Title: Ball tree

URL: https://en.wikipedia.org/wiki/Ball_tree

PageID: 31877832

Categories: Category:Machine learning, Category:Trees (data structures)

In computer science , a ball tree , balltree or metric tree , is a space partitioning data structure for organizing points in a multi-dimensional space. A ball tree partitions data points into a nested set of balls . The resulting data structure has characteristics that make it useful for a number of applications, most notably nearest neighbor search .

Informal description

A ball tree is a binary tree in which every node defines a D-dimensional ball containing a subset of the points to be searched. Each internal node of the tree partitions the data points into two disjoint sets which are associated with different balls. While the balls themselves may intersect, each point is assigned to one or the other ball in the partition according to its distance from the ball's center. Each leaf node in the tree defines a ball and enumerates all data points inside that ball.

Each node in the tree defines the smallest ball that contains all data points in its subtree. This gives rise to the useful property that, for a given test point t outside the ball, the distance to any point in a ball B in the tree is greater than or equal to the distance from t to the surface of the ball. Formally:

Where D B (t) {\displaystyle D^{B}(t)} is the minimum possible distance from any point in the ball B to some point t .

Ball-trees are related to the M-tree , but only support binary splits, whereas in the M-tree each level splits m {\displaystyle m} to 2 m {\displaystyle 2m} fold, thus leading to a shallower tree structure, therefore need fewer distance computations, which usually yields faster queries. Furthermore, M-trees can better be stored on disk , which is organized in pages . The M-tree also keeps the distances from the parent node precomputed to speed up queries.

Vantage-point trees are also similar, but they binary split into one ball, and the remaining data, instead of using two balls.

Construction

A number of ball tree construction algorithms are available. The goal of such an algorithm is to produce a tree that will efficiently support queries of the desired type (e.g. nearest-neighbor) in the average case. The specific criteria of an ideal tree will depend on the type of question being answered and the distribution of the underlying data. However, a generally applicable measure of an efficient tree is one that minimizes the total volume of its internal nodes. Given the varied distributions of real-world data sets, this is a difficult task, but there are several heuristics that partition the data well in practice. In general, there is a tradeoff between the cost of constructing a tree and the efficiency achieved by this metric.

This section briefly describes the simplest of these algorithms. A more in-depth discussion of five algorithms was given by Stephen Omohundro.

k -d construction algorithm

The simplest such procedure is termed the " k -d Construction Algorithm", by analogy with the process used to construct k -d trees . This is an offline algorithm , that is, an algorithm that operates on the entire data set at once. The tree is built top-down by recursively splitting the data points into two sets. Splits are chosen along the single dimension with the greatest spread of points, with the sets partitioned by the median value of all points along that dimension. Finding the split for each internal node requires linear time in the number of samples contained in that node, yielding an algorithm with time complexity O (n log n) {\displaystyle O(n\,\log \,n)} , where n is the number of data points.

Pseudocode

Nearest-neighbor search

An important application of ball trees is expediting nearest neighbor search queries, in which the objective is to find the k points in the tree that are closest to a given test point by some distance metric (e.g. Euclidean distance). A simple search algorithm, sometimes called KNS1, exploits the distance property of the ball tree. In particular, if the algorithm is searching the data structure with a test point t , and has already seen some point p that is closest to t among the points encountered so far, then any subtree whose ball is further from t than p can be ignored for the rest of the search.

Description

The ball tree nearest-neighbor algorithm examines nodes in depth-first order, starting at the root. During the search, the algorithm

maintains a max-first priority queue (often implemented with a heap), denoted Q here, of the k nearest points encountered so far. At each node B , it may perform one of three operations, before finally returning an updated version of the priority queue:

If the distance from the test point t to the current node B is greater than the furthest point in Q , ignore B and return Q .

If B is a leaf node, scan through every point enumerated in B and update the nearest-neighbor queue appropriately. Return the updated queue.

If B is an internal node, call the algorithm recursively on B' s two children, searching the child whose center is closer to t first. Return the queue after each of these calls has updated it in turn.

Performing the recursive search in the order described in point 3 above increases likelihood that the further child will be pruned

entirely during the search.

Pseudocode

Performance

In comparison with several other data structures, ball trees have been shown to perform fairly well on

the nearest-neighbor search problem, particularly as their number of dimensions grows. However, the best nearest-neighbor data structure for a given application will depend on the dimensionality, number of data points, and underlying structure of the data.

References

v

t

e

2–3

2–3–4

AA

(a,b)

AVL

B K-Dimensional

K-Dimensional

B+

B*

B x

Binary search Optimal Self-balancing

Optimal

Self-balancing

Dancing

HTree

Interval

Order statistic

Palindrome

(Left-leaning) Red–black

Scapegoat

Splay

T

Treap

UB

Weight-balanced

Binary

Binomial

Brodal

d -ary

Fibonacci

Leftist

Pairing

Skew binomial

Skew

van Emde Boas

Weak

Ctrie

C-trie (compressed ADT)

Hash

Radix

Suffix

Ternary search

X-fast

Y-fast

Ball

BK

BSP
Cartesian
Hilbert R
k -d (implicit k -d)
M
Metric
MVP
Octree
PH
Priority R
Quad
R
R+
R*
Segment
VP
X
Cover
Exponential
Fenwick
Finger
Fractal index
Fusion
Hash calendar
iDistance
K-ary
Left-child right-sibling
Link/cut
Log-structured merge
Merkle
PQ
Range
SPQR
Top