

# Fast Indexing Method

Dongliang Xu  
22th.Feb.2008

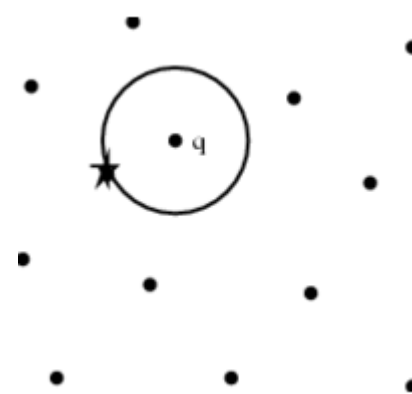
# Topics (Nearest Neighbor Searching)

- **Problem Definition**
- **Basic Structure**
  - Quad-Tree
  - KD-Tree
  - Locality Sensitive Hashing
- **Application: Learning**
  - BoostMap: A Method for Efficient Approximate Similarity Rankings
- **Application: Vision**
  - A Binning Scheme for Fast Hard Driver Based Image Search\*
  - Fast Pose Estimation with Parameter Sensitive Hashing

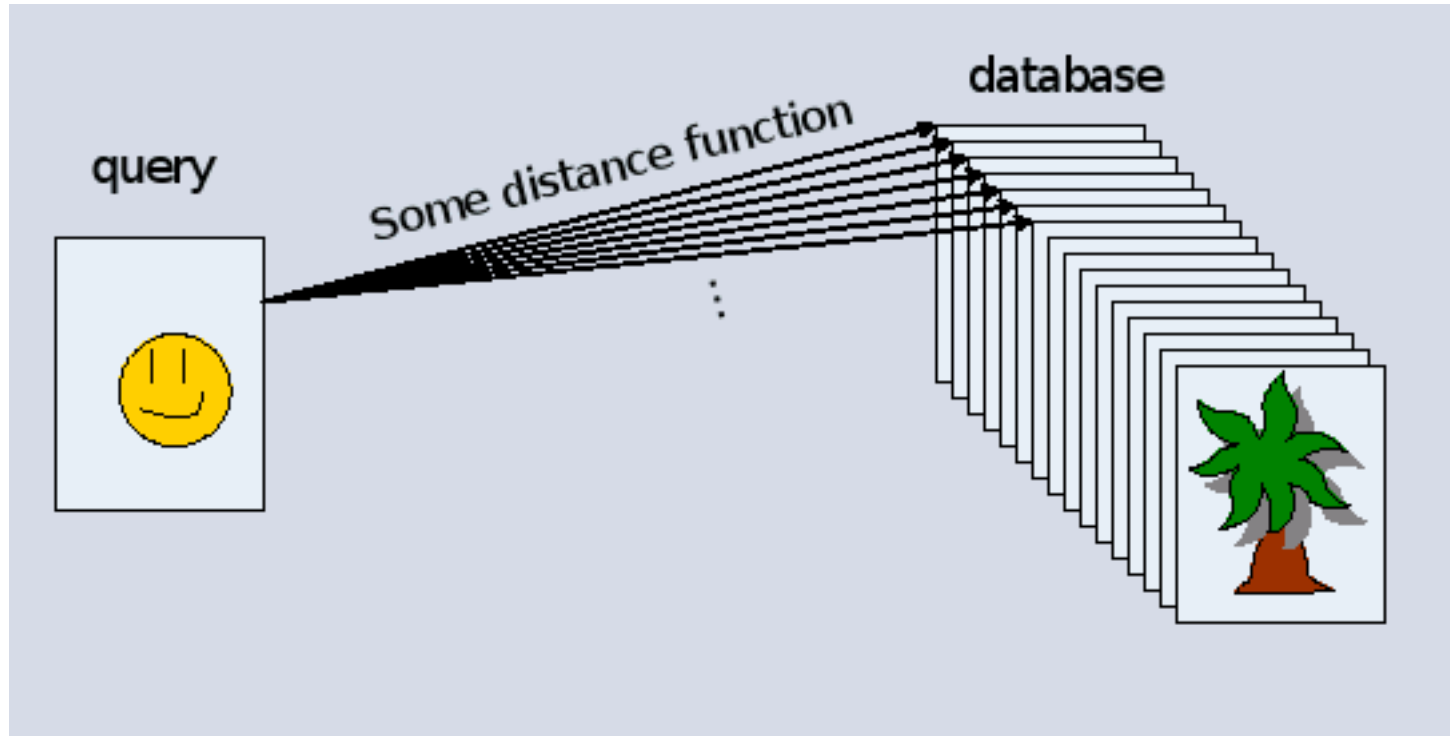
# The Nearest-Neighbor Search Problem

- **Input Description:** A set  $S$  of  $n$  points in  $d$  dimensions; a query point  $q$ .
- Which point in  $S$  is closest to  $q$ ?

( Linear scan approach has query time of  $\Theta(dn)$  )



# The Nearest-Neighbor Search Problem



# The Nearest-Neighbor Search Problem: Application

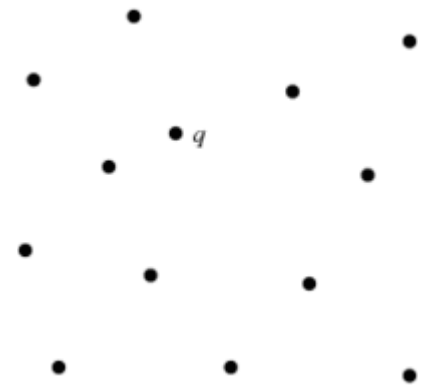
Depends on the value of  $d$ :

- low  $d$ : graphics, vision, natural language, etc
- high  $d$ :
  - similarity search in databases (text, images etc)
  - finding pairs of similar objects (e.g., copyright violation detection)
  - useful subroutine for clustering
  - Classification

# The Nearest-Neighbor Search Problem

- Efficient solutions have been discovered for the case when the points lie in a space of constant dimension.

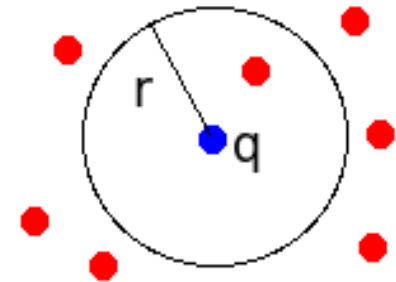
(For example, if the points lie in the plane, the nearest-neighbor problem can be solved with  $O(\log n)$  time per query, using only  $O(n)$  storage.)



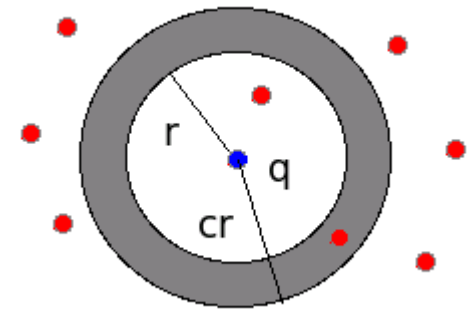
- Unfortunately, as the dimension grows, the algorithms become less and less efficient. More specifically, their space or time requirements grow exponentially in the dimension.

# The Nearest-Neighbor Search Problem

- **r-Near Neighbor:** for any query  $q$ , returns a point  $p \in P$   
s.t.  $\|p-q\| \leq r$  (if it exists)



- **c-Approximate r-Near Neighbor:** build data structure which, for any query  $q$ :
  - If there is a point  $p \in P$ ,  $\|p-q\| \leq r$
  - it returns  $p' \in P$ ,  $\|p'-q\| \leq cr$



# Metric space

Metric Space: In mathematics, a metric space is a set where a notion of distance (called a metric) between elements of the set is defined. The metric space which most closely corresponds to our intuitive understanding of space is the 3-dimensional Euclidean space.

1.  $d(x, y) \geq 0$  (non-negativity)
2.  $d(x, y) = 0$  if and only if  $x = y$  (identity of indiscernibles)
3.  $d(x, y) \neq 0$  implies  $D(x, y) > 0$  (isolation)
4.  $d(x, y) = d(y, x)$  (symmetry)
5.  $d(x, z) \leq d(x, y) + d(y, z)$  (triangle inequality).

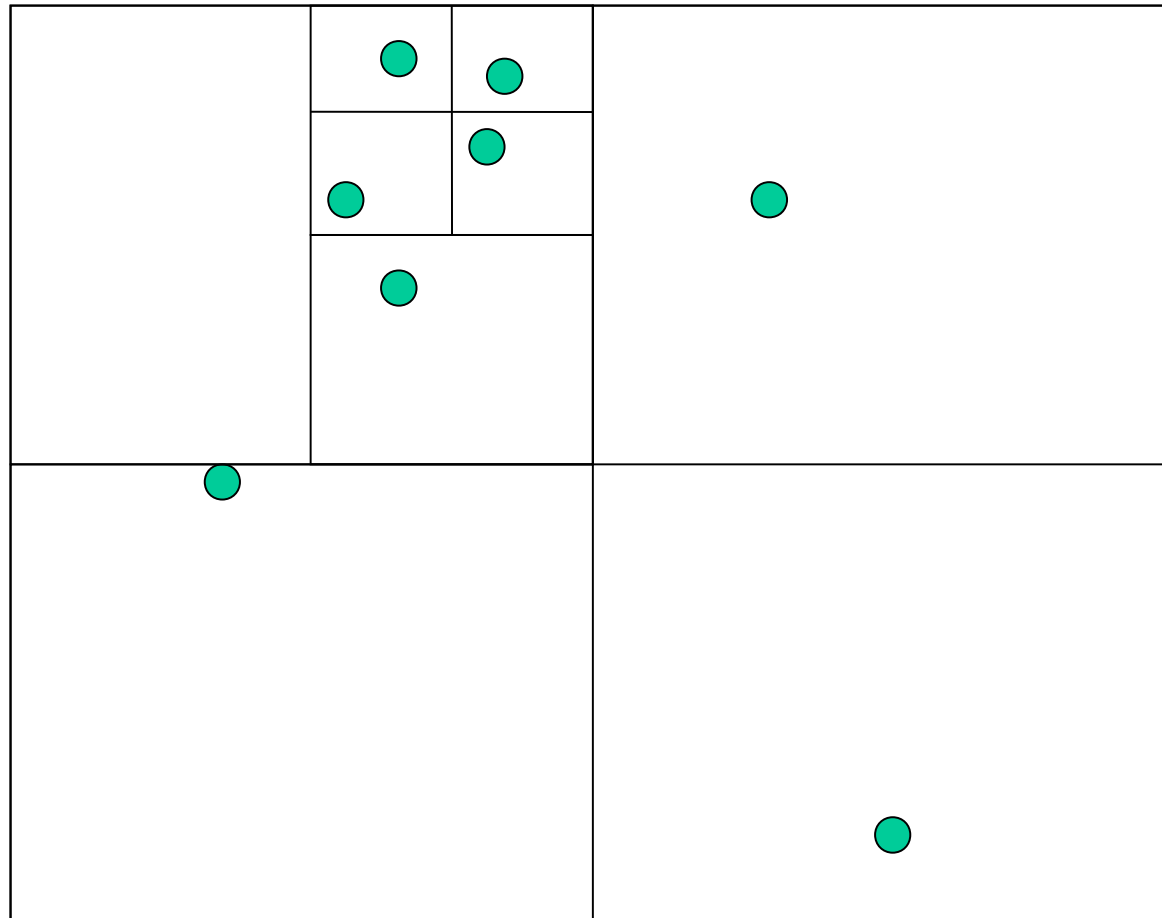


# Topics (Nearest Neighbor Searching)

- **Problem Definition**
- **Basic Structure**
  - Quad-Tree
  - KD-Tree
  - Locality Sensitive Hashing
- **Application: Learning**
  - BoostMap: A Method for Efficient Approximate Similarity Rankings
- **Application: Vision**
  - A Binning Scheme for Fast Hard Driver Based Image Search\*
  - Fast Pose Estimation with Parameter Sensitive Hashing

# Quad-Tree

- Split the space into  $2^d$  equal subsquares.



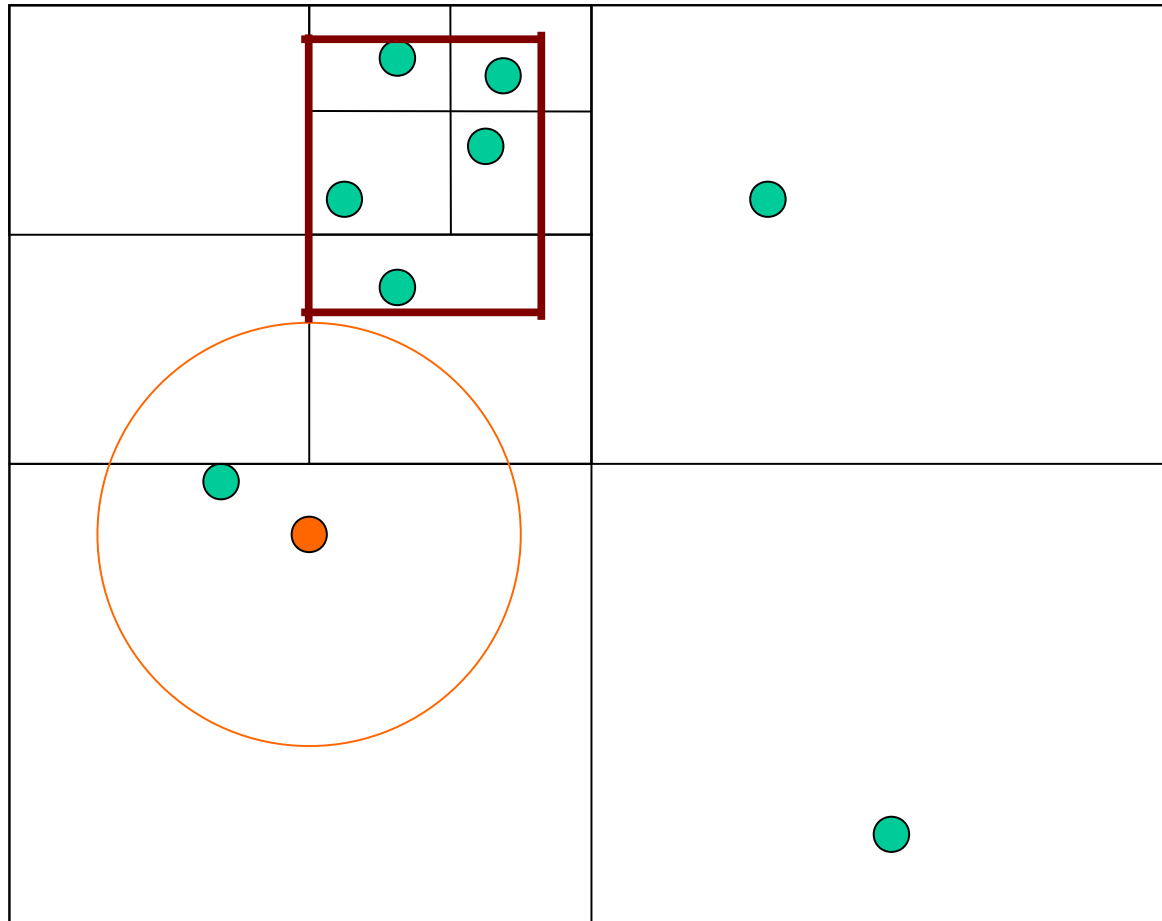
# Quad-Tree:Build

- **Split the space into  $2^d$  equal subsquares**
- **Repeat until done:**
  - only one point left
  - no point left
- **Variants:**
  - split only one dimension at a time

# Quad-Tree:Query

- Near neighbor (range search):
  - put the root on the stack
  - repeat
    - pop the next node  $T$  from the stack
    - for each child  $C$  of  $T$ :
      - if  $C$  is a leaf, examine point(s) in  $C$
      - if  $C$  intersects with the ball of radius  $r$  around  $q$ , add  $C$  to the stack (**bounding box**)

# Quad-Tree



# Quad-Tree

- Start range search with  $r = \infty$
- Whenever a point is found, update  $r$
- Only investigate nodes with respect to current  $r$

# Quad-Tree


- Simple data structure
- Versatile, easy to implement
- Disadvantages:
  - Empty spaces: if the points form sparse clouds, it takes a while to reach them
  - Space exponential in dimension
  - Time exponential in dimension, e.g., points on the hypercube

# Topics (Nearest Neighbor Searching)

- **Problem Definition**
- **Basic Structure**
  - Quad-Tree
  - **KD-Tree**
  - Locality Sensitive Hashing
- **Application: Learning**
  - BoostMap: A Method for Efficient Approximate Similarity Rankings
- **Application: Vision**
  - A Binning Scheme for Fast Hard Driver Based Image Search
  - Fast Pose Estimation with Parameter Sensitive Hashing



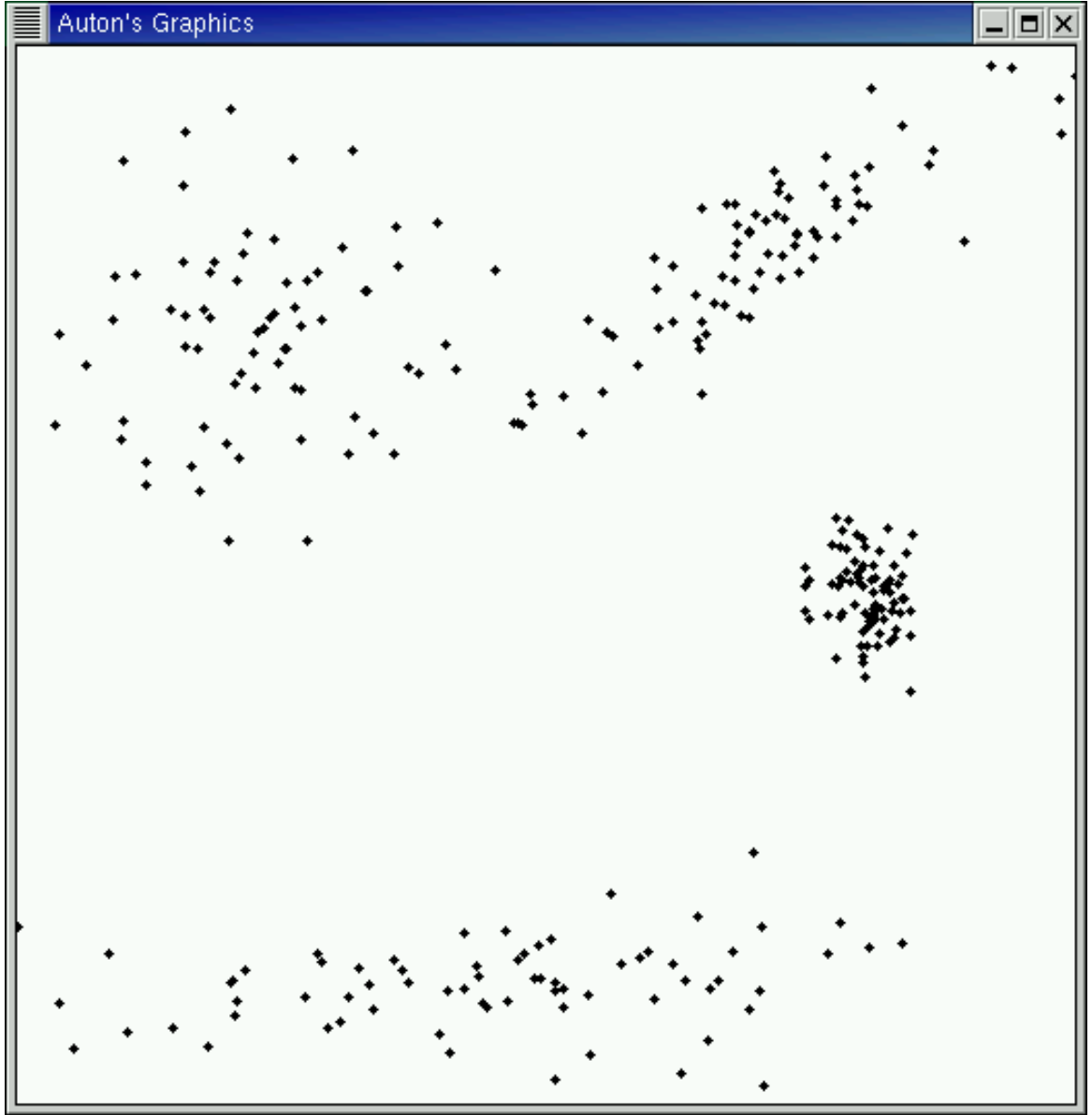
# Motivation: Space issues

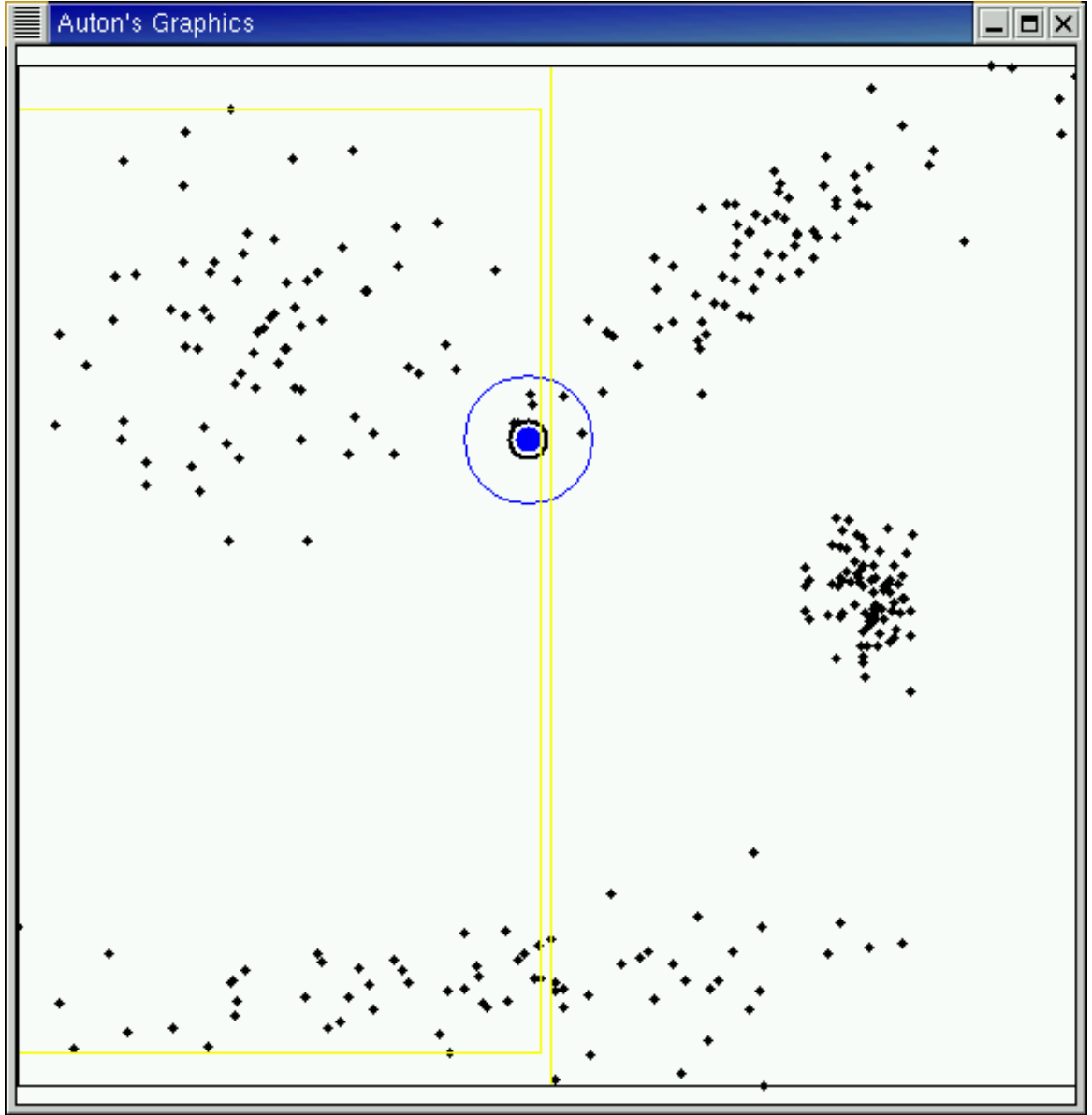
			

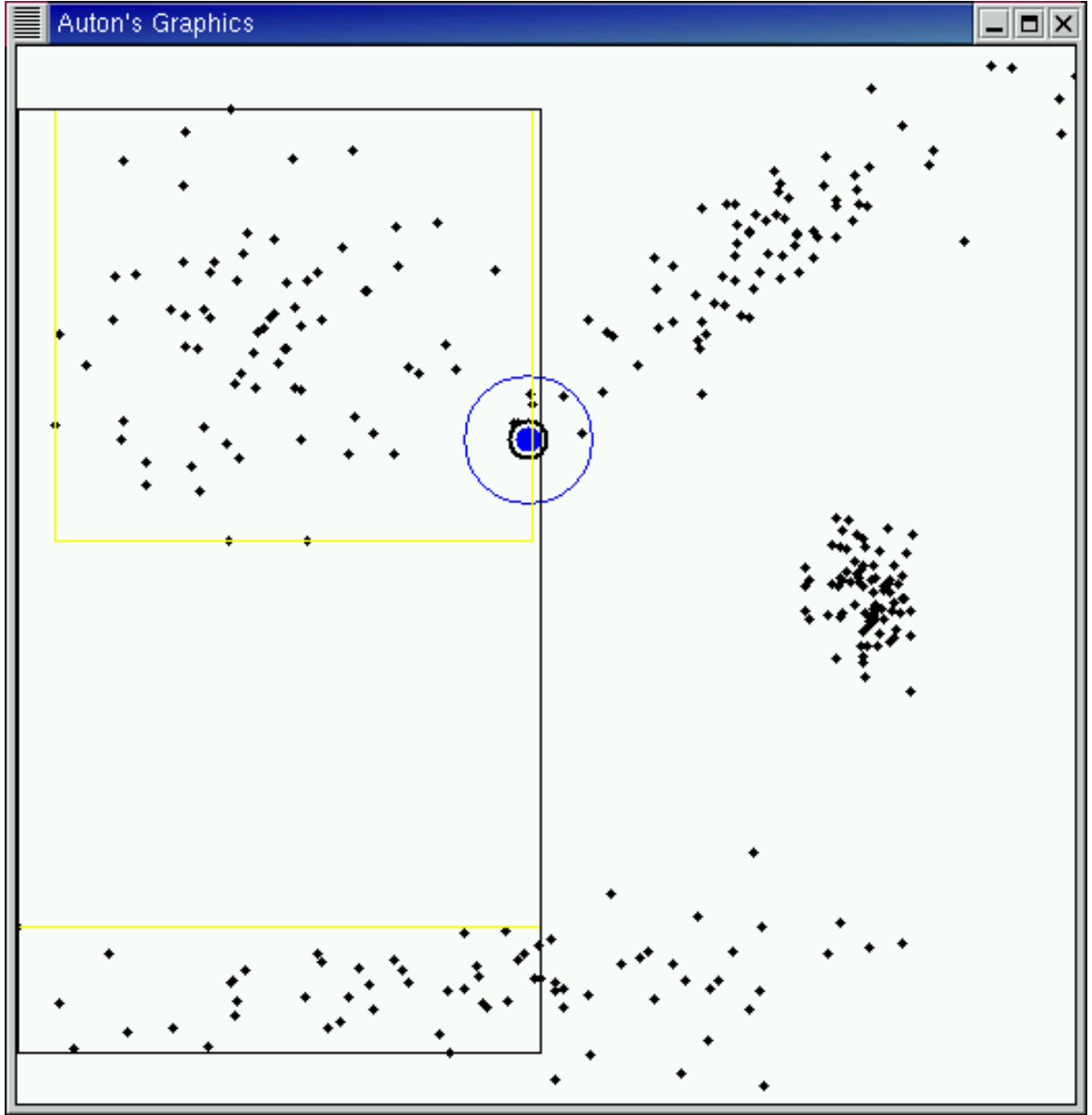
# KD-Tree [Bentley'75]

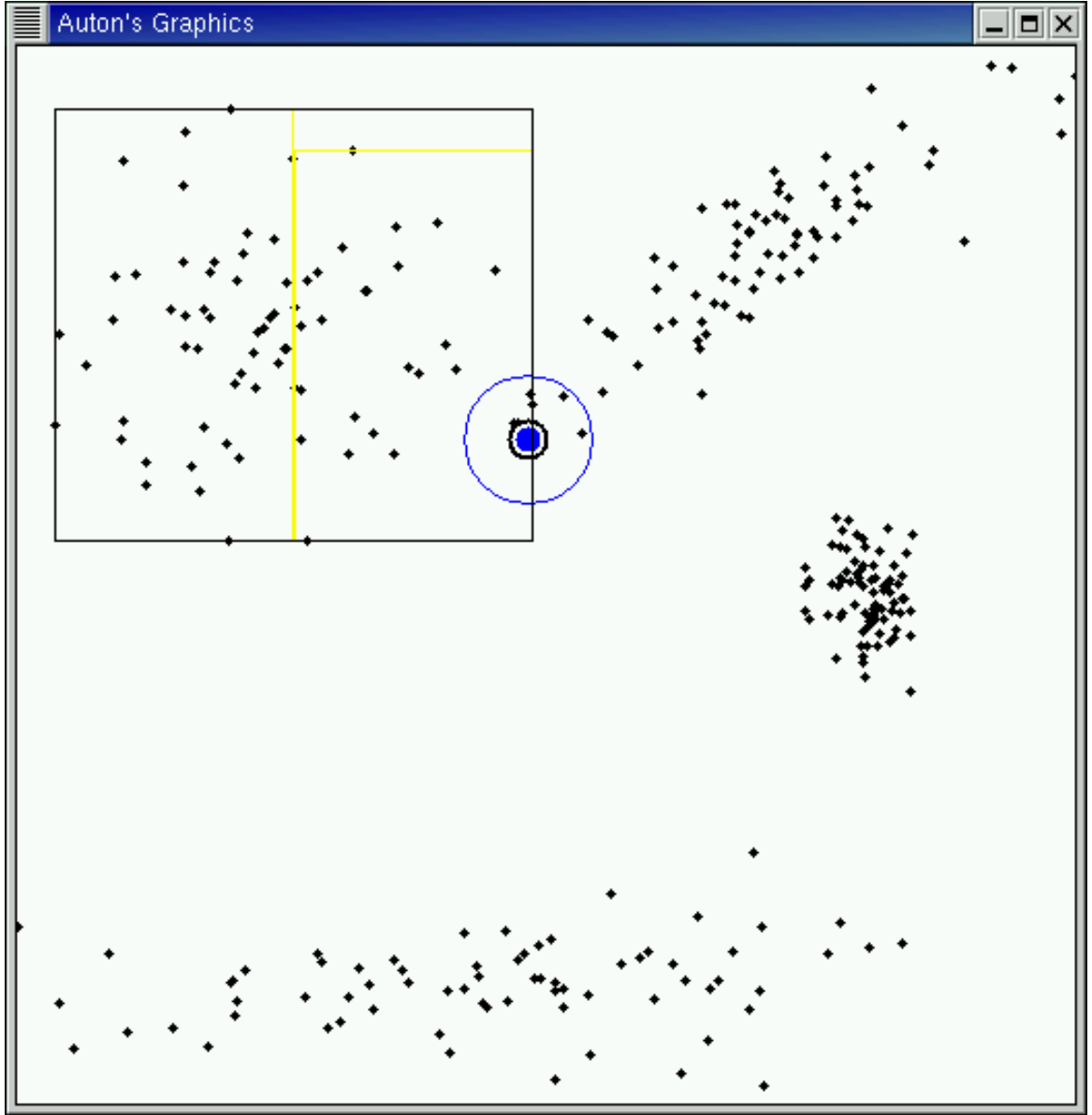
- **Main ideas:**
  - only one-dimensional splits
  - instead of splitting in the median, random position or split “carefully” (many variations)
  - near(est) neighbor queries: as for quadtrees
- **Advantages:**
  - no (or less) empty spaces
  - only linear space
- **Exponential query time still possible**

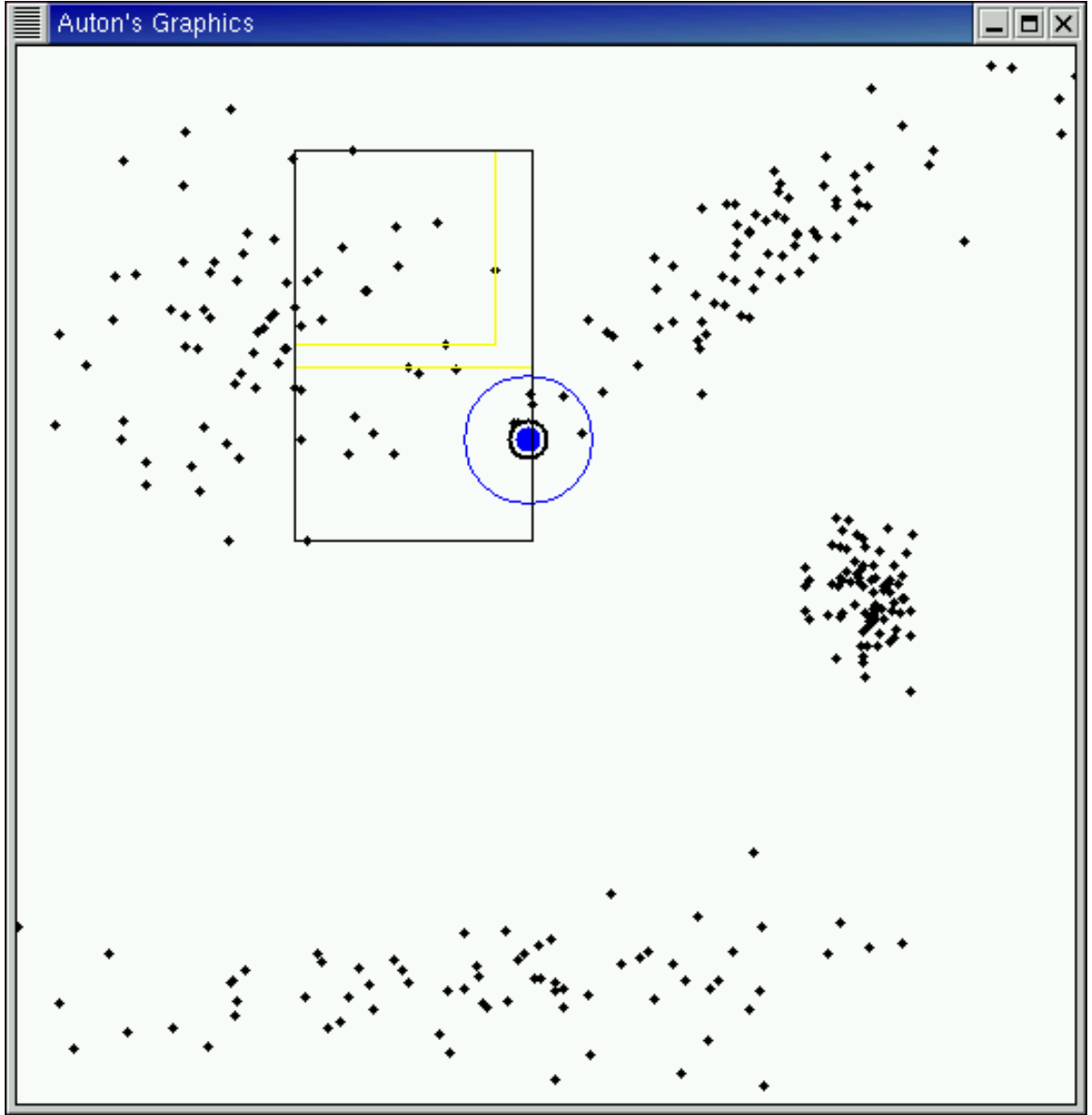
# KD-Tree: Animation



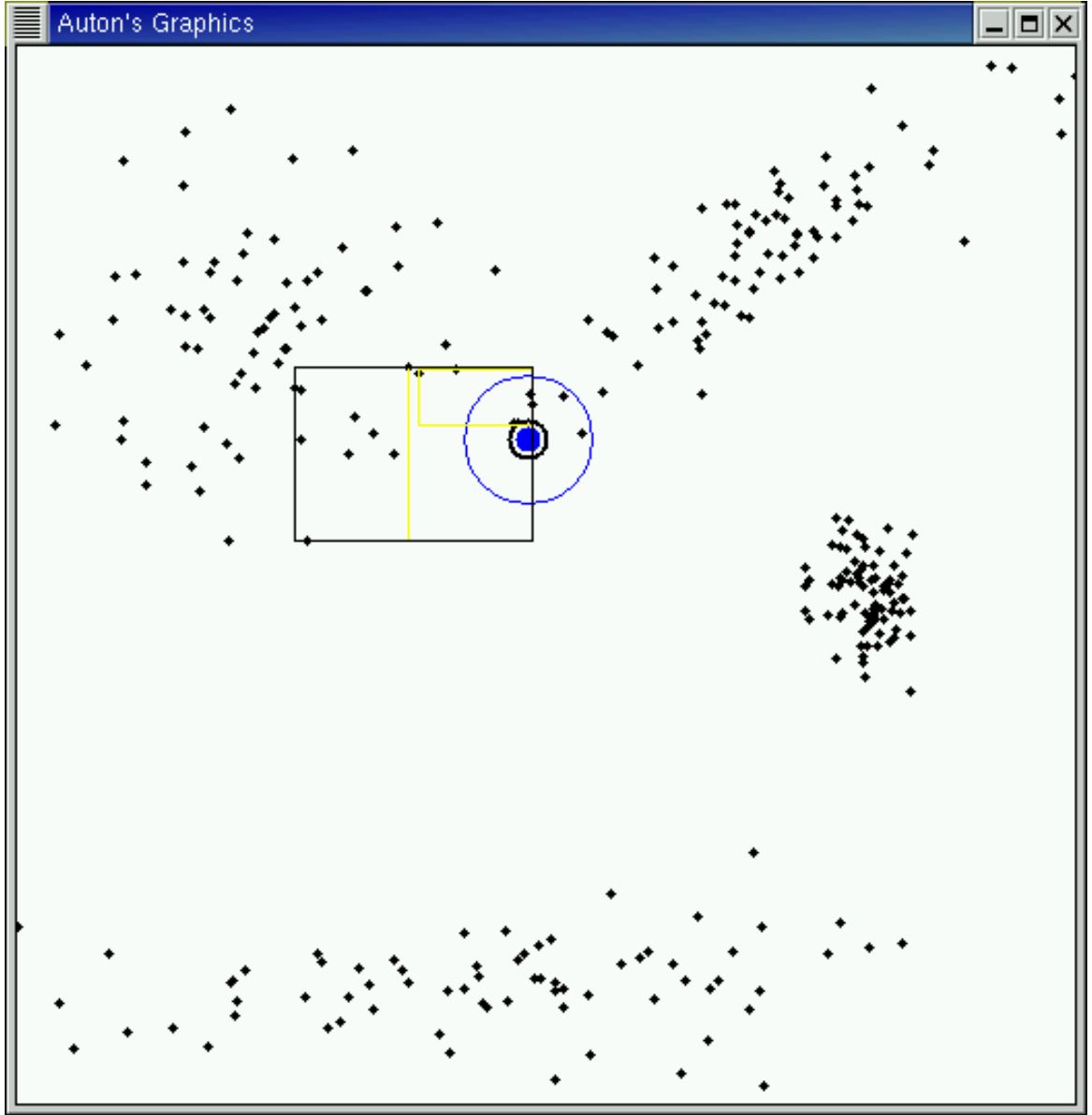


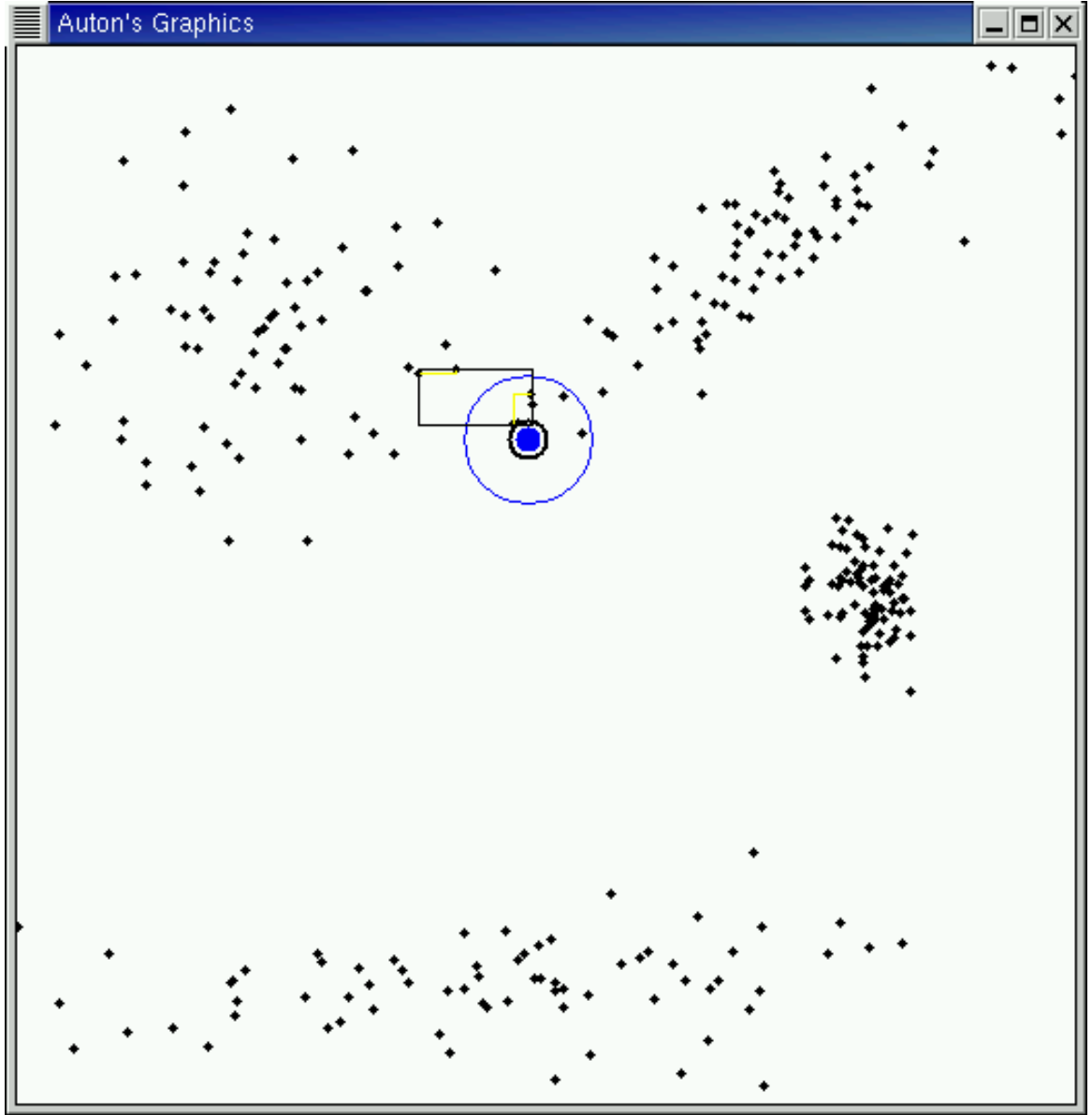


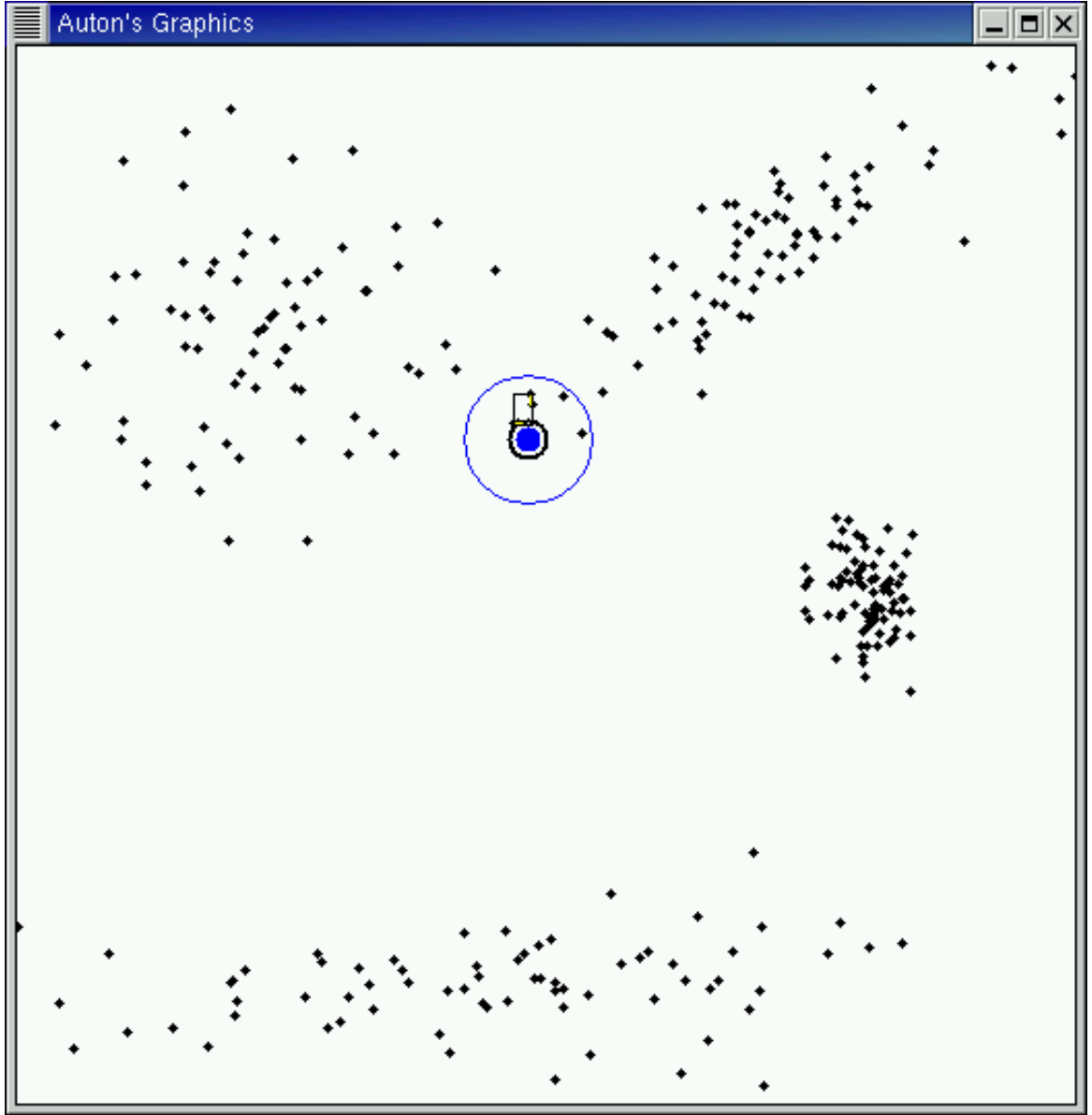


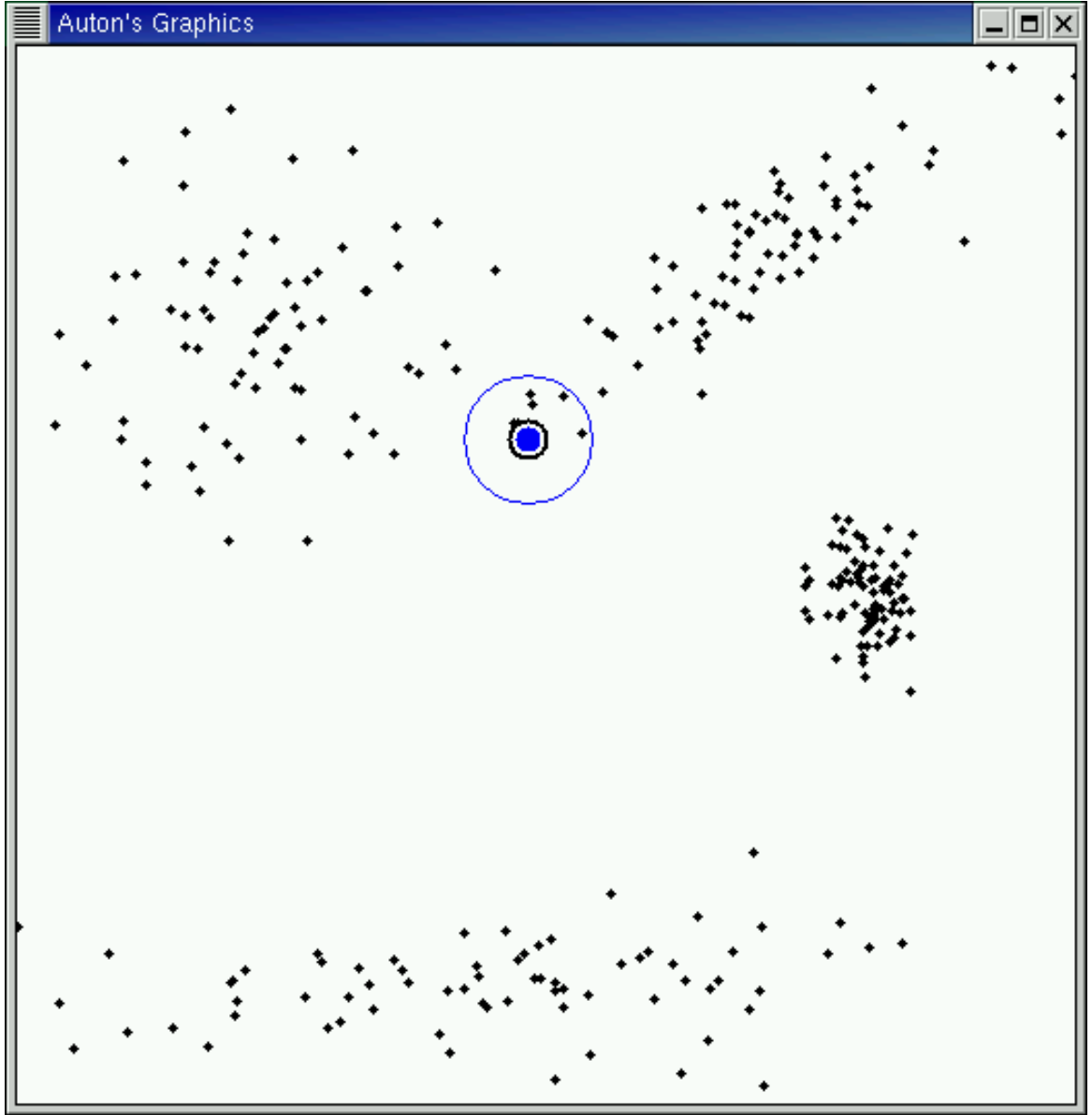


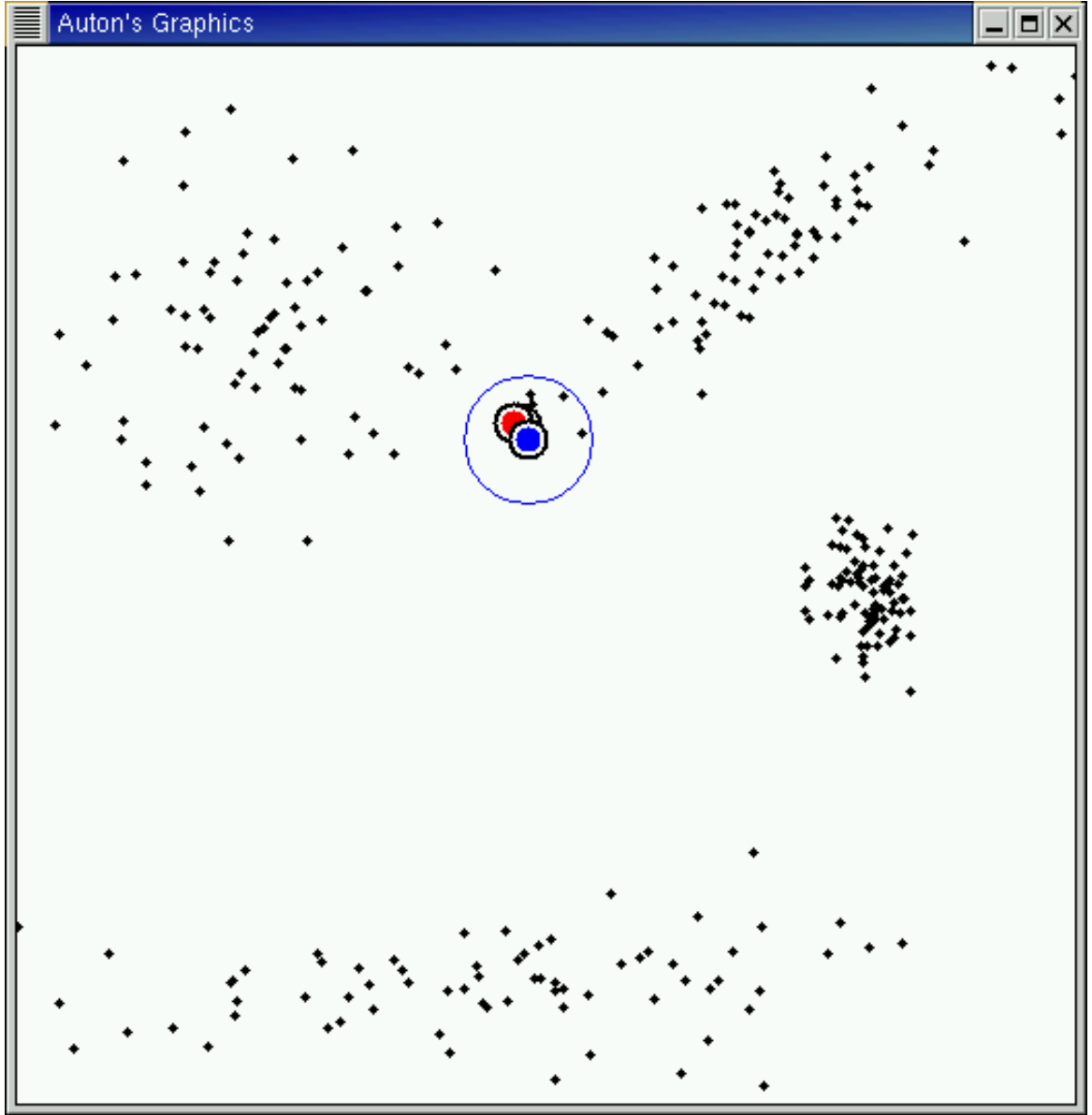


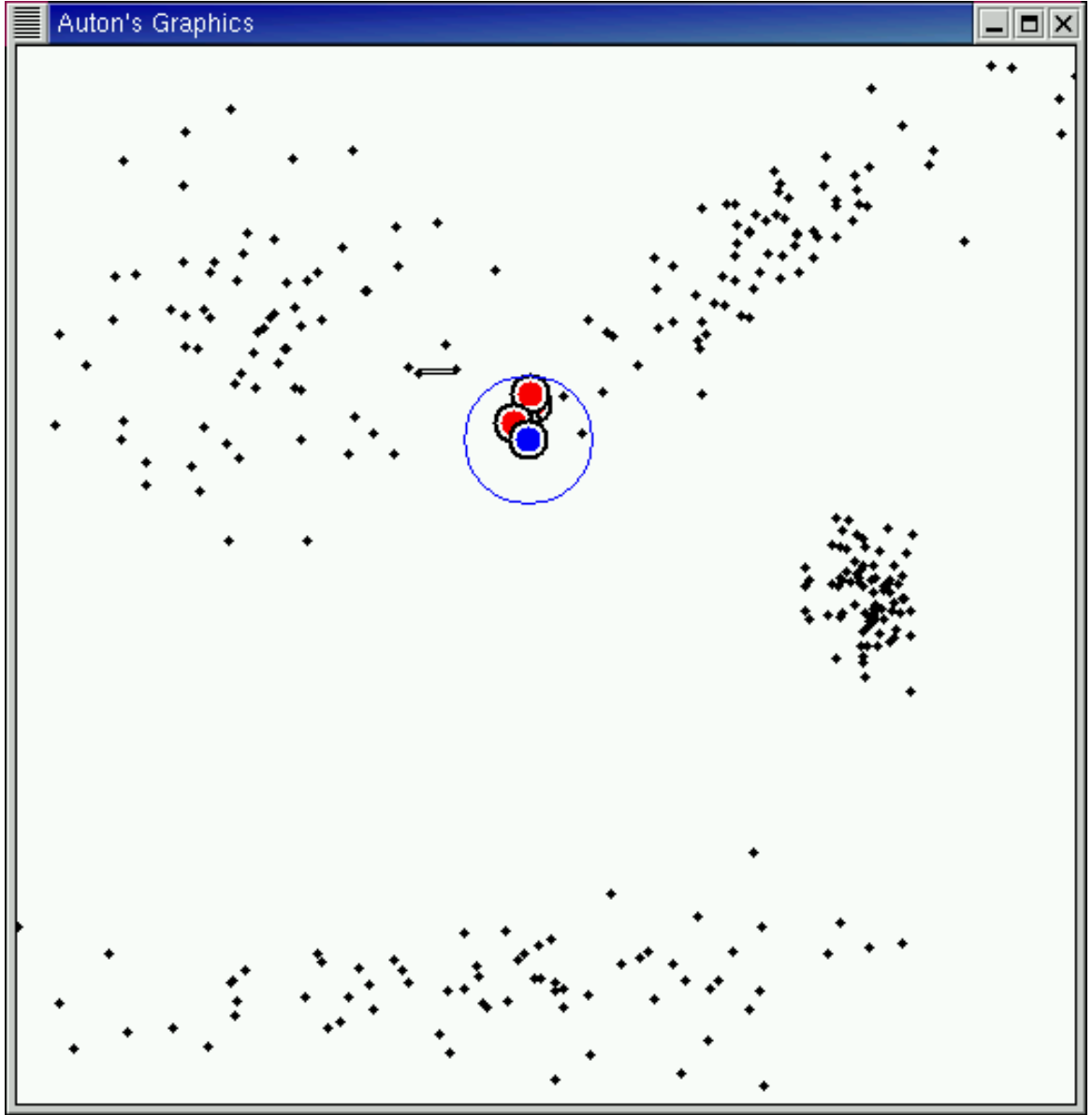


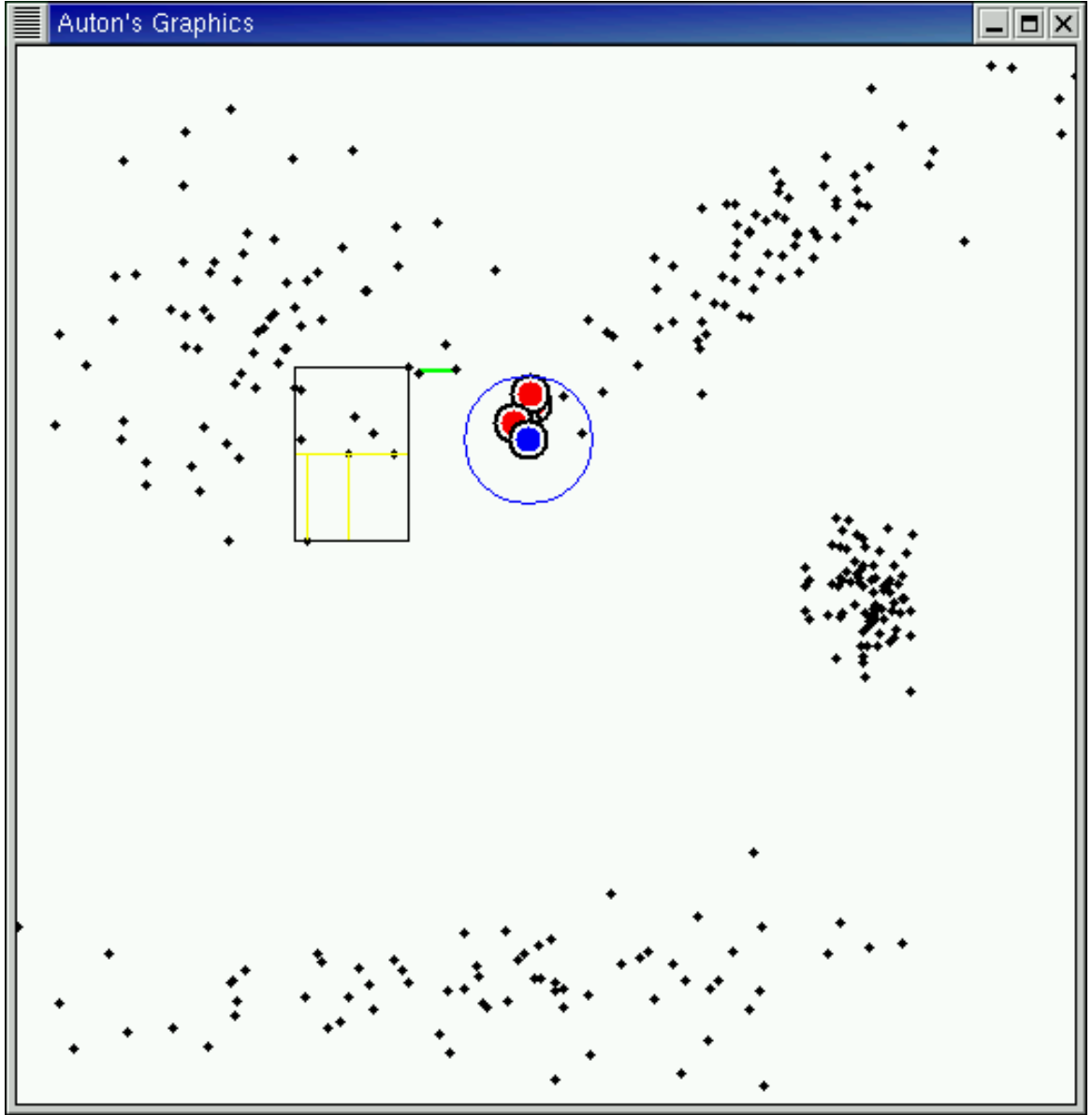


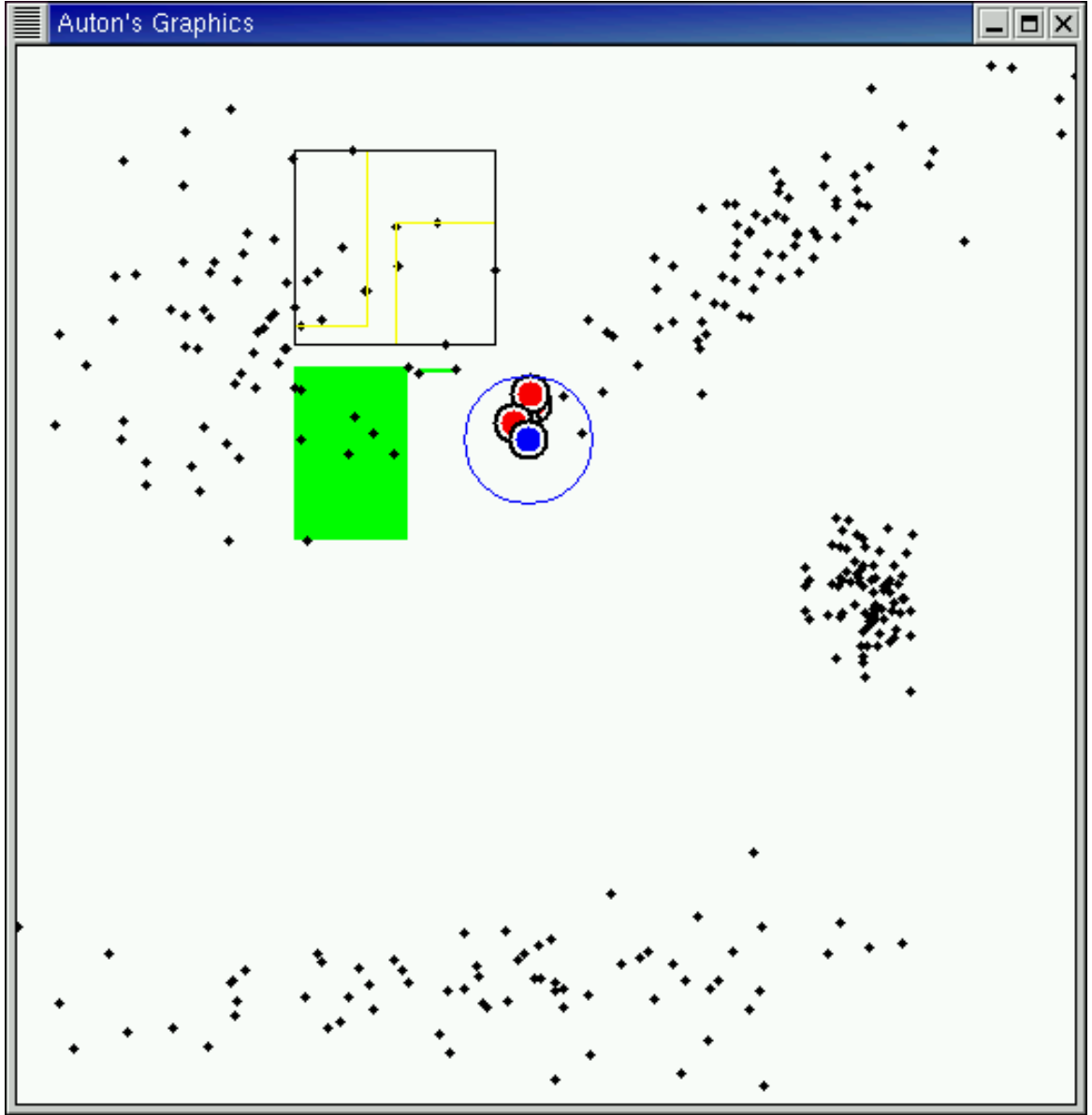




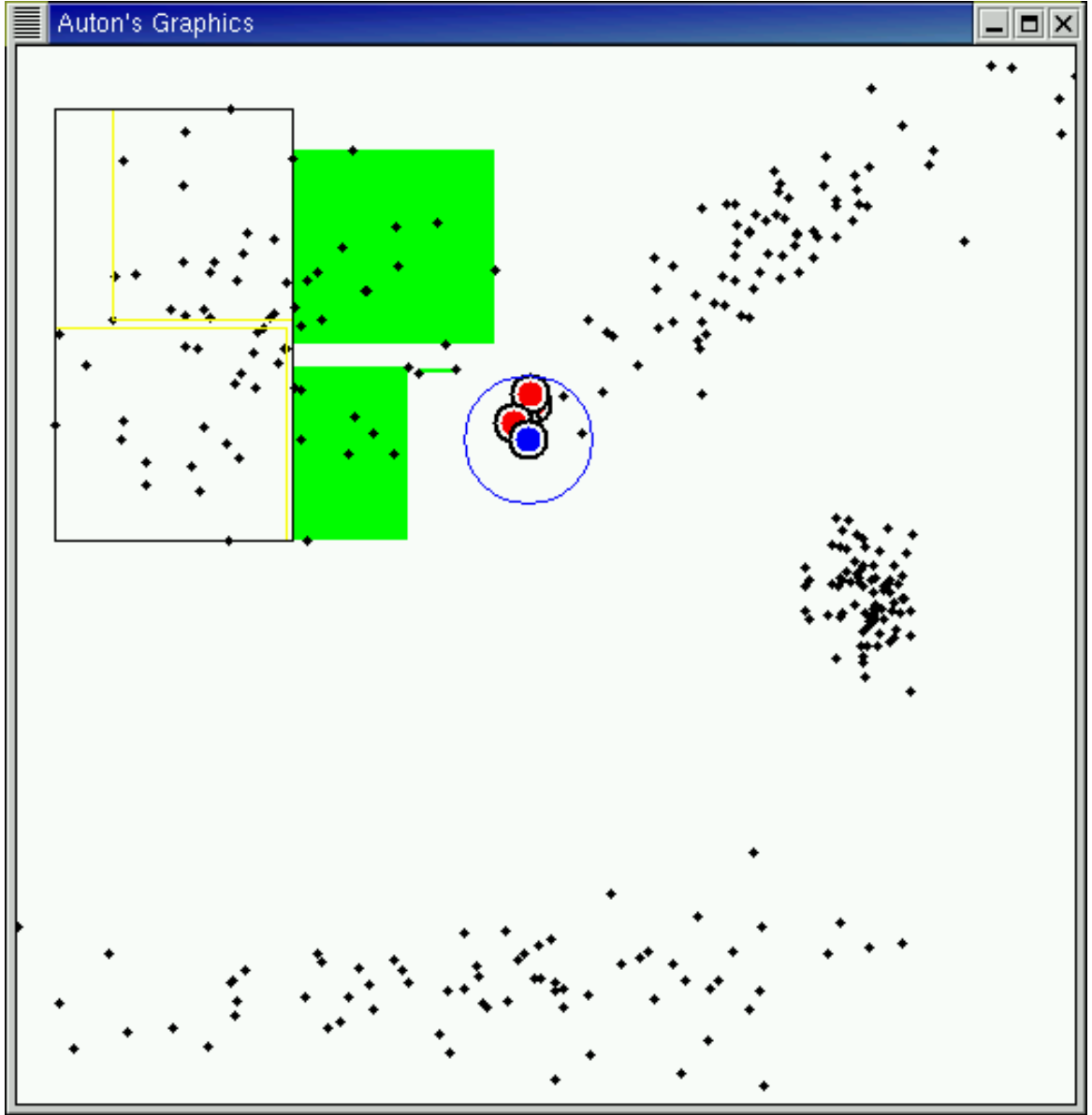


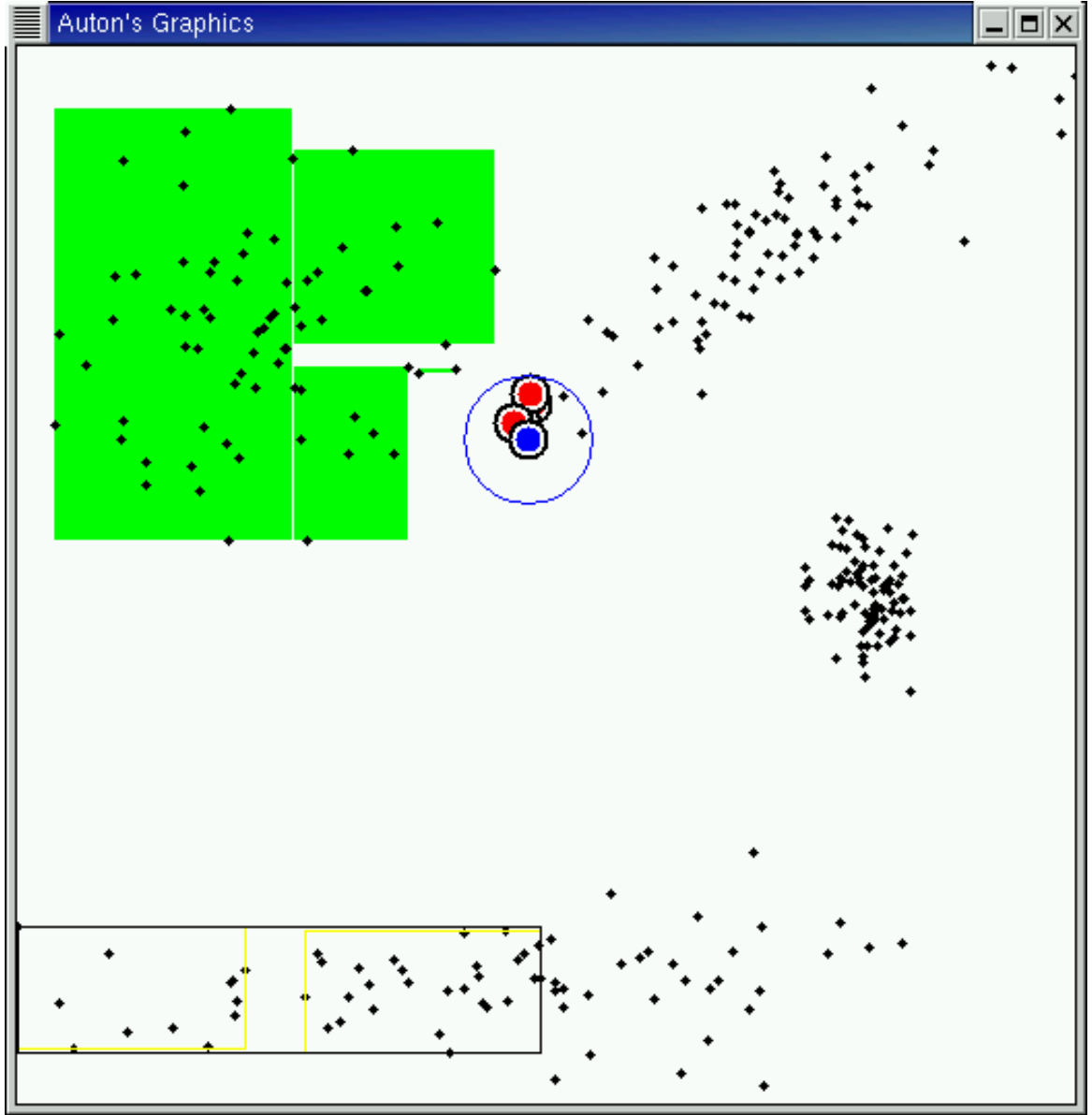


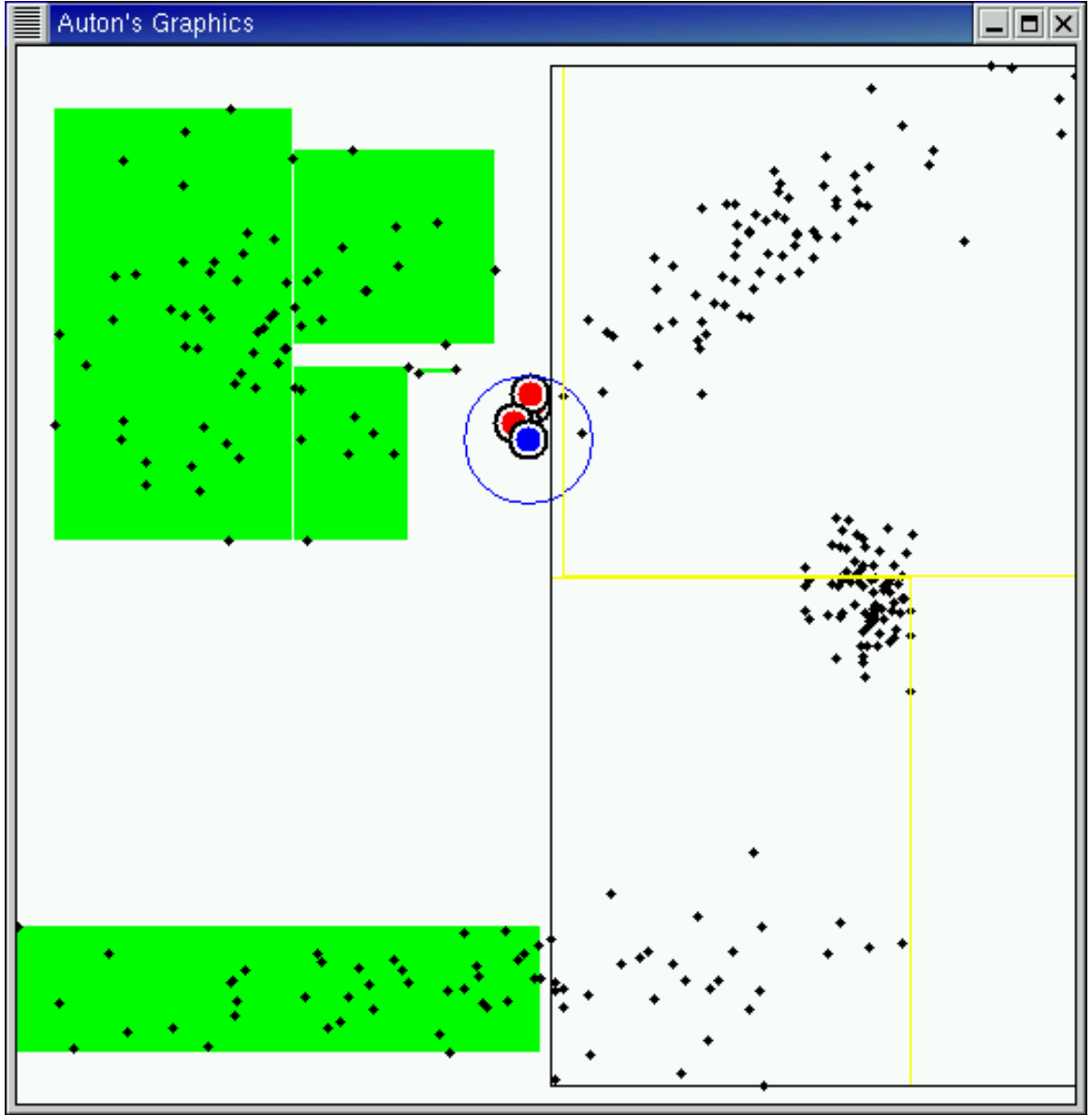


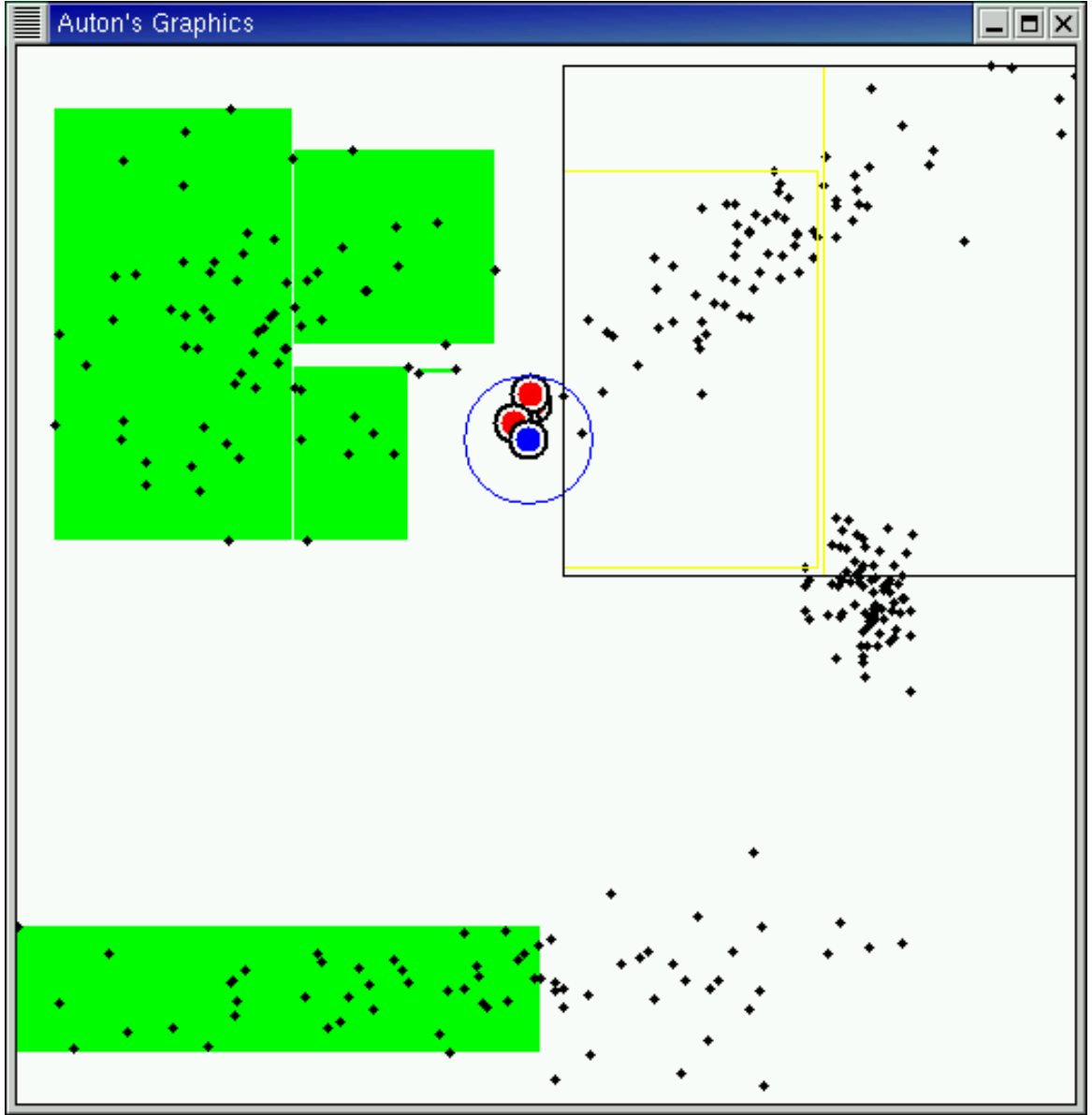


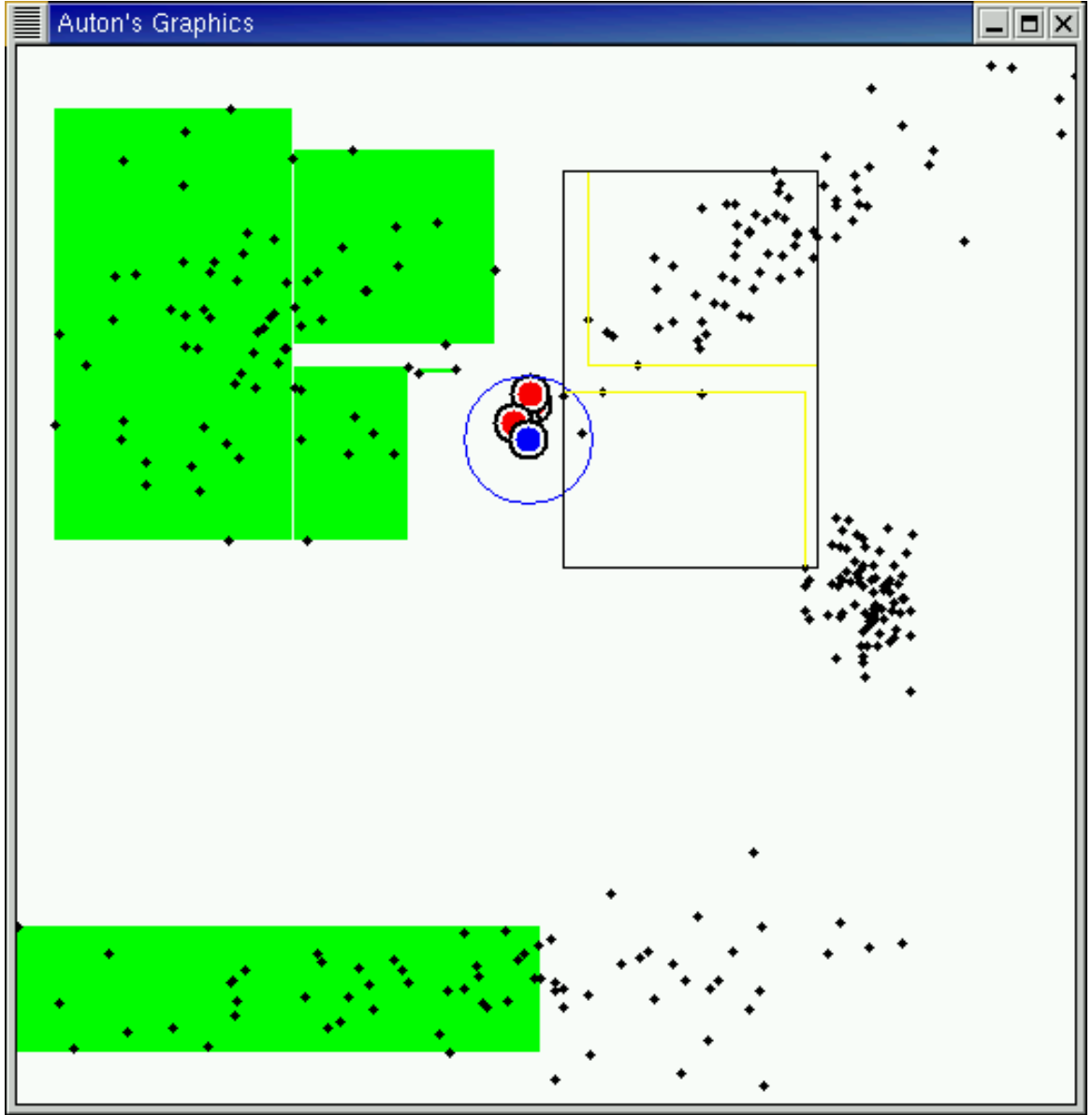


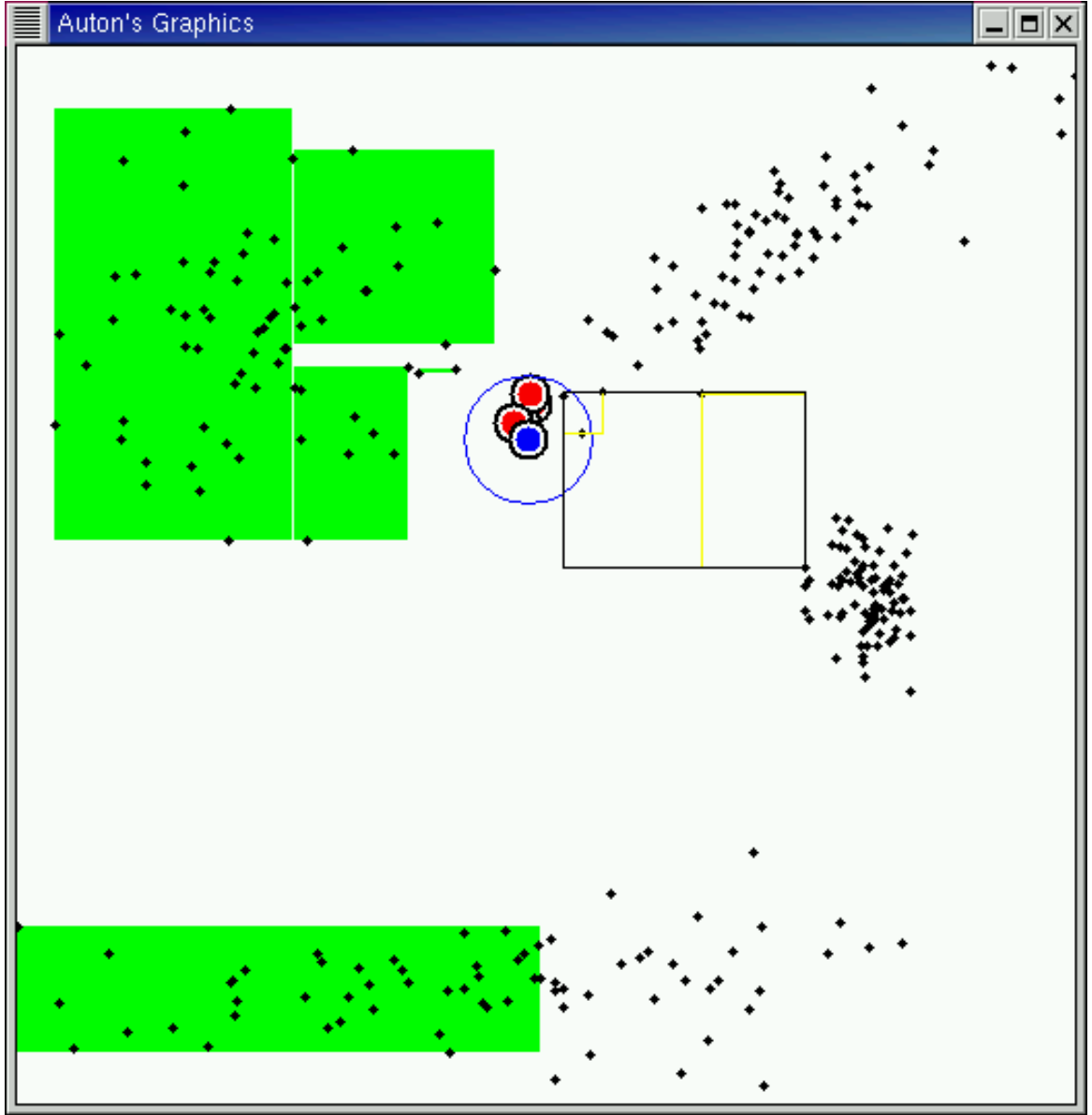


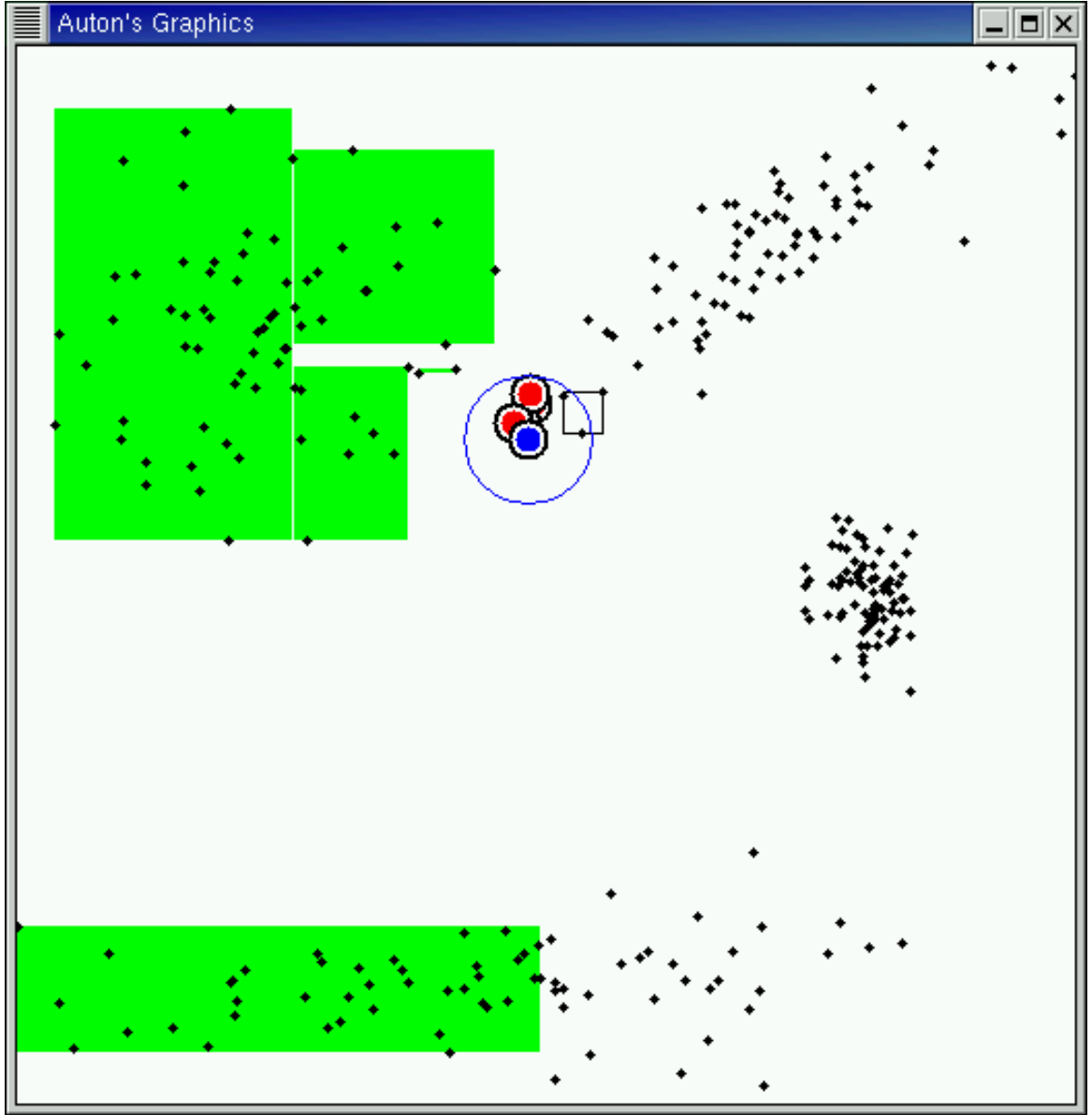


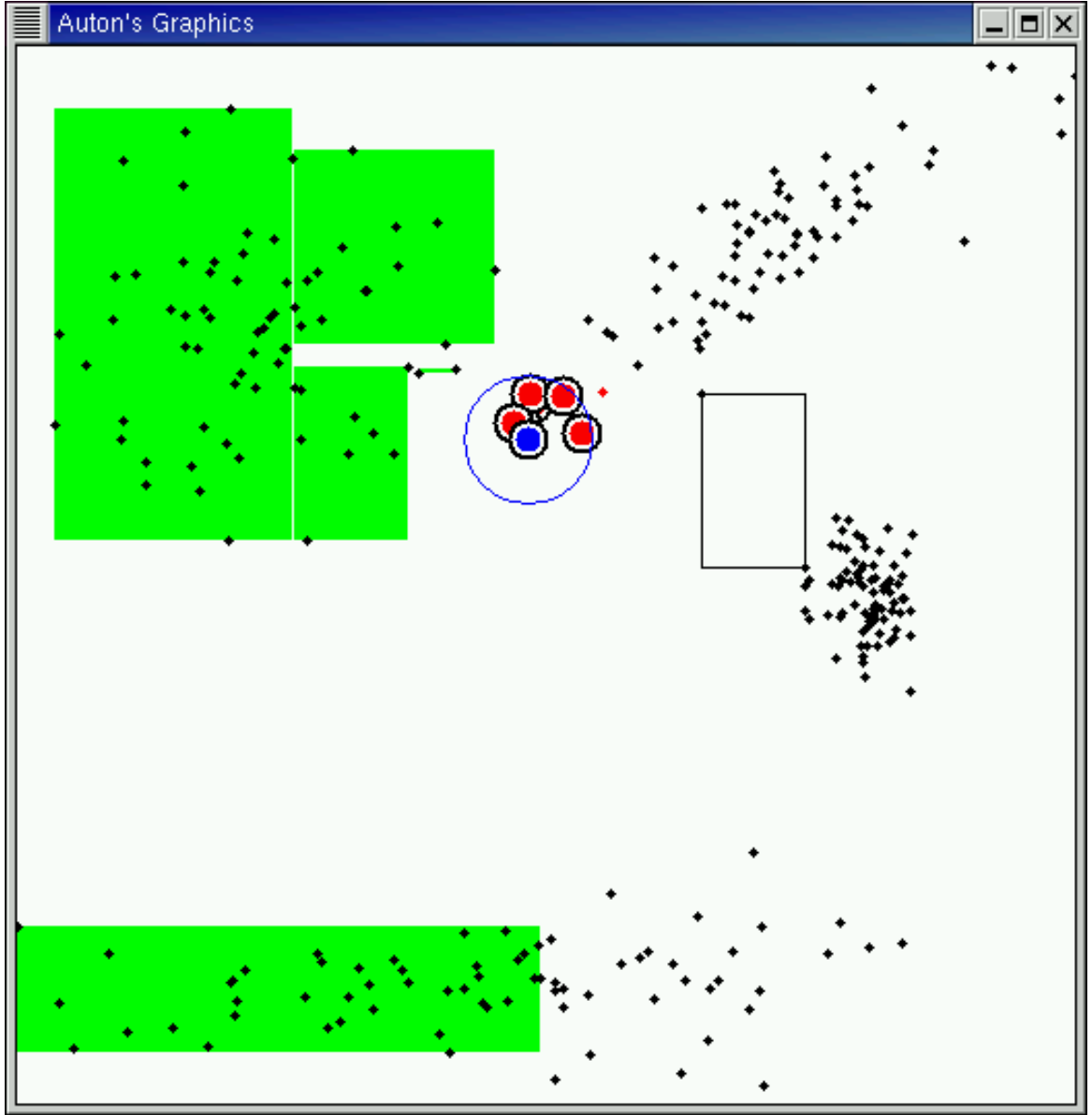




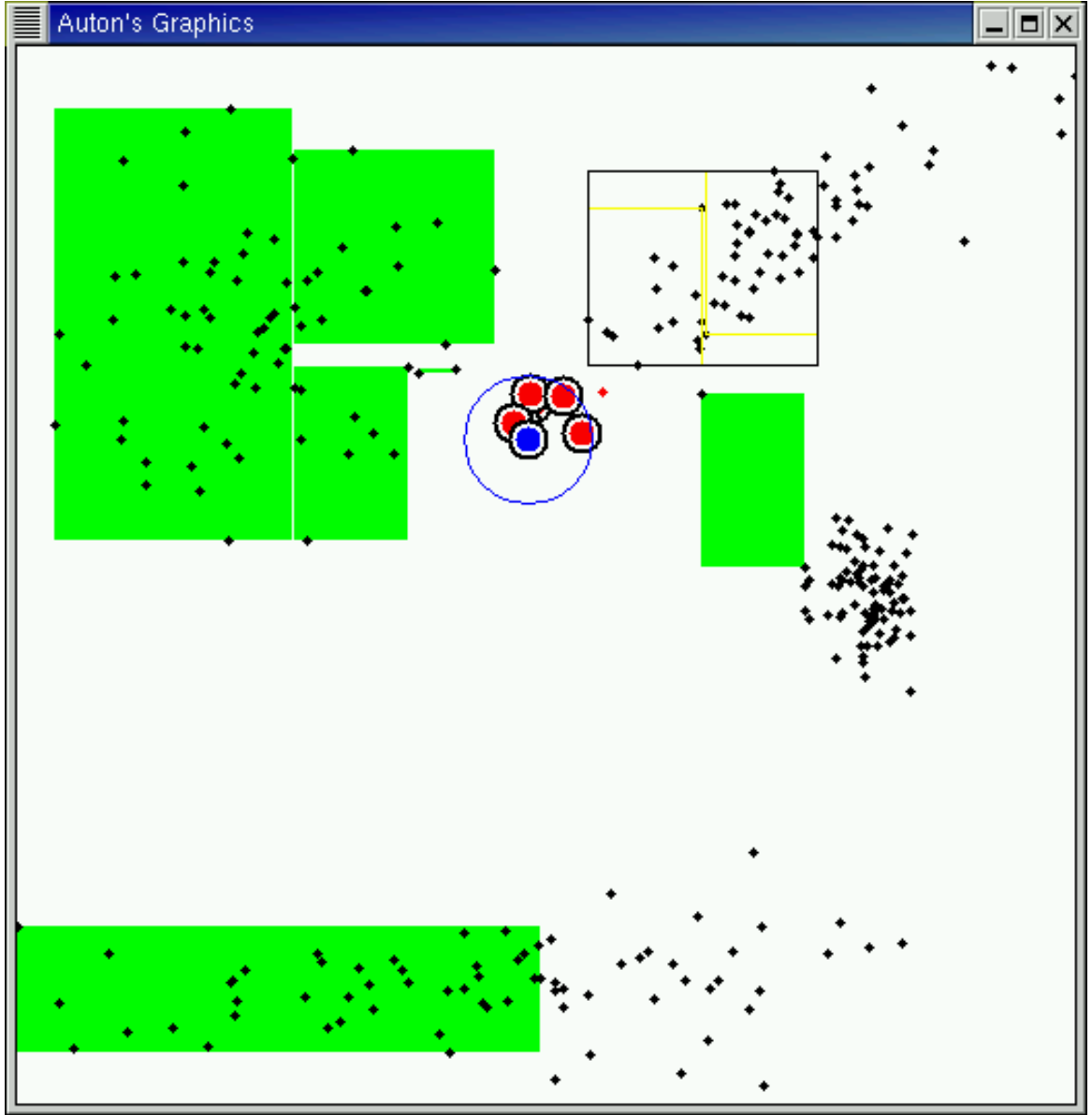


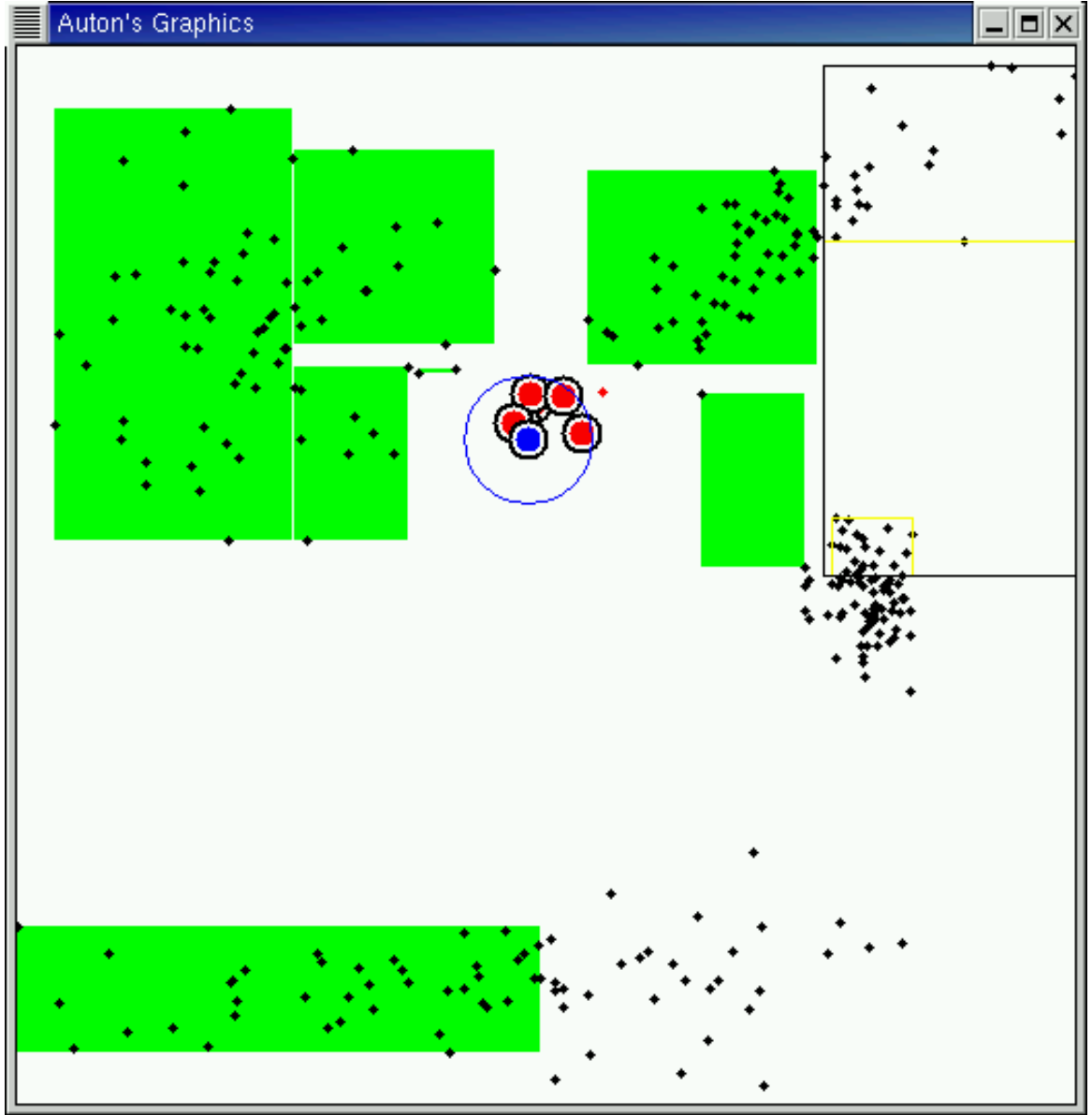


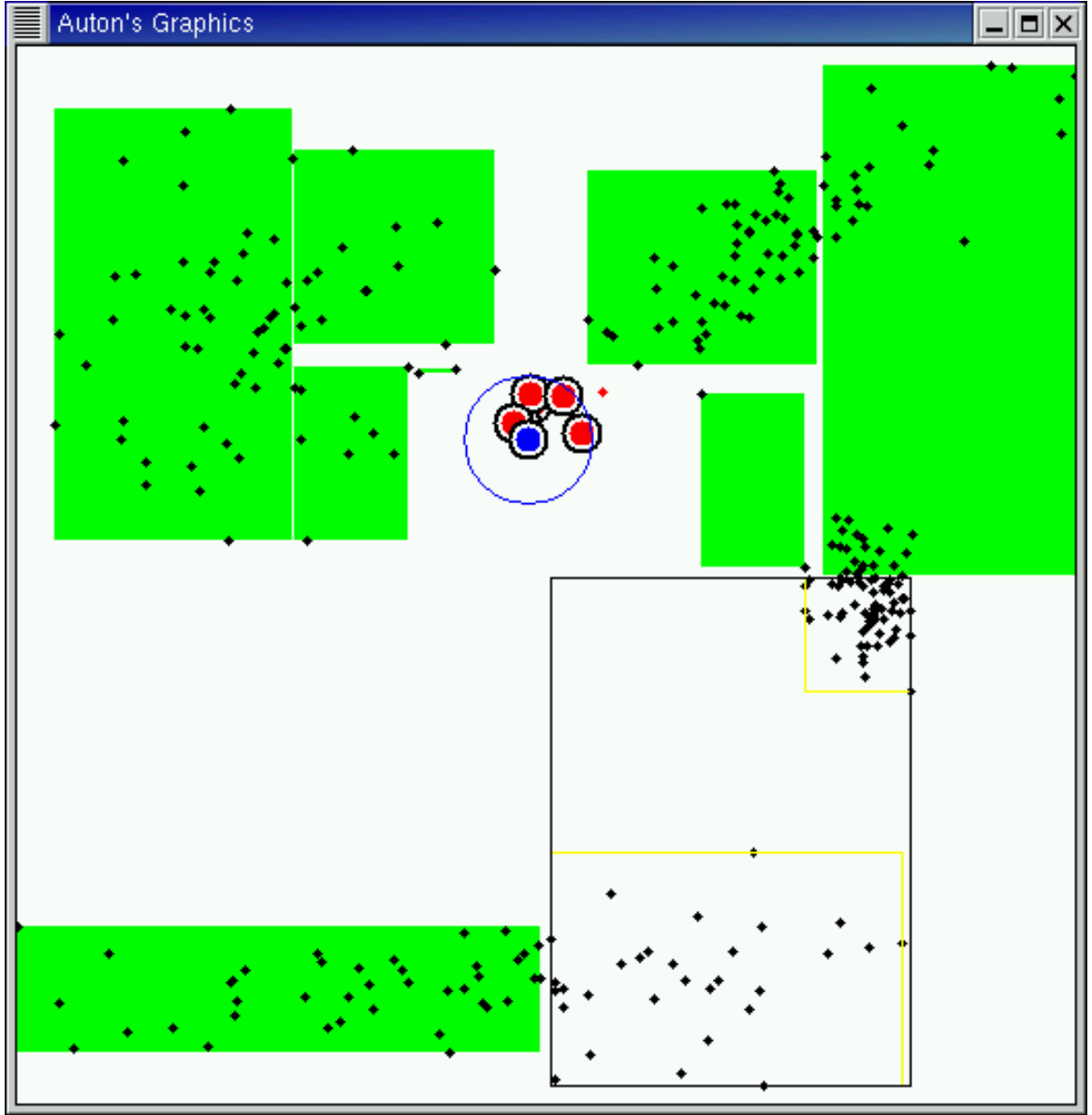


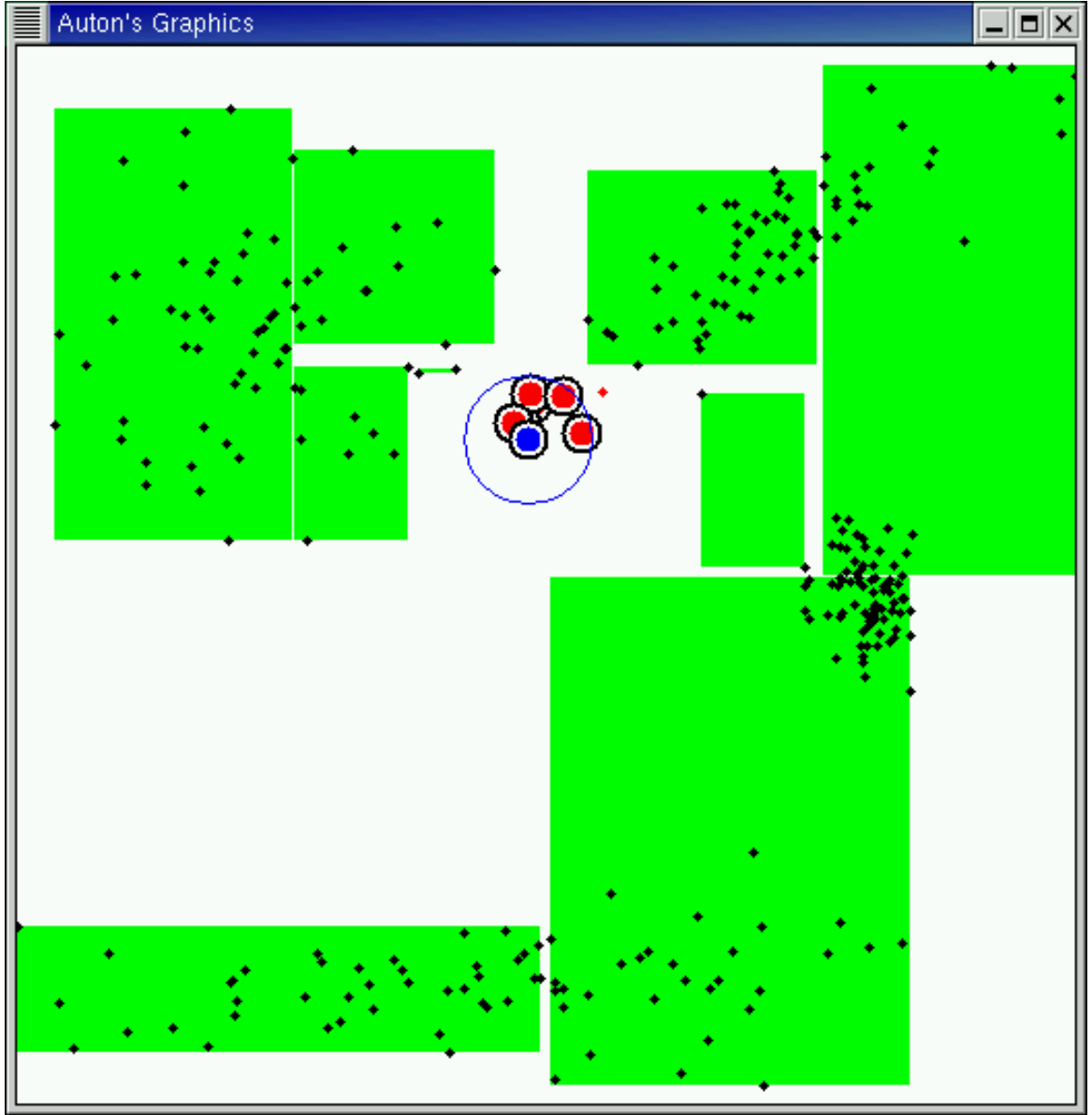












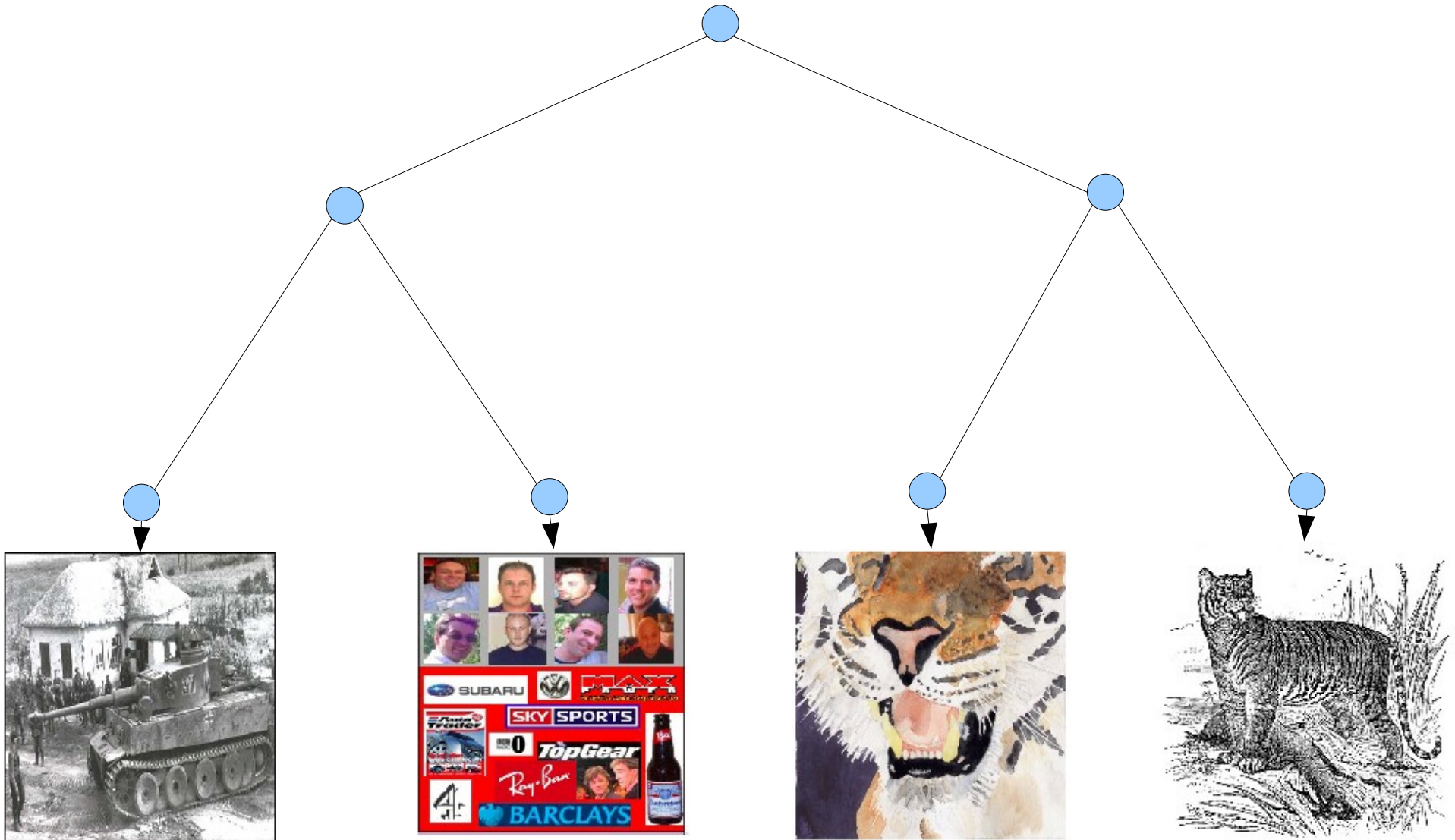
# KD-Tree: Exponential Query Time

- What does it mean exactly ?
  - Unless we do something really stupid, query time is at most  $dn$
  - Therefore, the actual query time is  
 $\text{Min}[ dn, \text{exponential}(d) ]$
- Object retrieval with large vocabularies and fast spatial matching  
James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman
- <http://www.cgg.cvut.cz/members/havran/>

# Topics (Nearest Neighbor Searching)

- **Problem Definition**
- **Basic Structure**
  - Quad-Tree
  - KD-Tree
  - **Locality Sensitive Hashing**
- **Application: Learning**
  - BoostMap: A Method for Efficient Approximate Similarity Rankings
- **Application: Vision**
  - A Binning Scheme for Fast Hard Driver Based Image Search\*
  - Fast Pose Estimation with Parameter Sensitive Hashing

# Tree-Structure

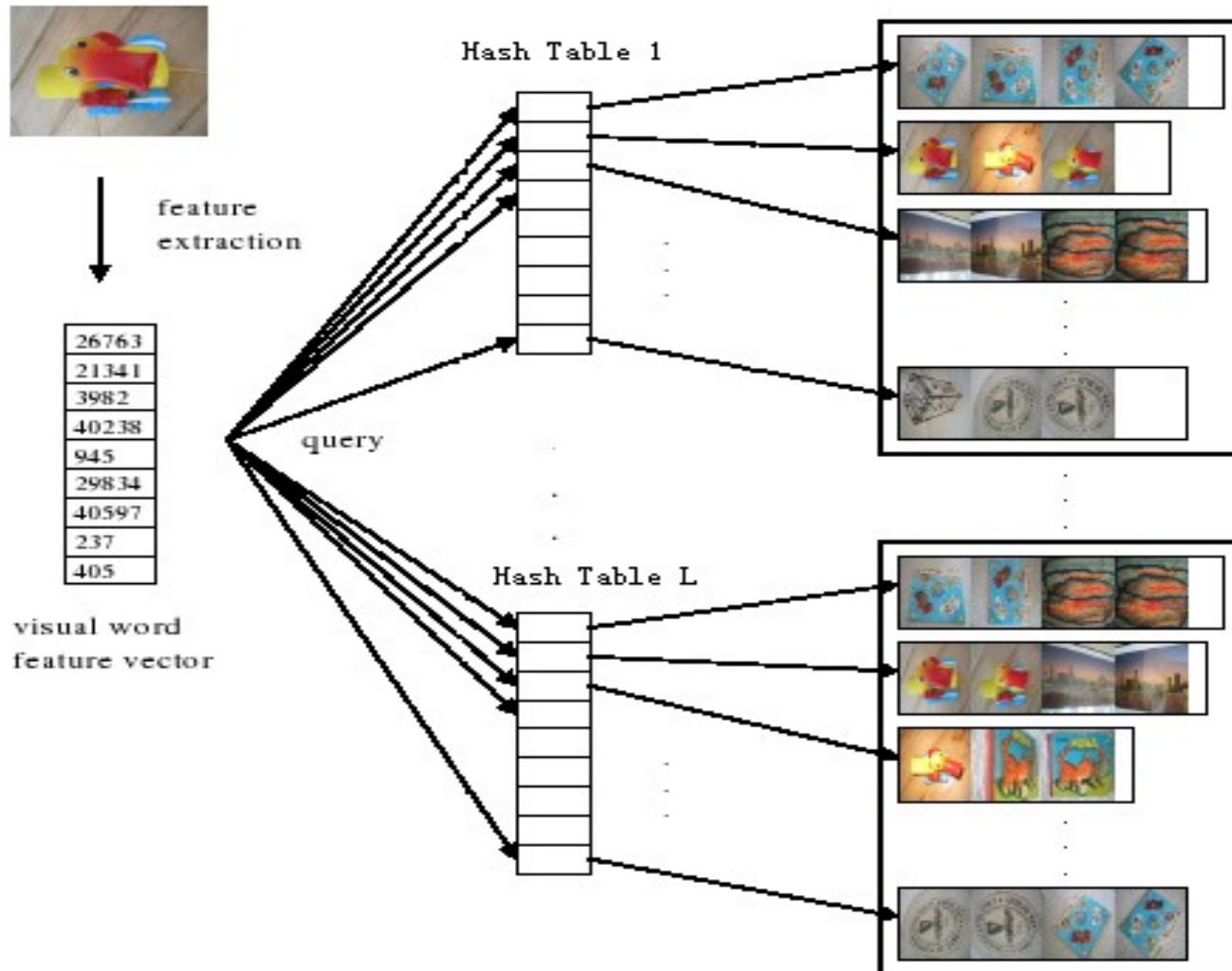


# Motivation: Curse of Dimension

- The tree structure is still quite bad though, when the dimension is around 20-30
- Unfortunately, it seems inevitable (both in theory and experiments) “Curse of Dimension”



# Hash Table



# Locality Sensitive Hashing [Indyk-Motwani'98]

- Hash functions are *locality-sensitive*, if, for a random hash random function  $h$ , for any pair of points  $p, q$  we have:
  - $Pr[h(p)=h(q)]$  is “high” if  $p$  is “close” to  $q$
  - $Pr[h(p)=h(q)]$  is “low” if  $p$  is “far” from  $q$

*The probabilities are based on the functions from the family  $H$ .*



# Locality Sensitive Hashing

- A family  $H$  of functions  $h: \mathbb{R}^d \rightarrow U$  is called  **$(r, cr, P1, P2)$ -sensitive**, if for any  $p, q$ :
  - if  $\|p-q\| < r$  then  $\Pr[ h(p)=h(q) ] > P1$
  - if  $\|p-q\| > cr$  then  $\Pr[ h(p)=h(q) ] < P2$

Now, we consider NN with parameter  $r, \epsilon$ . Set  $r_1=r$ ,  $r_2= (1+\epsilon)r$ , where  $c=(1+\epsilon)$ .

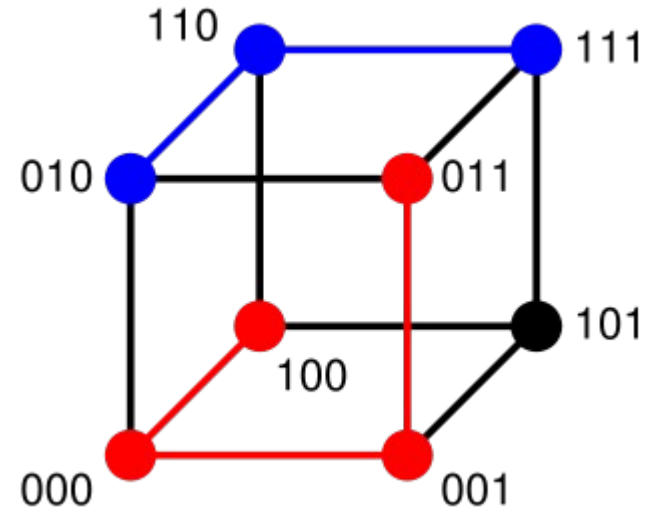
# LSH:Function Exist?

- Consider the hypercube, i.e.,
  - points from  $\{0, 1\}^d$
  - Hamming distance  $D(p, q) = \#$  positions on which  $p$  and  $q$  differ
- Define hash function  $h$  by choosing a set  $I$  of  $k$  random coordinates, and setting

$$h(p) = \text{projection of } p \text{ on } I$$

# LSH: Hamming Distance

- Take
  - $d=10, p=0101110010$
  - $k=2, l=\{2,5\}$
- Then  $h(p)=11$



3-bit binary cube

Two example distances: 100- $\rightarrow$ 011 has distance 3 (red path); 010- $\rightarrow$ 111 has distance 2 (blue path)

- *Probabilities:*

$$Pr[ h(p)=h(q) ] = 1-D(p,q)/d$$

# LSH: Preprocessing

**Algorithm:** Preprocessing,  $O(\ln)$

**Input:** A set of points  $P$ ,  $l$  (number of hash tables)

**Output:** Hash tables  $T_i$ ,  $i = 1, \dots, l$

Foreach  $i = 1, \dots, l$

    Initialize hash table  $T_i$  by generating  
    a random hash function  $G_i(\cdot)$

Foreach  $i = 1, \dots, l$

    Foreach  $j = 1, \dots, n$

        Store point  $P_j$  on bucket  $G_i(P_j)$  of hash table  $T_i$

# LSH: Approximate Nearest Neighbor Query

**Algorithm** Approximate Nearest Neighbor Query,  $O(l)$

**Input** A query point  $q$ ,  $M$  (number of approximate nearest neighbors)

**Output**  $M$  (or less) approximate nearest neighbors

$S \leftarrow \emptyset$

Foreach  $i = 1, \dots, l$

$S \leftarrow S \cup \{\text{points found in } G_i(q) \text{ bucket of table } T_i\}$

Return  $M$  nearest neighbors of  $q$  found in set  $S$

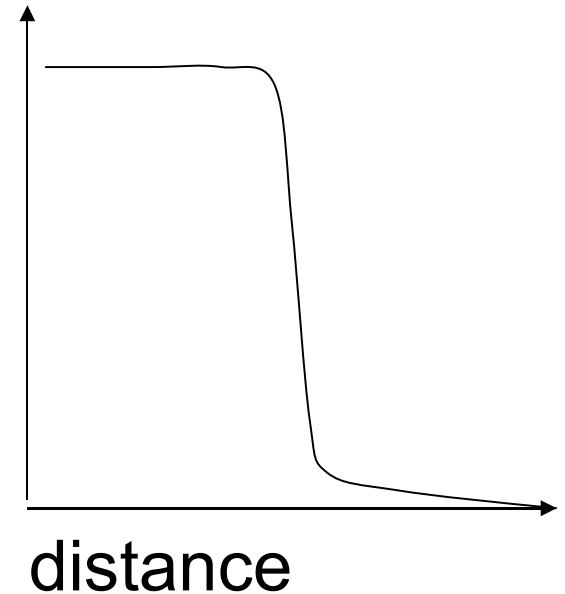
/\*Can be found by main memory linear search\*/

# LSH: Analysis (more proof and analysis in GIM99')

- By proper choice of parameters  $k$  and  $l$ , we can make, for any  $p$ , the probability that  $h_i(p) = h_i(q)$  for some  $i$

look like this:

- $K = \log_{(1/p_2)}(n/B)$  where  $B$  is size of bucket
- $l = \left(\frac{n}{B}\right)^v$  where  $v = \frac{(\ln(1/p_1))}{(\ln(1/p_2))}$





# Topics (Nearest Neighbor Searching)

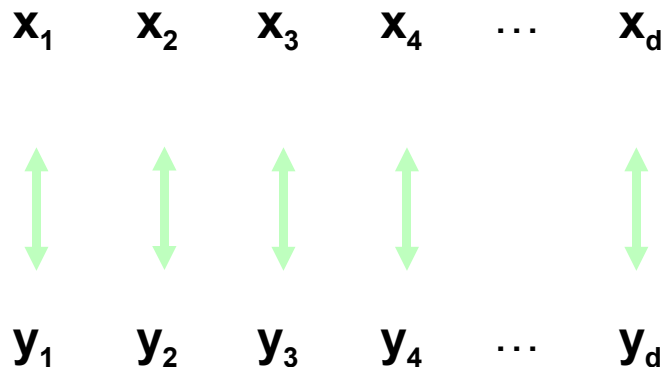
- **Problem Definition**
- **Basic Structure**
  - Quad-Tree
  - KD-Tree
  - Locality Sensitive Hashing
- **Application: Learning**
  - **BoostMap: A Method for Efficient Approximate Similarity Rankings**  
(Thanks for Prof. Athitsos' help)
- **Application: Vision**
  - A Binning Scheme for Fast Hard Driver Based Image Search\*
  - Fast Pose Estimation with Parameter Sensitive Hashing

# Motivation: Non-Metric Distance

- Distance function may be non-metric.
- Each query requires  $n$  distance calculation for a database of size  $n$ .
- What if the distance function is very complicated and expensive computationally.
- **The Solution: BoostMap**  
BoostMap is a method that can reduce the number of expensive distance calculations down to some  $d \ll n$ .  
It works for ANY distance function.

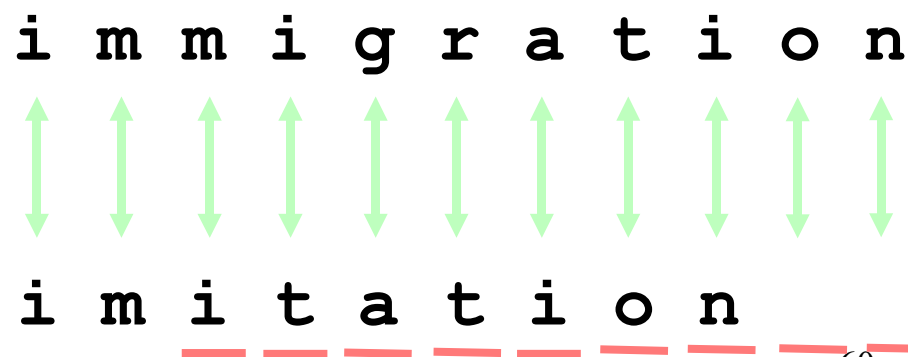
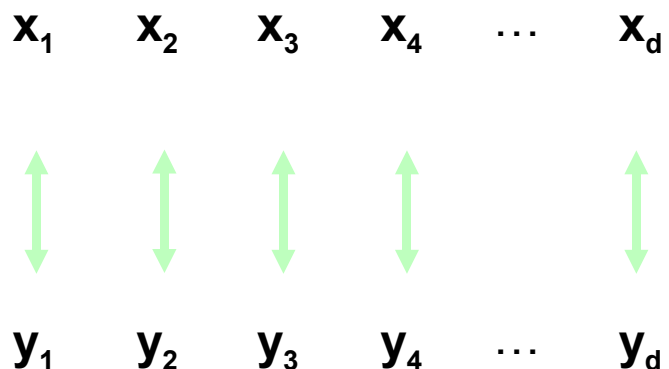
# Expensive Distance Measures

- Comparing  $d$ -dimensional vectors is efficient:
  - $O(d)$  time.



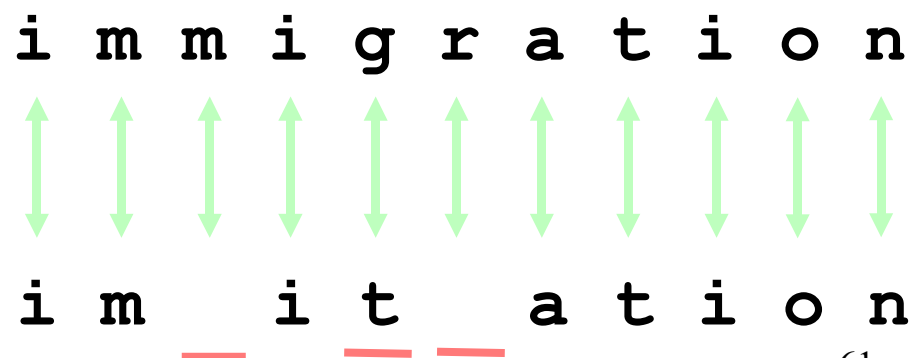
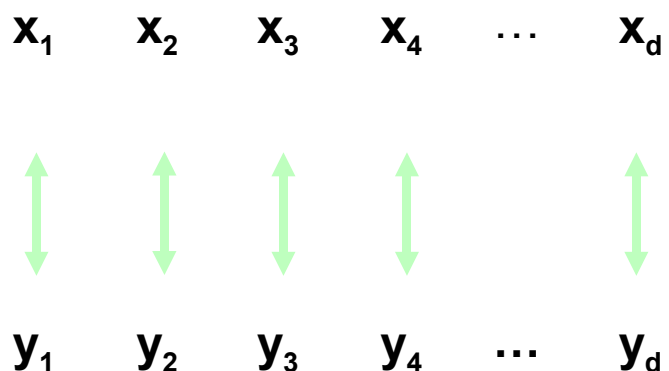
# Expensive Distance Measures

- Comparing  $d$ -dimensional vectors is efficient:
  - $O(d)$  time.
- Comparing strings of length  $d$  with the edit distance is more expensive:
  - $O(d^2)$  time
- Reason:alignment.



# Expensive Distance Measures

- Comparing d-dimensional vectors is efficient:
  - $O(d)$  time.
- Comparing strings of length  $d$  with the edit distance is more expensive:
  - $O(d^2)$  time.
- Reason: alignment.



# Hand Shape Classification

Database (80,640 images)



query



# Hand Shape Classification

Database (80,640 images)



query



Chamfer distance: 112 seconds per query

# Embeddings

database

$x_1$

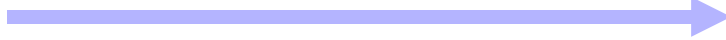
$x_2$

$x_3$

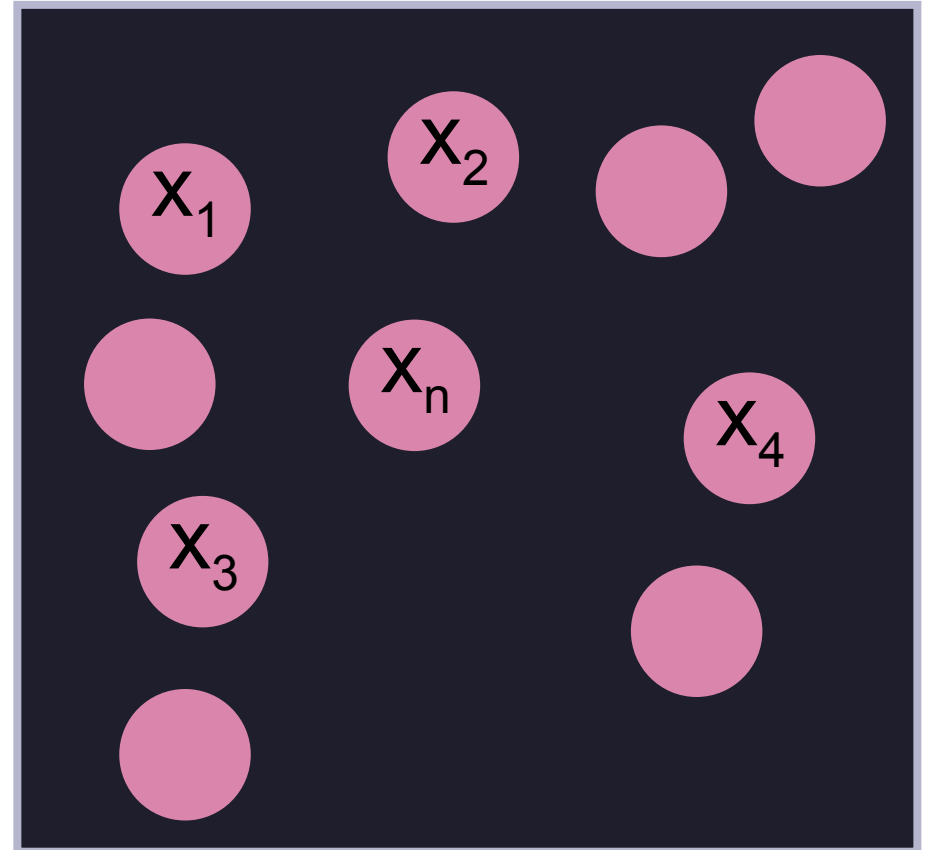


$x_n$

embedding  
 $F$



$\mathbb{R}^d$





# Embeddings

database

$x_1$

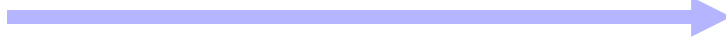
$x_2$

$x_3$



$x_n$

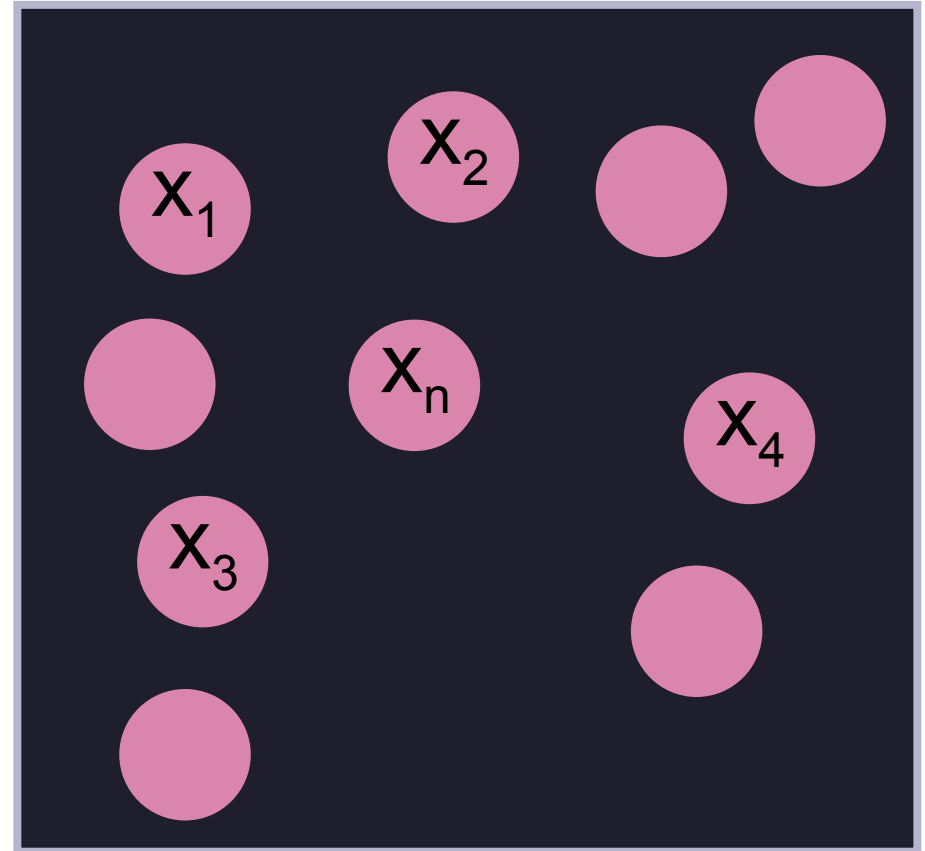
embedding  
 $F$



query

$q$

$\mathbb{R}^d$



# Embeddings

database

$x_1$

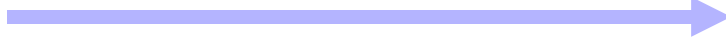
$x_2$

$x_3$

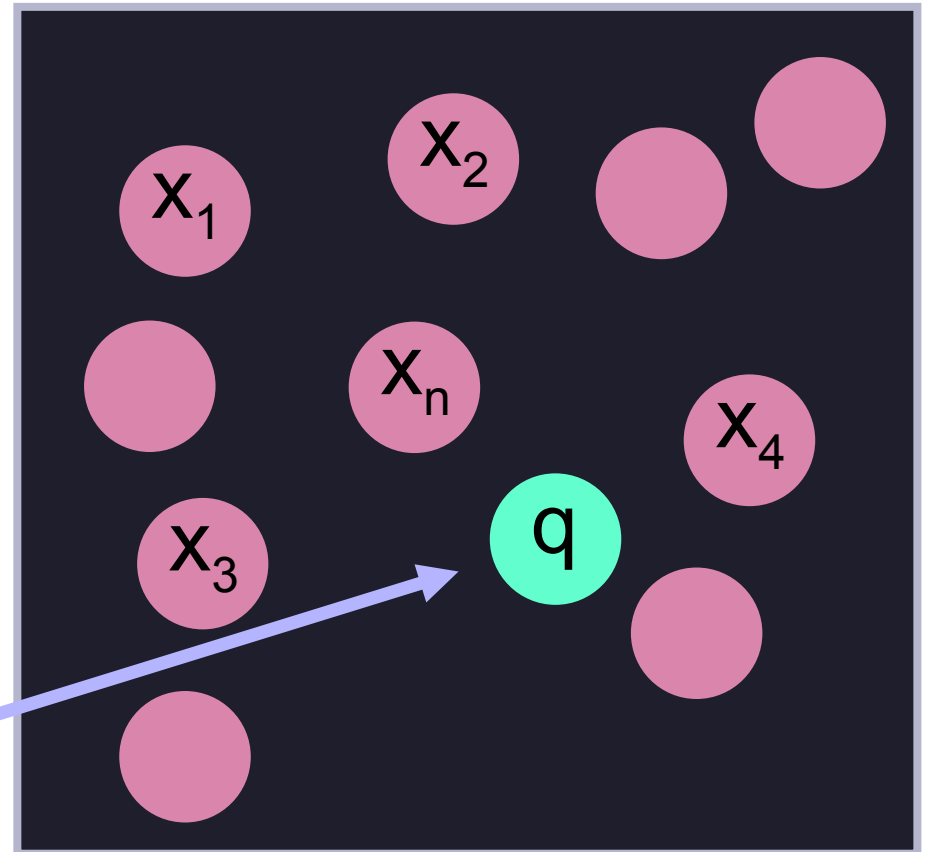


$x_n$

embedding  
 $F$



$\mathbb{R}^d$



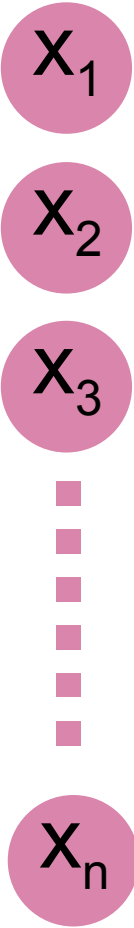
query

$q$

- Measure distances between vectors (typically much faster).

# Embeddings

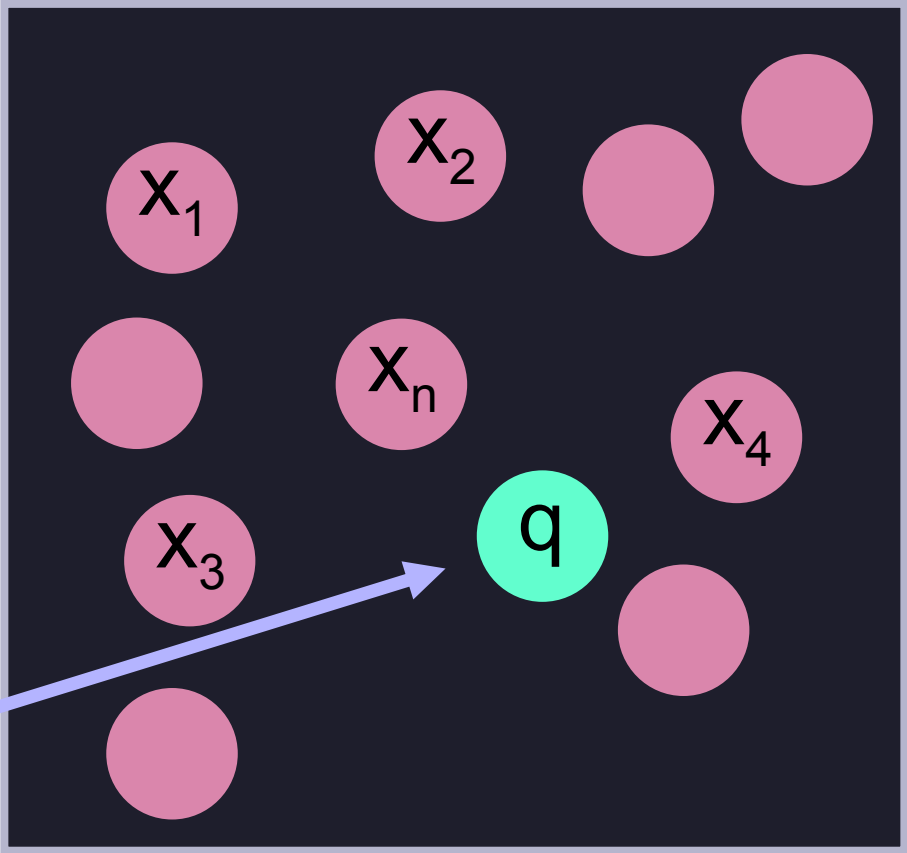
database



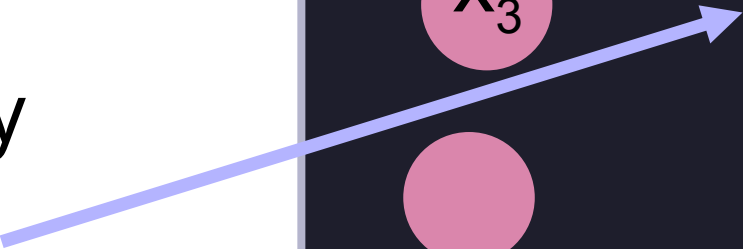
embedding  
 $F$



$\mathbb{R}^d$



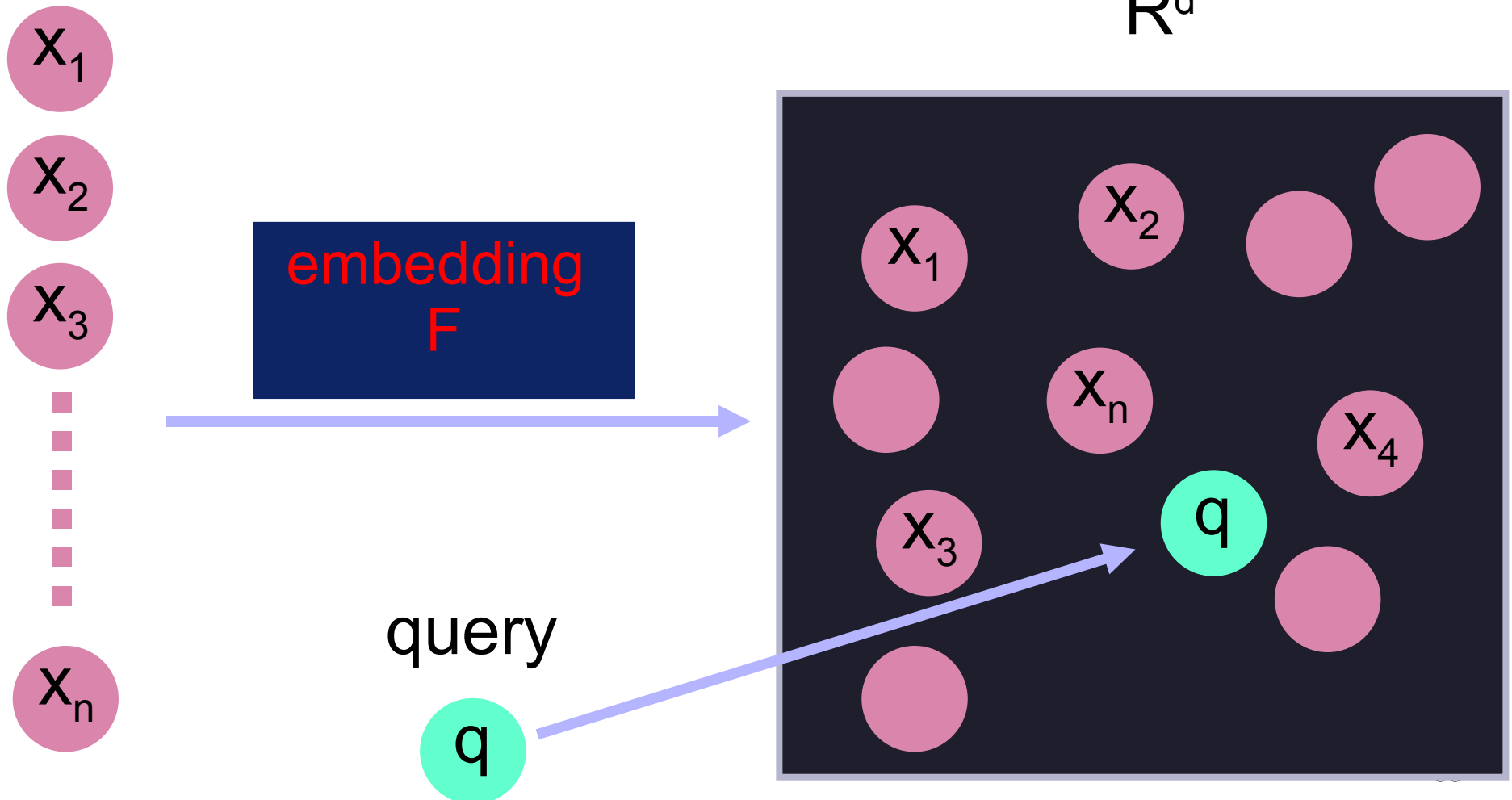
query



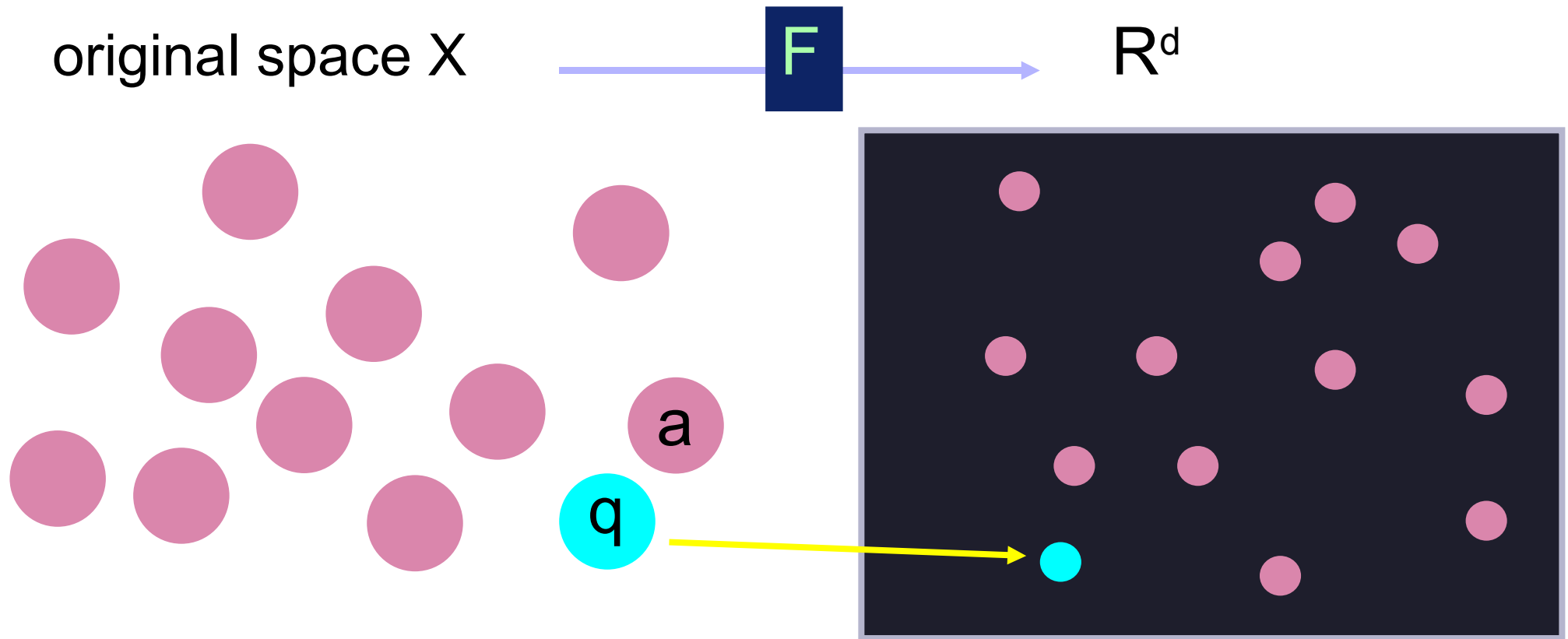
# Embeddings

- Measure distances between vectors (typically much faster).
- **Caveat: the embedding must preserve similarity structure.**

database

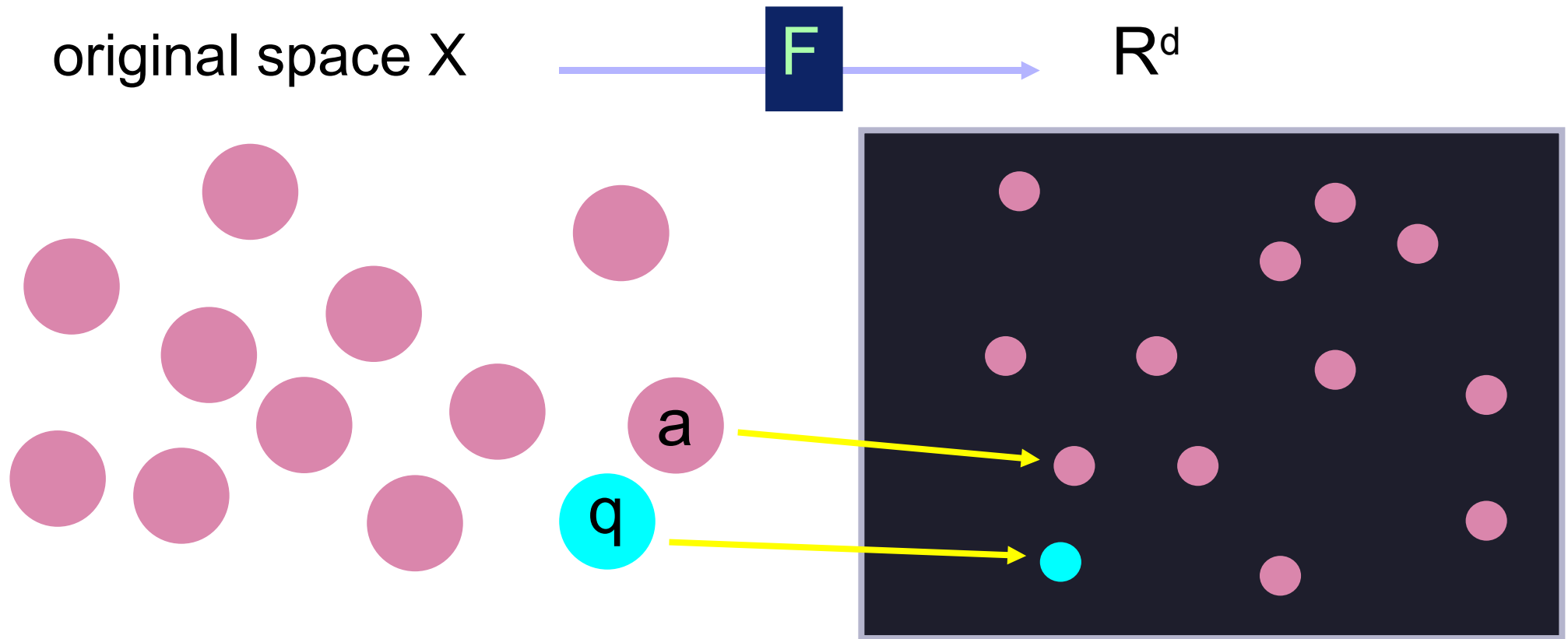


# Ideal Embedding Behavior



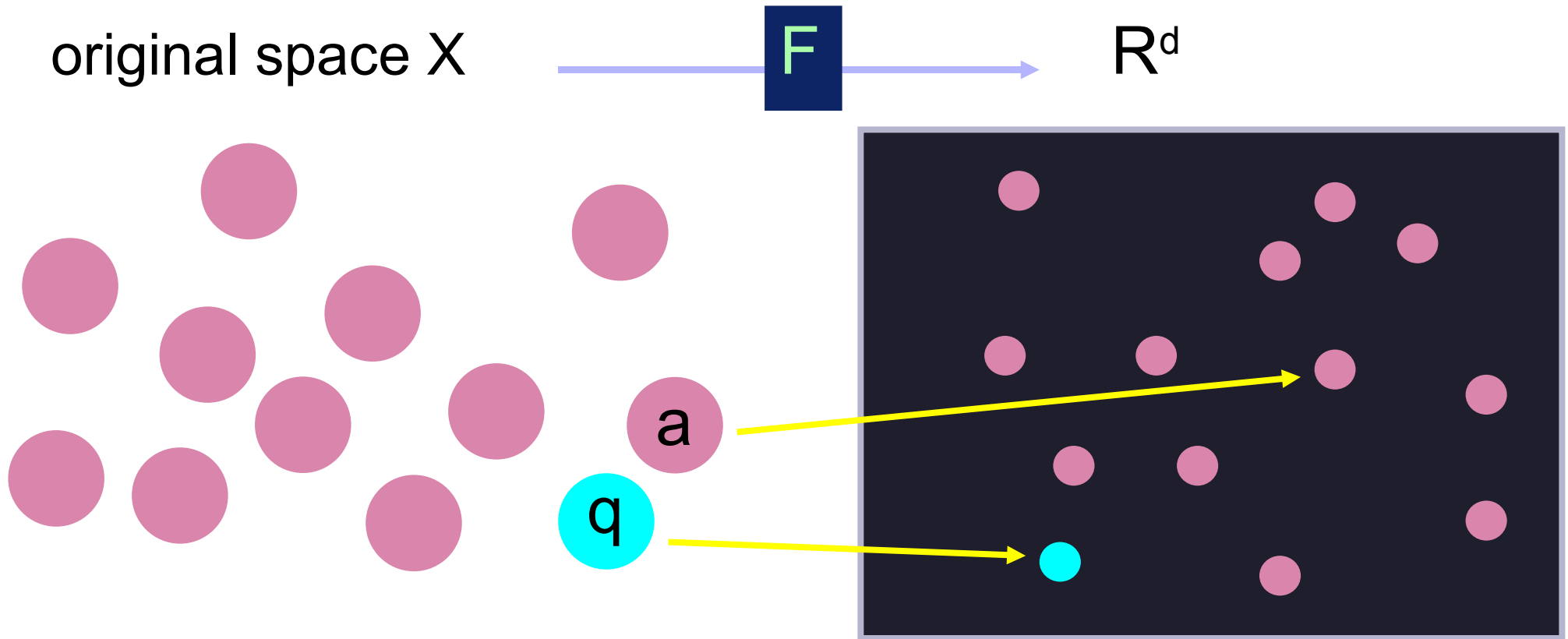
For any  $q$ : we want  $F(\text{NN}(q)) = \text{NN}(F(q))$ .

# Ideal Embedding Behavior



For any  $q$ : we want  $F(\text{NN}(q)) = \text{NN}(F(q))$ .

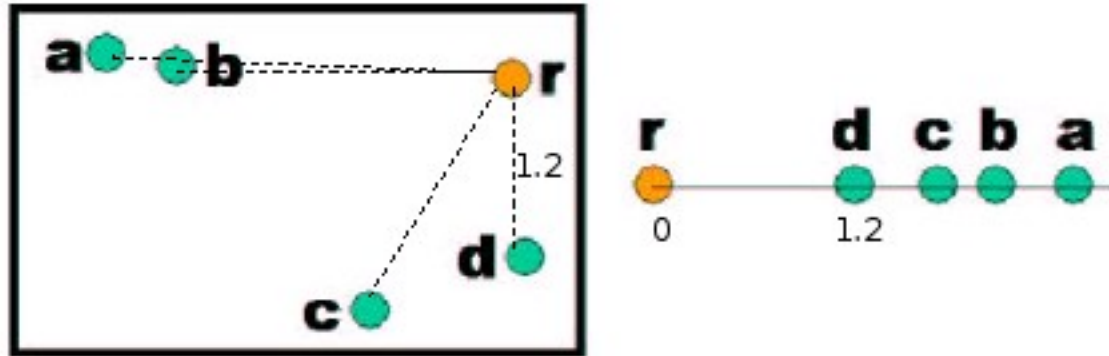
# Ideal Embedding Behavior



For any  $q$ : we want  $F(\text{NN}(q)) = \text{NN}(F(q))$ .

- BoostMap: 1D Embeddings

- Use a reference object  $r$

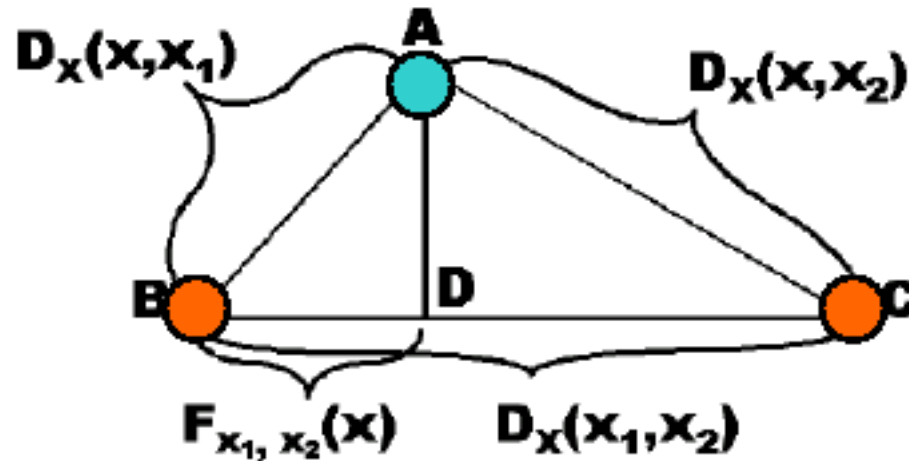


A set of five 2D points (shown on the left), and an embedding  $F$  of those five points into the real line, using  $r$  as the reference object.



- BoostMap: 1D Embeddings

- Use “pivot points”

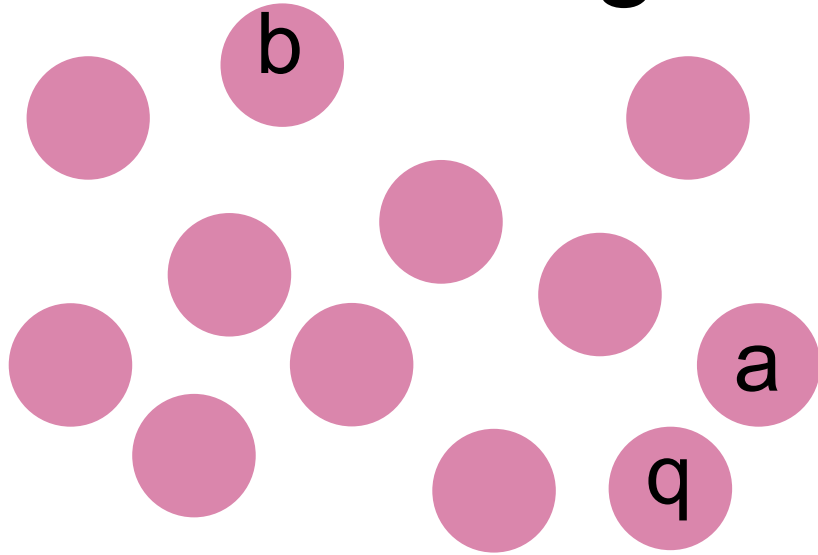


Select the pair  $(x_1, x_2)$  and construct the triangle using  $(x, x_1, x_2)$ .

The length of line segment BD is equal to  $F^{(x_1, x_2)}(x)$

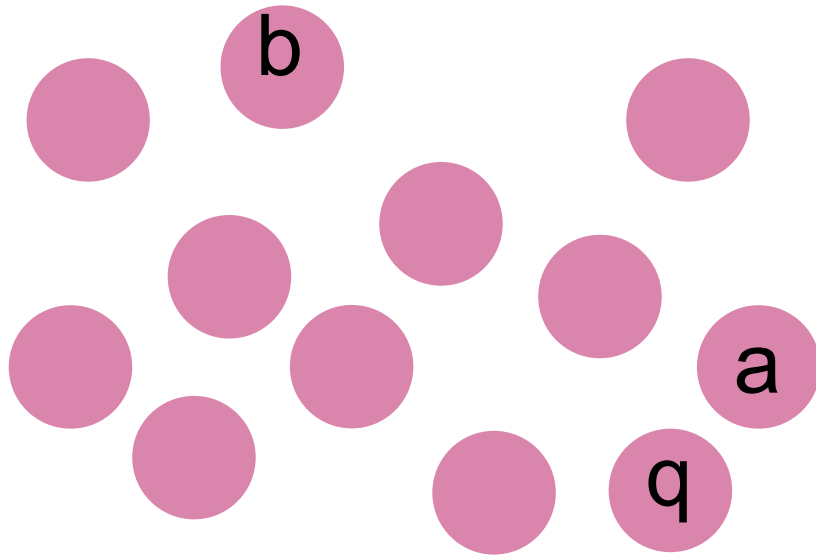
(Triangle inequality?)

# Embeddings Seen As Classifiers



Classification task: is q  
closer to a or to b?

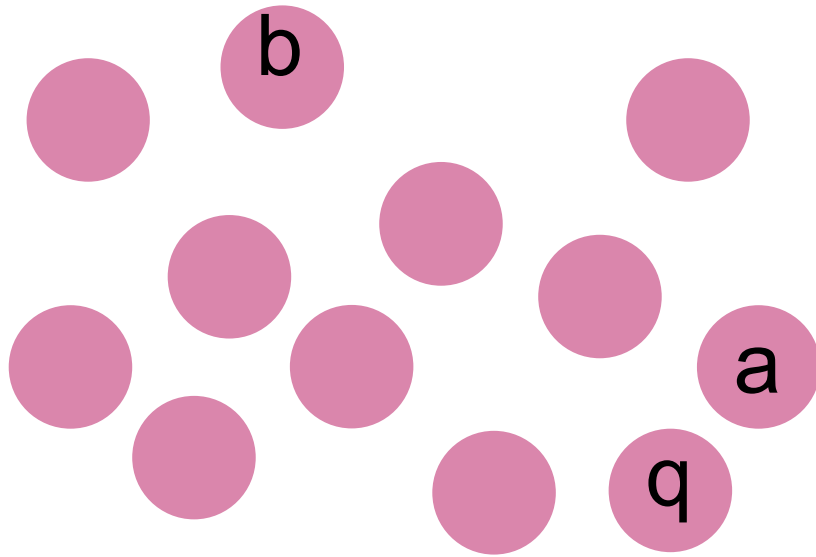
# Embeddings Seen As Classifiers



Classification task: is  $q$  closer to  $a$  or to  $b$ ?

- Any embedding  $F$  defines a classifier  $F'(q, a, b)$ .
  - $F'$  checks if  $F(q)$  is closer to  $F(a)$  or to  $F(b)$ .

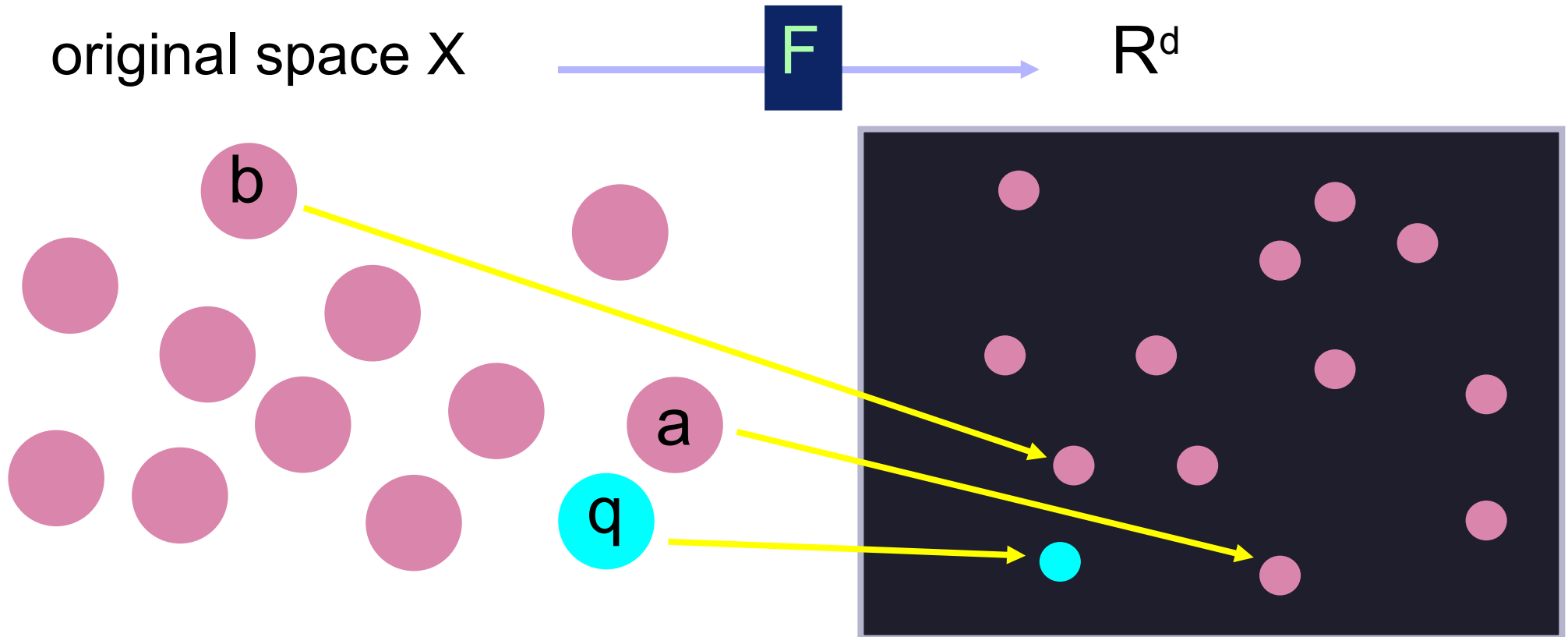
# Classifier Definition



Classification task: is  $q$  closer to  $a$  or to  $b$ ?

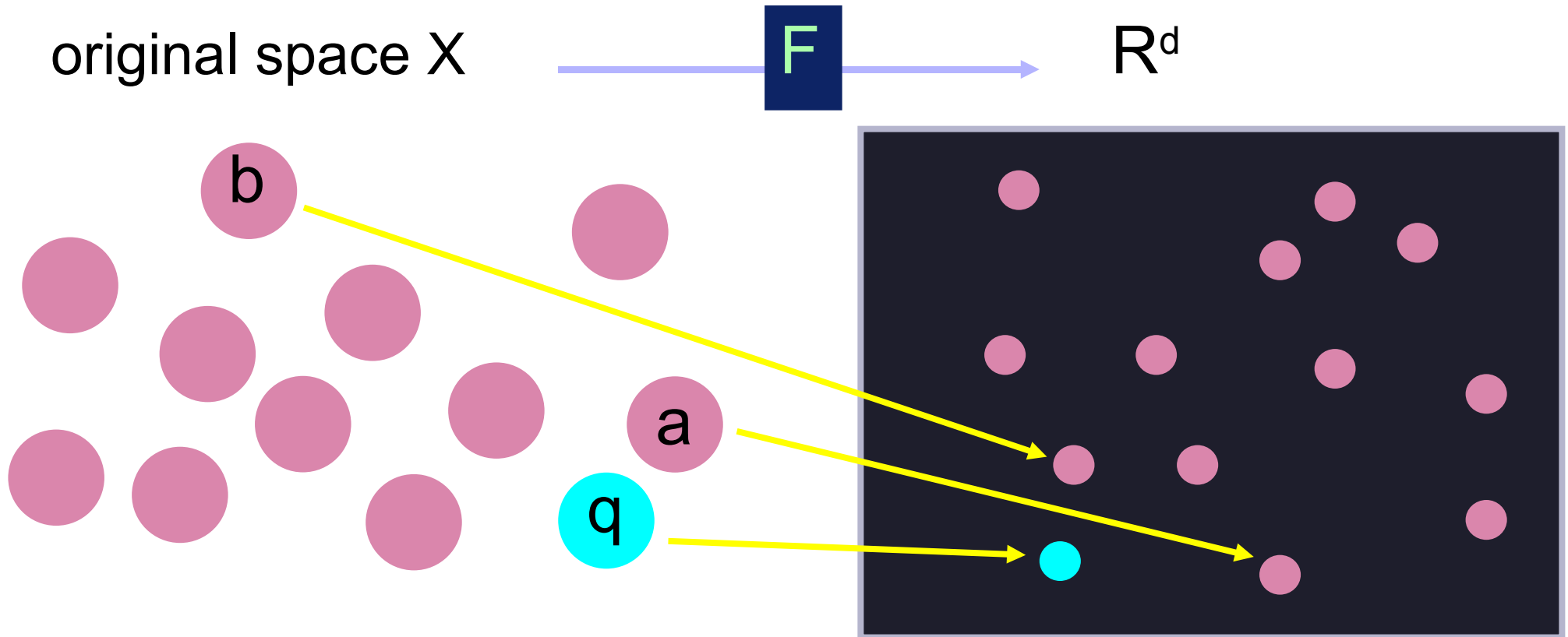
- Given embedding  $F: X \rightarrow \mathbb{R}^d$ :
  - $F'(q, a, b) = \|F(q) - F(b)\| - \|F(q) - F(a)\|$ .
- $F'(q, a, b) > 0$  means “ $q$  is closer to  $a$ .”
- $F'(q, a, b) < 0$  means “ $q$  is closer to  $b$ .”

# Key Observation



- If classifier  $F'$  is perfect, then for every  $q$ ,  $F(\text{NN}(q)) = \text{NN}(F(q))$ .
  - If  $F(q)$  is closer to  $F(b)$  than to  $F(\text{NN}(q))$ , then triple  $(q, a, b)$  is misclassified.

# Key Observation



- Classification error on triples  $(q, \text{NN}(q), b)$  measures how well  $F$  preserves nearest neighbor structure.

# Optimization Criterion

- **Goal: construct an embedding  $F$  optimized for  $k$ -nearest neighbor retrieval.**
- **Method: maximize accuracy of  $F'$  on triples  $(q, a, b)$  of the following type:**
  - $q$  is any object.
  - $a$  is a  $k$ -nearest neighbor of  $q$  in the database.
  - $b$  is in database, but **NOT** a  $k$ -nearest neighbor of  $q$ .
- **If  $F'$  is perfect on those triples, then  $F$  perfectly preserves  $k$ -nearest neighbors.**

# Overview of Strategy

- Start with simple 1D embeddings.
- Convert 1D embeddings to classifiers.
- Combine those classifiers into a single, optimized classifier.
- Convert optimized classifier into a multidimensional embedding.



# 1D Embeddings as Weak Classifiers

- 1D embeddings define *weak classifiers*.
  - Better than a random classifier (50% error rate).

# 1D Embeddings as Weak Classifiers

- 1D embeddings define *weak classifiers*.
  - Better than a random classifier (50% error rate).
- We can define lots of different classifiers.
  - Every object in the database can be a reference object.
  - Each pair also can work as 'pivot'.\*

# 1D Embeddings as Weak Classifiers

- 1D embeddings define *weak classifiers*.
  - Better than a random classifier (50% error rate).
- We can define lots of different classifiers.
  - Every object in the database can be a reference object.
  - Each pair also can work as 'pivot'.\*

Question: how do we combine many such classifiers into a single *strong* classifier?

# 1D Embeddings as Weak Classifiers

- 1D embeddings define *weak classifiers*.
  - Better than a random classifier (50% error rate).
- We can define lots of different classifiers.
  - Every object in the database can be a reference object.
  - Each pair also can work as 'pivot'.\*

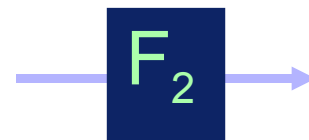
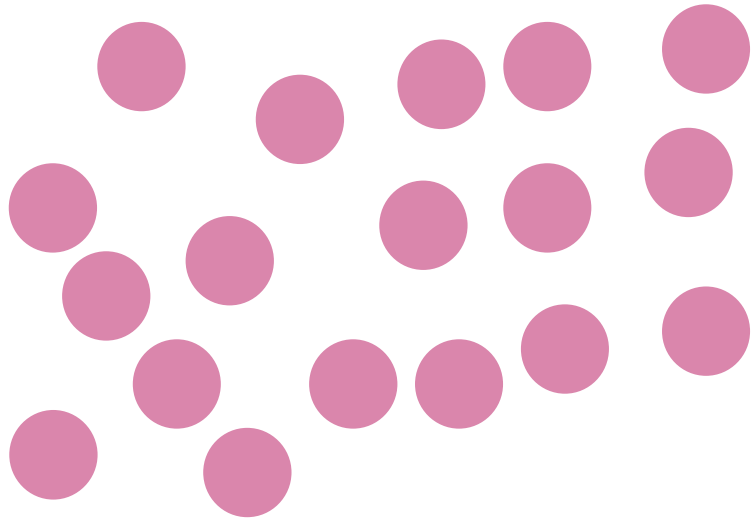
Question: how do we combine many such classifiers into a single *strong* classifier?

Answer: use AdaBoost.

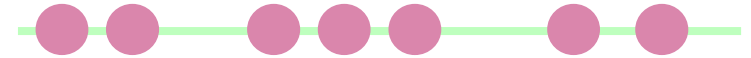
- AdaBoost is a machine learning method designed for exactly this problem.

# Using AdaBoost

original space  $X$



Real line

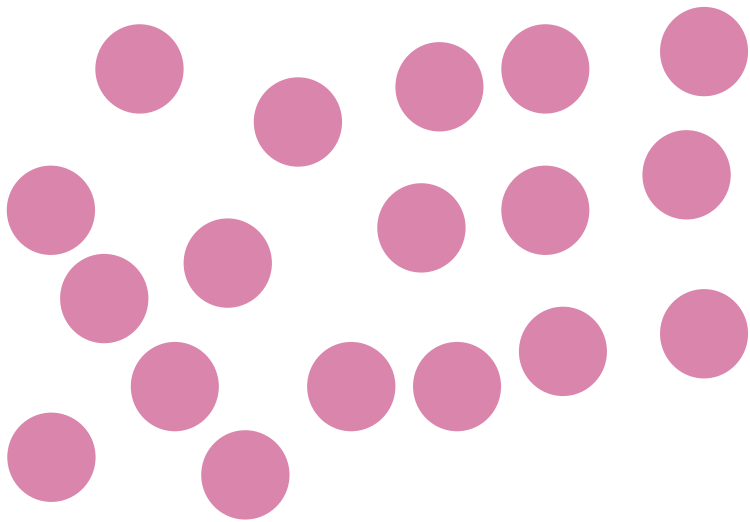


- Output:  $H = w_1 F'_1 + w_2 F'_2 + \dots + w_d F'_d$ .
  - AdaBoost chooses 1D embeddings and weighs them.
  - Goal: achieve low classification error.
  - AdaBoost trains on triples chosen from the database.

# BoostMap : Input

- A training set of  $T = ((q_1, a_1, b_1), \dots, (q_t, a_t, b_t))$  of  $t$  triples of objects from  $X$
- A set of labels  $Y = (y_1, \dots, y_t)$ , where  $y_i \in (-1, 1)$  is the class label of  $(q_i, a_i, b_i)$   
(no triples where  $q_i$  is equally far from  $a_i$ ,  $b_i$ )
- A set  $C \subset X$  of candidate objects. Elements of  $C$  can be used to define 1D embeddings. (as ref object or pivot points)
- A matrix of distances from each  $c \in C$  to each  $q_i$ ,  $a_i$ , and  $b_i$  included in one of the training triples in  $T$ .

original space  $X$



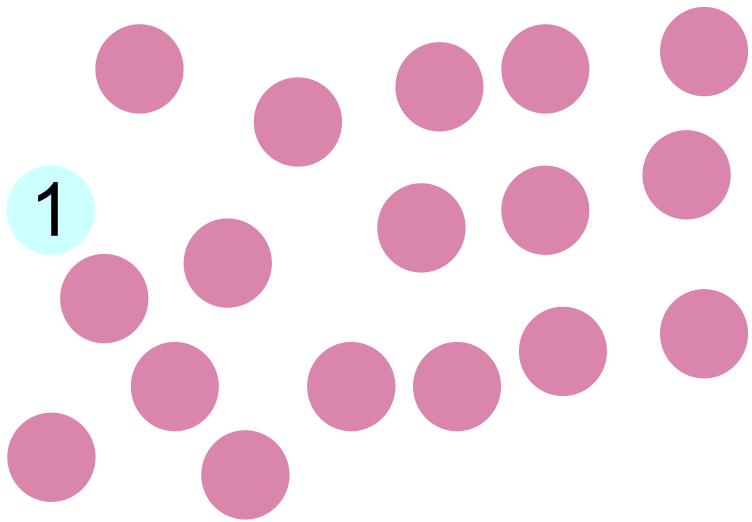
Training triples

Weights

$(q_1, a_1, b_1)$	.....	$1/m$
$(q_2, a_2, b_2)$	.....	$1/m$
$(q_3, a_3, b_3)$	.....	$1/m$
$\vdots$		$\vdots$
$(q_m, a_m, b_m)$	.....	$1/m$

- Training round 0.
- Classifier:  $H = \text{"I don't know"}$ .
- Embedding:  $F = 0$
- Distance:  $D(F(x), F(y)) = 0$ .
  
- Weights: all equal to  $1/m$  (example:  $m = 100,000$ ).

## original space X



## Training triples

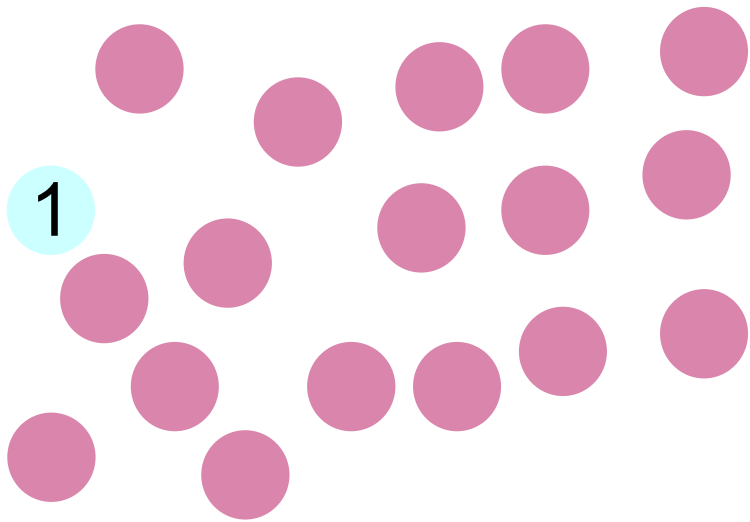
$(q_1, a_1, b_1)$	.....	$1/m$
$(q_2, a_2, b_2)$	.....	$1/m$
$(q_3, a_3, b_3)$	.....	$1/m$
	⋮	⋮
$(q_m, a_m, b_m)$	.....	$1/m$

## Weights

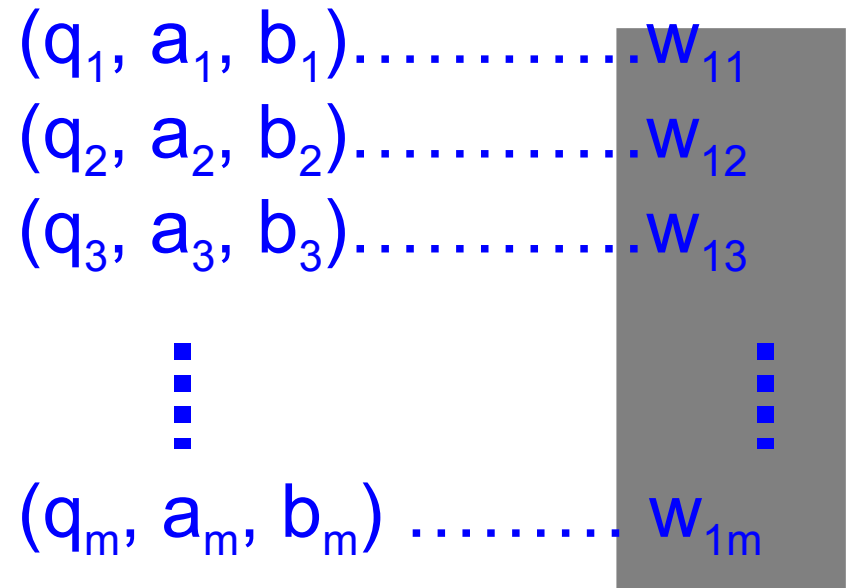
- Training round 1.
- Classifier:  $H = a_1 F'_1$ .
- Embedding:  $F = (F_1)$ .
- Distance:  $D(F(x), F(y)) = a_1 |F_1(x) - F_1(y)|$ .



## original space X

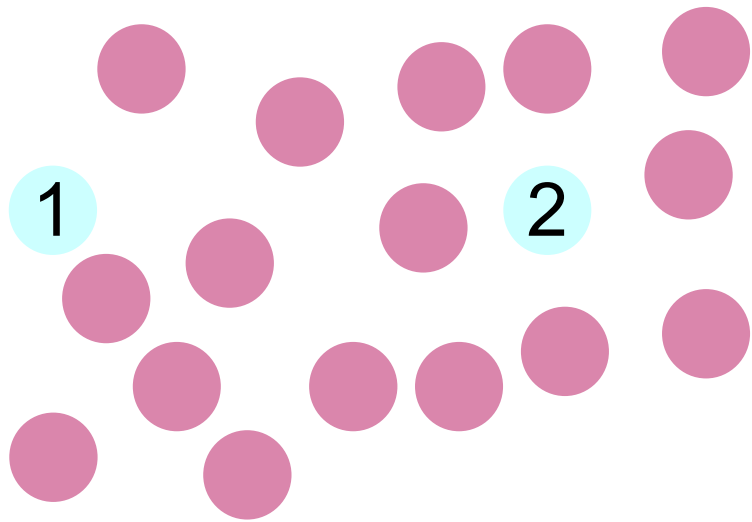


## Training triples

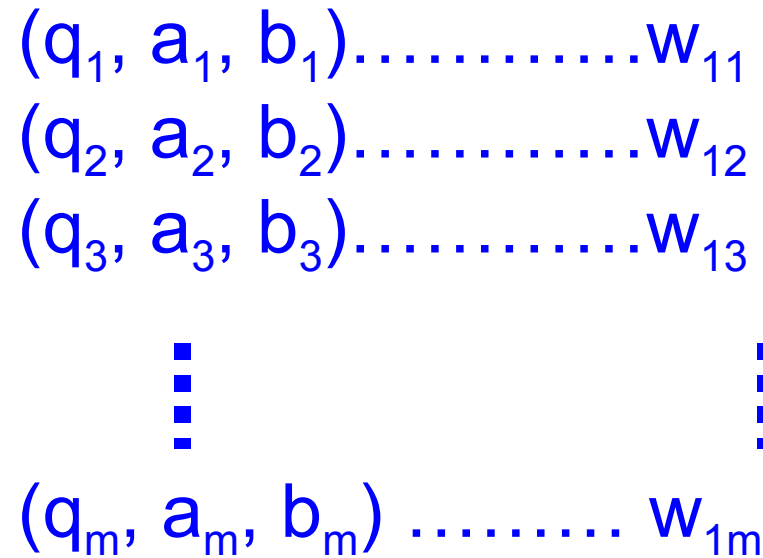


- Training round 1.
- Classifier:  $H = a_1 F'_1$ .
- Embedding:  $F = (F_1)$ .
- Distance:  $D(F(x), F(y)) = a_1 |F_1(x) - F_1(y)|$ .
- Weights: higher for incorrectly classified triples.

# original space X



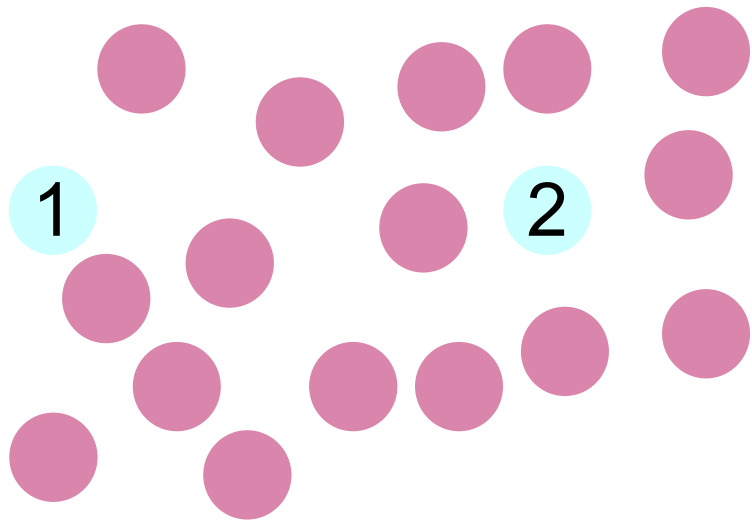
# Training triples



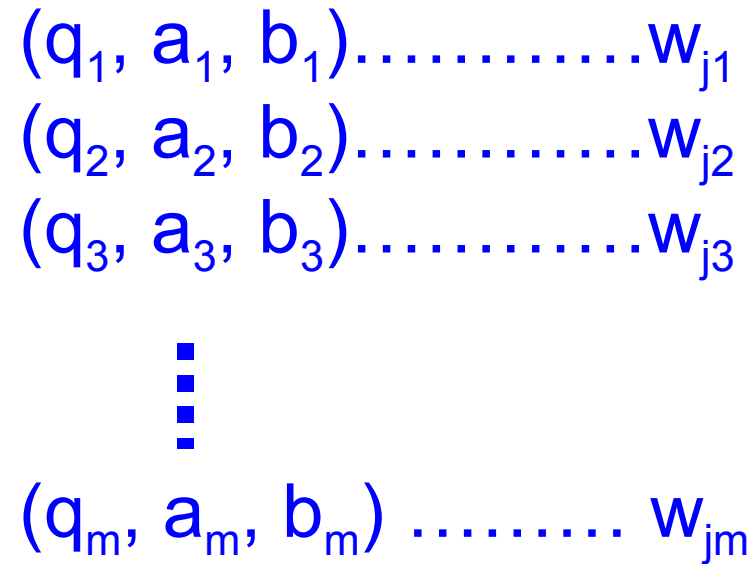
# Weights

- Training round 2.
- Classifier:  $H = a_1 F'_1 + a_2 F'_2$ .
- Embedding:  $F = (F_1, F_2)$ .
- Distance:  $D(F(x), F(y)) = a_1 |F_1(x) - F_1(y)| + a_2 |F_2(x) - F_2(y)|$ .

# original space X



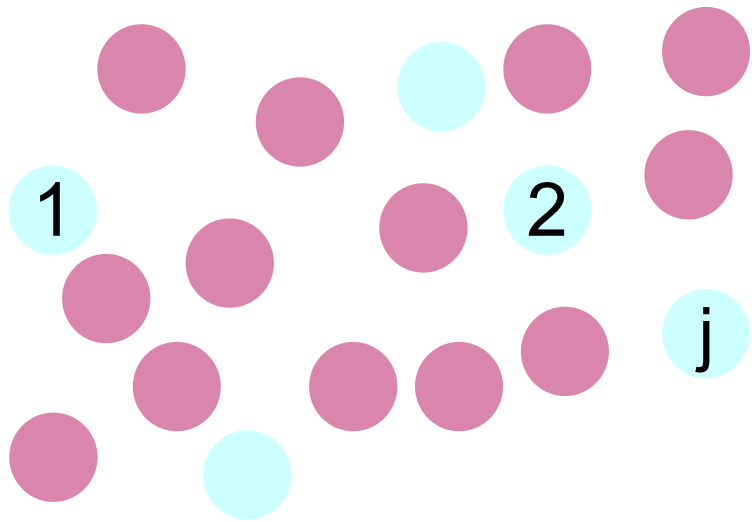
# Training triples



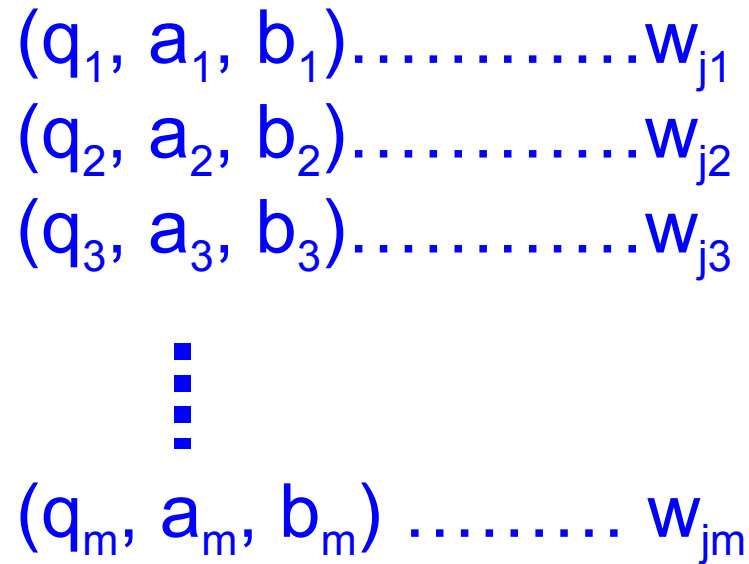
# Weights

- Training round  $j$ .
- Classifier:  $H = a_1 F'_1 + a_2 F'_2 + \dots + a_j F'_j$ .
- Embedding:  $F = (F_1, F_2, \dots, F_n)$ .
- Distance:  $D(F(x), F(y)) = a_1 |F_1(x) - F_1(y)| + a_2 |F_2(x) - F_2(y)| + \dots + a_j |F_j(x) - F_j(y)|$ .

# original space X



# Training triples



# Weights

- Training round  $j$ .
- Classifier:  $H = a_1 F'_1 + a_2 F'_2 + \dots + a_j F'_j$ .
- Embedding:  $F = (F_1, F_2, \dots, F_n)$ .
- Distance:  $D(F(x), F(y)) = a_1 |F_1(x) - F_1(y)| + a_2 |F_2(x) - F_2(y)| + \dots + a_j |F_j(x) - F_j(y)|$ .
- Stop when accuracy stops improving ( $a_j = 0$ ).

# BoostMap: Summary

- Maximizes amount of nearest neighbor structure preserved by the embedding.
- Based on machine learning, not on geometric assumptions.
- Combines efficiency of measuring distances in vector spaces with ability to capture non-metric structure.

# Topics (Nearest Neighbor Searching)

- **Problem Definition**
- **Basic Structure**
  - Quad-Tree
  - KD-Tree
  - Locality Sensitive Hashing
- **Application: Learning**
  - BoostMap: A Method for Efficient Approximate Similarity Rankings
- **Application: Vision**
  - **A Binning Scheme for Fast Hard Driver Based Image Search\***
  - Fast Pose Estimation with Parameter Sensitive Hashing

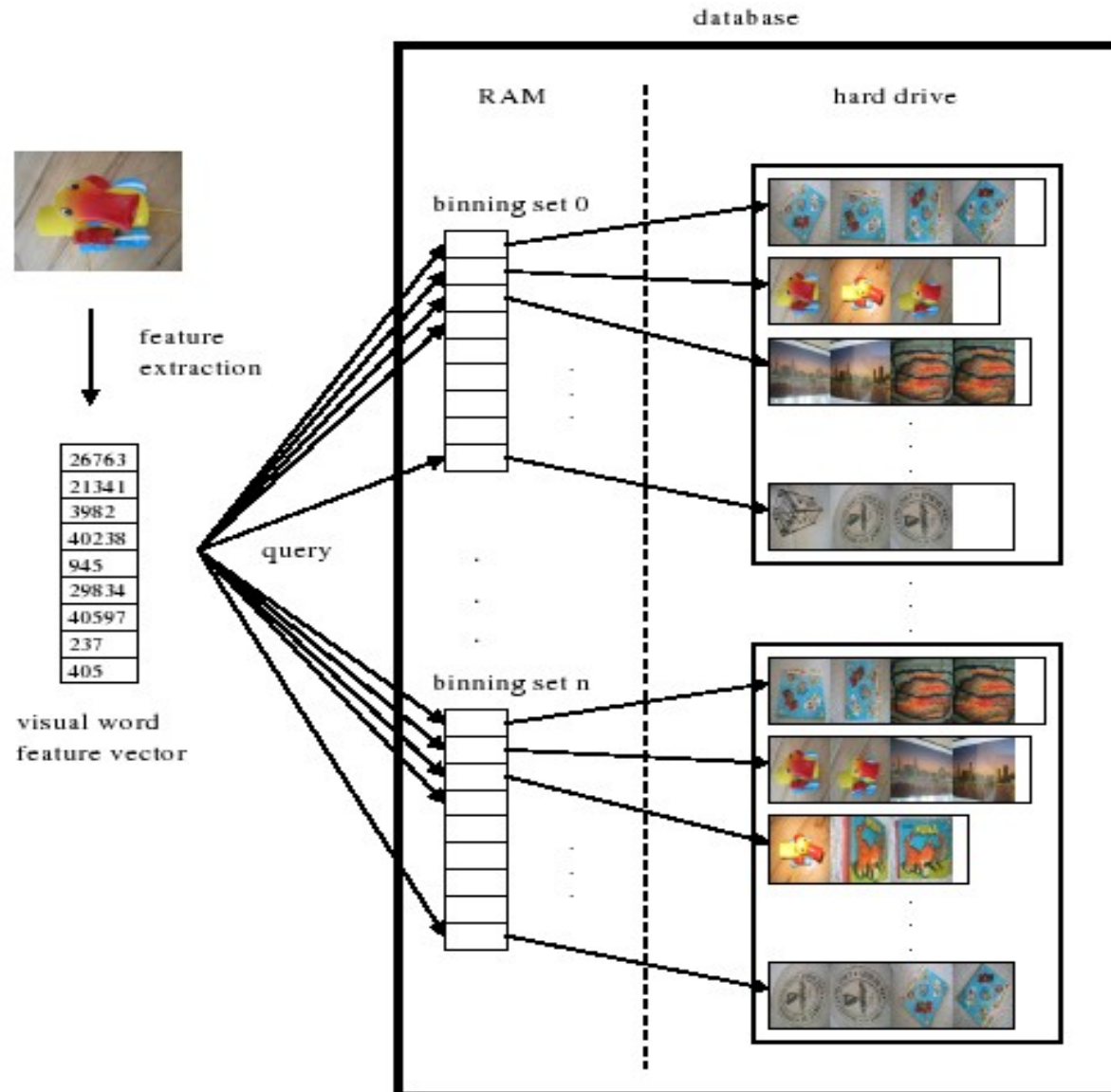
# A Binning Scheme for Fast Hard Drive Based Image Search

# Motivation: Beyond the RAM limits

- Investigate how to scale a content based image retrieval approach beyond the RAM limits of a single computer and to make use of its hard drive to store the feature database.
- The scheme cuts down the hard drive access significantly and results in a major speed up



# A Binning Scheme for Fast Hard Drive Based Image Search



# A Binning Scheme for Fast Hard Drive Based Image Search

- The algorithm is largely inspired by the success of Locality Sensitive Hashing for nearest neighbor search.
- Database consists of multiple independent binnings.
- Each binning is defined by a number of prototypes where a prototype is a vector representing an image.
- The images are assigned to the bin corresponding to the closest prototype, which is used as a proxy in the search.

# A Binning Scheme: Analysis

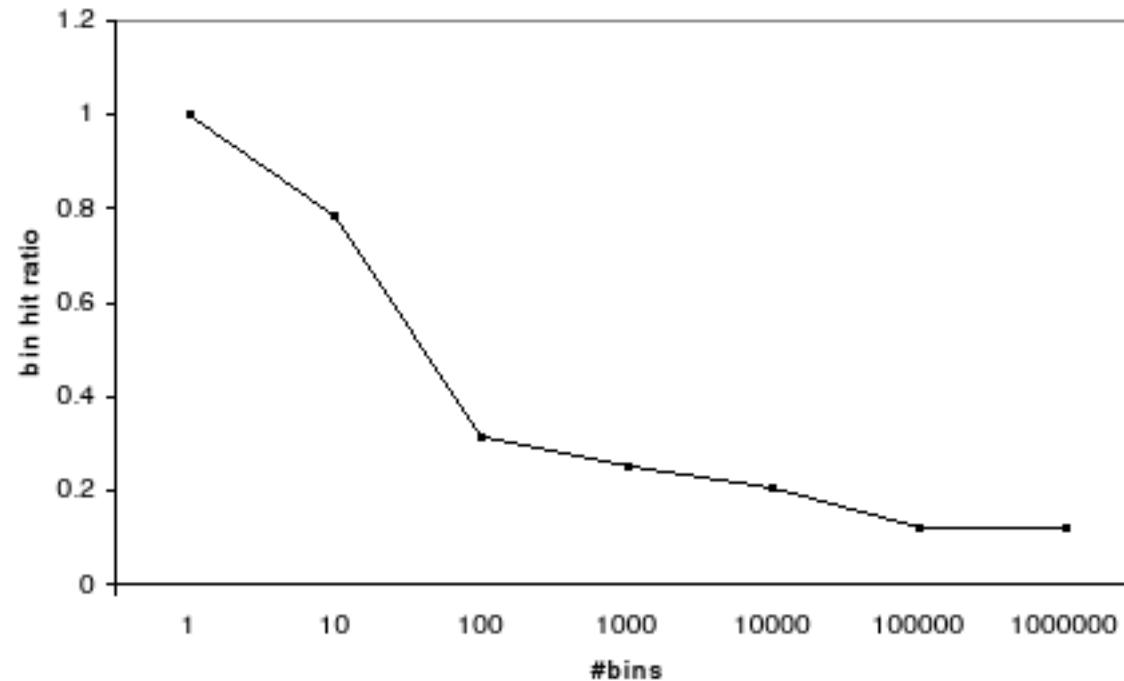


Figure 3. Binning hits: A higher number of bins increases the risk of a bin miss. By using multiple binnings this can be compensated for.

# A Binning Scheme: Analysis

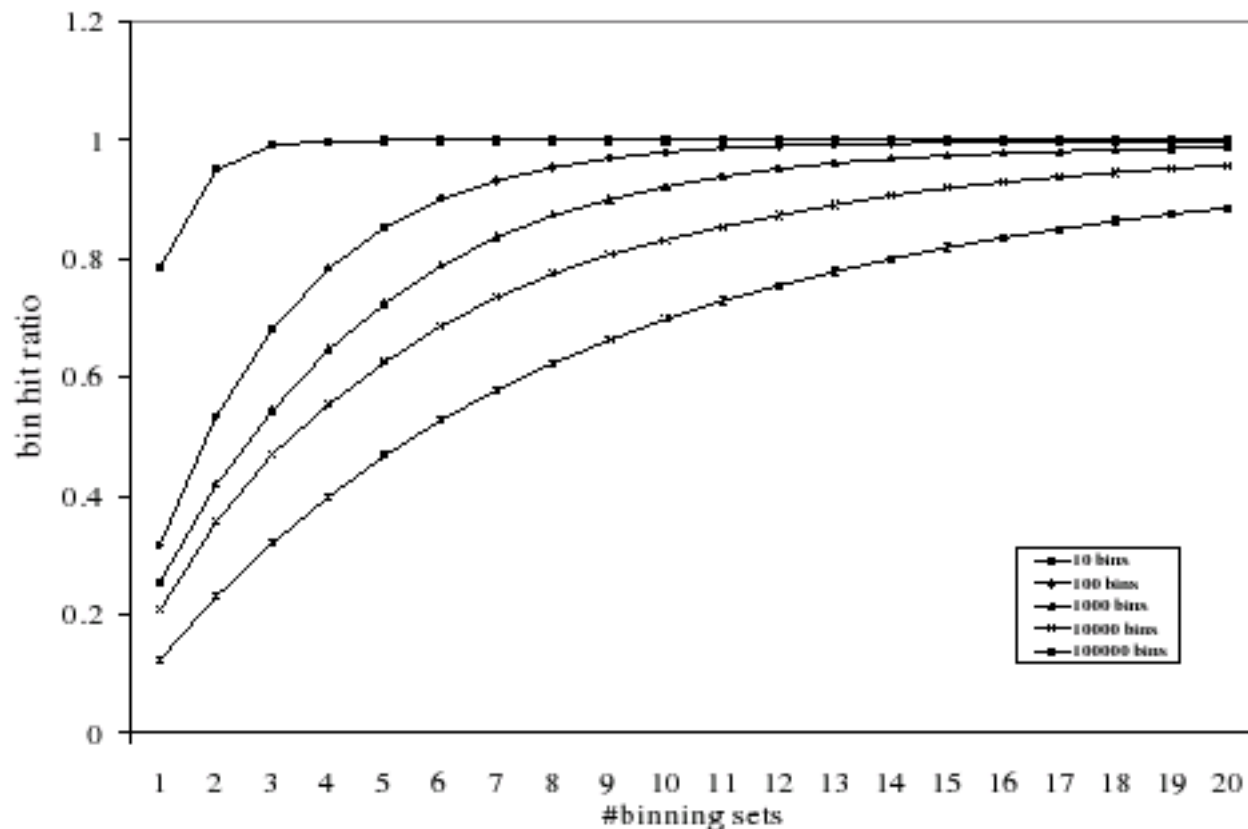


Figure 4. The effect of chaining multiple independent binnings. Each additional set increases the bin hit ratio. The experiment with different numbers of bins shows the tendency of converging to 1.

# Topics (Nearest Neighbor Searching)

- **Problem Definition**
- **Basic Structure**
  - Quad-Tree
  - KD-Tree
  - Locality Sensitive Hashing
- **Application: Learning**
  - BoostMap: A Method for Efficient Approximate Similarity Rankings
- **Application: Vision**
  - A Binning Scheme for Fast Hard Driver Based Image Search\*
  - **Fast Pose Estimation with Parameter Sensitive Hashing**

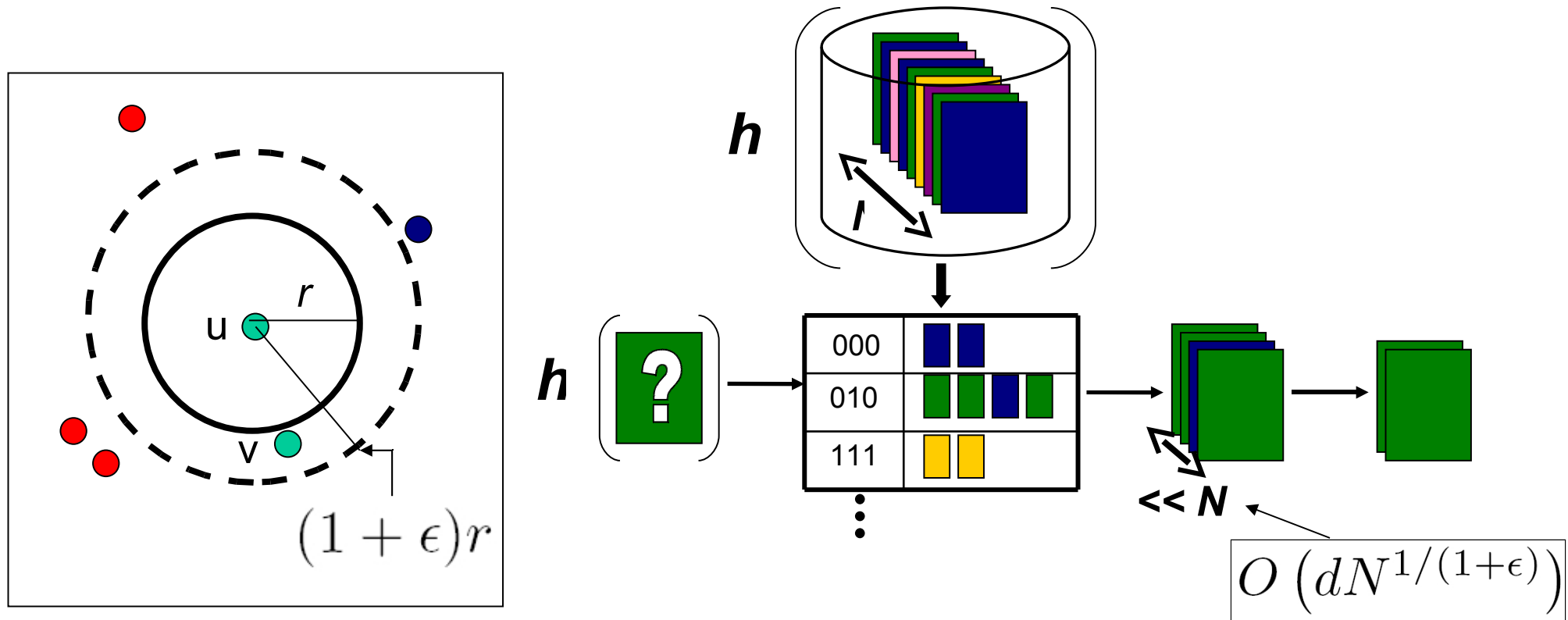
# Fast Pose Estimation with Parameter Sensitive Hashing (Learning Silhouette Features for Control of Human Motion)

Liu Ren, Gregory Shakhnarovich , Jessica K. Hodgins, Hanspeter Pfister , Paul A. Viola

# Motivation: Hidden State Space

- Approximate not the actual distance between objects, but a hidden state space distance.
- $(x, \theta)$   $x$  is feature vector extracted from the image and  $\theta$  is a parameter vector.

# Sub-linear time search with LSH



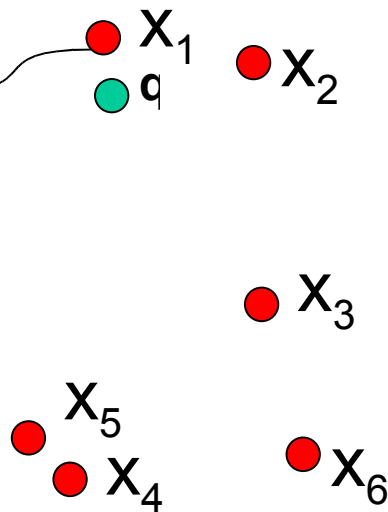
if  $d(\mathbf{u}, \mathbf{v}) \leq r$  then  $\Pr_{\mathcal{H}}(h(\mathbf{u}) = h(\mathbf{v})) \geq p_1$   
 if  $d(\mathbf{u}, \mathbf{v}) > (1 + \epsilon)r$  then  $\Pr_{\mathcal{H}}(h(\mathbf{u}) = h(\mathbf{v})) \leq p_2$

need  $p_1 > p_2$  and  $p_1 > 1/2$



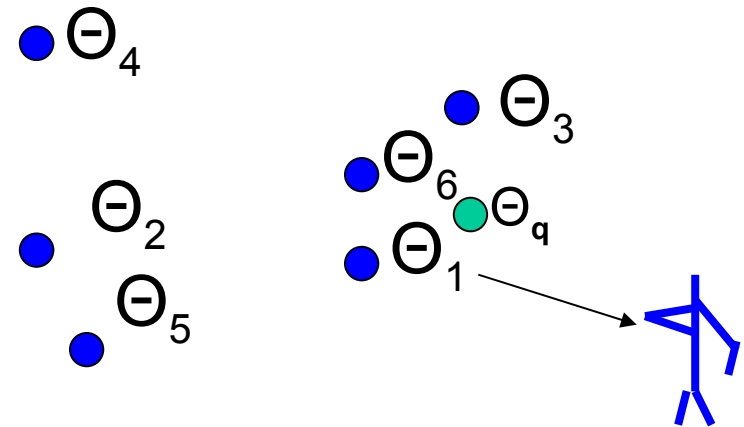
# Indexing for parameter estimation

$$(\mathbf{x}_1, \theta_1), \dots, (\mathbf{x}_N, \theta_N)$$



Input space

Index with LSH and randomized hash functions that respect *input space* locality



Parameter space

This work: learn hash functions that respect *parameter space* locality

# Learning PSH functions

Posed as a paired classification problem:

For each pair of examples  $(\mathbf{x}_i, \mathbf{x}_j)$  assign label

$$y_{ij} = \begin{cases} +1 & \text{if } d_{\theta}(\theta_i, \theta_j) < r, \\ -1 & \text{if } d_{\theta}(\theta_i, \theta_j) > R, \\ \text{not defined} & \text{otherwise,} \end{cases}$$

# Learning PSH functions

- Interpret a binary hash function  $h$  as a classifier:

$$\hat{y}_h(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} +1 & \text{if } h(\mathbf{x}_i) = h(\mathbf{x}_j) \\ -1 & \text{otherwise.} \end{cases}$$

$p_2(h)$  -> probability of false positive

$1-p_1(h)$  -> probability of false negative

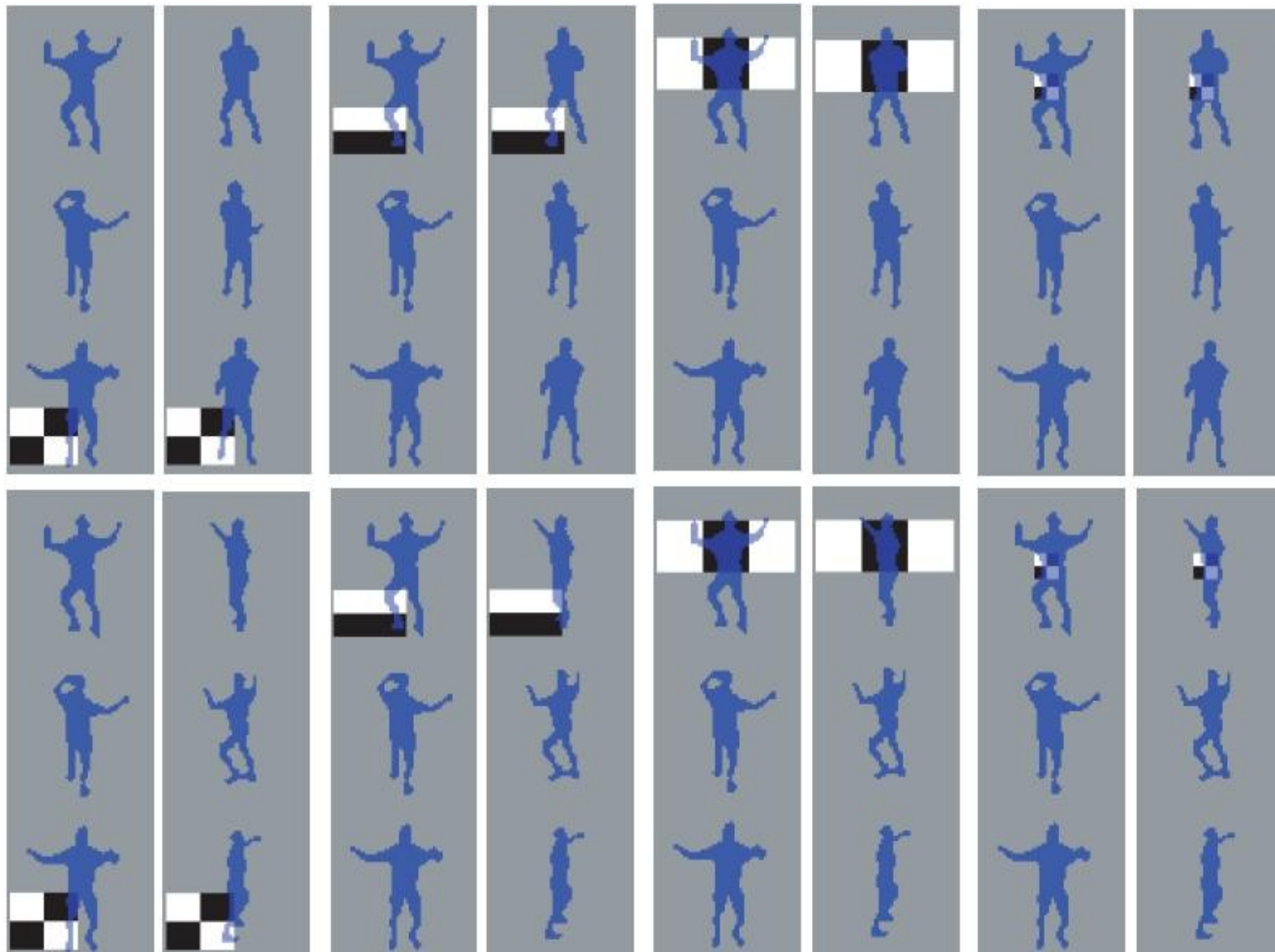
Examples collide, but not similar in parameter space

Examples similar in parameter space, but no collision

# Learning PSH functions

- Assemble some decision stumps for hash functions that have high accuracy on paired problem for database examples
- Set threshold so that #false positives + #false negatives minimal (obtained with two passes over training examples)

$$h_{\phi, T}(\mathbf{x}) = \begin{cases} +1 & \text{if } \phi(\mathbf{x}) \geq T, \\ -1 & \text{otherwise.} \end{cases}$$



# An Ensemble Classifier

Question: how do we combine many such classifiers into a single *strong* classifier?

# An Ensemble Classifier

Question: how do we combine many such classifiers into a single *strong* classifier?

Answer: AdaBoost

# LSH

LSH proceeds by randomly selecting  $k$  functions among those features chosen by AdaBoost, thus defining a  $k$ -bit hash function:

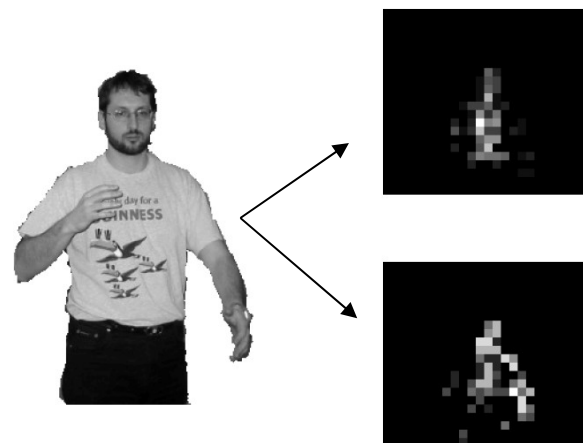
$$g(x) = [h_1(x), h_2(x), \dots, h_k(x)]$$

The entire database is indexed by a hash table with  $2^k$  buckets



# Pose estimation with PSH

- Describe images with multi-scale edge histograms(silhouette)
- Learn PSH functions
- Enter training examples into hash tables
- Query database with LSH
- Estimate pose from approximate NN using locally weighted regression



Animation

# Discussion

- Select the split position for KD-Tree in special domain.
- LSH eats much more space.
- Non-metric space in computer vision.
- Applying BoostMap to other distance functions.
- Applying BoostMap to other domains.
  - Natural Language Processing
  - Biological sequences.
- How to guess radius parameter for different problem
- Other Application of PSH
- Two spaces as input in PSH