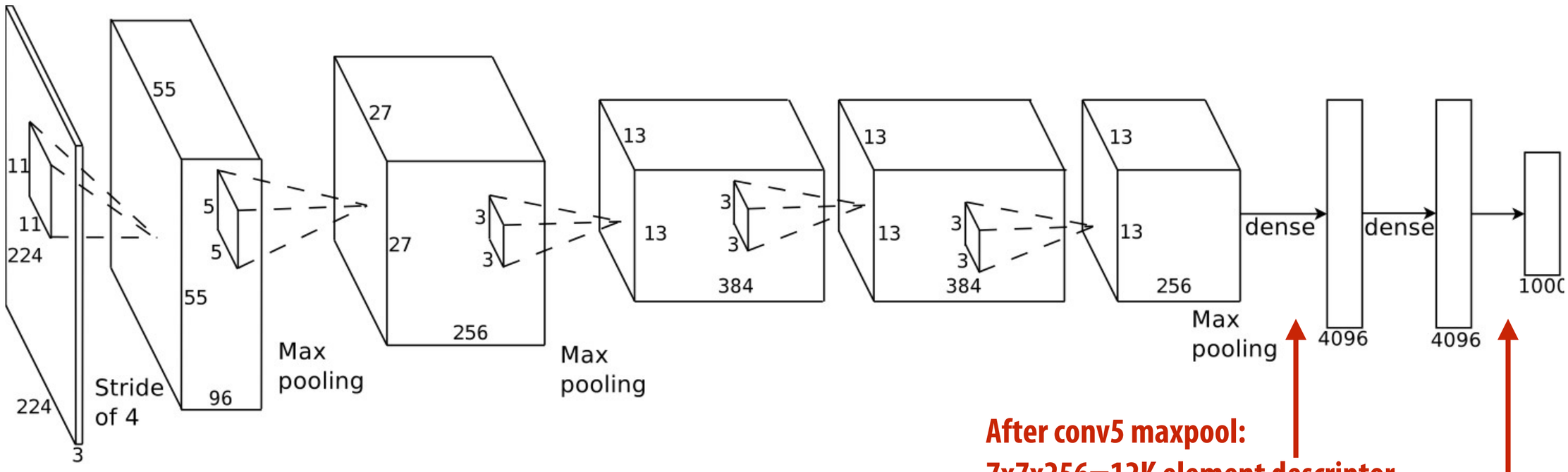


Aside: using approximate k-NN to perform image retrieval, not just keypoint matching

- **Database:**
  - K-D tree of features appearing in database images
  - e.g., SIFT descriptor: $K = 128$
- **Search procedure:**
  - Compute SIFT keypoints for query image
  - For each key point descriptor
    - Find ANN descriptor in database (or k-NN)
    - Add "vote" for DB image containing feature (e.g., vote weighted by distance)
  - Rank database images by final score

Recall: deep networks learn useful intermediate representations

- Image retrieval using descriptor produced by deep network
- Evaluate object classification deep net: e.g., AlexNet
  - Use intermediate output of the network as descriptor
- Use approximate K-NN technique to find similar images in database
  - Common optimization is to further compress descriptor into a small bit code (enables approximate K-NN search via hashing: not discussed today)



These low-D descriptors retain useful information for image classification, embedding is also useful for "semantic" retrieval

Note: last class we discussed compact embeddings learned for image compression. (Likely very useful for visually similar retrieval)

Another aside: retrieval using bit codes

Benefits of NN search in hamming space

1. **Efficient distance computation:**
   - **Hamming distance: number of bits that differ between two *b*-bit codes**

```
int hamming_distance(bitstring x, bitstring y) {
    return count_bits( xor(x, y) );
}
```

2. **Compact database representation:**
   - ***bn* bits to store bitcodes for *n* images in database**
   - **Recall SIFT descriptor: 512 bits per keypoint, hundreds/thousands of keypoints per image!**

K-NN search (K=5) in hamming space:

- 12.9M elements in database
  - Each element corresponds to full-image descriptor
- Quad-core CPU

[Torralba et al. 2008]

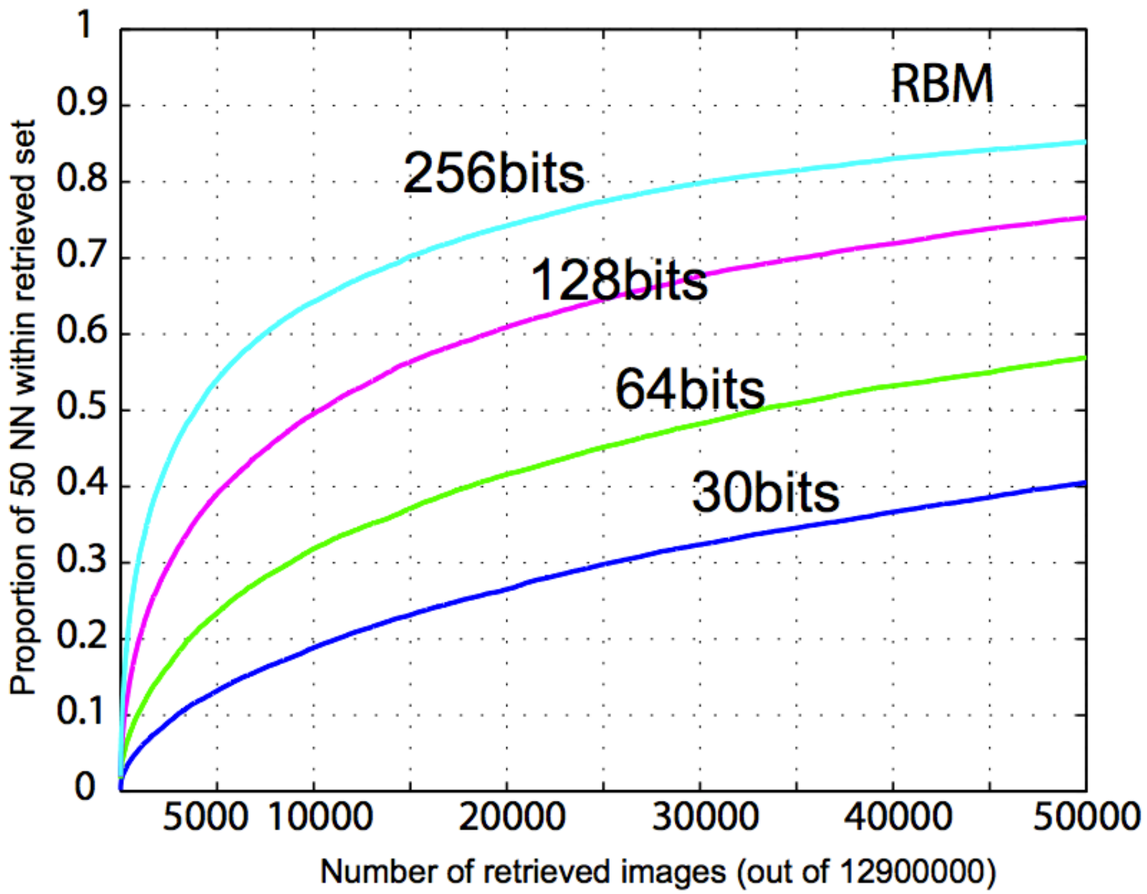
- Brute-force search for top 5 nearest neighbors:
  - 30-bit codes: 400 MB of memory, 74 ms
  - 256-bit codes: 3.2 GB of memory, 0.23 sec

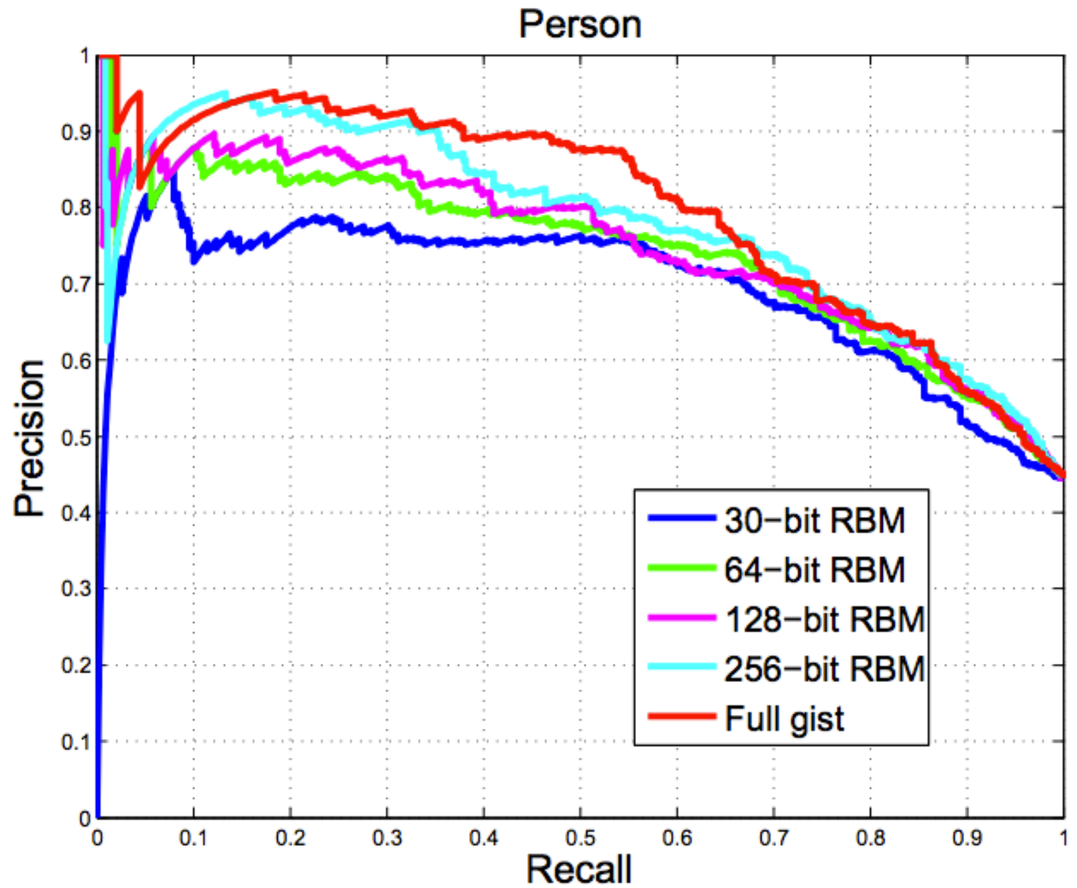
- Two orders of magnitude faster than brute force (and also K-NN tree search) on database containing full-representation GIST (384-float-element) descriptors *

* Unfair comparison: should have compared to approximate k-NN implementation to be more fair since bitcode search results are not the same (see next slide)

Bitcode search "performance"

- Baseline: GIST full image descriptor (384 floats)
- Experiment (left): compute top 50 NN in GIST-space, then measure how many of these NN appeared in the NN results in hamming space
- Experiment (right): object detection by transferring class label (person) from NN's to query image (does query picture contain a person?)





[Torralba et al. 2008]

Benefits of NN search in hamming space

1. **Efficient distance metric computation:**
   - **Hamming distance: number of bits that differ between two *b*-bit codes**
2. **Compact database representation:**
   - ***bn* bits to store bitcodes for *n* images in database**
3. **Potential for using binary code directly as hash table index for O(1) search**

Simple problem formulation

- Find all images within hamming distance *r* from query
- Search process: (assume 2^b indices in hash table)

```
Compute b-bit key for query
For all indices within distance r from query:
    Add images in hashtable[index] to result set
```

- Simple example: *r*=0, just check one bucket

Problem

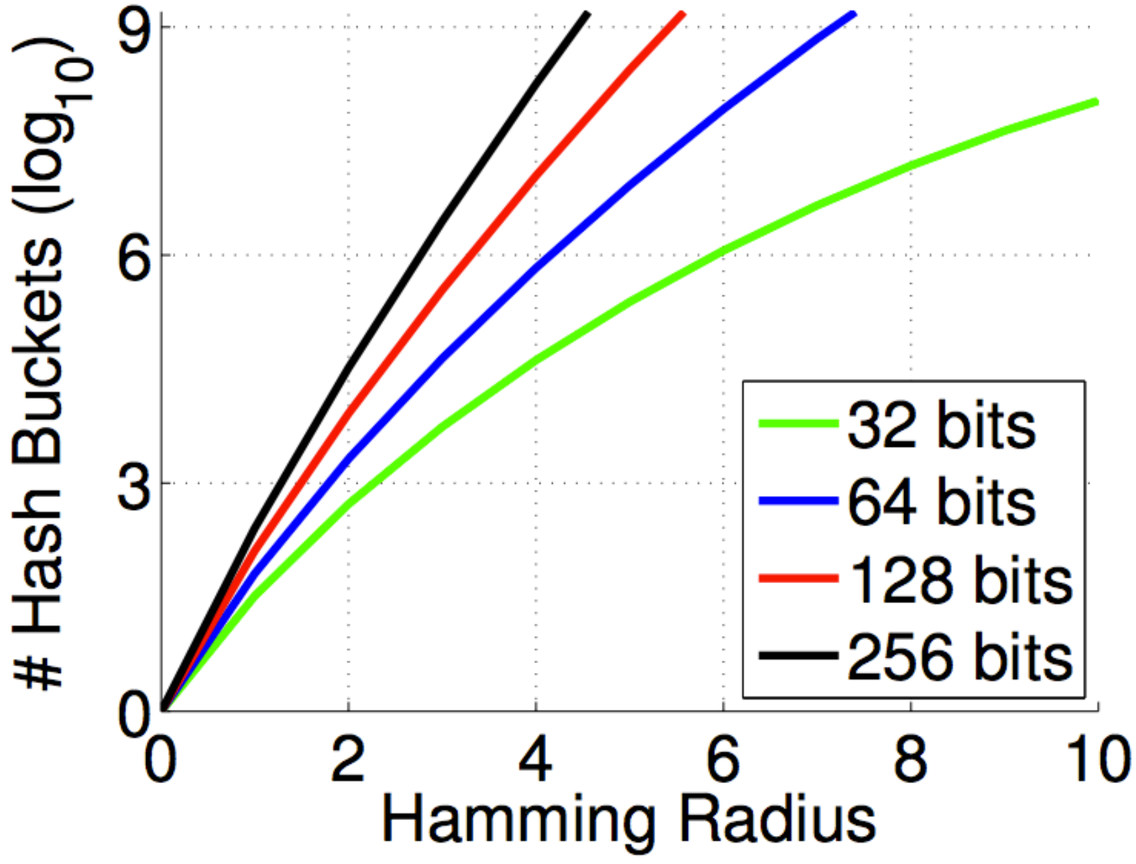
- Number of buckets to check increases rapidly with *r*
  - Volume of the "hamming ball" of radius *r*

- Number of candidate buckets:

$$L(b, r) = \sum_{k=0}^{r} \binom{b}{k}$$

[Norouzi et al. 2012]

- Example: *b*=64, then about 1B buckets for *r*=7
  - If database is smaller than 1B elements, most of these indices will be empty
  - Consider database of millions of elements: faster to just run brute-force linear search through database!



Multi-index hashing: to improve k-NN search in hamming space

[Norouzi et al. 2012]

- **Basic intuition:**
  - Divide query bit string into *m* disjoint *b*/*m*-bit substrings
  - Bit strings that are close in one of the substrings might be close overall
- **Key idea:**
  - If binary codes *x* and *y* differ by less than *r* bits, then in one of their *m* substrings they must differ by less than floor(*r*/*m*) bits.
  - Proof by pigeon-hole principle (if they differed by more than *r*/*m* bits in each substring, then overall *x* and *y* must differ by more than *r* bits

Efficient k-NN using multi-index hashing

- For each set of length-*m* substrings, find substrings of within Hamming radius of floor(*r*/*m*)

- This is a much easier problem!
  - Previously: search needed to examine $L(b, r)$ hash buckets
  - Now need to examine only $L(b/m, \lfloor r/m \rfloor)$ buckets in *m* different hash tables
  - E.g., r=7, m=4, then only need to search with radius 1 in each of the substrings

Full algorithm

- Build *m* hashtables using the length *b*/*m* substrings of elements in the original database

- Given *b*-bit query:
  - For each of the *m* substrings of the query:
    - Find radius floor(*r*/*m*) neighbors and add them to candidate set (using hashtable corresponding to current substring)
    - The candidate set is a superset of the true set of elements within hamming distance *r*, so compute actual set by executing full Hamming distance computation for all elements in candidate set (brute force linear scan)

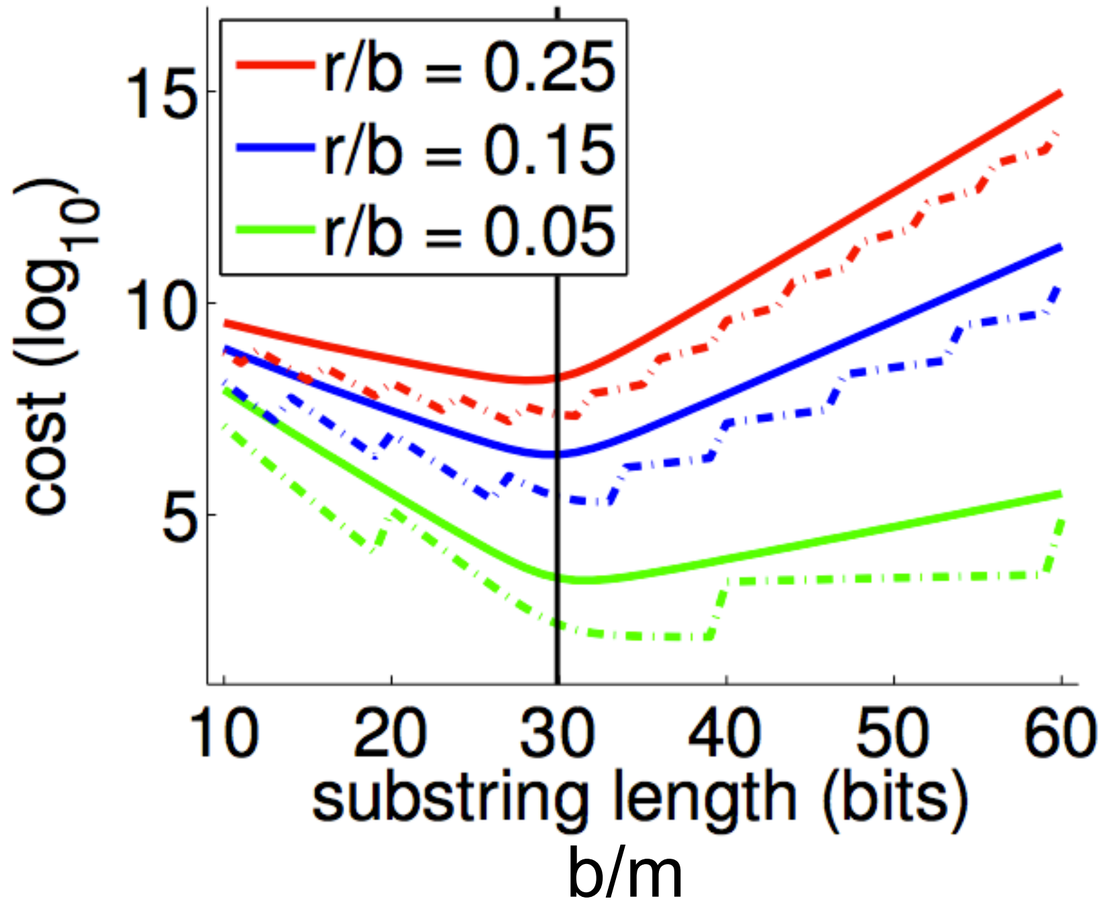
- Storage cost:
  - *bn* bits to represent all descriptors in hash table
  - *m* hash tables referring to these descriptors ($mn\lg_2 n$)
  - In practice, optimal $m=b/\lg_2 n$ so overall storage cost near linear in *n*

How to choose *m*?

- Trade-off between having large substrings (and thus a tight candidate set, but many bucket lookups in substring searches) and having small substrings (cheap substring search but very loose candidate set)
  - Consider $m=b$, substrings are of length 1, but all neighbors in candidate set!

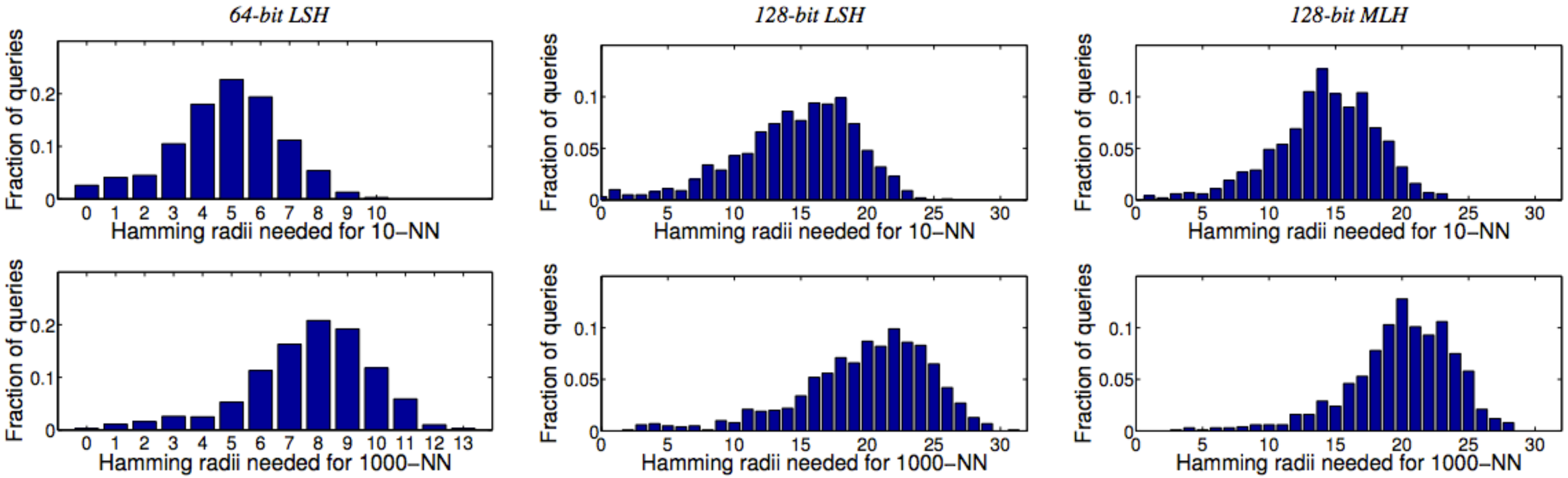
Figure at right:

- Database size: 1B descriptors
- 128-bit codes (b=128)



How to determine *r* from *k*?

- Algorithm finds all database elements within Hamming distance *r*, but we often want *k* nearest neighbors to a query (not all elements within a fixed distance)
- Problem: binary codes not uniformly distributed across Hamming space, so cannot just pick an *r* corresponding to *k* (*r* required to contain knn depends on query)



- Solution: progressively increase *r* until k-NN are found.

Fast image retrieval using bitcodes

Query image

Compute full descriptor

SIFT
SURF
HOG
etc.

Compute *b*-bit binary descriptor (embedding in hamming space)

Matrix-vector multiplication

Neural network evaluation
etc.

search database of binary descriptors

10's of thousands of hamming distance computations

Compute intensive

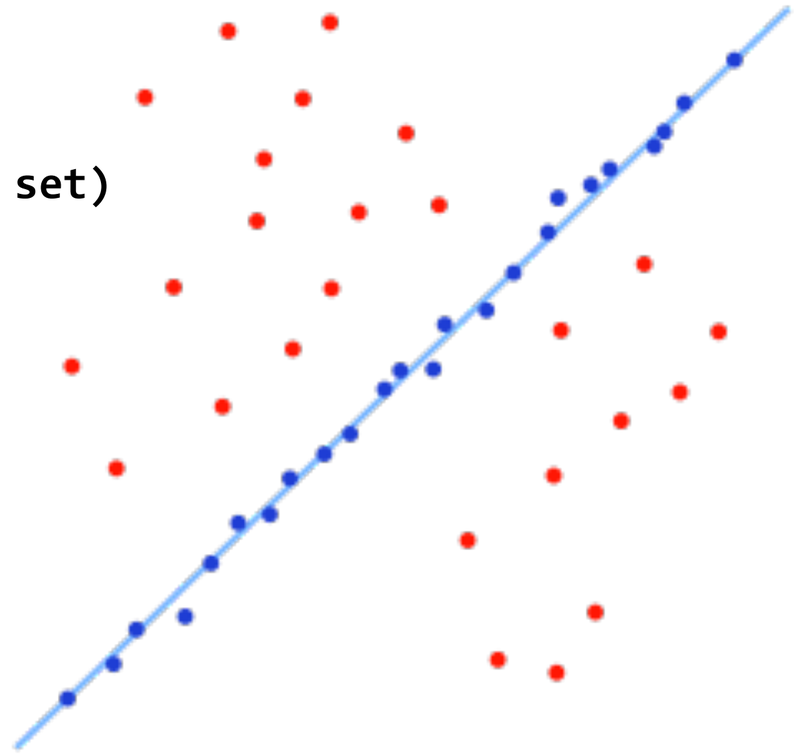
memory intensive

Background part 3: RANSAC

RANSAC

- RANdom Sample And Concensus
- Goal: fit model to collection of noisy data points

```
For i=0 to K:
   Perform random subsampling of datapoints (hypothetical inlier set)
   Fit model Mi to hypothetical inlier set
   For each datapoint d:
      Compute e = error(d, Mi)
      if e < threshold:
          d is in consensus set (it is consistent with model)
   If consensus set for Mi is larger than that for Mbest:
      Mbest = Mi
```



Red data points: outliers
Blue data points: consensus set

Let w = number of inliers / number of data points
= probability of selecting an inlier at random

So:

w^N = probability of selecting all inliers in hypothetical inlier set of size N

$(1-w^N)^K$ = probability that no iteration selects set of all inliers after K RANSAC iterations

Image credit: Wikipedia

3D reconstruction from large photo collections

Step 1:

- Find pairs of "matching" images
  (images that view the same scene points)

Step 1: find matching image pairs from collection

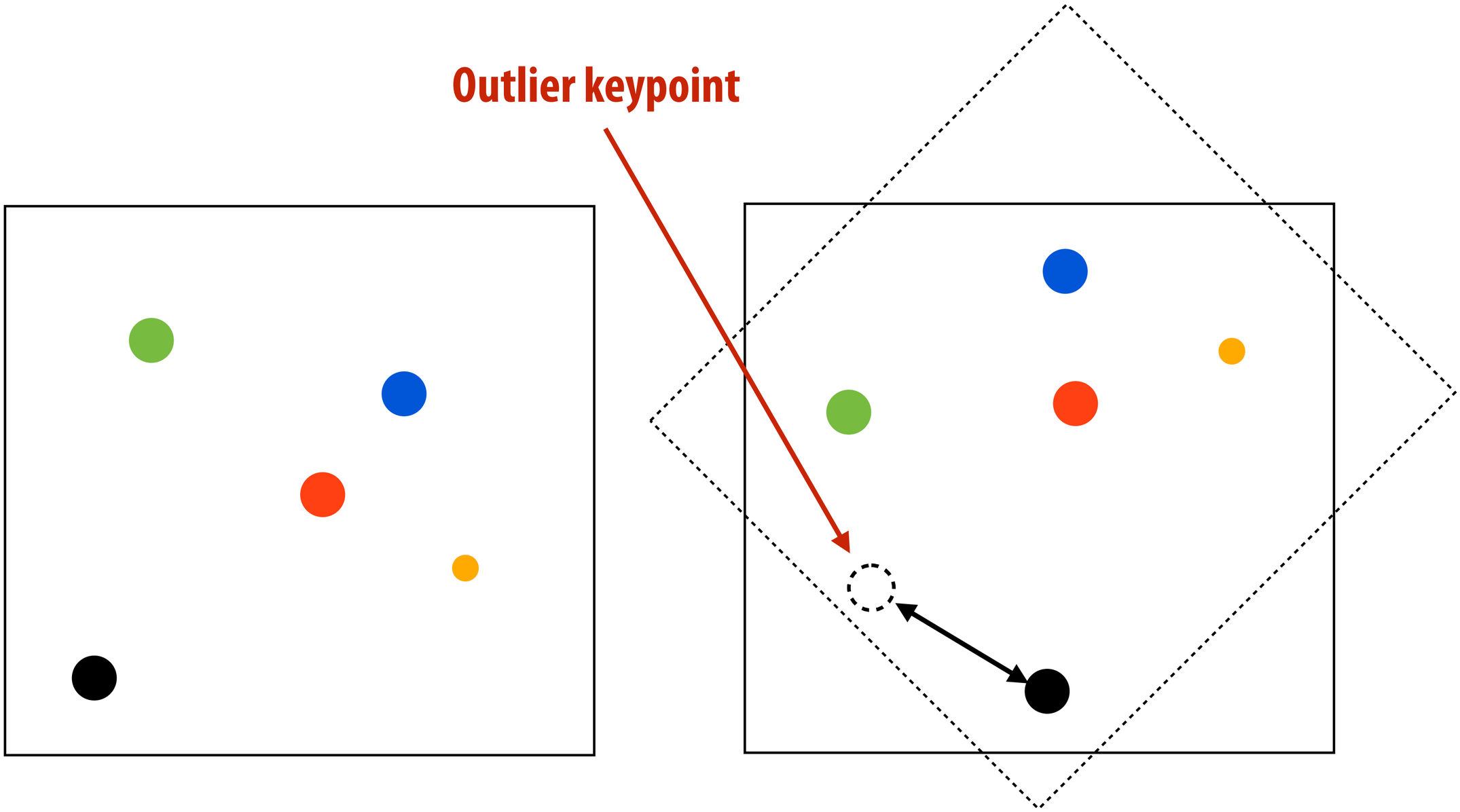
1. Compute feature points for all images (SIFT keypoint descriptors: 128-elements)
   - Thousands of keypoints per image
2. For each pair of images (I, J), determine if a match exists:
   - Find potentially matching keypoints (similar descriptors)

```
for each keypoint i in I:
   // d1, d2 are distance to first nearest neighbor j1 and second NN j2
   (d1, d2) = perform approximate nearest neighbor (ANN) lookup for i
              from keypoints in J
   if (d1/d2 < threshold)
      i in I and j1 in J are candidates for being the same surface point
```

   - Output: pairs of matching keypoints in image I and J
   - Verify matching keypoints: attempt to find geometric relationship between the two viewpoints: estimate a fundamental matrix (3x3 matrix, rank 2) for the image pair using RANSAC:
     - Select matching keypoints at random, estimate F-matrix
     - If there are insufficient inlier keypoints, repeat

   Recall: for keypoint visible at point p_1 in image 1 and p_2 in image 2, p_2 should lie on the epipolar line of p_1: $p_1^{\mathrm{T}} F p_2 = 0$

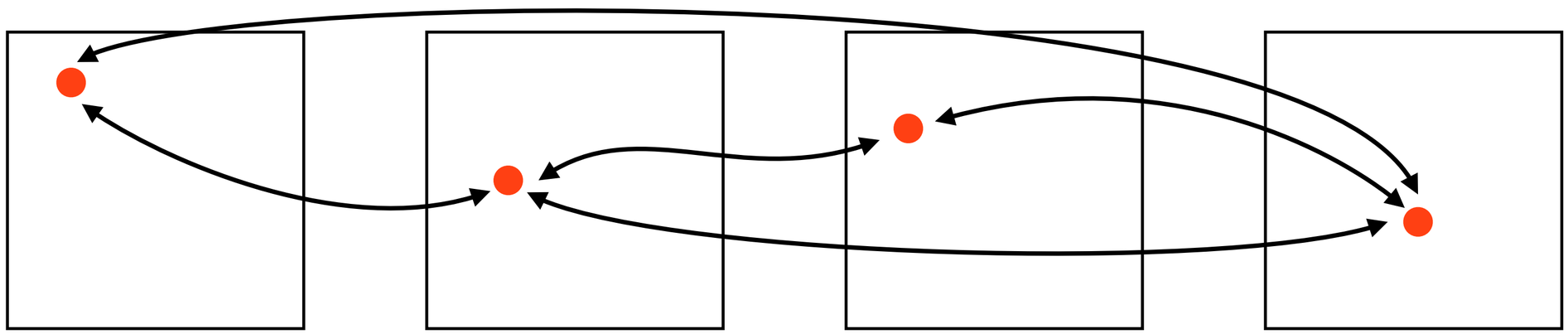
Geometric verification (example in 2D)



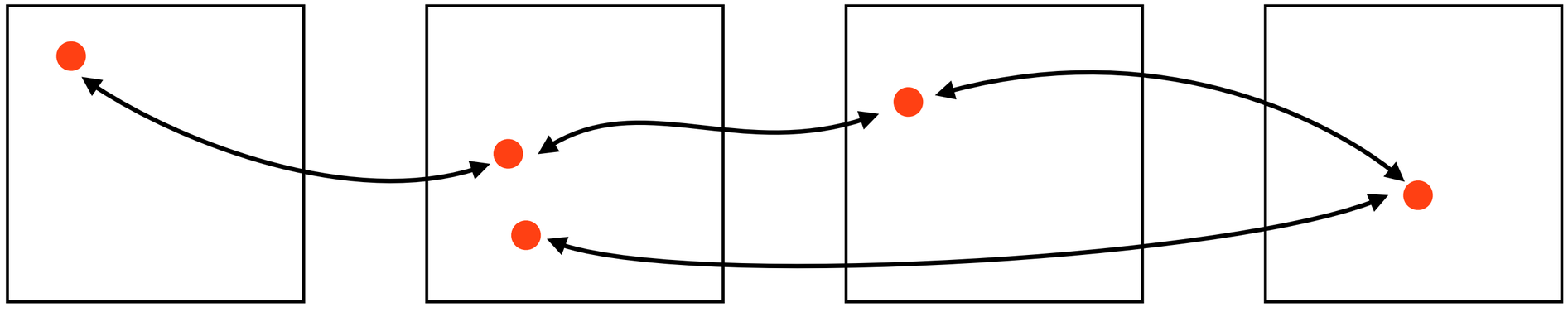
Step 2: organize matches into tracks

Track = connected set of matching keypoints

- **A track corresponds to a single point in the scene**
- **All images in a track are different views of that scene point**
- **Track must contain at least two keypoints**



Consistent track: black arrows indicate matching keypoints in difference images

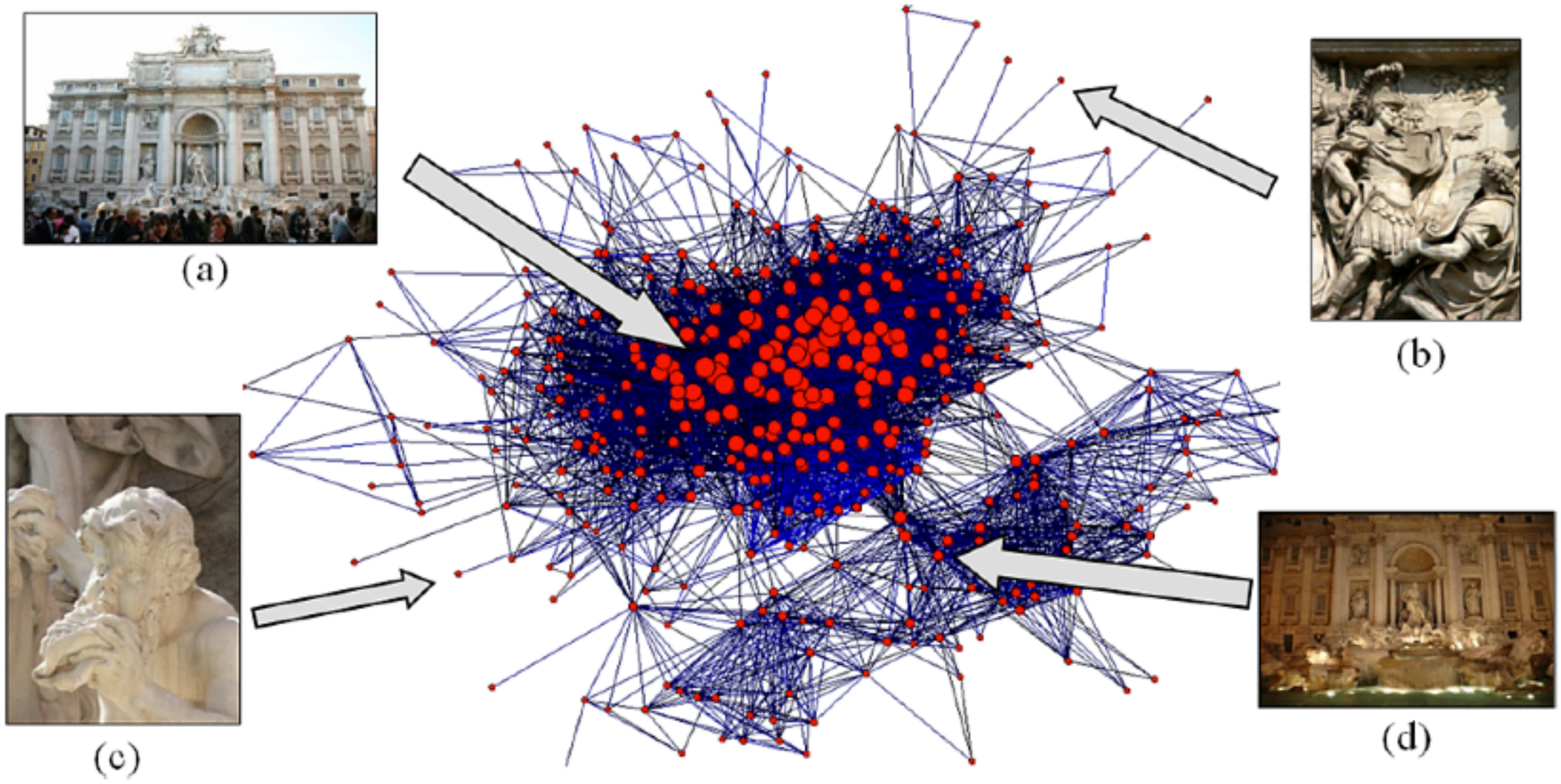


Inconsistent track: contains two keypoints in one image
(clearly, all both keypoints in this image cannot correspond to same scene point)

Image connectivity graph

Graph nodes = images

Graph edges = images that contain matching keypoints



In this example, the two densely connected regions correspond to daytime and nighttime photos

Image credit: Snavely et al. 2007

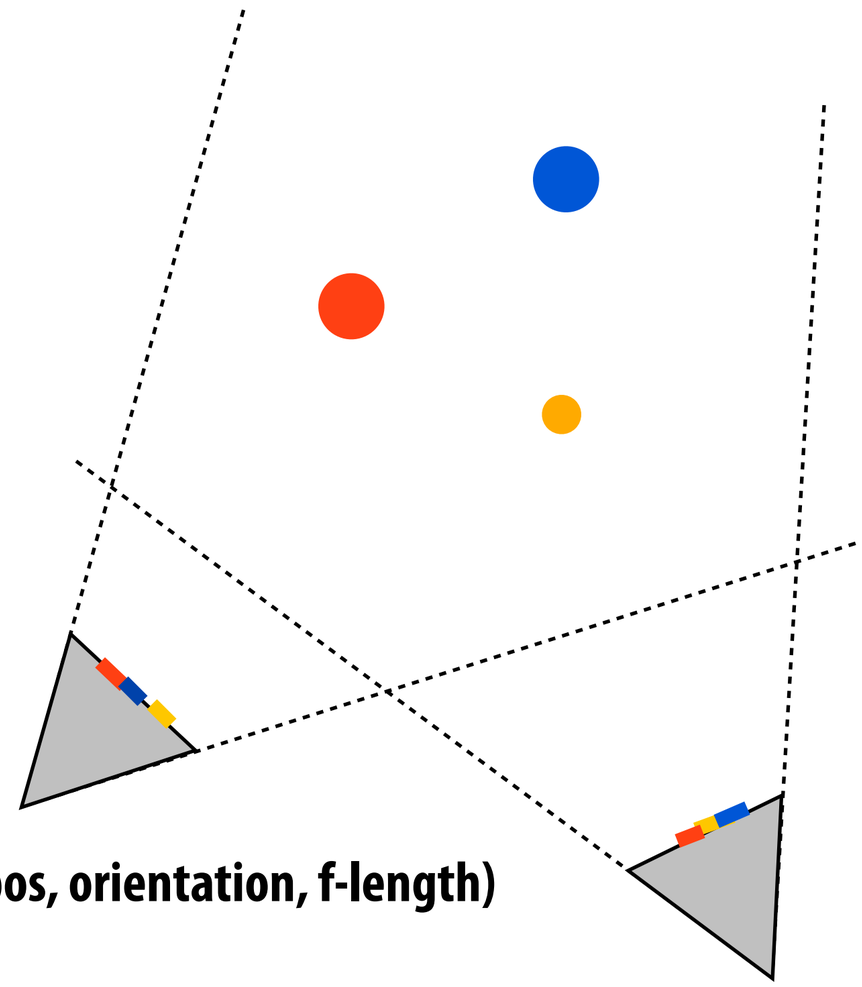
Step 3: structure from motion (SfM)

- Given image match graph and a set of tracks, estimate:
  - Camera parameters for each image (position, orientation, focal length)
  - 3D scene position of each track

- Goal: minimize track re-projection error:
  - Error = SSDs between projection of each track and the corresponding feature in the image.

$$\underset{\theta, X}{\operatorname{argmin}} \sum w_{ij} \left\| P_{\theta_i} X_j - f_{ij} \right\|$$

- Non-linear least squares problem (bundle adjustment)



Where:

P_{θ_i} is the projection matrix into the *i*'th image (depends on camera pos, orientation, f-length)

X_j is the 3D scene position of track *j*

f_{ij} is the 2D keypoint location of track *j* in image *i*

w_{ij} is a binary indicator: designating whether a keypoint for track *j* exists in image *i*

Incremental SfM approach

- Incrementally solve for camera positions, one camera at a time:
  - Begin with data that algorithm is most confident in
- Initialization:
  - Pick pair of images with large number of feature matches and also wide baseline, estimate camera pose from these matches *
  - Triangulate shared tracks to estimate 3D position of scene point corresponding to track
  - Run two-frame bundle adjustment to refine camera poses and 3D position of track
- Add next camera:
  - Choose camera that observes most number of tracks with known positions
  - Estimate camera pose from track matches using RANSAC
  - Run bundle adjustment to refine only new camera and positions of tracks it observes
  - Add new tracks to scene (observed by new camera but not yet in scene)
    - Triangulate positions of new tracks using two cameras with maximum angle of separation
  - Run bundle adjustment to globally refine all camera and track position estimates

* Snavely et al. initialize with image pair that has at least 100 keypoint matches, and for which the smallest percentage of matches are inliers to an estimated homography relating the two images

Algorithm summary

- For each image, compute matching images
  (purpose: find pairs of images containing the scene points)
  - Embarrassingly parallel across images
  - Naively $O(N^2)$, but we discussed faster search strategies

- Organize matching keypoints into consistent tracks (purpose: eliminate contributions from matches that are not consistent with others)
- Until no new camera positions can be estimated:
  - Pick next camera to estimate (currently unregistered image)
  - Refine estimate globally using bundle adjustment (non-linear least squares optimization)

Accelerating match finding

- A naive formulation of match finding is $O(N^2)$: for each image check for match against all other images
  - Large image collections → large N
  - Establishing a match is expensive: (it requires finding a geometric fit via estimating fundamental matrix): ~ a few matches per core per second
  - N=1,000,000, 10 matches per second per core = 3,100 CPU years

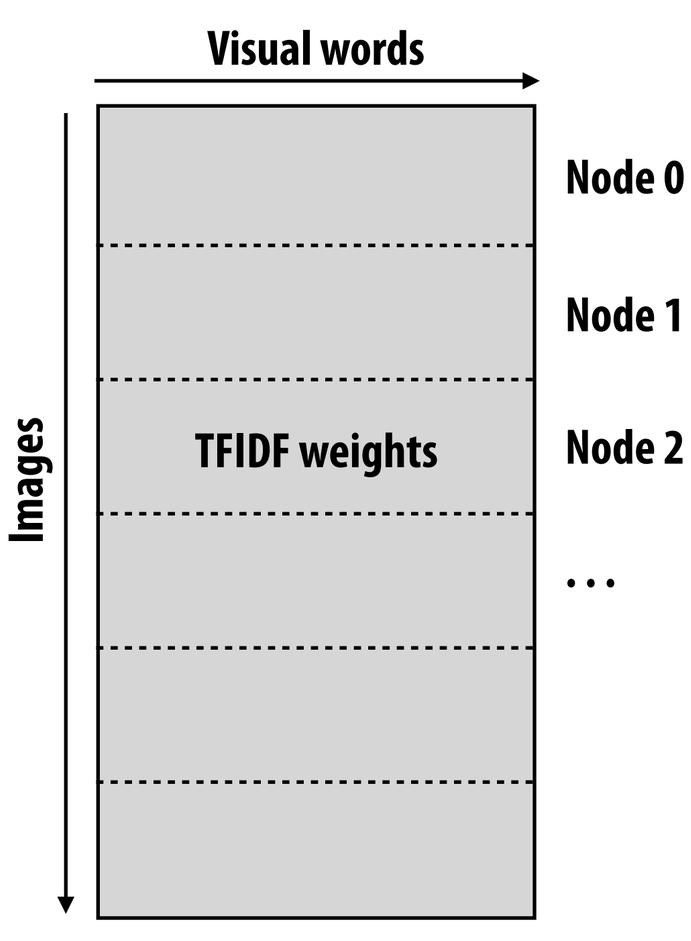
- Must avoid performing expensive check on all possible matches!

- This is a retrieval problem ("quickly find most likely matches")

Accelerating match finding

- Step 1: use fast retrieval techniques to find candidate matching images
  - e.g., use inverted index with TF-IDF weighting to get k-NN for query image
  - Alternative: reduce image to compact bitcode representation

- For each of the *k* candidates, perform expensive geometric verification step
  - Reduce complexity of expensive operations to O(*kN*), where k << N



Parallelization on a distributed system:

1. Partition images across nodes
2. In parallel, compute features/BOW + term-frequencies for all images
   - SIFT features for 1M images: ~ 1-2 TB
   - BOW representation for 1M images: ~ 13 GB
3. Global reduction to compute IDF for each visual word
4. Broadcast IDF information to all nodes
5. Broadcast TFIDF table to all nodes (13 GB)
6. Each node computes top-k NN for the images it owns

Improving match finding for 3D scene reconstruction

- Assume primary goal is to produce a high-quality 3D scene reconstruction (not to compute position of camera for every image in the database)

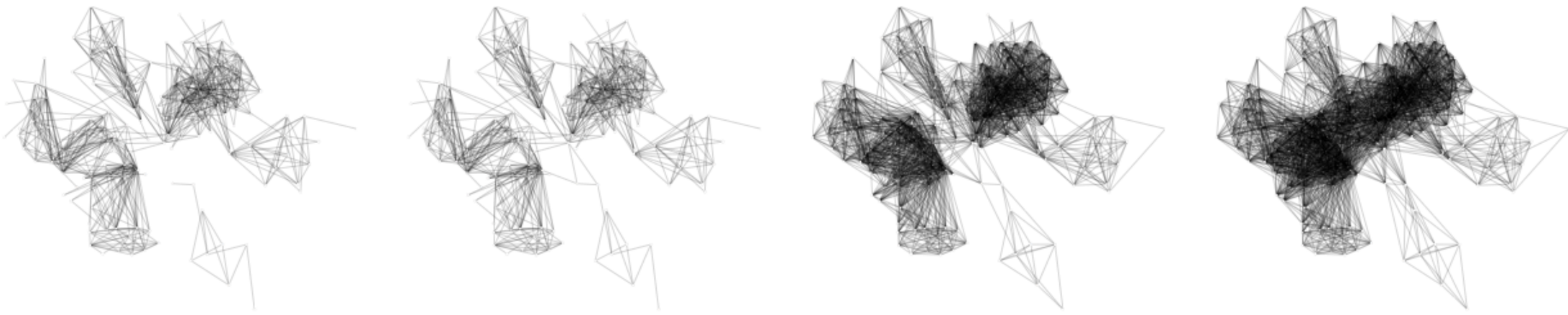
- Want a match graph that is sufficiently dense to enable 3D reconstruction:
  - Want as few connected components in match graph as possible (note: each connected component will be its own 3D scene after reconstruction)
    - Prefer a single, large scene reconstruction, not many "pieces" of scene
  - Want multiple views of the same track (i.e., want multiple images containing the same features to aid robustness of bundle adjustment)

Building a match graph

[Agarwal 2009]

- Step 1: Compute k nearest neighbors using acceleration structure, $k = k_1 + k_2$
- Step 2: Perform geometric verification of top k_1 matches, add graph edge when verification succeeds
- Step 3: Verify next k_2 matches, but only verify image pair (I,J) if image I and image J are in different components of the graph
- Step 4: Densify the graph using several rounds of "query expansion"

```
For each image I
  For each neighbor J of I in graph
    For each neighbor K of J in graph
      If I and K are in different components: verify (I, K)
```



Initial Matches | CC Merge | Query Expansion 1 | Query Expansion 4

Putting it all together (distributed implementation)

1. In parallel across all nodes, compute features and term frequencies
2. Compute IDF weights via reduction, broadcast to all nodes
3. Broadcast TFIDF information (weight table) to all nodes

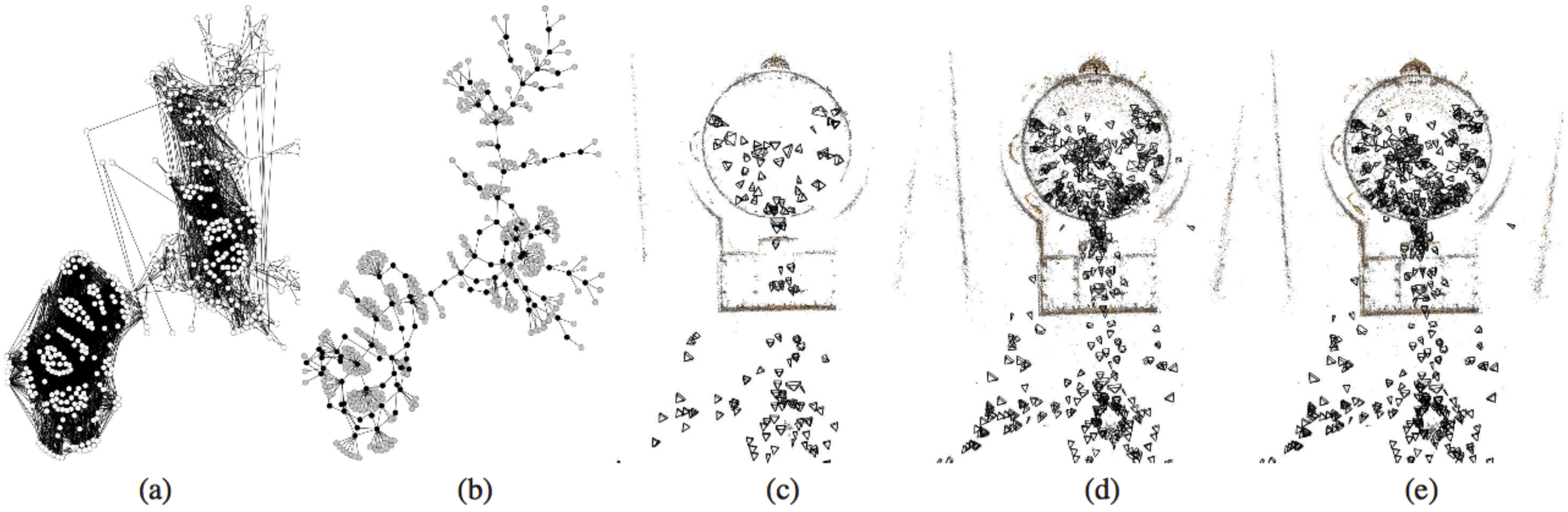
In this example: build inverted index (other search accelerations possible)

4. Independently compute $K=k_1+k_2$ NN on all nodes (all-to-all communication of BOW vectors)
5. For each image *i*, verify top k_1 candidates to create initial match graph (parallel across images)
6. Compute match graph connected components (sequentially on one node is easiest)
7. For each image *i*, verify next k_2 candidates if candidate is not in same graph connected component (dynamic parallelization) as *i*
8. For each image *i*, verify further matches based on candidates returned from query expansion
   - Repeat for N rounds, or until convergence
9. Generate tracks:
   - Each node generates tracks for the images it owns (in parallel across nodes)
   - Then merge tracks across nodes (parallel reduction, or sequentially on home node)
10. Compute graph skeletal set (next slide)

Match graph sparsification

[Snavely 2008]

- All images do not contribute accurately to coverage/accuracy of 3D reconstruction
- For efficiency, we'd like to compute SfM using a minimal set of images (the "skeletal set") that yields similar reconstruction quality as the full match graph



Match graph

Skeletal set (black nodes are skeletal set images, gray nodes are remaining images)

Reconstruction from skeletal set

Adding additional images with pose estimation

final result post bundle adjustment

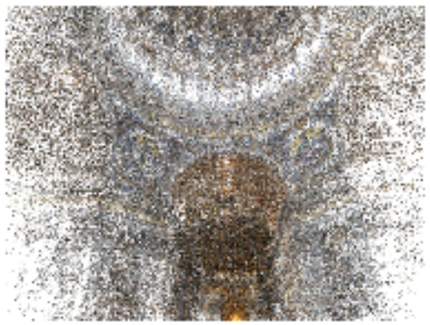
Result: 2x to 50x improvement in reconstruction performance

Image credit: Snavely et al. 2008

Results

"Building Rome in a Day" Agarwal et al. 2009

Data set	Images	Cores	Registered	Pairs verified	Pairs found	Time (hrs) Matching	Skeletal sets	Reconstruction
Dubrovnik	57,845	352	11,868	2,658,264	498,982	5	1	16.5
Rome	150,000	496	36,658	8,825,256	2,712,301	13	1	7
Venice	250,000	496	47,925	35,465,029	6,119,207	27	21.5	16.5









Building Rome on a Cloudless Day

Reconstruction from 2.8M images on a single PC in one day (Frahm et al. ECCV 2010)

Step 1: reduce all images to 512 bit binary code (64 bytes/image)

- **image → GIST descriptor+RGB → binarize via locality sensitive binary code**
- **millions of image descriptors can fit in RAM**

Step 2: cluster all images based on bitcode descriptor (seed cluster centers with GPS data if available)

Step 3: validate clusters using geometric verification

- **pick *n* images closest to cluster center, see if large fraction of pairs pass geometric verification (reject cluster if not)**

Step 4: compute single "iconic image" per cluster (image with most successful verifications with other images in the cluster)

- **remove images from cluster if they don't geometrically match the iconic image**

Step 5: build match graph from iconic images (using image search, or if available, compute GPS position of clusters by averaging geotags from images, then attempt to match all iconic images within 150m

Step 6: use SfM techniques to estimate camera pose, 3D pose for iconics

Step 7: incorporate keypoints from non-iconic images into estimate (no SfM run since matches already known)

Building Rome on a Cloudless Day

Reconstruction from 2.8M images on a single PC in one day (Frahm et al. ECCV 2010)

Dataset	Gist & Clustering	SIFT & Geom. verification	Local iconic scene graph	Dense	total time
Rome & geo	1:35 hrs	11:36 hrs	8:35 hrs	1:58 hrs	23:53 hrs
Berlin & geo	1:30 hrs	11:46 hrs	7:03 hrs	0:58 hrs	21:58 hrs

Dataset	total	LSBC clusters	#images iconics	#images verified	#images 3D models	#images largest model
Rome & geo	2,884,653	100, 000	21,651	306788	63905	5671
Berlin & geo	2,771,966	100, 000	14664	124317	31190	3158

Related task: location recognition

Given a new image, how can we leverage an existing 3D reconstruction to estimate the camera's location and orientation?

Query image

Database image (keypoints shown)

- First-thought solution:
  - For each keypoint in query image, finding matching tracks in scene database of all images (recall: tracks correspond to scene features)
    - Possible implementation: ANN lookup using KD-tree built over database
  - Then attempt camera pose estimation for query given the collection of matches

Left image credit: Mark Ordonez (via Flickr), Right image credit: Li et al. 2010

Observation

- Not all scene database features are equally useful in matching images
- Many scene features appear in many images
  - Example below: clock face on tower is most frequently observed point in database (many tourist images of Dubrovnik, Croatia on Flickr contain this feature)

Observation

- Not all scene database features are equally useful in matching images
- Many scene features appear in many images
  - Example below: clock face on tower is most frequently observed point in database (many tourist images of Dubrovnik, Croatia on Flickr contain this feature)

Idea: use up-front knowledge of likelihood of scene points to appear in images… to accelerate image feature matching

Idea: analyze image database to accelerate matching

- Previously in this lecture: organize database feature points into index (e.g., KD-tree) to accelerate search

- New idea: leverage co-occurrence and frequency of occurrence
  - We do not seek to find all matches in the database with the query, only enough matches to estimate query image's camera pose
  - Co-occurrence: it is sufficient to search over a small subset of scene points (since many scene points co-occur in the same images and are similarly useful for pose estimation)
  - Frequency of occurrence: search for the points that are most likely to be in the query image

K-coverings of scene images

- Subsample database: compute scene keypoint set that is a K-covering of all images in the database
  - K-cover: set of points such that at least K points are present in each image
  - Simple greedy algorithm to compute K cover:

```
S = {}    // set of points in covering
sort all scene points by number of images they appear in
while K-cover not reached by S:
    add point P appearing in largest number of images into S
```

- Precompute two K-coverings for image database
  - P^s: 5-covering, capped to at most 2,000 points
  - P^c: 100-covering

Localization algorithm

- Query = list of feature points
- Database = list of feature points for all images in collection
- Idea: rather than search database for matches to points in query image, search query list for matches database feature points
- Simple algorithm: tests database points against query image in priority order

```
Compute Kd tree for points in query
Initial prioritization of database points:
   Highest priority points: Ps
   Next highest priority points: Pc
   Remaining points: priority = number of images point is visible in

while additional matching points are required:
    Attempt to match highest priority point against query's points
    if match found:
        for each DB image I containing matched point:
            Increase priority of all DB points in I
```

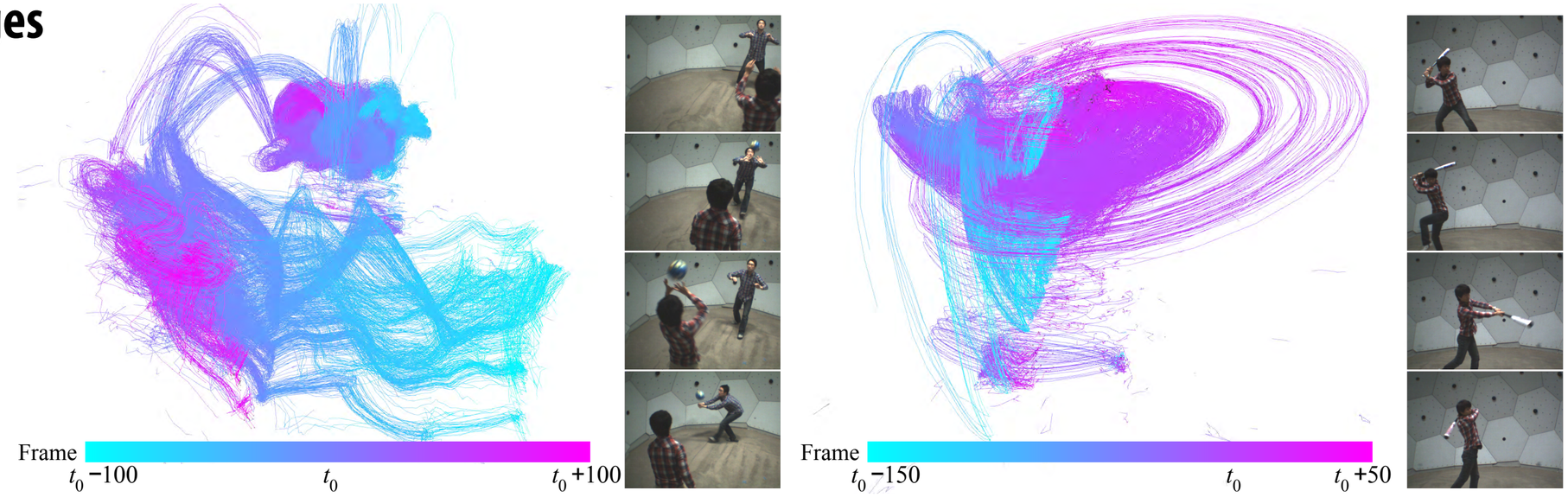
Dynamic reprioritization of DB points based on co-occurrence with matched points.

Recap: how the algorithm works

- Test the most likely to match images from the database first
  - Recall: only need a few matches to estimate 3D camera pose of query
- Once a match is found, leverage co-occurrence of points in database images to predict new matching points
- Desirable system behavior: optimize for the common case
  - Common query images get found very quickly
  - Uncommon query images take longer to localize
  - Memory efficient: don't need to store an additional acceleration structure for the entire database of images

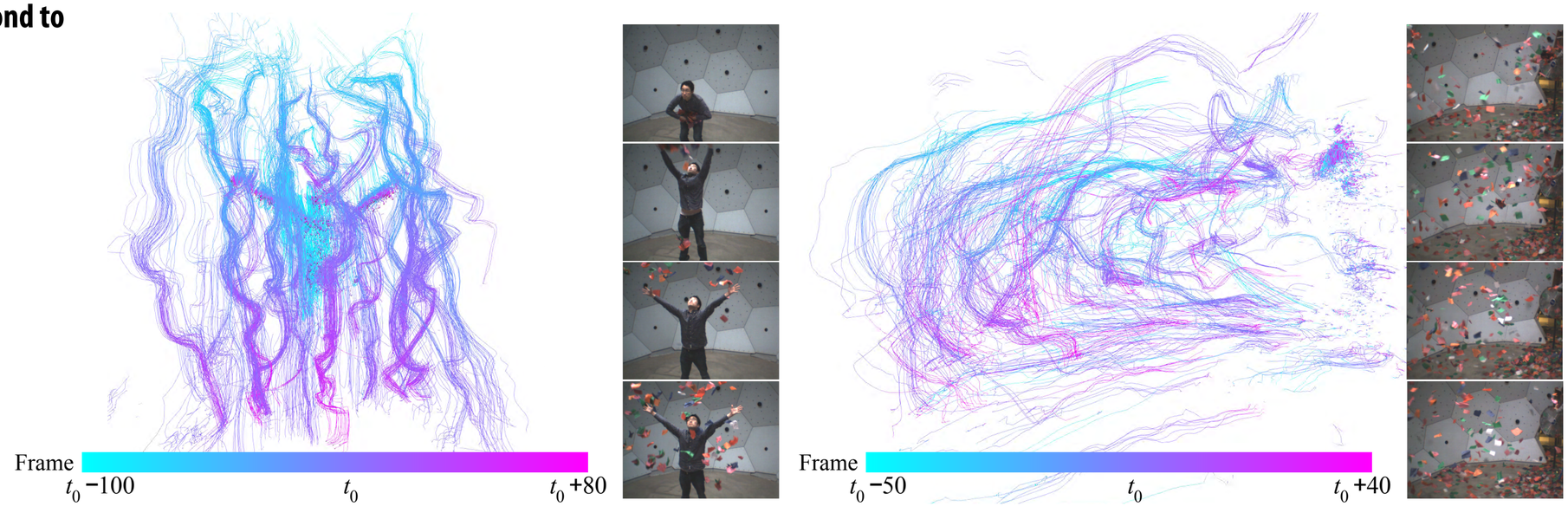
Modern challenges

- Interest in 3D reconstruction of moving scenes
  - Capturing dynamic environments (navigation, localization)
  - Capturing humans (performance capture, entertainment, VR)
- Correspondence problem is more difficult because scene keypoints may not share the same 3D position across images



(b) Volleyball (c) Bat swing

Another challenge is latency:
Autonomous agents typically want to respond to changes in world in ms
(also true of VR)



[Image credit: Joo et al. 2014]