Cooper Union

Masters Thesis

Thesis Title

Author:
David Katz

Supervisor:
Dr. Carl Sable

A thesis submitted in fulfilment of the requirements
for the degree of Masters of Electrical Engineering

in the

Research Group Name
Department or School Name

February 2015

Declaration of Authorship

I, David Katz, declare that this thesis titled, 'Thesis Title' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

UNIVERSITY NAME (IN BLOCK CAPITALS)

Abstract

Faculty Name
Department or School Name

Masters of Electrical Engineering

Thesis Title

by David Katz

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

Abbreviations

LAH **L**ist **A**bbreviations **H**ere

Physical Constants

Speed of Light $c = 2.997\ 924\ 58 \times 10^8\ \mathrm{ms}^{-\mathrm{S}}$ (exact)

Symbols

a	distance	m
P	power	W (Js^{-1})
ω	angular frequency	rads^{-1}

For/Dedicated to/To my...

Chapter 1

Background

1.1 Nearest Neighbors Search

1.1.1 Overview

Nearest neighbors search aims to solve the problem of finding the closest points in a vector space. This type of search has a variety of applications in pattern recognition, information retrieval and computer vision.

There are a variety of different types of vector spaces such as boolean valued, integer valued and mixed however I will focus only on real-valued vector spaces. In these spaces every dimension can be expressed by a real number. In this type of vector space, closeness can be defined by a variety of different distance metrics. Some distance common distance metrics between two N-dimensional points x and y are shown in 1.1. In low dimensional spaces the euclidean distance is typically used, and will be the focus of future sections.

Distance Type	Distance Function
Euclidean	$\sqrt{\sum_{i=1}^{N}(x_i - y_i)^2)}$
Manhattan	$\sum_{i=1}^{N} \lvert x_i - y_i \rvert$
Chebyshev	$\max \lvert x_i - y_i \rvert$

TABLE 1.1: Distance Metrics

1.1.2 Basic Search Algorithm

The most basic algorithm for a nearest neighbor search is a linear check across every single element in a set. To do this one must compute the distance between a query point and every single point in a dataset, and return the point with the minimum distance. For a dataset with N points of dimensionality D, the complexity of this operation is O(N*D). For large datasets this linear time approach is not feasable, especially if many queries need to be performed.

1.1.3 K-nearest Neighbors

The basic linear search algorithm can easily be extended to support a query which returns the K-nearest neighbors rather than simply the closest. This change requires the use of a priority queue. A priority queue guarantees amortized O(Log(N)) insert and delete-max operations and constant time check-max [1]. The comparator for priority metric for points will be their distance to the query point. If a closer point is found than the furthest of the top K, the delete-max operation can be performed, and the new closer point can be added to the priority queue. The cost of this change is very low as likely K will be small. If the most recently checked point is not one of the top K, then only the constant time check-max operation needs to be performed. If the point is closer than one of the top K, the delete-max and insert operations must be performed as well.

Because both of those operations are logarithmic, the cost of updating the heap will at most be O(Log(K)). In practice however the priority queue is updated very rarely. Assuming the points are searched in random order, the probability that a point being processed will be one of the current top K encountered is relatively low.

1.2 Approximate Nearest Neighbors

Computation of the exact nearest neighbors via a linear search algorithm is extremely costly. One way to improve this performance is to create an index. The goal of an index is to increase the speed of a nearest neighbors query at the cost of additional preprocessing and memory on the original dataset. In low dimensional spaces, one common index is the k-d tree described in more detail in 1.3. While the k-d tree supports average case O(log(N)) queries in low dimensional spaces, no index has beeen found which is guaranteed to return the exact set of neighbors in linear time ADD CITATION.

Additionally, for many applications it is not important that the result set be perfectly accurate. It may be advantageous to return a set which isn't guaranteed to be exact in significantly less time. For these reasons, approximate nearest neighbors are often computed instead of exact nearest neighbors. Approximate nearest neighbor algorithms work by creating an index on a dataset, and applying that index to return a result set quickly. The most common index types are constructed out of trees, hash tables, or graphs.

1.3 k-d Trees

1.3.1 Overview

The k-d tree was originally developed as "a data structure storage of information to be retrieved by associative searches" [2]. k-d trees are efficient both in the speed of associative searches and in their storage requirements.

A k-d tree is a binary tree which stores points in a k dimension space. Each node contains a single k-dimensional point, a split dimension, and up to two children nodes. Each node represents a hyperplane which lies perpendicular to the split dimension, and passes through the stored point. The left subtree of a node contains all points which lie to the left of the hyperplane, while the right subtree represents all points which lie to the right of the hyperplane. Thus, each node partitions all below it into two half-spaces. Because only a single split dimension is used, each splitting hyperplane is axis-aligned.

1.3.2 Construction

The construction of a k-d tree is performed recursively with input parameters of a list of points. Pseudo code is shown below in 1.

Axis selection can be performed in multiple ways. The classical approach is to deterministically alternate between each dimension. Another approach known as spatial median splitting selects the the longest dimension present in the current pointList to split on [3]. The downside of this method is that a linear traversal is required to select the split dimension. Another popular approach is to randomly select the split dimension with an equal probability of selection each dimension. This approach is often applied when

```
function KDTREE(pointList)
    splitDim = selectAxis()

    medianPoint = selectMedian(pointList, splitDim)
    leftList = select points ≤ medianPoint along splitDim
    rightList = select points > medianPoint along splitDim

    treenode node = new treenode()
    node.splitDim = splitDim
    node.splitPoint = medianPoint
    node.leftChild = kdtree(leftList)
    node.rightChild = kdtree(rightList)

    return node
end function
```

Algorithm 1: Construct k-d tree

using multiple k-d trees as because of the additional randomness, trees are more likely to be different [4].

While a linear time algorithm for determining the median of an unordered set is possible [5] a heuristic approach is typically used to approximate the median. A common heuristic is to take the median of five randomly chosen elements, however many other methods can be used such as the triplet adjust method [6].

At the termination of of 1, the root of the k-d tree is returned, and each node contains exactly one point. The runtime of this algorithm is O(Nlog(N)) where N is the number of points in pointList. While the median can be approximated in constant time, partitioning pointList along that median is an O(N) operation. Since the k-d tree is a binary tree in which each node holds one point, assuming it is relatively balanced, its height is O(log(N)).

1.3.3 Nearest Neighbor Query

A simple algorithm exists to apply the k-d tree to a nearest neighbor query. This algorithm is guaranteed to find the single closest point to the search query. Pseudocode for this algorithm is shown in 0.

function SEARCHKDTREE(kdTreeNode, searchPoint, currBest)

dim = kdTreeNode.splitDim

searchDir = searchPoint[dim] < kdTreeNode.splitPoint[dim]

searchFirst = searchDir ? kdTreeNode.left : kdTreeNode.right

searchSecond = searchDir ? kdTreeNode.right : kdTreeNode.left

searchkdtree(searchFirst, searchPoint, currBest)

if *distance(kdTreeNode.splitPoint, searchPoint) < distance(currBest, splitPoint)*

then

| currBest = kdTreeNode.SplitPoint

end

end function

The first part of the algorithm recursively steps down the tree until a leaf is reached. At each node, a comparison on a single dimension is performed to determine which side of the splitting hyperplane the search point lies so that the search can continue in that half space. When a leaf is reached, the point stored in the leafnode is set as the current closest point. The algorithm then recursively walks back up the tree, and at each node computes the difference between the current node's point and the searchpoint. If this distance is smaller than that of the current best, the current node point becomes the current best.

The algorithm then determines whether a closer point could potentially exist in the second unsearched subtree. Because all hyperplanes are axis aligned, this computation

is very simple. The closest possible point in the halfspace represented by the second subtree will lie a distance of ϵ from the hyperplane, where ϵ is very small. The distance of this point is the absolute value of the difference between the search point and split point along the current split dimension. If this distance is larger than the current best point, then the algorithm does not need to check the second subtree, as there is no possible closer point in that halfspace. If this distance is smaller however then the algorithm will search down the second subtree following the exact same procedure as before, treating the second child as the root.

Because of this comparison however, the worst case run time of this algorithm is O(N), as if all comparisons fail, then the entirity of the tree will be searched. As the dimensionality of the tree becomes larger, this check is more likely to fail, and k-d trees diminish in effectiveness.

1.3.4 Approximate Nearest Neighbors Query

The k-d tree nearest neighbor search algorithm can be extended into an approximate nearest neighbors search with two small changes. The first change is rather than storing a single point as the current best, one can use a priority queue storing the top K points encountered. This change was decribed in more detail in 1.1.3.

The other required change required to compute Approximate Nearest Neighbors is that a limit on the number of points to search must be applied. The algorithm will follow the exact same steps as 0 however when the search limit is reached the algorithm terminates. This means that every possible node a closer point could lie in would not be searched. However, the nodes that do get searched are searched in a best first order. In other

words the algorithm will try to examine the regions of space which are closest to the search query first before expanding outward.

1.3.5 Modification

Chapter 2

Related Work

2.1 Approximate Nearest Neighbor Frameworks

Many ANN frameworks exist for fast computation. Since no exact nearest neighbors algorithms are faster than linear time in high dimensional spaces, a variety of different ANN algorithms must be applied. The Fast Library for Approximate Nearest Neighbors (FLANN) makes use of many of these algorithms including k-d trees, k-means trees, and locality sensitive hashing [7]. However, each of these types of indexes have different properties, and some may be suited to some datasets more than others. Thus, one difficult task this framework can perform is automatic selection of the algorithm type and parameters. This is done by estimating the overall cost of an index in terms of memory consumption, index generation time, and query speed to achieve a given accuracy. The user of the system can also put weights on each of these costs to raise the relative importance of one ore more of these factors. By benchmarking on a small subset of a dataset, the framework can effectively select the most efficient index type apply the simplex optimization algorithm [8] to tune its parameters.

Also of importance with FLANN and other frameworks is that they are optimized for real world performance. This means that that benchmarks are taken in terms of real time, and memory consumption is minimized to the largest extent possible.

Appendix A

Appendix Title Here

Write your Appendix content here.

Bibliography

[1] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, 1976.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[3] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 27, page 126. ACM, 2008.

[4] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

[5] Nimrod Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM (JACM)*, 31(1):114–127, 1984.

[6] Sebastiano Battiato, Domenico Cantone, Dario Catalano, Gianluca Cincotti, and Micha Hofri. An efficient algorithm for the approximate median selection problem. In *Algorithms and Complexity*, pages 226–238. Springer, 2000.

[7] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[8] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.