Efficient Nearest Neighbour Search

3007/7059 Artificial Intelligence

School of Computer Science
The University of Adelaide

Nearest neighbour search

Given a set of points $\mathcal{X} = \{\mathbf{x}_i\}_{i=1}^{N}$ and a query point $\mathbf{y}$, both in M-dimensional space, find the nearest neighbour of $\mathbf{y}$ among $\mathcal{X}$.

The concept of "nearness" is based on a pre-defined distance metric, usually the Euclidean distance.

This is a fundamental Computer Science problem. It is also a very common routine in Artificial Intelligence applications.

A naive search algorithm scans each and every point in $\mathcal{X}$, incurring a computational cost of $\mathcal{O}(MN)$. This is not scalable to large N's (realistically N can be in the range of millions).

More sophisticated search algorithms have been devised to perform scalable nearest neighbour search. One of the earliest efforts is the Kd-tree (for "K-dimensional tree") method.

Constructing a Kd-tree

A Kd-tree is a binary tree, i.e., there are 2 child nodes per node.

Given a point set $\mathcal{X}$, the canonical way to build a Kd-tree is:

1. As the tree is descended, cycle through the dimensions along which to split the data points.
2. At a particular node, using only the dimension corresponding to the level in which the node is residing, find the median value of the points in the node.
3. Repeat until each node is a singleton node.

Note that the above steps lead to a balanced tree.

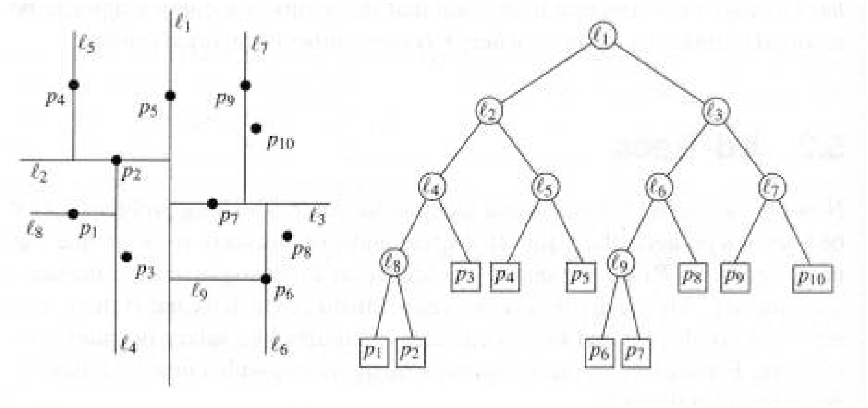
Variants differ in (1) how to choose the dimension to split at each node, and (2) how to choose the split value.

A recursive algorithm to construct Kd-trees

```
Function BuildKdTree($P$,$D$)
Require: A set of points $P$ of $M$ dimensions and current depth $D$.
if $P$ is empty then
    return null
else
    $d \leftarrow D \mod M$.
    $val \leftarrow$ Median value along dimension $d$ among points in $P$.
    Create new node $node$.
    $node.d \leftarrow d$.
    $node.val \leftarrow val$.
    $node.point \leftarrow$ Point at the median along dimension $d$.
    $node.left \leftarrow$ BuildKdTree(points in $P$ for which value at dimension $d$ is less than $val$, $D+1$).
    $node.right \leftarrow$ BuildKdTree(points in $P$ for which value at dimension $d$ is greater than $val$, $D+1$).
    Return $node$.
end if
```

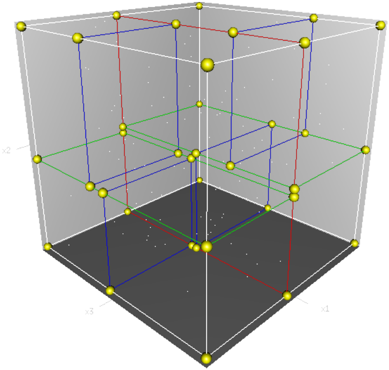
Example with data in 2D

In 2D a particular split defines a separating line (a 1-dimensional geometric entity).



Example with data in 3D

In 3D a particular split defines a separating plane (a 2-dimensional geometric entity).



More generally, for M dimensional data, a particular split defines a separating hyperplane of $(M - 1)$ dimensions.

Nearest neighbour search with Kd-trees

A Kd-tree partitions the M-dimensional space into a number of cells, each corresponding to a particular branch of the tree.

The speedup on nearest neighbour searches is enabled by ignoring whole branches whose lower bounds on the distance to the query point exceed the current best (smallest) distance.

Nearest neighbour search with Kd-trees (cont.)

Given a query point $\mathbf{y}$, a nearest neighbour search with a Kd-tree is accomplished by

1. Dropping $\mathbf{y}$ down the tree until a leaf node is reached.
2. Record the leaf node as the current best node.
3. Unwind the recursion of the tree, calculating at each node the lower bound of the distance to $\mathbf{y}$ of the other branch.
4. If the lower bound is larger than the current best distance, continue ascending the tree.
5. If the lower bound is not larger than the current best distance, descend the other branch until a leaf node is reached. Then update the best node if necessary.
6. Once the root node is reached, terminate the search and return the point in the best node as the nearest neighbour of $\mathbf{y}$.

A recursive algorithm to search Kd-trees

```
Function SearchKdTree($here$,$\mathbf{y}$,$best$)
Require: Current node $here$, query point $\mathbf{y}$, current best node $best$.
if $here$ is null then
    return $best$.
end if
if $best$ is null then
    $best \leftarrow here$.
end if
// Update the best node if necessary.
if distance($here$,$\mathbf{y}$) $<$ distance($best$,$\mathbf{y}$) then
    $best \leftarrow here$.
end if
// Determine on which side of the hyperplane the query point is located.
if $\mathbf{y}[here.d] < here.val$ then
    $child\_near \leftarrow here.left$, $child\_far \leftarrow here.right$.
else
    $child\_near \leftarrow here.right$, $child\_far \leftarrow here.left$.
end if
// Descend the near branch.
$best \leftarrow$ SearchKdTree($child\_near$,$\mathbf{y}$,$best$).
// Descend the far branch if lower bound does not exceed current best.
if distance_lb($here$,$\mathbf{y}$) $<$ distance($best$,$\mathbf{y}$) then
    $best \leftarrow$ SearchKdTree($child\_far$,$\mathbf{y}$,$best$).
end if
return $best$.
```

A recursive algorithm to search Kd-trees (cont.)

distance($node$,$\mathbf{y}$)

Computes the Euclidean distance of $\mathbf{y}$ to the point contained in node $node$, i.e.,

$$\text{distance}(node,\mathbf{y}) = \sqrt{\sum_{j=1}^{M}(node.point[j] - \mathbf{y}[j])^2}$$

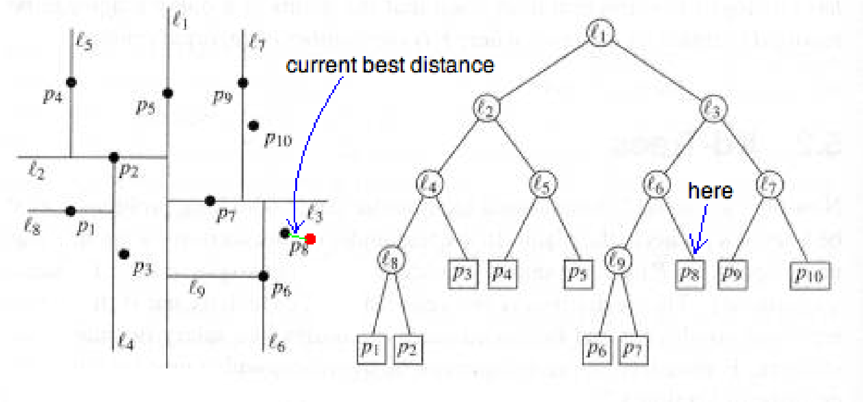
distance_lb($node$,$\mathbf{y}$)

Computes the lower bound of the Euclidean distances of $\mathbf{y}$ to points on the other branch. The lower bound is obtained as the distance along the dimension $node.d$ to the point contained in $node$, i.e.,

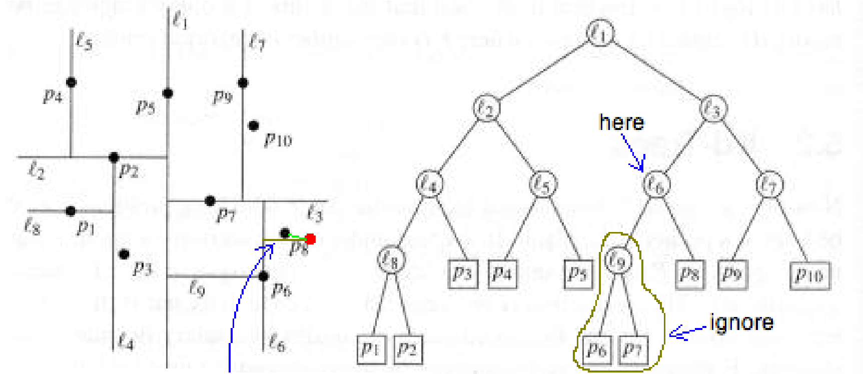
$$\text{distance\_lb}(node,\mathbf{y}) = \text{abs}(node.point[node.d] - \mathbf{y}[node.d])$$

Example 1

The query point is dropped down the tree until it reaches a leaf node.



Example 1 (cont.)

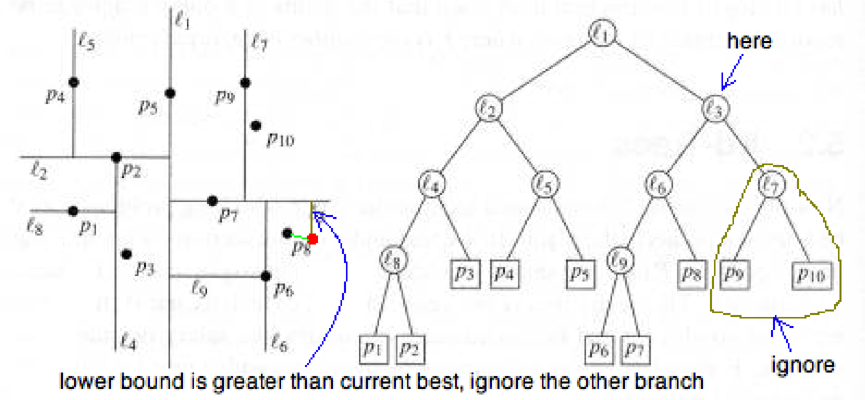


here

ignore

lower bound is greater than current best, ignore the other branch

Example 1 (cont.)



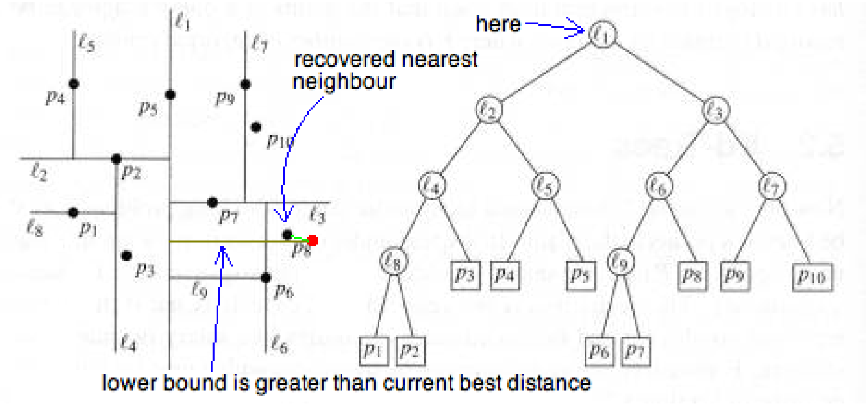
here

ignore

lower bound is greater than current best, ignore the other branch

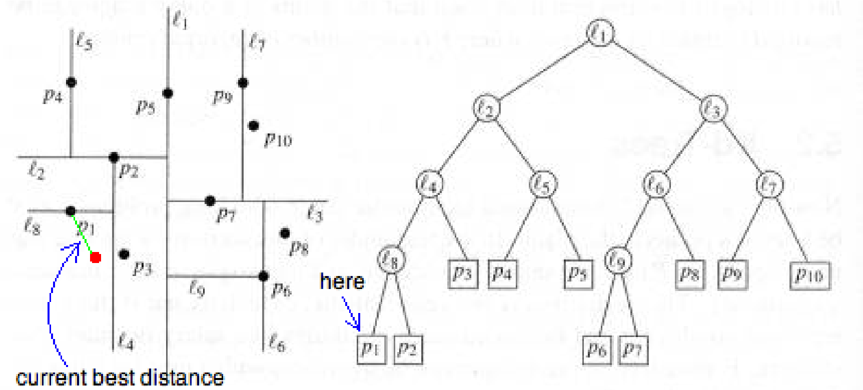
Example 1 (cont.)

Search terminates. Number of distinct full Euclidean distances computed is 4.



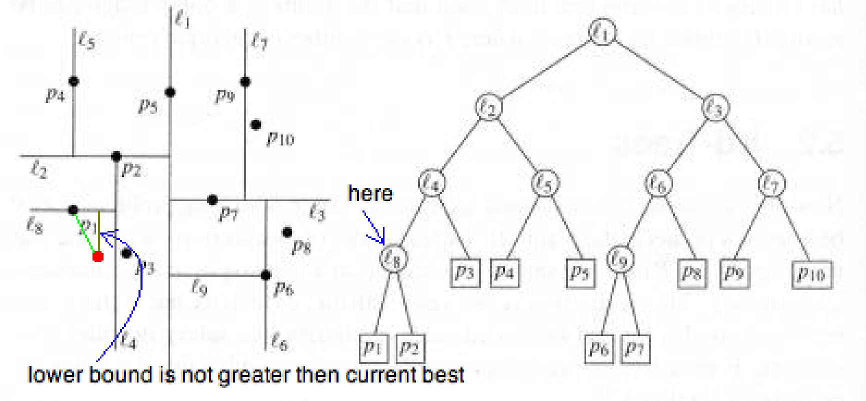
Example 2

The query point is dropped down the tree until it reaches a leaf node.



Example 2 (cont.)

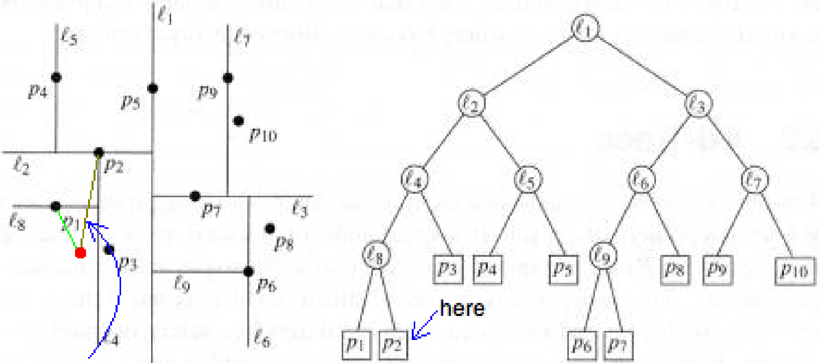
If a lower bound does not exceed the current best distance, we have to descend to the other branch.



here

lower bound is not greater then current best

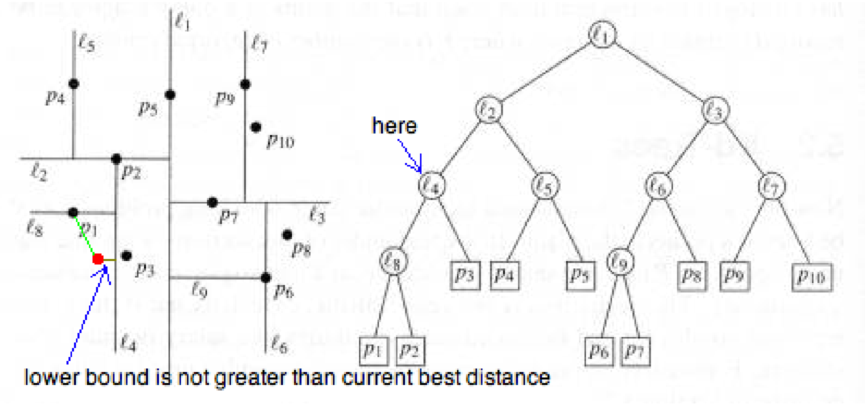
Example 2 (cont.)



distance is greater than current best distance, do not update

Example 2 (cont.)

If a lower bound does not exceed the current best distance, we have to descend to the other branch.

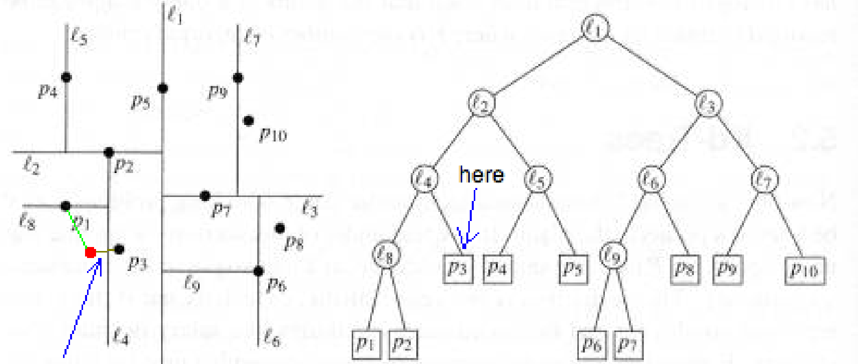


here

lower bound is not greater than current best distance

Example 2 (cont.)

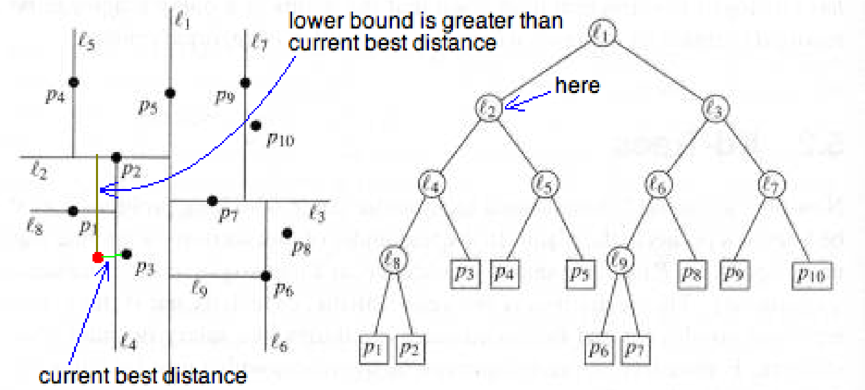
If a better solution is found, update the current best solution.



here

distance is smaller than current best distance, update

Example 2 (cont.)



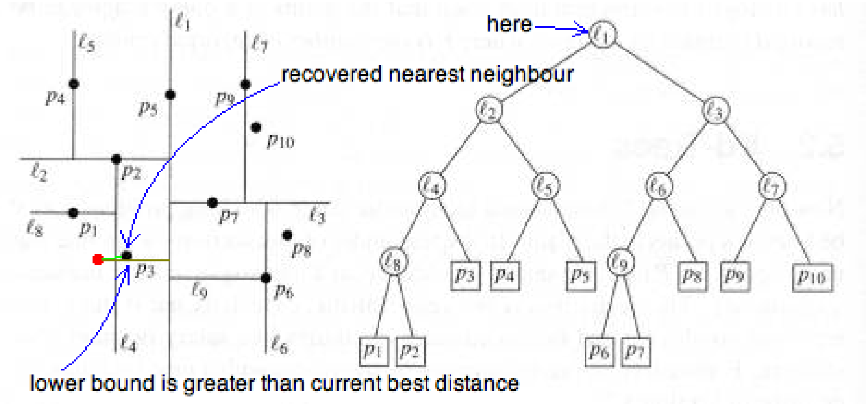
lower bound is greater than current best distance

here

current best distance

Example 2 (cont.)

Search terminates. Number of distinct full Euclidean distances computed is 7.



Theoretical behaviour

Building a balanced Kd-tree for N points takes $\mathcal{O}(N \log^2 N)$ if a $\mathcal{O}(N \log N)$ sort is used to compute the median at each level.

This can be improved if a more efficient sort procedure is used— However we are generally not concerned with this since building the tree can be done offline.

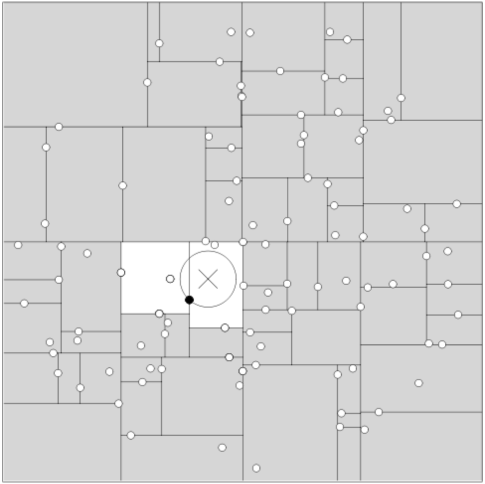
We are more concerned with the search performance— This depends on how many times we need to descend into the separate branches. Unfortunately this is hard to asymptotically analysed since the speed depends on how the data is distributed.

It is clear that at least $\mathcal{O}(\log N)$ computations of distances are necessary since we need to drop to at least one leaf node.

It is also clear that no more than N nodes are searched. In the worst case the algorithm reduces to a naive exhaustive search.

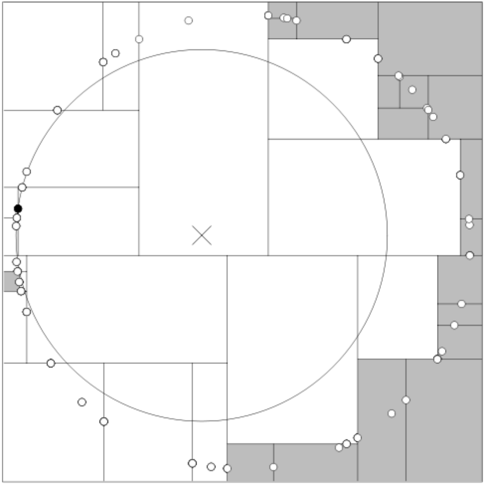
The distribution affects the search performance

A case where most of the branches are ignored (the grey boxes).



The distribution affects the search performance (cont.)

A case which forces almost all branches to be inspected.



Performance on high-dimensional data

The Kd-tree method for nearest neighbour search has a serious weakness— It is not efficient for high-dimensional data.

As a general rule Kd-trees can provide the highest gains in efficiency if the number of points N is much greater than 2^M, where M is the dimensionality of the data.

If the rule is not satisfied in most cases the Kd-tree search reduces to an exhaustive search.