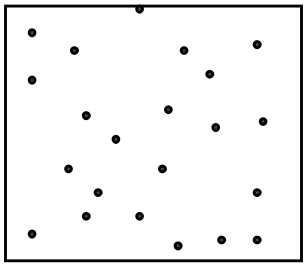
A Hashing Application: Finding Geometric Patterns

- Goal: determine if a particular pattern of points appears in a scene of many points
  - Finding a particular pattern of stars in a sky-photo
  - Finding a tumor in a catscan
  - Finding a pattern of atoms (an active site) on the surface of a protein
- Basic idea:
  - Try every possible way to place the pattern on the scene
  - The position that has the most *hits* is the best match
- *Hit* ≡ pattern point is "close enough" to a scene point

A Pattern Matching Example

- Does the *pattern* appear in the *scene*?
- Assume translation, but no rotation

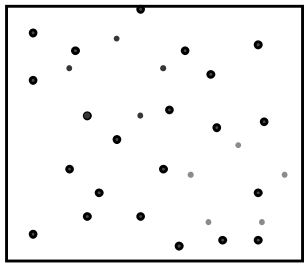
Pattern



Scene

Finding Geometric Patterns Efficiently

- Don't look at all pattern positions
- Just look at the ones with at least one hit
- For this example (translation only), we try each pattern point on each scene point
  - There are *mn* possibilities where *m* is the size of the pattern and *n* is the size of the scene



A Simple Pattern Matching Algorithm

```
                    for each pattern point p
                       for each scene point s
O(mn)                     Determine translation so that p falls on s;
O(mn)                     count = 0;
                          for each translated pattern point q
O(m²n²)                      Find the scene point r closest to q;
O(m²n)                       if (r is "close enough" to q) count++;
O(mn)                     Remember which p,s gives max count;
```

$T = O(m^2n^2)$

How do we detect a *hit*?

- Can't afford to check every point in the scene to see if it's close enough to a pattern point
- One solution is to *rasterize* the scene
  - Place a grid on the scene
  - Each grid-square is a bin
  - Points are "close enough" if they are in the same bin
- Disadvantage:
  - We've lost accuracy
  - We may miss some nearby points
- Advantage:
  - Instead of searching for arbitrary points, we search for occupied bins
  - The bins follow a regular pattern
  - The bins can be stored as an array **!**

An Improved Pattern Matching Algorithm

```
O(b)          Clear all the bins;
              for each scene point s
O(n)              Mark the bin for s as occupied;
              for each pattern point p
                  for each scene point s
O(mn)                 Determine translation so that p falls on s;
O(mn)                 count = 0;
                      for each translated pattern point q
O(m²n)                    if (q falls in an occupied bin) count++;
O(mn)                 Remember which p,s gives max count;
T = O(b+m²n)
```


Is it better?

- Total time is now $O(b+m^2n)$ instead of $O(m^2n^2)$
  - b is the number of bins
  - Note: m is small; n is *big*
- But we have lost accuracy and the array of bins can be expensive
  - Can be a very large array
  - Most bins are empty anyway
  - Note: could use a bit-array, but it's still large

Why check every pattern point?

- We could save time (a factor of *m*) by choosing just one pattern point *p* to translate to each scene point *s*, but…
- Some pattern points may be missing from the scene
  - Due to occlusion or errors in the scene data, for example
- The pattern may not align well with the bins

Improvement Using Hashing

Idea:

- Don't keep an array of bins
- Instead, place the *occupied* bins in a hash table
- Hash on the bin's array indices

Result:

- Initialization takes time O(n) instead of O(b)
- Still fast to test whether q falls in an occupied bin
  - A single test takes *expected* time O(1) instead of *worst-case* time O(1)
- Algorithm now runs in *expected* time $O(m^2n)$ instead of *worst-case* time $O(b+m^2n)$

What if we have to handle both translation and rotation?

- A single pattern point *p* and single scene point *s* aren't enough to determine a transformation
- Instead, we need 2 pattern points and 2 scene point
  - The first (p,s) pair gives the translation; the second gives the rotation
  - This leads to an algorithm that runs in expected time $O(m^3n^2)$
- If more-complicated transformations are allowed then more points are needed and the time bound grows
  - Example: For 3D translation and rotation, three points are needed and the time bound is $O(m^4n^3)$

Geometric Hashing

- Problem: A running time of $O(m^3n^2)$ may be impractical
- Idea: Use hashing to separate the runtime factors
- Preprocessing: [$O(m^3)$ time]

```
for each pair (p,q) of pattern points
   Transform all pattern points to reference frame (p,q)
      [i.e., reference frame with origin at p, + X-axis through q];
   for each pattern point r
      Determine bin containing the transformed r
      Place (p,q) in Hashtable, hashed by bin number of r
```


Geometric Hashing, Continued

- To test for a match: [$O(n^3)$ time]

```
for each pair (s,t) of scene points
   Find transformation with origin at s, + X-axis through t;
   for each transformed scene point u
   if u falls in an occupied pattern bin [tested using hashing]
      Increment the vote for that pattern transformation;
   Report a match if some transformation gets enough votes;
```

Geometric Hashing Advantages

- Advantages
  - Depending on problem $O(n^3)$ may be significantly better than $O(m^3n^2)$
  - If we want to search for multiple patterns, this affects the preprocessing time, but not the matching time
- Time is better in practice than big-O bound
  - Can often reject a pair (s,t) of scene points quickly because, for instance, they're too far apart to match any pair of pattern points

Usage of Geometric Hashing

- Applications to
  - Computer vision
  - Astronomy
  - Repair of CAD objects
  - Medical imaging
  - Fingerprint matching
  - Molecular docking

Reference:

- Lamdan & Wolfson, Geometric hashing: A general and efficient model-based recognition scheme, 1988; lots of papers since then