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 focus will be placed on real-valued vector spaces. In these spaces every dimension can be expressed by a real number. 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(ND). 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 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. 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 priority

queue 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.

The k-nearest neighbors (k-NN) result is extremely useful for pattern recognition on labeled datasets. For a classification task one common way to make a hard decision on a class is to use the class that appears most frequently in the top K. Thus, for datasets where test data is very similar to training data, this simple inference method can perform extremely well. k-NN can also be used for regression. Since the output is continuousin this case, the average of the results in the top K can be used.

k-nearest neighbors has a few limitations however. For one, it is an instance based learning technique. This means that it will only perform well when instances are similar to those from training and thus does not generalize well. Another issue is the computational cost of this method. While no training is required, each k-NN query takes linear times. Approximate nearest neighbors described in 1.3 attempts to address this. Another issue is that common distance metrics, such as euclidean distance, weight each dimension equally. Thus, in order to achieve reasonable results one must normalize all dimensions.

1.2 Fixed Radius Search

Another common search type is a fixed radius search. This type of search attempts to find all points within a distance R of a query point. The linear search algorithm can easily be adapted to this type of query. After computing the distance between the query point and each point in the training set, if this distance is found to be less then R that point can be added to the result set.

1.3 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 time and memory. In low dimensional spaces, one common index is the k-d tree described in more detail in 1.4. 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 [2].

However, 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. The most common index types for approximate nearest neighbor algorithms are constructed out of trees, hash tables, or graphs.

1.3.1 Tree Based Indexes

The main concept behind a tree based index is space paritioning. As such, these types of indexes tend to be extremely effective in low dimensionality settings, but do not scale as well to those of higher dimensionality. Generally, at the root of these indexes, the entire search space is present. As the tree splits, space is partitioned and only points which satisfy a split criteria will be present on each subtree. Thus, when searching to the leaf of these trees, one can find the space partition a query point lies in, and by recursively backtracking can gradually expand the search radius.

One of the most widely tree based indexes used are k-d trees described in detail in 1.4. The main advantage to k-d trees are that they are relatively fast to construct, can be easily modified, and have worst case linear space consumption.

K-means trees are another type of tree often used in practice [3]. These trees are constructed with a branching factor K. At each node, the k-means clustering algorithm is perfomed, separating remaining points into K clusters [4]. This branching continues until less than K points remain in a node, at which point the node becomes a leaf. To search a tree, one can move to a leaf by moving down the tree selecting the cluster with the closest mean. Each cluster's center is added to a priority queue. When a leaf is reached the algorithm continues the search at the closest center in the priority queue.

K-means trees are more expensive to construct than k-d trees, as the K-means algorithm is not guaranteed to converge quickly. Additionally, since all points are stored in the leafs and only cluster centers are stored at each node, the tree will be larger. K-means trees however tend to be more effective than k-d trees when high precision is required in the result set.

Quad trees are another algorithm commonly used for nearest neighbor searches [5]. In a two dimensional space, each point in a quad tree splits space into 4 different quadrants, similar to how the origin separates a standard x and y axis into four regions. The tree will be expanded this way and as such has a higher branching factor than k-d trees leading to lower depth. Quad trees can also be expanded into octrees for 3-D space and generalized into similar higher orders for even higher dimensionality [6]. Unfortunitely, in higher dimensional spaces of dimensionality D, each point splits space into 2^D regions. This often leads to many unused pointers since points will not likely lie in all of these regions. As such the memory cost of quad tree variants can become extremely large.

1.3.2 Hash Indexes

Many variations of hash indexes exist however the most widely used is locality-sensitive hashing (LSH). The goal of LSH is to use a variety of different hash functions to map similar points into the same buckets. Rather than using cryptographic hash functions which aim map entities into a bucket independent of their state, the hash functions used in LSH aim to match similar points into the same bucket with a high probability, and dissimilar points into the same bucket with a low probability. Formally, each hash function maps a D-dimensional vector v into a R buckets [7]. Many different types of hash functions can be used such as projection, lattice, and quantization based. Different types of hash functions have been studied and evaluated extensively [8].

Thus, to initialize an LSH index, one must pass every point through H hash functions, and store a key to each point within every bucket it falls into. A larger H leads to more information in the results however requires more processing time and memory consumption. Additionally, since all the information becomes compressed into these hash functions these types of indexes generalize well to higher dimensional spaces.

To query an LSH index one must pass a query point through all H hash functions, and search each bucket for collisions. The entries that most commonly collide have a smaller hamming distance in the new hash space, and will thus be treated as the most similar points.

While LSH scales extremely well to high dimensional spaces, one disadvantage is that its memory consumption tends to be much larger than that of tree based indexes in low dimensional spaces [7]. Another key disadvantage is that quality of the search queries cannot be changed, as this is dependent on H and R. In other words, LSH indexes have their maximum accuracy constrained during their construction, wheras tree based

indexes can have variable levels of accuracy on each query dependent on the number of nodes searched.

1.3.3 Graph Indexes

Graph based indexes tend to be the most expensive type to construct, however can support extremely fast queries. One common type is a k-nearest neighbor graph. In this type of graph, each node has exactly K edges, in which each node is connected to its k-nearest neighbors. A variety of different algorithms are available for constructing these types of graphs efficiently [9].

To perform a nearest neighbor query, a very common technique is a greedy traversal of the graph [10]. Given a query point, a randomly chosen node in the graph is chosen as the startpoint. Each neighbor is checked, and the next node traversed to is the one which is closest to the query point. The algorithm is terminated after a fixed number of moves, where a higher number of moves will have improved results. The K best nodes encountered are returned as the k-nearest neighbors. Often times random resets are incooperated to ensure that different parts of the graph are searched. Another common heuristic is to only search a subset of the the vertices at each node.

From experimental results these indexes tend to perform better than LSH and k-d trees [10]. However, one downside is that the offline construction of the graph is very expensive. Additionally, there is a large amount of randomness in the search algorithm, so there tends to be a large variance in the quality of the results obtained from queries with the same point.

1.4 k-d Trees

1.4.1 Overview

The k-d tree was originally developed as "a data structure storage of information to be retrieved by associative searches" [11]. 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 d dimension space. Each node contains a single d-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.

1.4.2 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 [12]. 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 selecting 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 [3].

While a linear time algorithm for determining the median of an unordered set is possible [13] 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 [14].

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)).

1.4.3 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 2.

```
function SEARCHKDTREE(kdTreeNode, searchPoint, currBest)
    If(leaf) kdTreeNode.splitPoint vs currBest; return;
    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,
searchPoint) then
|       currBest = kdTreeNode.SplitPoint
    end
    if HyperPlaneCheck(searchPoint, currBest, searchSecond) then
|       searchkdtree(searchSecond, searchPoint, currBest)
    end
end function
```

Algorithm 2: Nearest Neighbor Search k-d tree

The first part of the algorithm recursively steps down the tree until a leaf is reached. At each node, a comparison on the split dimension is performed to determine which side of the splitting hyperplane the search point lies so that the search can continue in the proper 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's distance, 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.

This algorithm can also be extended to perform a radius bounded search. Rather than checking the distance to the hyperplane compared to the furthest point in the top K one can check against a fixed radius R. Thus, this algorithm can also efficiently find all points within R of the query point.

1.4.4 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 2 however when the search limit is reached the algorithm terminates. This means that not every possible node a closer point could lie in would 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.

1.4.5 Modification

One key advantage of k-d trees is that they are very easy to modify. Inserting a node requires a traversal to a leaf node following the same procedure as described in 2. This search takes approximate O(log(N)) time if the tree is balanced. Once a leaf node is reached, a single comparison along one dimension needs to be performed in order determine whether the new node should be added as the left or right child. If random points are inserted the tree will remain relatively balanced as there will be an equal probability of being placed on each leaf. However heuristics can be used to help ensure that a tree remains balanced in a dyanmic environment [15].

Deletion on k-d trees can also be performed with relative ease. Again a downward traversal is performed until the target node is encountered. If the target node is a leaf, it can be removed trivially by removing the connection from its parent. If the target node is not a leaf, lazy deletion can be performed. The node will not be removed and will still serve to partition space, however it will never be compared against. If many nodes are lazy deleted however performance will slowly degrade, and reconstruction of the tree may be optimal.

1.5 Randomized k-d Tree Forests

With one k-d tree, one runs the risk of potentially very bad queries occuring when one of the earlier splitting hyperplanes lies close to the point of interest. By having a forest of

multiple k-d trees made from randomized split dimensions and splitpoints, the effects of this can be minimized. It has also been shown that k-d tree forests can lead to increased performance [16]. It is important to note that when searching multiple trees, the same heap must be used, and as such this object needs to properly be mutex locked. In doing so, as better results are found in each tree, more hyperplane checks will pass and less time will be wasted searching parts of the tree where better results do not lie. When searching trees sequentially rather than in parallel the performance was significantly weaker.

Another approached discussed to improve performance is the rotation of dimensions along different trees [16]. This allows the splitting hyperplanes to no longer be constrained to axis alignment, and can introduce a greater amount of variety in the forest. From experimental results [2] the optimal amount of trees was generally less than 20 though varied greatly on different datasets.

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 [2]. 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 index 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 and apply the simplex optimization algorithm [17] to optimize 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. For example, when possible bit array keys are used rather than pointers in order to save as much memory as 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] 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.

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

[4] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Applied statistics*, pages 100–108, 1979.

[5] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[6] Hanan Samet. An overview of quadtrees, octrees, and related hierarchical data structures. In *Theoretical Foundations of Computer Graphics and CAD*, pages 51–68. Springer, 1988.

[7] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the*

twentieth annual symposium on Computational geometry, pages 253–262. ACM, 2004.

[8] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11):1348–1358, 2010.

[9] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *Visualization and Computer Graphics, IEEE Transactions on*, 16 (4):599–608, 2010.

[10] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1312, 2011.

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

[12] 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.

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

[14] 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.

[15] Warren Hunt, William R Mark, and Gordon Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 81–88. IEEE, 2006.

[16] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

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