The Cooper Union

Albert Nerken School of Engineering

Masters Thesis

An Efficient Index for Computation of Approximate Nearest Neighbors with Query Specified Dimension Relevance Weights

Author:
David Katz

Advisor:
Dr. Carl Sable

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

in the

Department of Electrical Engineering

April 2015

Declaration of Authorship

I, David Katz, declare that this thesis titled, 'An Efficient Index for Computation of Approximate Nearest Neighbors with Query Specified Dimension Relevance Weights' 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: _____

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Carl Sable for the continuous support of my study and research, for his patience, motivation, enthusiasm, and immense knowledge.

THE COOPER UNION

Abstract

Department of Electrical Engineering

Masters of Engineering

An Efficient Index for Computation of Approximate Nearest Neighbors with Query Specified Dimension Relevance Weights

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.... 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... 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... 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... 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... 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... 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...

Contents

List of Figures

List of Tables

Table of Nomenclature

LSH	**L**ocality **S**ensitive **H**ashing
K-NN	**K**-**N**earest **N**eighbors
ANN	**A**pproximate **N**earest **N**eighbors
SMS	**S**patial **M**edian **S**plitting
SPM	**S**plit **P**robability **M**atching
WSMS	**W**eighted **S**patial **M**edian **S**plitting
MPDG	**M**ean **P**ercent **D**istance **G**ain
DDD	**D**eterministic **D**imension **D**epth
PDF	**P**robability **D**istribution **F**unction
CDF	**C**umulative **D**istribution **F**unction

Chapter 1

Introduction

Nearest neighbor search (NNS) on a vector space, described formally in Section 2.1.1 has a variety of applications including... The most basic algorithm for NNS, a linear search is described in Section 2.1.2. Unfortunitely, for large datasets upon which many NNS queries are being performed, a linear search will take too much time. Approximate nearest neighbor (ANN) algorithms address this concern by creating an index which allows computation of a result set in sublinear time as described in Section 2.3. While this result set is not guaranteed to be perfectly accurate, this limitation does not prevent ANN algorithms from being used for most NNS applications.

One challenge with modeling a data set in a vector space, is that the distance between two points, using one of the distance metrics described in Section 2.1.1 should represent the similarity of two items in the data set. As such, before performing an ANN query each dimension needs to be normalized proportional to its importance. For, different queries however blah...

Thus, our goal was to design an index which can efficiently compute ANNs on queries that specify the relevance of each dimension. To do so, we developed an system which

generates a large set of indexes optimized for different dimension weights, and efficiently selects the best set of indexes at query time. The particular index our system is based upon is the k-d tree. This index is described in detail in Section 2.4 and reasons for its selection in our system are described in Section 4.1.3.

To evaluate our system, we tested its performance against that of a conventional k-d tree, and a k-d optimized for the transformed vector space (which represents a theorestic best case performance of our heuristics described in Section 4.1.4 but cannot be constructed efficiently at query time). Our result quality metric described in Section 4.1.5 measures the amount the average distance of points in the result set increased compared to a linear search. Our system was then benchmarked on multiple datasets and with different types of ANN queries. Results are shown in Chapter 5

Chapter 2

Background

2.1 Nearest Neighbors Search

2.1.1 Overview

One common way to represent a collection of objects is as a set of points in a vector space with a fixed number of dimensions. Each object is represented as a vector, or point, in the vector space, and each dimension in the vector space represents a different feature of the object. The distance between two points in a vector space is analagous to the relatedness of two objects. Nearest neighbors search aims to find the closest points to a query point in a vector space. This type of search has a variety of applications in pattern recognition [1], information retrieval [2] and computer vision [3].

There are a variety of different types of vector spaces such as boolean valued, integer valued and mixed; however, we will focus on real-valued vector spaces. In these spaces, the value of every dimension in a vector can be expressed as a real number. Closeness can be defined by a variety of different distance metrics. Some common distance metrics

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 2.1: Distance Metrics

between two N-dimensional points, x and y, are shown in Table 2.1. In low dimensional spaces, the Euclidean distance is typically used [4], and will be the focus of future sections.

2.1.2 Basic Search Algorithm

The most basic algorithm for a nearest neighbor search performs a linear scan across every element in a set. Such a method computes the distance between a query point and every single point in a dataset, and returns 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.

2.1.3 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 [5]. Certain implementations of a priority queue allow even faster guarantees. For example, for binary heaps allow for average case constant time and worst case logarithmic insertion [6]. The priority

metric for points will be their distance to the query point. The first K points searched are automatically added to the priority queue. For future points searched, if a point is closer 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 ensuring that K elements always remain. 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 classification and regression 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 continuous in this case, the average of the results in the top K can be used.

The k-nearest neighbors has a few limitations, however [7]. 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 [8]. Another issue is the computational cost of this method. While no training is required, each k-NN query requires linear time with respect to the size of the dataset. Approximate nearest neighbors described in 2.3 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.

2.2 Fixed Radius Search

Another common search type is a fixed radius search [9]. 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.

2.3 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 Section 2.4. 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 [10].

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 [11].

2.3.1 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 [12]. As the tree splits, space is partitioned and only points which satisfy a split criteria will be present in 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, a process similar to the one described for k-d trees in Section 2.4 can gradually expand the search radius. K-d trees, described in detail in Section 2.4, are one of the most widely used tree based indexes. The main advantages of k-d trees are that they are relatively fast to construct, can be easily modified, and have worst case linear space consumption [12].

K-means trees are another type of tree often used in practice [11]. 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 [13]. 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 to the query point. As the search proceeds, each cluster's center is added to a priority queue, with its priority set as the distance from the query point. 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 [14]. 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 [15]. Unfortunately, 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.

2.3.2 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 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 one of R buckets [16]. Many different types of hash functions can be used, such as projection, lattice, and quantization-based hash functions. Different types of hash functions have been studied and evaluated extensively [17].

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 [16]. 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.

2.3.3 Graph Indexes

Graph based indexes tend to be the most expensive type to construct; however, they 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 [18].

To perform a nearest neighbor query, a very common technique is a greedy traversal of the graph [19]. 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 randomly selected subset of the connected nodes at each node.

From experimental results, these indexes tend to perform better than LSH and k-d trees [19]. 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.

2.4 k-d Trees

2.4.1 Overview

The k-d tree was originally developed as "a data structure for storage of information to be retrieved by associative searches" [20]. 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-dimensional 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 on one side of the hyperplane, while the right subtree represents all points which lie on the other side of the hyperplane. Thus, each node partitions all nodes below it into two half-spaces. Because only a single split dimension is used at each internal node, each splitting hyperplane is axis-aligned. This splitting procedure continues on each subtree until each node contains only one element. The procedure for axis selection on each split is described in section 2.4.2.

2.4.2 Construction

The construction of a k-d tree is performed recursively with input parameters of a list of points. Pseudo code is shown in Figure 2.1. Axis selection can be performed in multiple ways. The classical approach is to deterministically iterate between all N dimensions. Another approach, known as spatial median splitting, selects the the longest dimension present in the current pointList to split on [21]. 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 using multiple k-d trees; because of the additional randomness, trees are likely to be different [11]. While a linear time algorithm for determining the median of an unordered set is possible [22], 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 [23].

At the termination of the algorithm, the root of the k-d tree is returned, and each node contains exactly one point. The runtime of this algorithm has an average case O(Nlog(N)) running time where N is the number of points in pointList. While the median can be approximated in constant time, partitioning pointList along that median

```
function KDTREE(pointList)
    if pointList is empty then
        return null
    end if
    splitDim = selectAxis()

    medianPoint = selectMedian(pointList, splitDim)
    remove medianPoint from pointList
    initialize empty leftList, rightList
    for all points p in pointList do
        if p(splitDim) < medianPoint(splitDim) then
            leftList.Add(p)
        else if p(splitDim) > medianPoint(splitDim) then
            rightList.Add(p)
        else
            randomly add p to leftList or rightList with equal probability
        end if
    end for

    treenode node = new treenode()
    node.splitDim = splitDim
    node.splitPoint = medianPoint
    node.leftChild = kdtree(leftList)
    node.rightChild = kdtree(rightList)
    return node
end function
```

FIGURE 2.1: Pseudo code for Constructing a k-d tree

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)). Another key point is that points which match medianPoint along splitDim should be randomly assigned to a subtree. This will ensure that if many points have the same value along a given dimension all of these points will be distributed close to evenly between the two subtrees allowing the tree to remain balanced.

2.4.3 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 Figure 2.2. The inputs to this algorithm are the root of the tree, the query point, and dummy current best point located at infinity in each dimension.

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 than the current best 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

```
function SEARCHKDTREE(node, searchPoint, currBest)
    if node is a leaf then
        if dist(node.splitPoint, searchPoint) < dist(currBest, searchPoint) then
            currBest = node.SplitPoint
        end if
        return
    end if

    dim = node.splitDim
    Boolean searchDir = searchPoint[dim] < node.splitPoint[dim]
    searchFirst = searchDir ? node.left : node.right
    searchSecond = searchDir ? node.right : node.left
    searchkdtree(searchFirst, searchPoint, currBest)

    if dist(node.splitPoint, searchPoint) < dist(currBest, searchPoint) then
        currBest = node.SplitPoint
    end if
    if HyperPlaneCheck(searchPoint, currBest, searchSecond) then
        searchkdtree(searchSecond, searchPoint, currBest)
    end if
end function
```

FIGURE 2.2: Pseudo code to apply a nearest neighbor search using a k-d tree

by the second subtree could potentially lie within a distance of ϵ from the hyperplane, where ϵ is very small. The lower bound of the distance of the closest point, occuring when ϵ approaches zero, 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 running 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 be extended to support KNN with the same priority queue procedure described in Section 2.1.3. The check as to whether a closer point could potentially exist in the second subtree will then be performed against the furthest point in the top K. 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.

2.4.4 Approximate Nearest Neighbors Query

The k-d tree nearest neighbor search algorithm can be extended into an approximate nearest neighbors search. To do so, a limit on the number of points to search must be applied. The algorithm will follow the exact same steps as shown in Figure 2.2; however, when the search limit is reached the algorithm terminates. This means that not every possible node that might contain a closer point would necessarily be searched. However,

the nodes that do get searched are more likely to contain the closer points. In other words, the algorithm will examine the regions of space which are closest to the search query first before expanding outward.

2.4.5 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 the first part of Figure 2.2. This search takes approximate O(log(N)) time if the tree is balanced. Once a leaf node is reached, a single comparison along the split 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 on a balanced tree, the tree will remain relatively balanced as there will be an equal probability of placing points below each leaf. Heuristics can be used to help ensure that a tree remains balanced in a dyanmic environment [24].

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 not store any actual split point against which comparisons can be made. If many nodes are lazy deleted, performance will slowly degrade, and reconstruction of the tree may be advisable.

2.5 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. [25] has shown that k-d tree forests can lead to inproved performance. 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 if the search is to occur in parallel. 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 [25].

Another approach to improve performance is the rotation of dimensions along different trees [25]. This allows the splitting hyperplanes to no longer be constrained to axis alignment, and can introduce a greater amount of variety in the forest. Based on experimental results, the optimal amount of trees was generally less than 20, though the number varied greatly between different datasets [10].

Chapter 3

Related Work

3.1 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 more efficient ANN algorithms exist. 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 [10]. However, each of these types of indexes have different properties, and some may be better suited to certain datasets. 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 set weights to raise the relative importance of one or 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 to optimize its parameters [26].

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 in controlled test environment [10]. Additionally, physical memory consumption is another concern of these frameworks. To minimize physical memory consumption a variety of optimizations exist; for example, bit array keys are used rather than pointers.

Chapter 4

System Description

4.1 Overview

As mentioned in Section 2.1.3, before applying a nearest neighbors algorithm, one must normalize dimensions proportional to their relevance. Conventionally, if the relevance of a dimension or a set of dimensions were to be changed, one must perform a linear transformation on the every single point in the search space. When using an index type described in Section 2.3, this linear transformation requires a total reconstruction of the index for optimal nearest neighbors search performace, as the distance between all pairs of points in the index is now different.

The goal of this system is to support queries of dynamic dimension relevance in low dimensional spaces. Dynamic dimension relevance means that requester of a given query must provide both a search point, and the relevance of each dimension in the query. The system will then compute the ANNs using a modified Euclidean distance metric in which the distance in each dimension is weighted proportionately to the relevance. This metric is described formally in Section 4.1.2.

4.1.1 Normalization

Each dimension in the vector space representation of a dataset must be normalized to ensure that each dimension is initially weighted equally. The normalization scheme used performs a linear transform to set the mininum value of a dimension to zero and the maximum value to one. After finding the minimum and maximum of each dimension, one must normalize every dimension of every point following Equation 4.1.

$$d_{normalized} = \frac{d - d_{min}}{d_{max} - d_{min}} \tag{4.1}$$

This normalization technique is ideal for datasets which follow a relatively uniform distribution. However, the presence of outliers could greatly skew this normalization scheme. An alternative normalization strategy for these cases is to normalize the mean of each dimension to zero, and the standard deviation to one. This can be achieved by following the linear transformation outlined in Equation 4.2. This procedure is equivalent to finding the standard score or z-score of each dimension [27].

$$d_{normalized} = \frac{d - d_{mean}}{d_{stdev}} \tag{4.2}$$

4.1.2 Dimension Relevance

After normalization described in section 4.1.1, each dimension in the dataset is said to have equal relevance, and would have an equivalent contributribution to a standard Euclidean distance. As described in section 4.1 each query requires both a search point, and a dimension relevance vector (DRV). The DRV must contain the same number of dimensions as all points in a dataset. Each element in the DRV specifies a weight for a

single dimension. The DRV is then normalized to have a sum of one. Using the DRV, v, and two D-dimensional points x and y, the modified Euclidean distance metric is shown in Equation 4.3.

$$distance = \sqrt{\sum_{i=1}^{D}((x_i - y_i) \times v_i \times D)^2)} \tag{4.3}$$

For the case in which all dimensions are weighted equally, each element is $v_i = 1/D$. Thus, in this special case, the standard Euclidean distance is computed. This distance metric is also equivalent to transforming each dimension via multiplying by $v_i \times D$, and computing the standard Euclidean distance. By using this modified metric instead, this transformation does not need to be explicitly performed but is inherent in the distance calculation. It should also be noted that for computational purpose one can disclude the multiplication by D, as this can factor out of the distance metric as a constant. D was included such that this distance would better conceptually match the standard Euclidean distance.

4.1.3 Motivation for k-d Trees

We aim to tackle the the challenge of creating an ANN index which performs well when DRVs are specified at query time, and allows for queries of different quality requirements. We opted to base our index on k-d trees for a variety of different reasons. Graph indices, as described in Section 2.3.3 have a high offline construction cost, and compute the KNN of each node at construction time. This makes them poorly suited to adapt to a changing distance metric. Hash based indexes, described in Section 2.3.2 also have some downsides. Generally, the hash functions used act as quantizers and aren't tied directly to dimensions in a vector space. Additionally, the accuracy of hash based indexes is

tied to the number of hash functions, and the amount of buckets of each. As such this quality cannot be changed at query time.

Tree based indexes, described in Section2.3.1 tend to be the most memory efficient in low dimensional spaces. Additionally, the partitioning of tree indexes is inherentaly tied to the vector space representation. A specific advantage of k-d trees, described in 2.4 is that they are guaranteed to be linear in size, and splits can only occur orthogonal to a dimension.

4.1.4 Split Dimension Determination Heuristics

A common heuristic for determining the split dimension on k-d tree indexes is spatial median splitting (SMS) [21, 28]. SMS always selects the longest dimensions as the one to split on. The motivation for this technique is that each hyperrectangle should be as close to a cuboid as possible. In doing so, the check of whether or not a ball of radius R around a point intersects with an enclosing hyperplane is more likely to result in no intersection. When no intersection occurs, the k-d tree search algorithm described in Section 2.4.3, can eliminate a subtree without searching it. It is important to note that k-d trees can still function, even if the partitions are not square. However, because of the inability to prune subtrees as effectively, ANN searches will not be as efficient, and the quality of the result set under the constraint of a fixed number of searches will suffer. It is also important to note that on a vector space in which each dimension has been normalized, selecting a random dimension to split on with equal probability accomplishes a similar effect to SMS. On average, regions will tend to be close to cuboids. The advantage of this method is less offline computation in index construction and more variety in trees. This additional variety is advantageous when searching multiple trees [11].

When applying dimension weights via a DRV to a k-d tree search, the distance metric acts on a transformed space. As such, if the split dimension is selected with SMS or randomly with equal probability on each dimension, the regions would no longer be cuboids on average. Two different possible heuristics are applied to combat this. The first is split probability matching (SPM). The goal of this heuristic is to adjust the probability of splitting on each dimension to account for the fact that in the transformed space the dimensions are no longer normalized to the same weight. Therefore, by selecting split dimensions with probabilities equal to the weight of each dimension, the regions will tend to approach a cuboid shape in the transformed space. The second heuristic is weighted spatial median splitting (WSMS). WSMS is performed the same way as SMS, however the distance between the maximum and minimum of dimension is multiplied by the dimension weight from the DRV. Thus, this technique is equivalent to SMS on the transformed vector space.

4.1.5 Initial Tests

The dataset used for our tests included 100,000 8 dimensional points, in which the value for each dimension is pulled from U(0,1). 100 different randomly generated DRVs were generated via applying U(0,1) in each dimension and normalizing to a sum of one. 10 query points were generated for each DRV following the same procedure as the generation of the dataset. Thus, this test consists of 1000 queries on each index type. For this test, K was set to 50, while the max number of nodes searched was set to 500.

Initially two k-d trees were generated. One used SMS as its split dimension selection method, while the other selected randomly. Additionally, two forests of three trees were generated. In the first, each tree used SMS to selected split dimensions, while in the other split dimensions were selected randomly. For each of the generated DRVs, four

new indexes were created. A tree using WSMS, a tree using SPM, a forest of 3 trees using WSMS on each, and a forest of three trees using SPM on each.

A standard linear search was applied using the modified distance metric to determine the correct top K for each DRV and query point. An ANN query was then performed using the eight different indexes (four of which are the same across all query points, while four were initialized using the seed DRV for each of the 100 seed DRVs). The evaluation metric used is mean percent distance gain (MPDG). The average distance of each set of K elements was calculated. Then, using Equation 4.4 the average distance in each ANN set was compared to that of the linear search. A gain in this average distance represents a lower quality result.

$$MPDG = average\_distance_{index}/average\_distance_{linear} - 1 \tag{4.4}$$

Figure 4.1 displays the results of these initial tests. A single k-d tree using WSMS had the highest performance. Spatial median splitting and its variant tended to perform better than the randomized methods, and single trees performed better than forests. Figure 4.2, shows a similar test with a varying number of points searched. This figure however only includes the first results for a standard k-d tree using SMS and the variant using WSMS. From this plot it can be seen that WSMS has significantly better performance with a low number of points searched, and eventually both SMS and WSMS converge to very accurate results given enough searching. Another perspective which can be made from this plot is that under the constraint of a minimum quality, for any quality chosen, WSMS will return the result significantly faster. For example, if a MPDG of .15 is required, WSMS would reach that level of quality approximately three times faster. Thus, from these plots it is clear that both of these heuristics have the potential for

better results. Further evaluation is discussed in Chapter 5 to determine scenarios in which both of these heuristics are most effective.

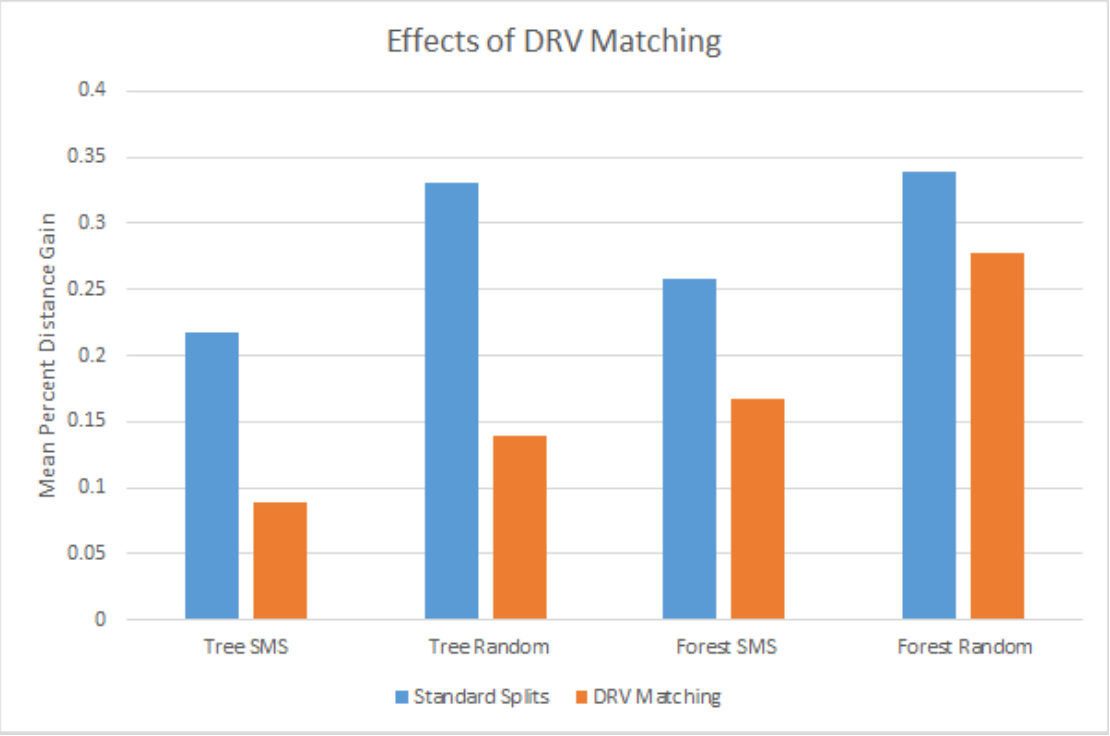

FIGURE 4.1: Initial tests of DRV matching heuristics

4.1.6 Tree Quality Metric

While SPM and WSMS can result in greatly improved ANN performance, these methods are not directly feasible in practice on a system which supports specifying a DRV at query time rather than during tree construction. As shown in Section 2.4.2 the cost of generating a k-d tree is Nlog(N), while the cost of a standard linear query is N. Thus, generating new trees to directly match the DRV of each query is not practical, as the tree construction cost is higher than a that of a linear search.

To work around this, we opted to construct a large set of k-d trees from a variety of different seed DRVs, whose selection process is detailed in Section 4.2.1. On receiving a query, our system can then select the tree or subset of trees whose seed DRVs best

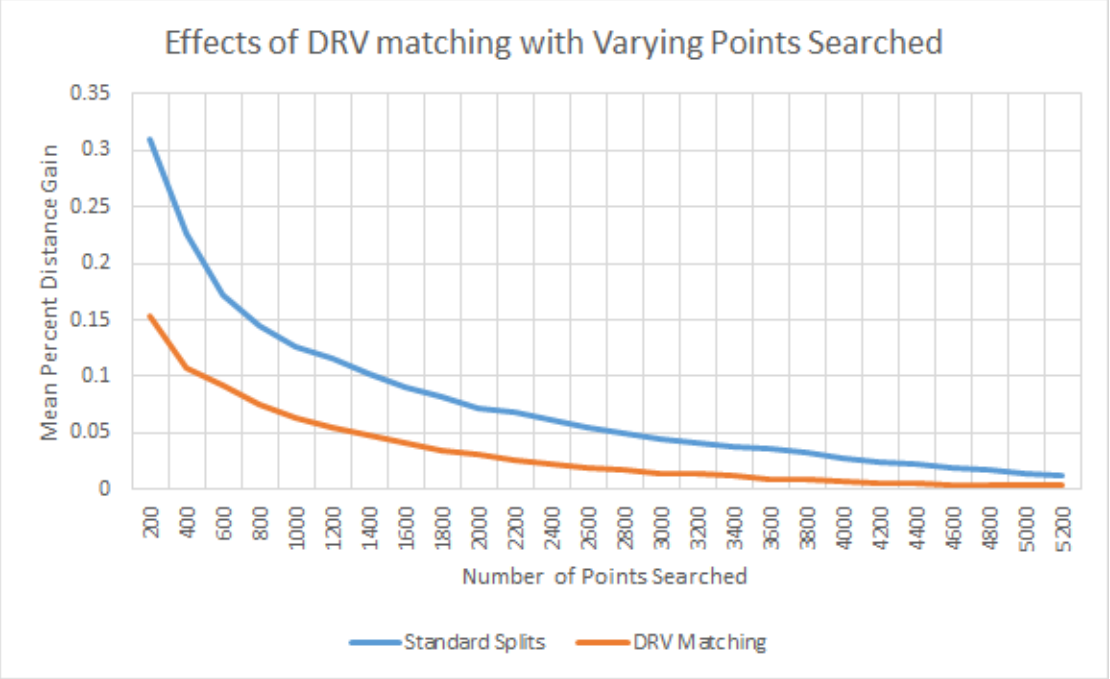

FIGURE 4.2: Effects of DRV matching in a single k-d tree using SMS/WSMS with varying number of points searched

match the query's DRV. In doing so, the set of trees searched will have been generated based on DRVs similar to that of the query, leading to improved performance for ANN queries.

The quality metric of a tree used is the modified Euclidean distance metric from equation 4.3. Rather than comparing two points however, the two entities compared are the query DRV and the seed DRV of a tree, both of which have been normalized to have a sum of one. The set of trees with the highest quality (smallest distance) can then be searched in parallel. Additionally, based on the result of this tree quality metric, the parallel search between multiple trees can be skewed towards the trees of highest quality, searching these trees more deeply.

There is of course a tradeoff between the size of the set of k-d trees, and system performance. More trees will result in high quality matches for more DRVs, and thus improved accuracy. However, each additional tree is linear in memory consumption, so the number

of these trees must be kept within reason. Additionally, each test of the tree quality metric is computationally equivalent to a comparison between two points. As such, the cost associated with computing the quality metric of each tree must be accounted for in performance benchmarks.

4.2 Detailed Implementation Overview

4.2.1 Index Construction

The initial dataset input to the system is an unordered set of N points containing D dimensions. The first step is to normalize all points in this set along each dimension using the scheme specified in Equation 4.1 or 4.2, per selection of the user. The user must also select their split dimension heuristic to be SPM or WSMS as described in Section 4.1.4. Both heuristics are evaluated in Chapter 5.

The user also needs specify two index size parameters: the deterministic dimension depth (DDD) which will be explained shortly, and the number of random trees. Both of these parameters impact the number of trees which will be generated. The use of the DDD allows the system to perform well on DRVs which put very heavy weight on a low number of dimensions, which is likely to be common in practice. To do so, a tree is generated with every possible subset of dimensions containing less than or equal to the DDD. For each of these subsets, the seed DRV has equal weight in each of these dimensions, and as such are equivalent to $1/D$. The number of trees needed to satisfy a DDD of R on a dataset with dimensionality D is shown in Equation 4.5. This number scales factorially with both D and R, so a very small R must be used if D is large to

ensure that the number of trees is reasonable. A large R however has the advantage of high quality tree matches on queries with R or less dimensions.

$$ntrees = \sum_{i=1}^{R} \binom{D}{i} \tag{4.5}$$

The second parameter specifies the number of random trees using all dimensions, generated in addition to those based on the DDD parameter. The number of random trees directly controls the number of trees generated with seed DRVS pulled from a uniform distribution. Each initial weight on each dimension is a number selected randomly from U(0,1) on each dimension, and the results are normalized to have a sum of one. A larger number of random trees will result in overall better matches on randomly selected DRVs at the cost of memory consumption. Other types of dimension split priors other than uniform could be used instead if additional information was known about the distribution of DRVs used. This possiblity is further considered in Chapter 7

A final tree is also added which is seeded from a uniform DRV to perform standard ANN queries. The total number of trees generated is thus the number of deterministic trees as specified by Equation 4.5 added with the number of random trees plus one. Each tree is generated following the algorithm in Section 2.4.2 with the modification that the split dimension is selected via SPM or WSMS rather than from a uniform distribution. Implementation details about weighted random number selection are described in Section 4.2.3. In our test environment, if the number of trees exceeds that which can be held in memory, they will be written to disk. In practice, this would not be an acceptable solution, as the cost of reading a tree from memory is linear, and disk reads are significantly slower than a linear seek performed in memory [29]. Our evaluation metric, however, considers only the quality of results against the number of nodes searched, so

this solution is acceptable for benchmarking purposes. Details about implementation in a live, distributed system will be discussed in Chapter 7.

The last step of construction after each tree is generated is to generate an index on the seed DRVs. Without an index, in order to determine the best tree(s) for each query a linear seek across each seed DRV would need to be performed. This would add an overhead of one comparison operation for each tree on every single ANN query. To avoid this overhead, the seed DRVS are cast into a vector space. After doing so, a k-d tree index is constructed on the seed DRVs following the standard procedure in section 2.4.2. At query time, this index can be applied to heuristically select the best set of trees via an ANN search against the DRV in sublinear time.

4.2.2 ANN Query

Each ANN query requires a query point, a DRV, the number of results to return (K), and the maximum number of nodes to search (S). Additional parameters shown in Table 4.1 can also be optionally supplied to tune the search. The first step of the algorithm, is to determine the top M trees to be used in the search. Using the k-d tree index on the seed DRVs described in Section 4.2.1, an ANN search is performed to determine these M trees and their respective quality scores against the DRV. By default M is set to 3, and $.4 \times total\_trees$ nodes are searched. However, both of these parameters can be adjusted. For example, by setting the percentage of trees to search to 1, a linear search is performed instead. Doing so is recommended for cases in which the dataset is large and the additional overhead from the linear search across all seed DRVs is negligible.

Since the initial quality metric is the distance between the query DRV and seed DRVs, a lower result of this metric represents a higher quality tree. To obtain a quality metric in

Optional Parameter	Default Value	Description
M	5	Maximum number of trees to search
SPS	.1× number of trees	Seed DRVs to search
TC	.5	Tree cutoff limit

TABLE 4.1: Optional Parameters

which a larger value represents high quality, $1/(distance+\epsilon)$ is used instead. Epsilon was set as 10^{-10} to avoid a division by zero error on perfect matches while having minimal effect on the metric. After M trees and their qualities are obtained, the qualities are normalized to have a sum of one.

Optionally, a tree pruning heuristic can be applied to remove trees of low quality in this set. This is advantageous for queries with a low number of dimensions in the query DRV, as some trees in the top M may be of significantly high quality than others. All trees with a normalized quality of less than $TC \times (1/M)$ are removed, where TC is the tree cutoff limit. By default TC is set to .5 but a value of 0 results in the heuristic not being applied, as no trees will pass this check. This heursitic has the additional benefit of reducing the set down to a single tree if a perfect or near perfect match is present. If a distance is extremely close to zero, the quality metric for that tree will be extremely high, and after normalization will push that of other trees towards zero. After trees are pruned, the qualities are re-normalized to a sum of one.

Since the cost of comparing the DRV against a set of split probabilities is equivalent to that of searching a point, the number of seed DRVs checked is subtracted from S for the next part of the query. The remaining searches are split among the selected trees proportional to the quality score of each tree. In our testbed, the parallel search of these trees was emulated by a master thread switching between the search of each tree. As mentioned in Section 2.5, the same priority queue was used during this search. Additionally, a hash table was used on the id (a unique integer) of each point. After a

point was checked in one of the trees, the id was added to the hash table. Upon checking the hash table in constant average time, the index can determine whether or not a point has been checked, and can avoid rechecking it if so.

The mechanisim for switching between between threads in our testbed is based on a single master. At the start of an ANN query, M trees and their qualities are known. A search is started on each tree in parallel; however, upon entering the critical area where a point would be searched the thread is put to sleep. Using the algorithm described in Section 4.2.3, a random tree is selected with probabilities proporitional to each tree's quality. This tree's thread is awakened, a single point is checked, and the thread is put back to sleep the next time it reaches the critical region. When the maximum number of searches is reached, the master forces the search thread on each tree to return. When all threads have returned, the top K will reside in the max priority queue shared between the threads. Thus, with this method, the effect of searching the trees in parallel is emulated. Implementation considerations for a true parallel or distributed implementation are discussed in Chapter 7.

4.2.3 Weighted Random Number Selection

At multiple points in our system, weighted random number selection along D dimensions is performed, and as such it was important for the implementation to be efficient. Given D weights normalized to a sum of one, these weights represent a probability distribution function (PDF) of each dimension being selected. By summing these PDFs, a cumulative distribution function (CDF) was generated containing the probability of a dimension of d or less being selected. To select a random dimension, a random double is drawn from a pseudorandom number generator uniformly distributed between zero and one. A binary search is performed to determine the largest entry in the CDF which is

greater than or equal to the selected number. Thus, this implementation requires O(D) additional memory, O(D) preprocessing, and O(log(D)) time after each random number generation.

Chapter 5

Results

5.1 Full System Tests

An initial test of the completed system was performed using parameters similar to those in Figure 4.1. The complete conditions were as follows: 100000 8 dimensional points were generated following the same procedure in Section 4.1.5. The conditions for the ANN searches performed used a K of 20, and limited the number of nodes searched to 500. This means that only about .5% of the nodes were searched. This test was performed using 80 different randomly generated DRVs, with 20 random points generated for each again following the procedure in Section 4.1.5. This means that each index was tested against 1600 different queries. The same MPDG error metric is also used, which measures the ratio of the average distance between the points in each query's result set and that of a linear search. For our system, the default parameters described in Section 4.2 were used.

From Figure 5.1, it is clear that the SMS based k-d trees tend to perform significantly better than those with random weights. Thus, even though the offline construction

cost of the random method is lower (due to not requiring a linear seek across each dimension), it is worth investing in WSMS, as this is is a one time cost and leads to significant improvement in result quality. In fact, a single k-d tree using SMS had higher overall performance than our system when using SPM. It is also important to note that the query DRVs used were pulled from a uniform distribution using the same method described in Section 4.1.5. As such, many of these DRVs were not drastically different from a standard uniform split. Thus, while our data structure lead to a performance improvement for both splitting heuristics, the baselike k-d tree still performed rather well.

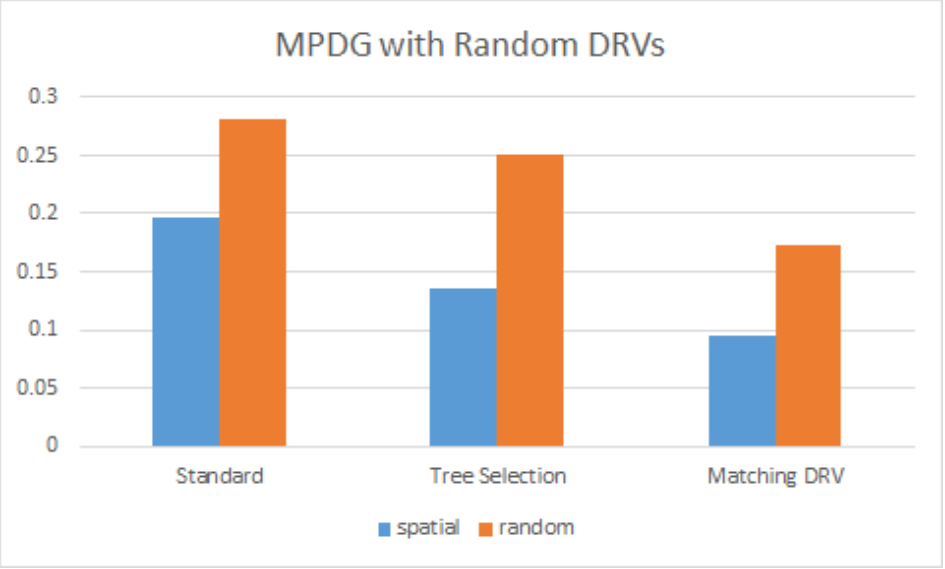

FIGURE 5.1: Full system test with random DRVs from uniform distribution

The real improvements came from extreme queries which only use a small subset of dimensions. The same parameters for our system were chosen with the exception of setting the DDD to 3. Following Equation 4.5, 56 deterministic trees were used in addition to the 100 random, and 1 generated from a uniform seed DRV. The method of selecting the query DRVs was changed to only include a small subset of dimensions. The method for doing so was to require at least a single dimension to be selected, and to

determine whether or not other dimensions should be selected with a bernoulli random variable. Thus, the distribution of the number of selected dimensions given a selection probability p and D dimensions is described by Binomial(D-1,p) + 1. This test was run with p = .125. After selecting a subset of dimensions, the weight of each dimension was drawn randomly from a uniform distribution and normalized to a sum of one. Of note is that when only one dimension is selected it will have a weight of 1 in the DRV, and it would be ideal to only split on that dimension. Fortunately, a seed DRV of all single dimension cases is included in our system, and as such there is guaranteed to be a tree which perfectly matches these DRVs, and only splits on that single dimension. In this case, the k-d tree performs equivalent to a binary search tree, and true nearest neighbors can be computed in O(log(N)) [30]. When more than one dimension is selected, while a perfect match tree likely won't exist (since weights are equal in seed DRVs), the two dimensional matching seed DRV will perform well if the query DRV is close to equal in the two dimensions, while the single dimension DRV will perform well when one dimension has a much larger weight than the other. Additionally, k-d trees tend to be very effective with a low number of dimensions.

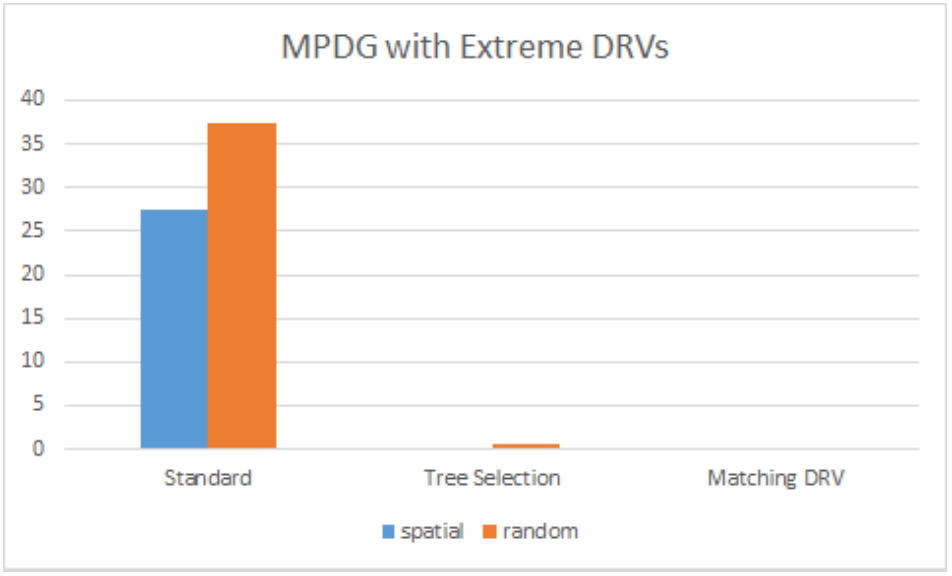

FIGURE 5.2: Full system test with DRVs containing a low number of dimensions

As shown in Figure 5.2, with extreme queries our system's performance was extremely close to that of a single tree with a matched seed DRV, while the standard k-d tree performance suffered greatly. The true performance of our system likely lies somewhere in the middle of the cases shown in Figures 5.1 and 5.2, as the true distribution of DRVs in seach queries is unknown. Also of note is that seed DRVs with all sets of three dimensions were present, and close to 90% of the selected DRVs had three or less dimensions. However, even when four to five dimensions are selected performance is still expected to be strong, as the weights on a one or more of the dimensions will likely be low.

Because of the significantly stronger performance of WSMS compared to SPM, future tests are performed only using the WSMS heuristic. The key advantage to SPM is that all split dimensions are selected in constant time, wheras those in WSMS require a linear seek across all dimensions. However, this cost is only associated with offline construction of the trees which is only performed once. Since our benchmark attempts to optimize result quality on queries ignoring offline computation, WSMS is a superior heuristic according to our MPDG metric. However, in 6.2, we discuss that on a live system it may be necessary to reconstruct trees at times and as such, when performance tuning, one should consider the merits of a heuristic with lower tree construction cost. Additionally, for future tests three different indexes were generated. The first is a standard k-d tree for baseline performance. The second is our system, using the standard parameters described above. The final index generated is a single k-d tree using WSMS with a seed DRV matching that of the query DRV. This case represents hypothetical best case performance with our heuristics.

5.2 Change of Dimensionality

One important factor to test is how our system performs on data sets with different dimensionalities. The data sets were generated following the procedure from 4.1.5; however, the process was replicated to generate data sets with different number of diemnsions.

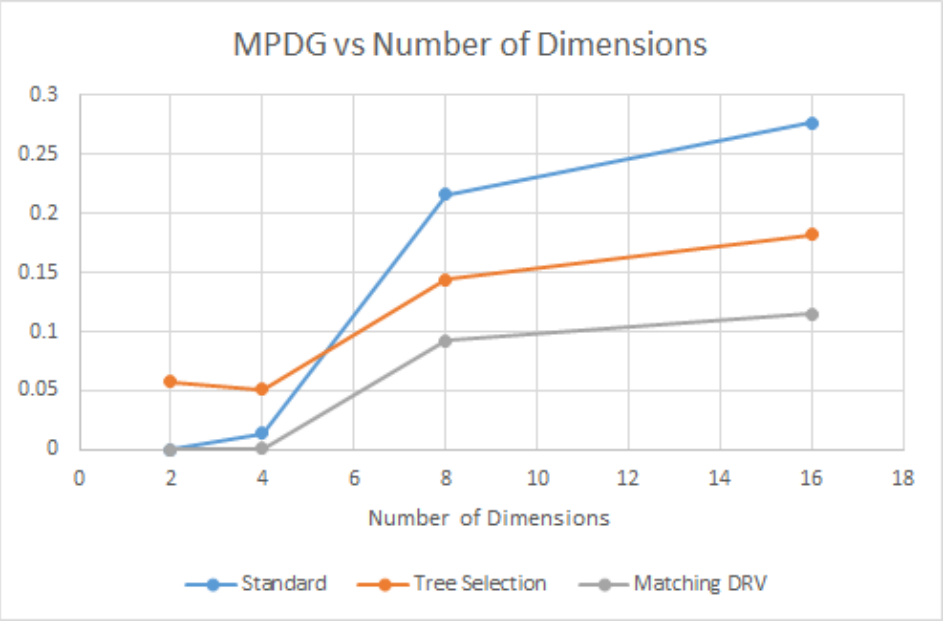

FIGURE 5.3: Full system test with data set of varying dimensions

The results for this test are shown in Figure 5.3. For each index, the MPDG increases with dimensionality. This is to be expected, as performance for k-d trees begins to degrade at higher dimensionalities. Of note is that our system actually performed worse than a standard k-d tree for very low dimensionality. This is likely because some searches were used to select the best trees. Additionally, the k-d tree has the advantage of searching in a single tree, while searches in our index were split between multiple, and as seen in Figure 5.3 the performance of forests tends to be slightly worse than that of a single tree. With a higher number of dimensions our system performs better than

k-d trees, as the value of selecting trees with better seed DRVs is higher with more dimensions.

5.3 Size of Dataset

We also tested the performance of our system on data sets of varying size. Again the same procedure from 4.1.5 was followed, to generate datasets of dimensionality 8 with varying sizes.

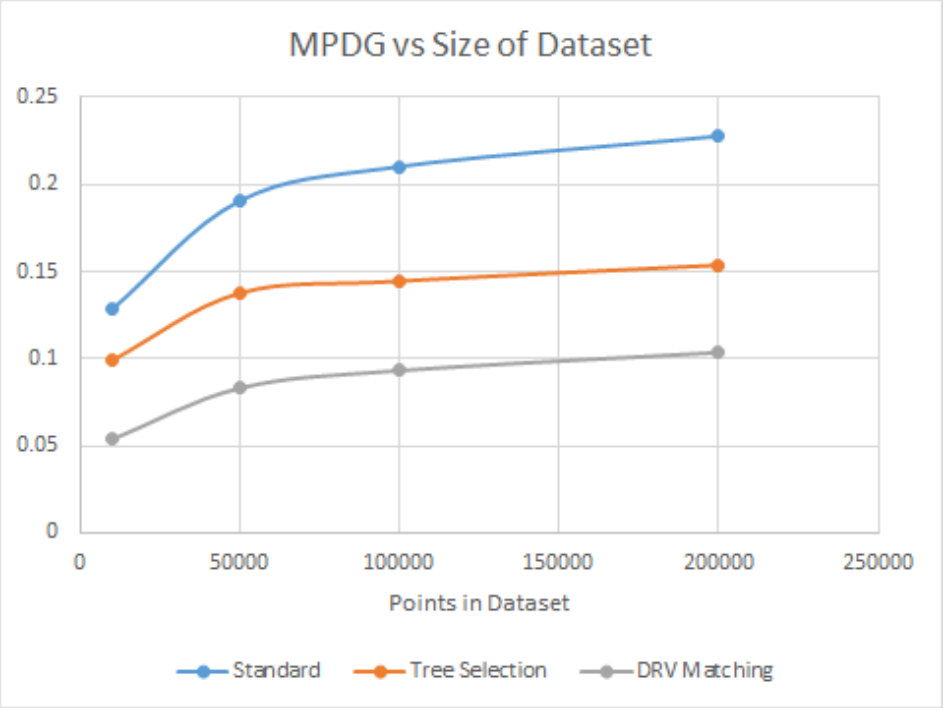

FIGURE 5.4: Full system test with data set of varying size

The results of this test are shown in Figure 5.4. For each of the indexes, the MPDG increases with the size of the dataset. This is expected, searching a fixed number of nodes on a larger dataset means that a lower percentage of nodes are searched. Of note is that the standard k-d tree index's performance decreased at a faster rate than that of our index and the tree with a matched seed DRV. Thus, the benefits of our system are larger for bigger datasets.

5.4 Number of Trees

The number of trees in our system is expected to have a direct impact on in its performance. We tested our system against the same dataset described in 4.1.5, generating our index with a varying number of random trees.

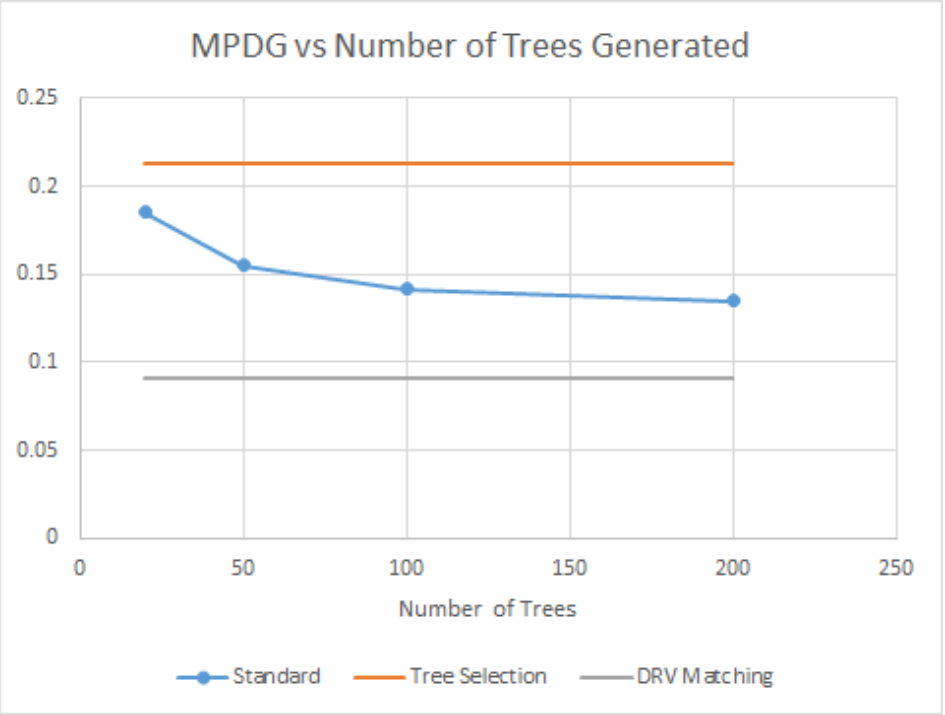

FIGURE 5.5: Full system test with varying number of random trees in our system

The results are shown in Figure 5.5. The orange line represents the performance of the standard k-d tree, while the grey line represents the performance of a tree with a matched seed DRV. With a larger number of trees, the overall quality of the selected trees will increase, since with more trees there are more chances for high quality matches. However, this effect is diminishing. As the number of trees increases, the performance gain from doing so drops.

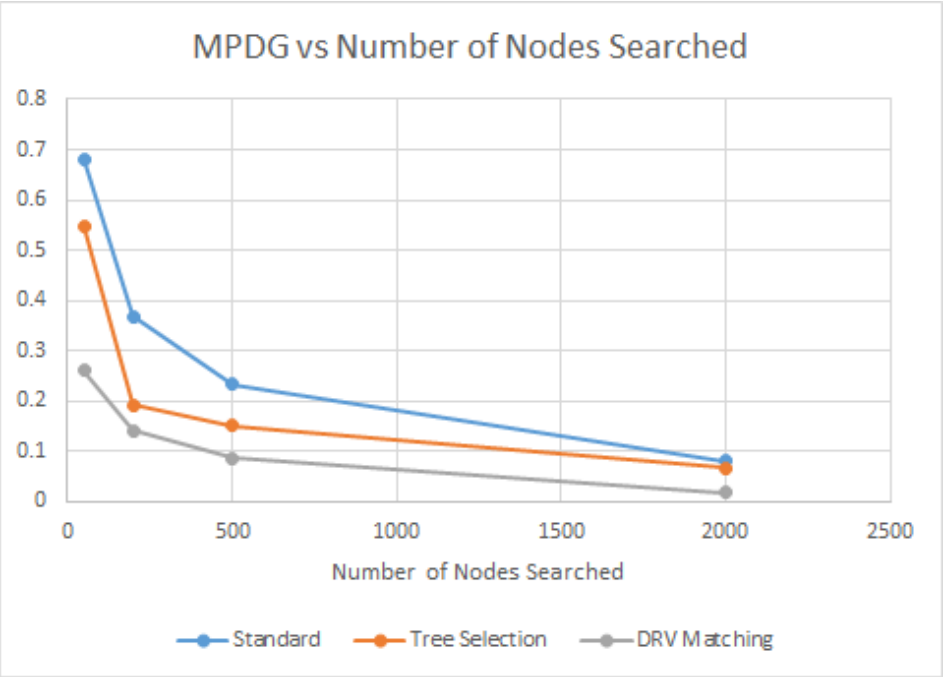

FIGURE 5.6: Full system test with varying number of nodes searched

5.5 Nodes Searched

We also considered the performance of our system for varying numbers of nodes searched. The results for this test are shown in Figure 5.6. As expected, for each index, the MPDG decreases with an increase in the number of nodes searched since a larger percentage of nodes are searched. This effect is dimishing however. Initially, an increase in the number of nodes searched leads to a large decrease in MPDG, wheras a similar increase when a large number of nodes are already searched has little effect. Also of importance is that for a very small number of nodes searched our system's performance is close to that of a standard k-d tree. This is because a high proportion of the alloted searches are used to select which trees to search. As the number of searches increases so does our system's performance relative to the standard k-d tree. Additionally, at a large enough number of searches, the standard k-d tree eventually catches up to our index as both converge to zero error. Thus, our index was most effective when about .2% to .5% of the nodes

were searched.

5.6 Size of Result Set

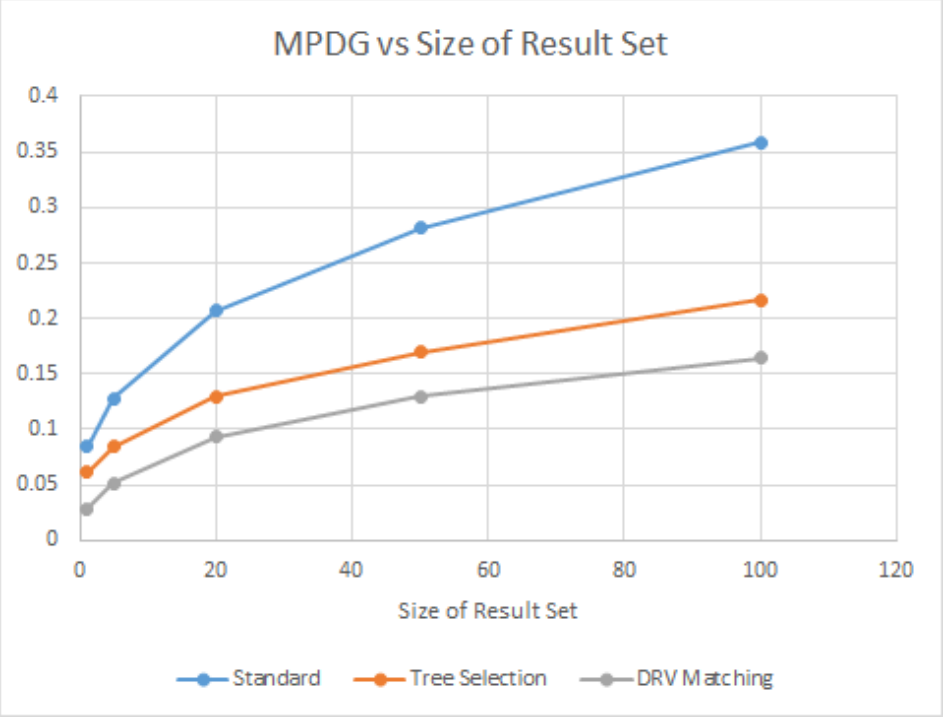

FIGURE 5.7: Full system test with varying size of result set (K)

The effect of the size of the result set (K) on MPDG was also tested. Results are shown in Figure 5.7. For all indexes, the quality of the results decreased with larger K. With a larger K, the index must find points with an increasingly large radius from the query point. Since the ANN searches are performed most densely around the query point, the quality of further points is expected to be lower. It should also be noted, that a because all branches must be searched which may contain a point closer than the current furthest point in the top K, with a larger K the stored furthest distance will be larger, and less branches of a tree can be pruned. Of note is that with larger K, the quality of the standard k-d tree falls at a faster rate than that of our system. This is because as the search expands outwards, since the split dimensions in our system are

more optimal, closer points with the modified distance metric will be searched sooner, and more branches will be pruned allowing the search to expand deeper.

5.7 Number of Trees Per Query

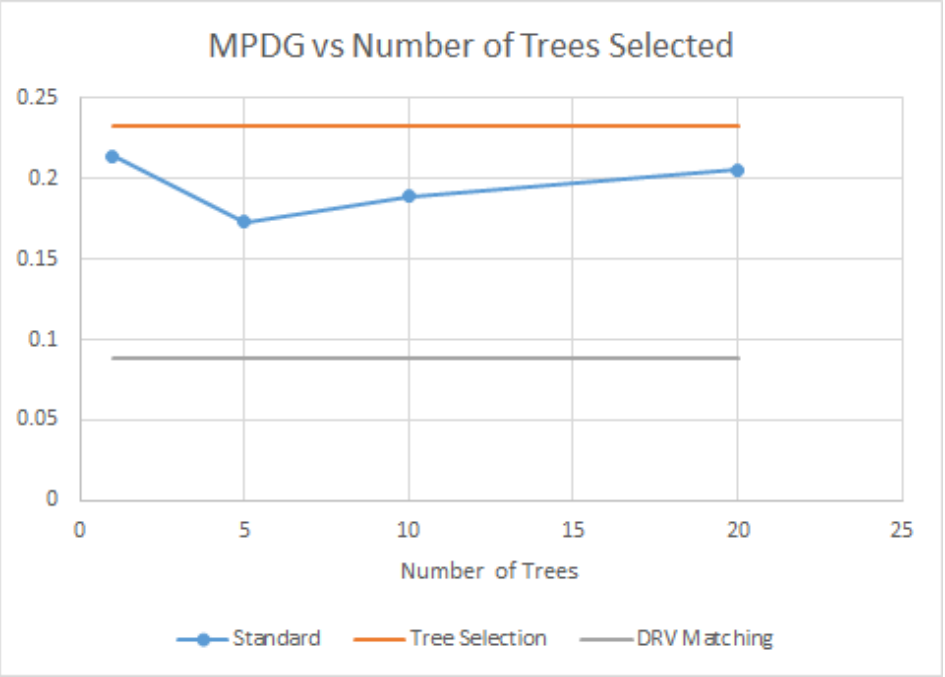

FIGURE 5.8: Full system test with varying number of trees retrieved per query

The number of trees searched per query (M) also has an effect on performance. The results are shown in Figure 5.8. While with a perfect DRV match searching a single tree has the best performance, this is not the case when the DRV of a query does not perfectly match a tree's seed DRV. Thus, when our system searches only one tree, because of the mismatch in these DRVs although performance is better than a standard k-d tree, it is not as high as when searching 5 trees in parallel. When searching 5 trees, even though none of the trees are likely perfect matches, they add additional variety to the search, and likely introduce DRVs which are both too high and too low in each dimension rather than having a biased search. However, increasing the number of trees too much prevents

our system from searching as far in each tree and the performance begins to decrease. Thus, the system should be tuned on this data set with around 5 trees.

5.8 Alternative Dataset

While it was demonstrated that our system consitently performs better than a standard k-d tree on the previous data set of uniformly distributed data, we also tested our system on a new data set with a drastically different distribution. The data set selected was the Corel Image Features Color Histogram data set [31]. This data set of over 68040 images has 32 dimensions, each of which represents the density of a range of colors in an image. Similar images can then be computed based on these color densities from these 32 dimensional representations of the image. It is important to note that this dataset was highly non uniform. For most of the images a large number of the 32 dimensions had a value of 0 or close to zero. Therefore, using the distance between the mininum and maximum elements is a poor metric of the length of a dimension. Thus, our system was also benchmarked using the variance of a dimension as a measure of its spread during SMS.

The results for the same search conditions as described in Section 5.1 are shown in Figure **??**. In this figure, all query DRVs were selected randomly with uniform weights on each dimension. Interestingly, when using the poor maximum-minimum metric for SMS and WSMS, the tree with a perfectly matched DRV actually performed more poorly than our system. This is likely because even though the seed and query DRVs matched, because of the flawed SMS dimension length metric poor split dimensions were selected. When using variance however, performance is similar to that of our random dataset.

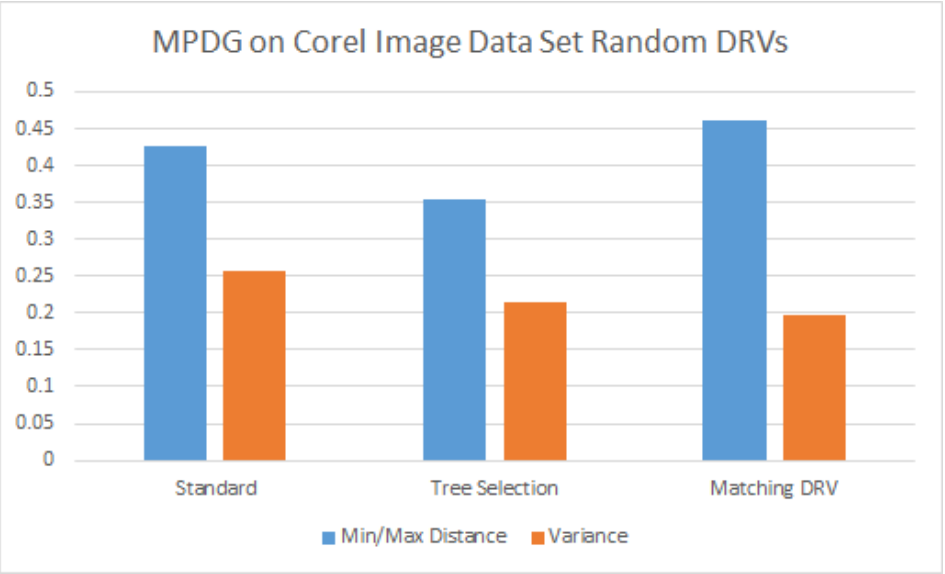

FIGURE 5.9: Full system test on Corel Image Features Data Set with random DRVs from uniform distribution

The performance of our system lies in between that of a standard k-d tree and a k-d tree with a matched seed DRV.

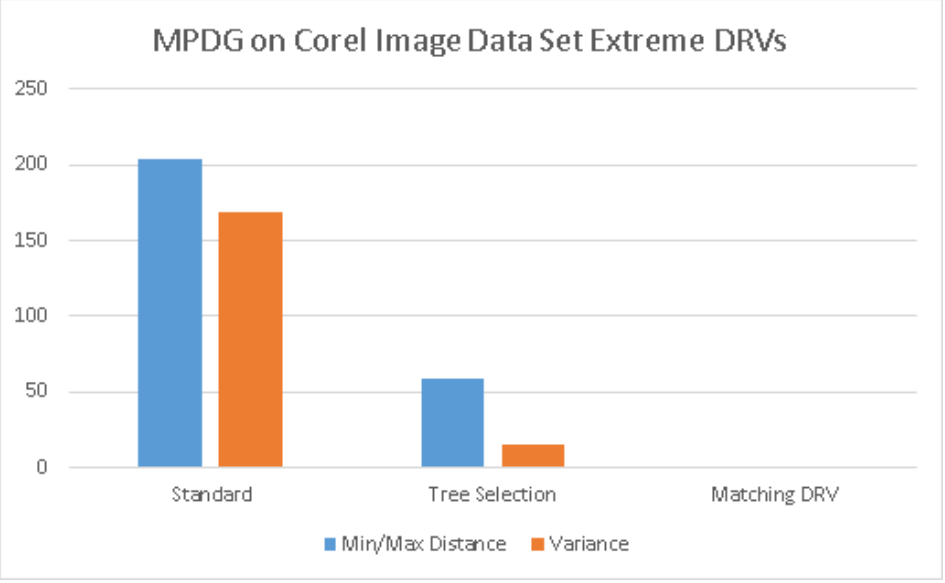

FIGURE 5.10: Full system test on Corel Image Features Data Set with DRVs with a low number of dimensions

This dataset was also tested with queries with a low number of dimensions shown in Figure 5.10. The same procedure for generating seed DRVs from Section 5.1 was used;

however, since this dataset had 32 dimensions instead of 8, p was set to .03. Following a binomial distribution, this means that just under 80% of the seed DRVs had 3 or less dimensions. Additionally, because of the larger number of dimensions, DDD was set as 2 rather than 3. With 100 random trees, and 1 additional tree with uniform seed DRV, our system used a total of 629 trees for this test. Additionally, query points were pulled from the data set rather than generated randomly. From Figure 5.10, it is clear that the trees generated with matching seed DRVs again have nearly zero MPDG. This is because these trees only use a low number of dimensions, where k-d trees are most effective. Our system's performance is still significantly better than that of the standard k-d tree, especially when using the variance for WSMS. Because of the larger number of dimensions, we were restricted from setting the DDD to be larger, as increasing it to 3 would result in 4960 additional trees. Thus, our index can be most effective on queries whose DRVs contain a low number of dimensions (or those with high weights on a small subset of dimensions) when the dimensionality of the data set is lower, as the DDD parameter can be increased with lower memory cost.

Chapter 6

Future Work

6.1 Heuristic Tuning

Although we have demonstrated that the selected heuristics provide significant performance improvements over a standard k-d tree, they are likely not the optimal heuristics for every dataset. One way to improve performance would be to tune split heuristics specific to the distribution of a dataset. For example, for a dataset with a uniform distribution across all dimensions, spatial median splitting will perform very well. However, for a dataset which consists of distinct Gaussian distributed clusters, an alternative heuristic could lead to improvements. In general, more complicated heuristics such as surface area heuristics have demonstrated improved performance with a higher offline computational cost [24]. If these heuristics were adapted to the constraint of DRVs, further improvements in the quality of the result set given a fixed number of searches could likely be made. Tuning this heuristic would effect the theoretical best case scenario when the seed and query DRVs match.

Another route for potential improvement is to improve the tree selection heuristic. In the current implementation, a distance metric is used to provide a scalar value which represents the quality of a tree. However, another potential approach would be to keep track of how well a seed DRV matches a query DRV in all dimensions. This would mean using a vector based distance metric rather than a scalar. In doing so, rather than simply using the trees which best match the DRV overall, a set of trees could be extracted which attempt to make sure each dimension is represented with proportional relevance across all of the trees. In doing so, one could avoid selecting a set of trees in which the same dimensions are over or under represented in each.

6.2 Real World Considerations

There are a variety of aspects which must be considered when attempting a real world implementation of this algorithm. The first is the large memory cost. As shown in Section 5, a larger amount of trees results in improved performance, as the overall quality of the top trees will be higher. However, there is a diminishing return of this effect, so memory consumption should be balanced and tuned. One proposed method of tuning is to develop a cost metric which considers both memory usage, and the quality of results. Using a small initial set of requested or randomly generated queries, one can use this data to minimize this cost metric, and select the optimal amount of trees. Other algorithmic parameters can be tuned in a similar manner. By examining the performance relative to a linear search on a subset of queries, one can attempt these queries with different parameters to see if performance can be improved. [10] suggests some methods for automatic parameter selection such as putting weights on different cost aspects of the algorithm and generating an objective function which represents the

total cost. The parameters which minimize this function based on a set of sample queries would be considered optimal.

If the dataset is large, a single server implementation is likely not possible as the memory consumption of this algorithm can be hundreds of times larger than the original dataset. To counteract this, we propose a distributed approach with N trees split across M compute nodes, with some duplication factor D. The duplication factor represents the number of different nodes on which a single tree exists in memory. A larger duplication factor will further increase memory cost, but will allow the system to be more robust in the event of a node's failure. [32] discusses many types of potential architectures for a system with distrubed memory, and considerations of each.

In our suggested distributed implementation, each node will store as many trees as it can fit into memory, and will also store the seed DRVs used to construct each tree, and the nodes on which each of these trees exist. All query requests will be made to a master node. This node acts as a load balancer and routes each query to the least busy node. Performance concerns with load balancing are discussed in [33]. When a request arrives at a node, that node can compute the best set of trees to search, and the least busy nodes which hold those trees. This selection process can be augmented by adding an additional cost to trees whose nodes are currently busy. In doing so, the search can be performed with slightly lower quality to avoid waiting on nodes. This system could also be implementated with no duplication (D = 1). While this would result in less memory consumption, in the event of a node shutting down all of its trees would need to be reconstructed in others. During this time, those trees would be unavailable, which would temporarily hinder performance on queries which would benefit from them.

After the top trees and their nodes are selected for a query, a distributed priority queue

and hashtable must be used to perform a parallel search on them. Considerations for these data structures are discussed in [34] and [35]. It is also important to ensure that the searches happen in parallel for the best results. For example, if one node searches its allocated amount before another, results will not be as accurate as if the searches are interleaved between the two nodes. In our testbed, searches were perfectly interleaved relative to the quality of each tree. Thus, real world performance will likely not reach that of our test results.

Another important consideration with this distributed system is continuous performance tuning. In our proposed construction of the data structure, no information was available about the frequencies of different DRVs in each query, and as such a uniform distribution of seed DRVs was used. However, by storing all requested DRVs, one can learn which type of queries are most popular. The system can then eliminate the trees which are least used, and replace them with trees that better match the popular queries. This dynamic adjustment will allow the system's performance to improve over time as more data is gathered about the types of searches performed.

Another potential concern would be a dataset large enough such that it cannot fit in memory on a single node. In this case, subtrees must be included on separate nodes containing parts of the dataset. Performance concerns on a distributed k-d trees are discussed in [36]. Another potential solution is to perform the ANN queries on disk. This is far from ideal as disk read and write times are significantly larger than those of memory [37]. The primary cost encountered would become page reads and writes, which k-d trees are not optimized for. In this case, a similar system to ours could be constructed using R-trees, a similar tree to k-d trees which is optimized to minimize disk costs [38].

Lastly, one must consider performance on a changing dataset. In particular, datasets tend to grow over time, and as such it is important to support efficient insertion of new data. To insert a new item, it must be added to every single k-d tree. As mentioned in section 2.4.5, the insertion of a new element into a k-d tree is O(Log(N)). However, over time trees are likely to become unbalanced, and the quality of results will begin to degrade. One approach of avoiding the issue would be to constantly rebuild trees which have a large number of inserted nodes. This could be performed one tree at a time, so the system could remain live while updating its trees in the background. However, when this cost is introduced, the offline construction cost of trees becomes more imporant, as trees will be constructed during system use. Thus, for a continuously growing data structure there may be some value in the random splitting heuristic which, although having weaker query performance, could be constructed faster due to not requiring a linear seek on each dimension.

Chapter 7

Conclusion

Bibliography

[1] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.

[2] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[3] Oren Boiman, Eli Shechtman, and Michal Irani. In defense of nearest-neighbor based image classification. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[4] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and image processing*, 14(3):227–248, 1980.

[5] 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.

[6] Svante Carlsson, J Ian Munro, and Patricio V Poblete. An implicit binomial queue with constant insertion time. In *SWAT 88*, pages 1–13. Springer, 1988.

[7] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is nearest neighbor meaningful? In *Database TheoryICDT99*, pages 217–235. Springer, 1999.

[8] David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.

[9] Matthew T Dickerson and R Scot Drysdale. Fixed-radius near neighbors search algorithms for points and segments. *Information Processing Letters*, 35(5):269–273, 1990.

[10] 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.

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

[12] Peter N Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, volume 93, pages 311–321, 1993.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] Chris Cheadle, Marquis P Vawter, William J Freed, and Kevin G Becker. Analysis of microarray data using z score transformation. *The Journal of molecular diagnostics*, 5(2):73–81, 2003.

[28] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o (n log n). In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 61–69. IEEE, 2006.

[29] John Ousterhout and Fred Douglis. Beating the i/o bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review*, 23(1):11–28, 1989.

[30] Nor Bahiah Hj Ahmad. Binary search tree.

[31] Michael Ortega-Binderberger, jul 1999.

[32] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Distributed Shared Memory-Concepts and Systems*, pages 42–50, 1991.

[33] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301, 1989.

[34] M Frans Kaashoek and David R Karger. Koorde: A simple degree-optimal distributed hash table. In *Peer-to-peer systems II*, pages 98–107. Springer, 2003.

[35] Anne Rogers, Martin C Carlisle, John H Reppy, and Laurie J Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):233–263, 1995.

[36] Mohamed Aly, Mario Munich, and Pietro Perona. Distributed kd-trees for retrieval from very large image collections. In *Proceedings of the British Machine Vision Conference (BMVC)*, 2011.

[37] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[38] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.