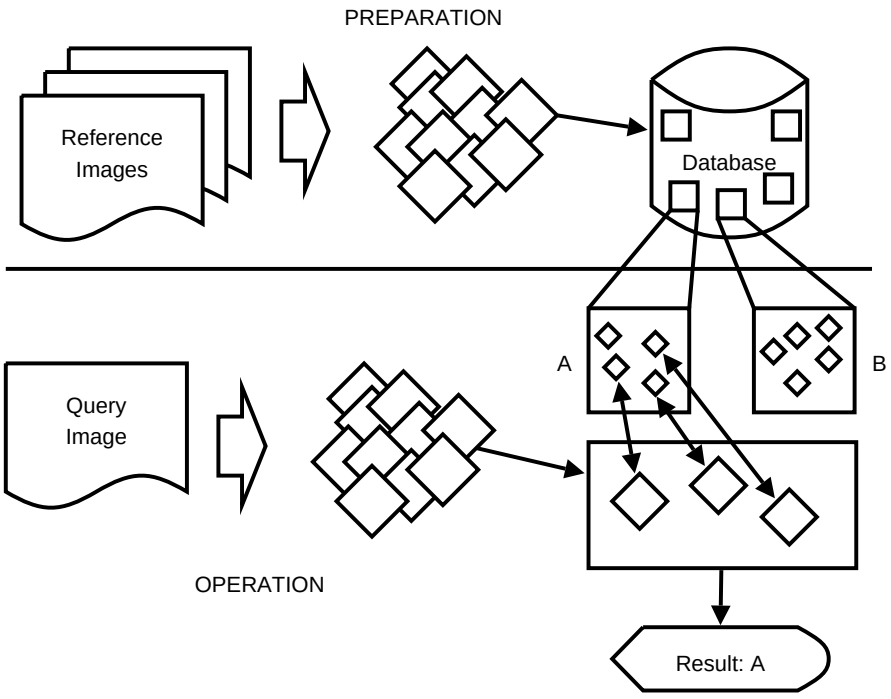
University of Cambridge
Engineering Part IB

Information Engineering Elective
Paper 8 Image Matching

Handout 4: Search



Roberto Cipolla and Matthew Johnson
May 2018

Data Structures

The overarching goal of this system is to accurately retrieve a source image from a database for each presented query image. We have discussed ways of describing images succinctly, using interest points and descriptors. Now we will discuss how to use these descriptions for search and retrieval.

In the introduction to the course, the analogy of text retrieval was used to introduce the way that we are going to achieve our goal of image retrieval. The interest points, or "words", and their descriptors, or "definitions", will be stored in a database. With each query image, the system will extract interest points and descriptors and then look in the database for those interest points and descriptors which are similar. The source image with the most matches is then returned.

While this system seems quite straightforward, there are two problems which aren't without a clear solution. The first is how to define "similar" when the entities we are comparing are multi-dimensional vectors. The second is how to store the source image descriptions such that they can be efficiently searched.

Similarity and Searching

For the purpose of defining similarity, we have a multitude of metrics to choose from. However, the most common used is the L_2 Norm, otherwise known as a Euclidean distance metric, measured using the following formula

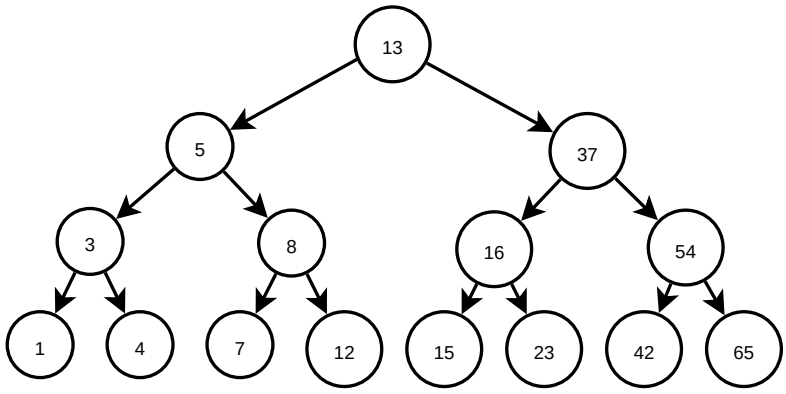
$$E(\vec{x}, \vec{y}) = \sqrt{\sum_d (\vec{x}_d - \vec{y}_d)^2}$$

While this does work well for comparing the individual vectors, without a data structure we must resort to searching through all the feature points in the database images for the best match of a query feature. This is called a **linear search**, and is prohibitively expensive to compute. Instead, we must find a way of storing the data to make searching faster.

A data structure organizes data such that it is more efficient to store, access and search. The simplest data structure is a list of items, such as an array of numbers. The most complex are almost impossible to visualize. We will be looking, however, at tree-based data structures, and how they can applied to tackling the problem of nearest neighbour retrieval.

Binary Search Trees

A **binary search tree** is a data structure which stores scalar values in such a way as to make them very efficient to search. It is a directed graph (meaning that it has nodes that are connected with one-way connections) with the stipulation that every node can only have one incoming edge and at most two out-going edges. What makes them effective search structures is the cunning way in which they are organized.



The left subtree contains nodes with values that are all less than the current node's value, and the right subtree values which are all greater than its value. Therefore, to search, you start at the root, and if your query value is less than the node value you go left, if greater than, you go right. If you get to the bottom and haven't found your number, it isn't in the tree.

This way, instead of performing an operation for all N items of data, you are only examining at $\log_2 N$ items, which is significantly better!

Binary Search Tree, cont.

The structure of a node is quite simple:

```
structure Node
{
    scalar value;
    Node leftLeaf;
    Node rightLeaf;
}
```

The algorithm to search the tree is a `while` loop:

```
procedure search(Node root, scalar query) returns boolean
{
    Node current = root;
    while(current is something)
    {
        if(current.value == query)
            return true;
        if(query < current.value)
            current = current.leftLeaf;
        else current = current.rightLeaf;
    }
    return false;
}
```

Binary Search Trees, cont.

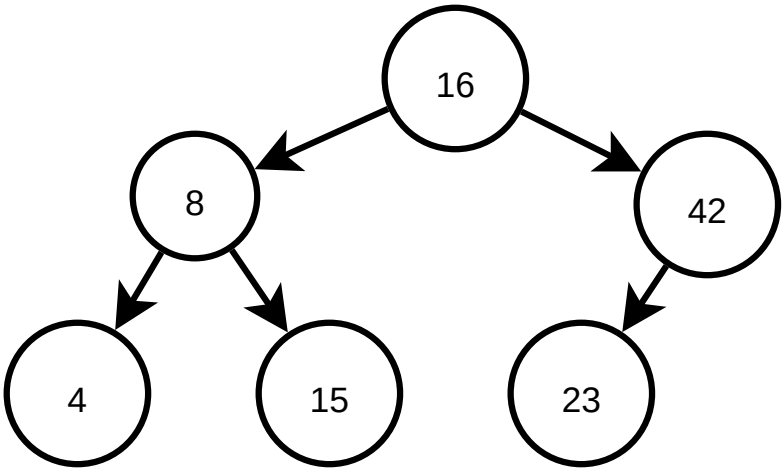
The procedure to add a new node to the tree is recursive:

```
procedure add(Node current, scalar value) returns Node
{
    if(current is nothing)
        return new Node(value, nothing, nothing);
    if(value == current.value)
        return current;
    if(value < current.value)
        current.leftLeaf =
            add(current.leftLeaf, value);
    else current.rightLeaf =
            add(current.rightLeaf, value);
    return current;
}
```

Let's practice building a binary search tree. Here is a list of numbers:

`{16, 42, 8, 4, 23, 15}`

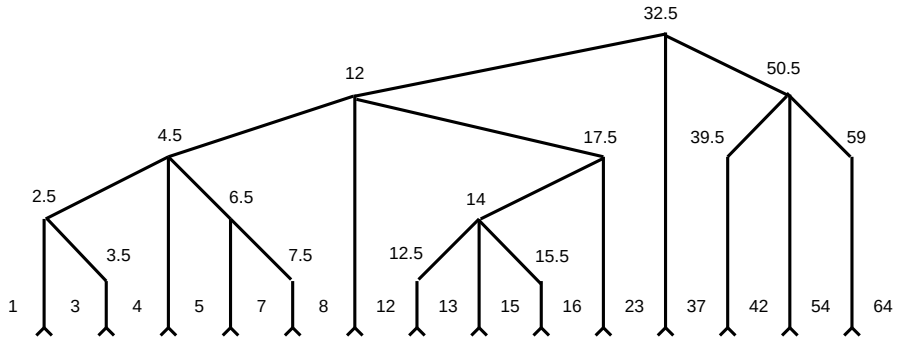
Build a binary tree below:



Metric Trees

Binary search trees are excellent, but in their current form we can only use them for scalar values. Also, constructing them so that they are optimal (balanced, with all subtrees at $\log N$ length) is quite difficult. We need a way to generalize the concept to vector values while retaining the property of speedy search.

The answer to this problem is the **metric tree**. The way in which a metric tree works is simple. At each level of the tree you find the two data values which are furthest away from each other (using some metric). A line is drawn between those points, and all points in the space are projected onto that line. All those between the midpoint and the "left" end go in the left subtree, and the rest go in the "right" subtree. The process is then repeated for the subtrees.

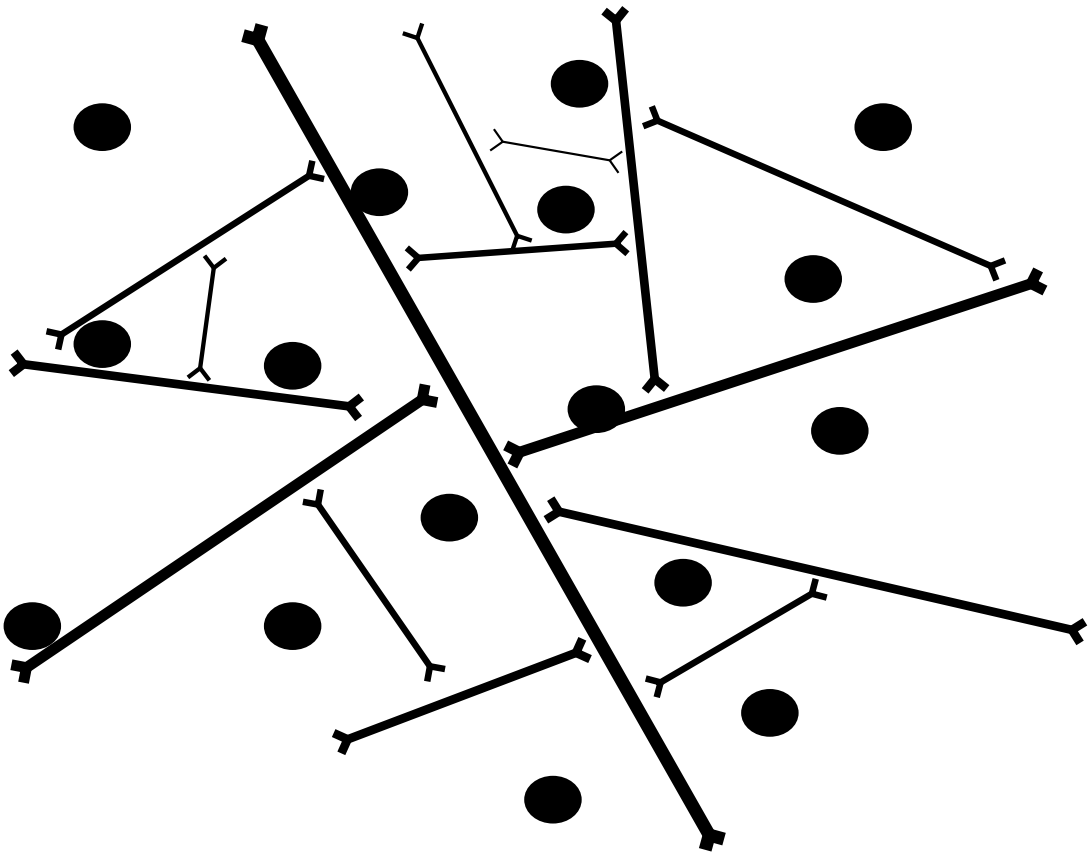


A metric tree in 1 dimension

In a metric tree, traversal decisions are made based on which side of a hyperplane the query value falls on, and as such they are scalable to any dimensionality.

Metric Trees

Here is an example of a two-dimensional metric tree. The thickness of the lines indicates what level of the tree they represent, with thicker lines being nearer the root. The circles are the data points.

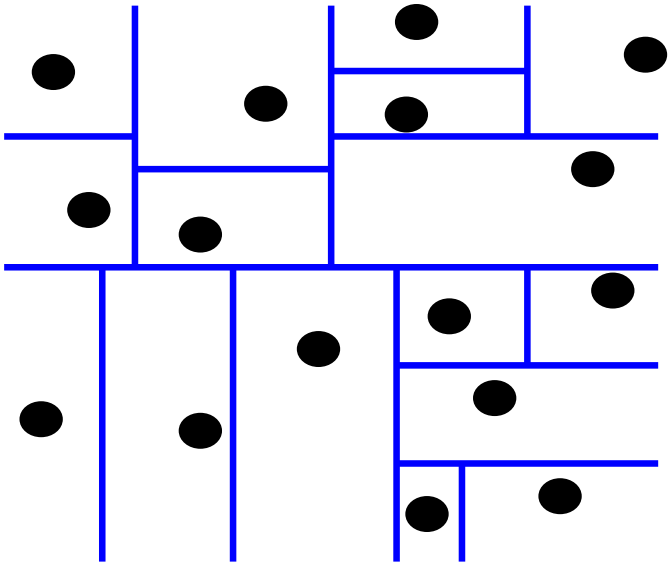


There are several problems with metric trees as a final solution to the need for a multi-dimensional search data structure. Namely, its dependence on projection makes it particularly slow to construct and algorithmically complex to model and traverse. Thankfully, there is a better data structure which has all the properties of a metric tree but without the hassle in construction and traversal.

K-Dimensional Trees

K-dimensional trees, or Kd trees as they are often referred to, are a kind of metric tree. The difference is that while the canonical metric tree uses a projection to a line between the two farthest points and then comparing values against the midpoint as its means of dividing the data into subtrees, the Kd tree uses a hyperplane with one dimensional constraint.

We begin by collecting all the points assigned to a subtree. We choose a dimension to split in, and a value to split at, and then split the data into two groups, with those which are less than the splitting value in the splitting dimension going to the left subtree and the rest to the right subtree.



K-Dimensional Trees, cont.

The node in a Kd tree is similar to that in a binary search tree:

```
structure KDNode
{
    hyperrectangle bounds;

    vector value;

    int splitDimension;
    scalar splitValue;

    KDNode leftLeaf;
    KDNode rightLeaf;
}
```

Traversing the tree follows the same mode as that used in binary search trees:

```
procedure search(KDNode root, vector query) returns boolean
{
    KDNode current = root;
    while(current is something)
    {
        if(current.value == query)
            return true;
        if(query[current.splitDimension] < current.splitValue)
            current = current.leftLeaf;
        else current = current.rightLeaf;
    }
    return false;
}
```

K-Dimensional Trees, cont.

Constructing a Kd tree is more complex, and cannot be accomplished piecemeal but must be done with all of the data which will be stored in the data structure. At each subtree, a method of choosing a splitting dimension and a splitting value must be applied. The two standard ways of choosing the dimension are to either (A) use a round-robin system or (B) to choose the dimension with the highest variance. Choosing the splitting value is typically done by either (A) using the mean value or (B) using the median value. Once a split has been determined, the points are split into their sub-tree groups and the process is repeated for each subtree.

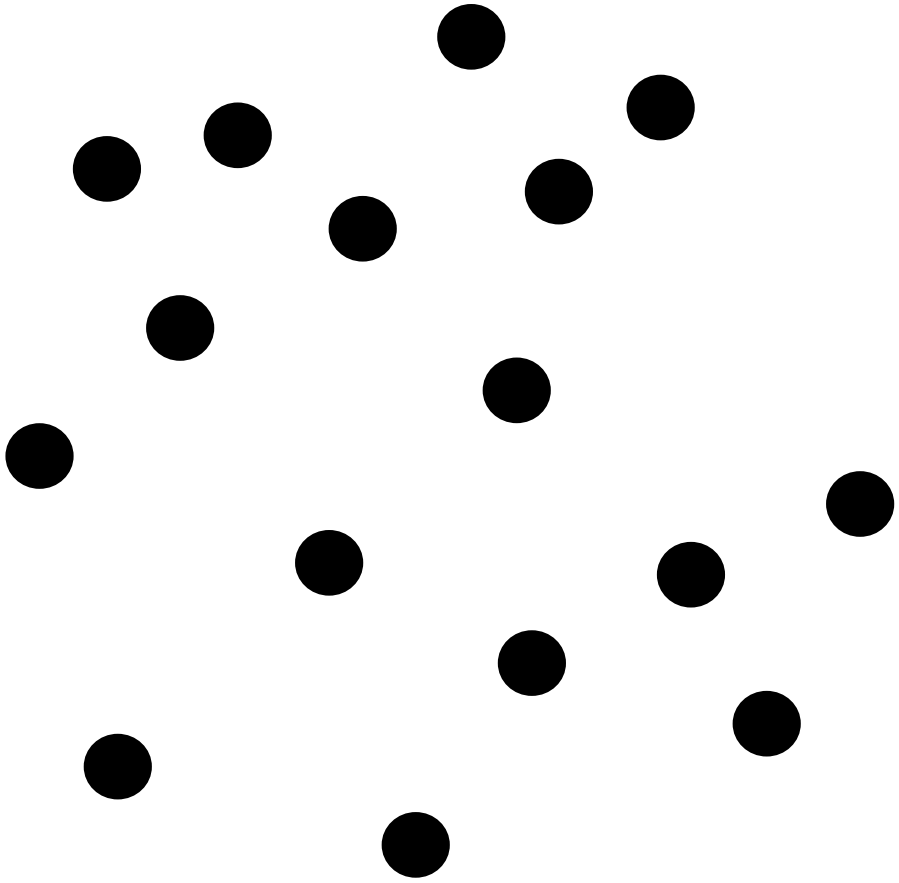
K-D Trees

```
procedure BuildKDTree(list<vector> data)
  returns KDNode
{
    if(data.Count == 1)
        return new KDNode(nothing, -1, 0, data[0],
          nothing, nothing);
    hyperrectangle bounds = constructHyperrectangle(data);
    int splitDimension = chooseSplitDimension(data);
    scalar splitValue = chooseSplitValue(data, splitDimension);
    list<vector> leftData = new list<vector>();
    list<vector> rightData = new list<vector>();
    foreach(vector dataPoint in data)
    {
        if(dataPoint[splitDimension] < splitValue)
            leftData.Add(dataPoint);
        else rightData.Add(dataPoint);
    }
    KDNode leftLeaf = BuildKDTree(leftData);
    KDNode rightLeaf = BuildKDTree(rightData);

    return new KDNode(bounds, splitDimension, splitValue,
      leftLeaf, rightLeaf);
}
```

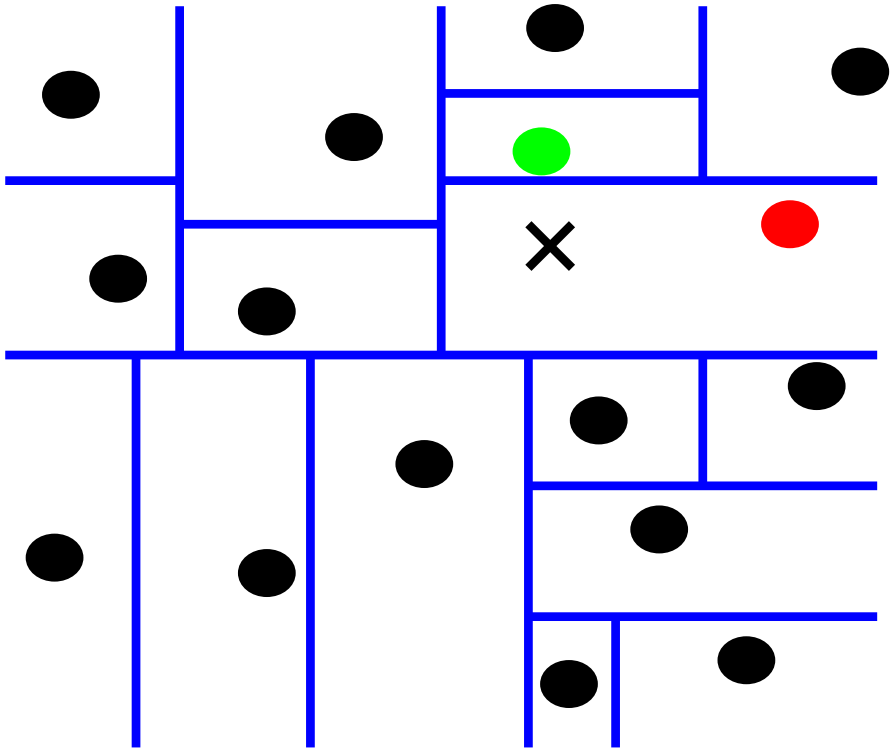
K-Dimensional Trees, cont.

Let's practice by building a Kd tree using the round-robin split dimension and the median point split value methods.



Nearest Neighbour Search

How do we use Kd trees for nearest neighbour searching? Well, the first intuition is to simply use the query value as our search query, descend the tree to a leaf, and then use the value at that leaf as our nearest neighbour. However, there is a problem with doing this.



The 'X' is our search point. The correct result is highlighted in green, but were we to use our naïve approach, the returned result is the point highlighted in red, which is obviously incorrect.

Branch and Bound Searching

In order to avoid the boundary problems we've just encountered, we must do a more thorough search of the Kd tree to be certain that the true nearest neighbour has been found. This is achieved using a technique called **branch and bound searching**. As we descend the tree, we keep track of what choices we have made. Then, after reaching the leaf, we use the distance between the leaf value and the query value as an upper **bound** on distance, and then search the **branches** we ignored if the distance from the query value to that branch's hyperrectangle is less than the current upper bound. Every time the algorithm reaches a leaf node, it updates the upper bound to further prune the search.

Branch and Bound Searching

```
procedure BranchAndBound(KDNode root, vector query) returns vector
{
    KDNode current = root;
    list<KDNode> branches = new list<KDNode>();
    vector nearestNeighbour;
    scalar bound;
    ...
    /// Store branches ///
    /// bound from nearestNeighbour ///
    ...
    foreach(KDNode branch in branches)
    {
        if(distance(branch.hyperrectangle, query) >= bound)
            continue;
        current = branch;
        ...
        /// Store additional branches ///
        /// update bound and nearestNeighbour ///
        ...
    }
    return nearestNeighbour;
}
```

Approximate Nearest Neighbour

The problem with branch and bound searching is that it can often prove exhaustive and inefficient. Indeed, as the number of dimensions increases, an effect known as the curse of dimensionality dictates that the number of branches which need to be searched will increase until the search is practically a linear search of the dataset. Given that descriptors like the SIFT descriptor have over one hundred dimensions (typically 128D), this is a serious consideration.

The solution is to find a way of approximating the search. In this case, we create a certainty/speed tradeoff, where we trade certainty for increased speed. The approximate nearest neighbour technique for Kd trees most commonly used is called **best bin first searching**. It is a modified version of branch and bound searching, in which the list of branches not taken is modelled as a priority queue, in which the determining value is the distance from the query to the hyperrectangle of the subtree. This way, the closest subtrees are searched first, increasing the likelihood that the nearer neighbours are found first and more branches are pruned. Also, once one of the branches is too far away, the search can stop, as all subsequent branches will be even farther away. Approximation is implemented by limiting the number of leaf nodes visited. Once the limit is reached, the algorithm stops.