High Dimensional Similarity Search With Space Filling Curves *

Swanwa Liao *Mario A. Lopez* *Scott T. Leutenegger*

Department of Mathematics and Computer Science

University of Denver, Denver CO 80208.

{swliao,mlopez,leut}@cs.du.edu

Abstract

We present a new approach for approximate nearest neighbor queries for sets of high dimensional points under any L_t-metric, $t = 1, \dots, \infty$. The proposed algorithm is efficient and simple to implement. The algorithm uses multiple shifted copies of the data points and stores them in up to $(d+1)$ B-trees where d is the dimensionality of the data, sorted according to their position along a space filling curve. This is done in a way that allows us to guarantee that a neighbor within an $O(d^{1+1/t})$ factor of the exact nearest, can be returned with at most $(d+1)\log_p n$ page accesses, where p is the branching factor of the B-trees. In practice, for real data sets, our approximate technique finds the exact *nearest neighbor between 87% and 99% of the time and a point no farther than the third nearest neighbor between 98% and 100% of the time. Our solution is dynamic, allowing insertion or deletion of points in $O(d\log_p n)$ page accesses and generalizes easily to find approximate k-nearest neighbors.*

1 Introduction

Efficient nearest neighbor (NN) search for high dimensional data is an important problem in such diverse areas as data mining, image processing, image indexing, and pattern recognition [9, 4, 5]. The past decade has witnessed a significant amount of effort devoted to solving the nearest neighbor problem efficiently (see [12, 13, 2] for detailed surveys). Despite all this activity, for high dimensions most solutions remain disappointingly slow, often requiring as much or more work than a linear scan of the entire data set. This problem, which has been referred to as the "curse of dimensionality" [2], usually arises because a proposed technique has running time that is logarithmic in the number of data points but exponential in the number of dimensions. The difficulty of the problem has lead researchers to consider approximate solutions. In fact, as discussed in [6], in many applications an exact answer may not be needed.

In this paper we present a practical approximate NN algorithm for disk resident high dimensional data. Our algorithm, which uses a family of shifted space filling curves, provides extremely accurate answers with a number of page accesses which is linear in the number of dimensions, thus avoiding the aforementioned dimensionality curse. Furthermore, the maximal approximation error of the algorithm is theoretically bounded. Our method is simple and dynamic, i.e., it can efficiently accommodate insertions and deletions of points. The basic idea is to store the input in multiple B-trees using different linear orders of the n d-dimensional input points. Each linear order is obtained by sorting the input points along a space filling curve *after* they have been subjected to a simple transformation. An approximate nearest neighbor is obtained by examining neighbors of the (transformed) query in each of the linearly ordered sets and returning the best candidate examined. The main contributions of this paper are:

- A simple, efficient and effective approximate NN algorithm for disk resident data. The algorithm guarantees an upper bound that is nearly linear in d on the approximation error of the answer returned and requires $O(\log_p n)$ page accesses from each of up to $d+1$ B-trees with branching factor p. An obvious adaptation using balanced trees results in a solution for memory resident data that runs in $O(d^2 \log n)$ time and $O(dn)$ space, where $O(dn)$ is the size of the input.
- Experimental evidence that our algorithm provides near-optimal approximation with a variety of realistic input loads. We implement and conduct substantial performance studies with real data sets. We show that the practical results are

*This work has been supported in part by the National Science Foundation under grant IRI-9610240.

much better than what the theory allows. For the (real) data sets considered, our algorithm finds the actual nearest neighbor between 87% and 99% of the time, and a point no farther than the third nearest neighbor between 98% and 100% of the time. Thus, though an approximation, in practice we provide an almost exact NN algorithm in time linear in the number of dimensions.

- Experimental comparisons with SR-trees [7] and locality sensitive hashing [6].
- We explore a trade-off between space/processing time and quality of approximation. This tradeoff is achieved by carefully selecting and varying the number of B-trees storing the input data.
- Buffer simulations to show the impact of buffering on our method and on SR-trees.

We compare the performance of our approach with that of SR-trees [7], an index structure for high dimensional NN believed to outperform many of the previously proposed methods. Unlike our approach, SR-trees provide exact answers to the nearest neighbor problem but require significantly higher disk accesses.

An approximate method based on Locality-Sensitive Hashing (LSH) was presented in [6]. Our results differ from those of [6] in several respects:

- LSH is not guaranteed to find an answer, our method is.
- When LSH finds an answer, the distance from the true optimal is not bounded with 100% probability, ours is.
- Our experimental results show a smaller error for similar number of page accesses.
- LSH performance is better than our method if larger errors are allowed, but results in significantly more page accesses than our method, when errors of less than 10% are required.

The idea of using a family of space filling curves for approximate neighbor matching is not exclusively ours. Closely related to our work are the results of Shepherd et al [14] which we recently discovered. Our work was done independently and differs in the following:

- While both methods are based upon linearly ordering the transformed input points along a space filling curve, the transformations used are different. Shepherd et al [14] use random reorderings and scaling of the input coordinates. We use specific (deterministic) translation vectors to transform the input points.
- Our transformations (translations) allow us to guarantee a maximum approximation error. No such guarantee is available from [14] which states "at present, the ... (transformations) ... are essentially random. It is an open problem whether a set of optimal ... (transformations) ... could be devised for arbitrary data". Indeed, we get closer to this goal. We provide a set of transformations that, while not known to be optimal, result in answers guaranteed to be within a constant factor of the optimal.
- We include a much more extensive set of experiments, including a study of the impact of buffering on the number of disk accesses as well as a larger collection of data sets.

The rest of the paper is organized as follows. In Section 2 we describe the algorithm and derive theoretical bounds. Section 3 describes our experimental methodology and the data sets used. In Section 4 we present experimental results. We conclude in Section 5.

2 Algorithm Description

In this section we describe the algorithm, analyze its performance, and derive key theorems that provide the theoretical bound on the maximum approximation error. In the process, we describe an efficient version of the algorithm for memory-resident data.

We start with an intuitive explanation of the key ideas of our approach.

2.1 Key Idea

The L_t distance between points p and q in $\mathbb{R}^d$ is defined as $dist_t(p,q) = (\sum_{i=1}^{d} |p_i - q_i|^t)^{1/t}$ for $0 < t < \infty$ and $dist_\infty(p,q) = \max_{1 \leq i \leq d}\{|p_i - q_i|\}$, otherwise, where p_i denotes the i-th coordinate of p. Consider a recursive space filling curve, such as the Hilbert or z-curve [1, 12]. Such curves have the property that they linearize $\mathbb{R}^d$ in such a way that points that are close to each other along the curve are also close to each other under L_t distance. Thus, a natural approach for nearest neighbor searching is to sort the input points P according to their position on a Hilbert-curve. Given a query point q we find the position of q in the sorted list and report the closest of its two neighbors (predecessor and successor) along the curve. Unfortunately, it may happen that data that is close under L_t may not be close in Hilbert distance. Consider the top diagram of Figure 1. Assume that the top-right point in the lower-left quadrant is the query point and that the other points are data points. The point in the

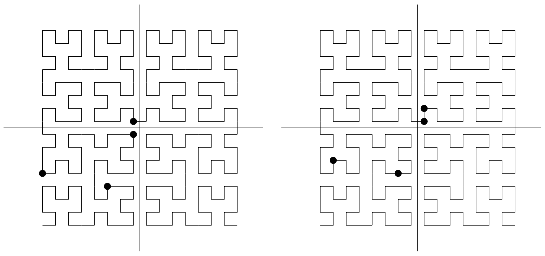

Figure 1: Effect of shifting the data.

upper-left quadrant is the true nearest neighbor of q. Unfortunately, both the predecessor and successor of q (the other two points in the lower-left quadrant) are closer to q in the linear Hilbert order. Thus, if we only check the immediate neighbors of q we would fail to find the true answer. The problem illustrated in this example results from the way in which the curve recursively divides space into compartments each of which is exhaustively traversed before proceeding to the next. This problem is inherent to many space filling curves and it is only worsened in higher dimensions.

Consider now what happens when we shift both query and all data points by $(1, 1)$, i.e., one unit up and to the right, as illustrated in the bottom diagram of Figure 1. In this case, the query point and the true nearest neighbor are now adjacent in Hilbert order. If we shift the data several times, we are more likely to find the true nearest neighbor by looking at the Hilbert predecessors and successors of q on every shifted list. Examining more than one predecessor and successor only improves the quality of the approximation.

As simple as it sounds, when the shifts are carefully controlled, this idea results in an algorithm with strong performance guarantees. We describe this algorithm in the next section.

2.2 Notation and Algorithm Description

For the rest of the paper we use the notation:

P: the set of points in the data set.

d: number of dimensions of the points in P.

n: number of points in P.

$v^{(j)}$: the d-dimensional vector $(j/(d+1), \dots, j/(d+1))$ for $j = 0, \dots, d$.

H_j: Hilbert-sorted list of $\{P[i] + v^{(j)}, i = 1, \dots, n\}$, i.e., the points of P shifted by $v^{(j)}$ and sorted in Hilbert order.

HL-set: set of $d+1$ Hilbert-sorted lists $H_0, \dots, H_d$.

The preprocessing phase of the algorithm simply builds the *HL*-set of P by independently building each of the H_j lists. Each list is then stored in a B^+-tree, modified so that the largest (resp. smallest) point of the left (resp. right) sibling is duplicated in the node.

A nearest neighbor query is conducted as follows. Search each B^+-tree to find the leaf level node whose range contains the query point. Examine all points in these $d+1$ nodes and return the point nearest to q as the approximate nearest neighbor.

Although in theory (as shown later) all $d+1$ lists are needed to guarantee an upper bound on the error of the approximation, we found that in practice a small subset of the *HL*-set suffices in order to achieve a small approximation error. Thus, the number of trees actually built may be smaller than $d+1$. The trade-off between memory/error and the number of built indices is further discussed in Section 4.

Updating the *HL*-set is equally simple. For updates, simply insert/delete the target point into/from each of the (up to) $d+1$ trees.

2.3 Cost of Searching and Updating

We assume that the root pages of the B-trees are kept in memory, as they need to be accessed with every operation. Notice that no more than $d+1$ pages need to be thus buffered. In this case each index lookup requires at most $h-1$ page accesses where $h = \lceil \log_p n \rceil$ and p is the branching factor of the B-trees. Thus, the total number of disk accesses is at most $(d+1) \times (h-1) = O(d \log_p n)$. Further buffering will likely reduce this number by holding in memory a portion of the top levels of the B^+-trees.

An update is comprised of up to $d+1$ independent updates in the corresponding B^+-trees. Thus an update requires $O(d \log_p n)$ page accesses.

2.4 Approximation Error

In this section we discuss the theoretical basis of our algorithm which allows us to derive an upper bound on the approximation error.

Let I be an interval on the real line such that I^d contains all the input points. To simplify our analyses we assume that the data has been scaled so that $I = [0, 1)$. Thus, the shifted data of the *HL*-set fits in $[0, 2)^d$. A d-dimensional Hilbert curve can be viewed as a recursive subdivision of $[0, 2)^d$ into hypercubes with the property that the intersection of the curve with any one such hypercube is a *single contiguous* curve segment. All hypercubes at the same level of recursion have the same size. We refer to these hypercubes as r-regions.

Definition 1 *An r-region is an open ended hypercube in $[0,2)^d$ with side length of $r = 2^{1-m}$ having the form: $\Pi_{i=0}^{d-1}[a_i r, (a_i+1)r)$, where each a_i, $0 \le i < d$, and m are in $\mathbb{N}$.*

It is not hard to see that at the i-th level of recursion the Hilbert curve consists of 2^{di} disjoint segments, each of which corresponds to a unique r-region of size $r = 2^{1-i}$. We now discuss some properties of the *HL*-set.

Observation 1 *Let L be one of the Hilbert ordered lists of the* HL*-set and $r = 2^{1-i}$, for some $i \in \mathbb{N}$. If we divide $\mathbb{R}^d$ into equal sized r-regions, then each r-region contains one and only one contiguous subsequence of L.*

The quality of our approximation hinges on the capability of finding a tight r-region R with the query point centrally located, i.e., "far enough" (at a distance no less than αr) from each border of R.

In the following definition $\bmod(x,r) = x - \lfloor x/r \rfloor r$. For example $\bmod(1.8, 0.25) = 0.05$.

Definition 2 *A point p is α-central in an r-region iff for each $i = 0 \ldots d-1$, we have $\alpha r \le \mathrm{mod}(p_i, r) < (1-\alpha)r$, or equivalently, $2\alpha r \le \mathrm{mod}(p_i + \alpha r, r)$, where $0 \le \alpha < 0.5$.*

The following lemma is from Chan [3], whose proof we reproduce for completeness.

Lemma 1 *Suppose d is even. For any point $p \in \mathbb{R}^d$ and $r = 2^{-m}$ ($m \in \mathbb{N}$), there exists $j \in \{0, \ldots, d\}$ such that $p + v^{(j)}$ is $(1/(2d+2))$-central in its r-region cell.*

Proof. Suppose, on the contrary, that $p + v^{(j)}$ is not $(1/(2d+2))$-central for any $j = 0, \ldots, d$. Then, for each j, there is an index $i(j) \in \{0, \ldots, d-1\}$ with $\mathrm{mod}(p_{i(j)} + j/(d+1) + r/(2d+2), r) < r/(d+1)$, or equivalently, $\mathrm{mod}((d+1)2^m p_{i(j)} + 2^m j + \frac{1}{2}, d+1) < 1$. By the pigeonhole principle, there exist two distinct indices j, $j' \in \{0, \ldots, d\}$ with $i(j) = i(j')$. Letting $z = (d+1)2^m p_{i(j)} + 1/2$, we have $\mathrm{mod}(z + 2^m j, d+1) < 1$ as well as $\mathrm{mod}(z + 2^m j', d+1) < 1$. This can happen only if $2^m j \equiv \mathrm{mod}(2^m j', d+1)$. Since 2^m and $d+1$ are relatively prime, we must have $j = j'$, a contradiction! ■

Henceforth we assume d is even (for odd d, simply replace d by $d+1$).

Chan's powerful result combined with the properties of Hilbert curves discussed previously allows us to guarantee the existence of a tight r-region that contains the query point and its nearest neighbor. Every other point in the same r-region is not significantly farther than the nearest neighbor. The following lemma and theorem constitute the theoretical basis of our algorithm. In the process, we describe a memory resident data structure that allows us to find approximate nearest neighbors efficiently.

Lemma 2 *Let $C_\varepsilon(q)$ denote the closed hypercube of size $(2\varepsilon) \times \ldots \times (2\varepsilon)$ centered on q. Given a point $q \in [0,1)^d$ and ε, where $0 < \varepsilon < 1/(2d+2)$, there exists j, $0 \le j \le d$, such that $C_\varepsilon(q + v^{(j)})$ is contained in an r-region with r satisfying $r/(4d+4) \le \varepsilon < r/(2d+2)$.*

Proof. Note that $r/(4d+4) \le \varepsilon < r/(2d+2)$ defines a unique r. If we divide R^d into equal sized r-regions with r as defined above, then Lemma 1 guarantees the existence of j such that $q' = q + v^{(j)}$ is $(1/(2d+2))$-central in its r-region. Hence, the hypercube $C_{r/(2d+2)}(q')$ is completely contained in this r-region. But by construction $\varepsilon < r/(2d+2)$, so $C_\varepsilon(q')$ is also contained in the r-region. ■

Lemma 2 states that if a closed hypercube C is shifted $d+1$ times in a certain fashion, and if we were to find a tightest r_j-region that encloses the shifted C for each shift, then $\min(r_0, r_1, \ldots r_d)$ is bounded above by some small deterministic value.

Definition 3 *Let $c > 1$, a point $p \in P$ is a c-nearest neighbor of q if $dist_t(p,q) \le c \cdot dist_t(q^*, q)$, where $q^* \in P$ is the true nearest neighbor of q in L_t-metric.*

Theorem 1 *Let $c = d^{1/t}(4d+4)+1$, using $O(d^2 n \log n)$ pre-processing time and $O(dn)$ space, a c-nearest neighbor of q can be found in $O(d^2 \log n)$ time.*

Proof. The data structure alluded to in the theorem consists simply of the *HL*-set of P with each list stored in a sorted array. The approximation algorithm uses a binary search to locate q in each list H_j and computes the distance from q to its predecessor and successor in H_j. It reports the best distance among the up to $2(d+1)$ points examined. Let δ be the distance between the query q and its true nearest neighbor in P. Lemma 2 guarantees the existence an r-region (in one of the shifted lists) that contains the hypercube $C_\delta(q + v^{(i)})$ and such that $r/(4d+4) \le \delta < r/(2d+2)$, for some $0 \le i \le d$. Clearly, such r-region contains the (shifted) true nearest neighbor of q. Furthermore, because of contiguity of the curve, if any two points belong to an r-region every point on the curve in between does also. Thus, the r-region that contains both q and its nearest neighbor must also contain either the predecessor or successor of q (which, as shown experimentally in Section 4 is almost always the true nearest neighbor). By searching on all $(d+1)$ lists we are guaranteed to find a neighbor whose L_∞ distance to q

is at most $r \leq (4d+4)\delta$. Since for any two points p and q, $dist_t(p,q) < d^{1/t} d_{\inf}(p,q)$, then the approximate nearest neighbor found by our search is an $O(d^{1+1/t})$-approximate neighbor. ■

Adapting this result to disk-resident data is straightforward. We simply store each sorted array of the *HL*-set in a separate B$^+$-tree. Locating the predecessor and successor of q in a sorted list requires $\lceil \log_p n \rceil$ page accesses, where p is the branching factor of the tree.

Corollary 1 *Let $c = d^{1/t}(4d+4)$ and let p be the number of d-dimensional points that can be stored in a disk page. There is a disk-resident data structure that can be built using $O(dn)$ disk accesses which can be used to find a c-approximate neighbor of q with no more than $O(d \log_p n)$ disk accesses.*

Since a buffer is almost always invariably used, in practice, the number of disk accesses required to perform a query is substantially smaller than $d\lceil \log_p n \rceil$. Furthermore, even with a reasonably small buffer size, multiway sort can be used to sort each Hilbert list in $O(n)$ page accesses. We study the impact of a buffer in Section 4.

2.5 Approximate k-Nearest Neighbors

The approach to find a good approximation to the k nearest neighbors of a query q is similar to the method outlined above. Instead of looking at only one predecessor and successor, we examine k predecessor and k successors for each list H_j. From the (up to) $2(d+1)k$ points examined we select the best distinct k points.

An argument similar to the one presented at the end of the previous section guarantees that the distance from q to the k-th neighbor returned is within a factor of $O(d^{1+1/t})$ of the true k-th distance. Notice that for arbitrary k the number of pages examined per list is $\lceil 2k/p \rceil$, where p is the capacity of a node. These pages can be retrieved efficiently by allocating them contiguously on disk.

3 Experimental Methodology

We base our experimental results on several real data sets. The first three correspond to feature vectors extracted from the same set 68,040 Corel images[1]. These data sets are the color histogram which is 32 dimensional, the texture set which is 16 dimensional, and the moments set which is 9 dimensional. For these data sets we conjecture that uniform queries do not make sense. Thus, we chose to remove 1,000 of the points from the data set and used these as the query set. The 1000 points were chosen using a uniform random distribution between 1 and 68,040. The final result is a collection of data sets of 67,040 points and query sets of 1,000 points.

The second data set (provided indirectly by B.S. Manjunath [8]) is composed of 275,465 sixty dimensional feature vectors representing texture information of blocks of large aerial photographs [2]. Again, we conjecture queries would likely not be uniform and hence removed 5,465 points at random to use as query points. The final result was a data set (Landsat) of 270,000 sixty dimensional points and a query set from which we used 1000 points.

We also created reduced dimensionality data sets to study the impact of dimension. These were obtained by using the first 4, 10, and 20 dimensions of each point of the color histogram data, and the first 4, 10, and 30 dimensions of each landsat point.

We compared our approximate algorithm to the exact 1-NN SR-tree algorithm [7, 11]. We downloaded the code from Katayama's web site (`www.rd.nacsis.ac.jp/~katayama/homepage/research/srtree/English.html`) and modified it to create a trace of the pages accessed. We then simply ran 1000 queries and calculated the average number of pages accessed. We also collected the page trace as input to our buffer simulations. No attempt at calculating confidence intervals was made due to the length of experimental run time.

For fully packed B$^+$-trees [10] the number of page accesses is deterministic, the height of the tree minus 1, given the data set size. Thus we were able to verify our experimental results analytically. For the approximate solution we also collected page access traces which we used in the buffer simulations.

3.1 Buffer Simulations

We coded a simple LRU trace-drive buffer simulation. Our simulations assume the buffer is initially empty and use the page access traces collected in our experiments as input. The simulation reports the number of accesses and the number of misses. Given that we used a single trace for each simulation there was no opportunity to collect confidence intervals.

4 Experimental Results

In the first experiment we study the effect of increasing the number of dimensions for the color histogram

[1] The data sets and the definitions of the features can be retrieved from `http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html`

[2] The data can be retrieved from `http://vivaldi.ece.ucsb.ucsb.edu/users/wei/gaborfeatures/airphotoFeatures/`

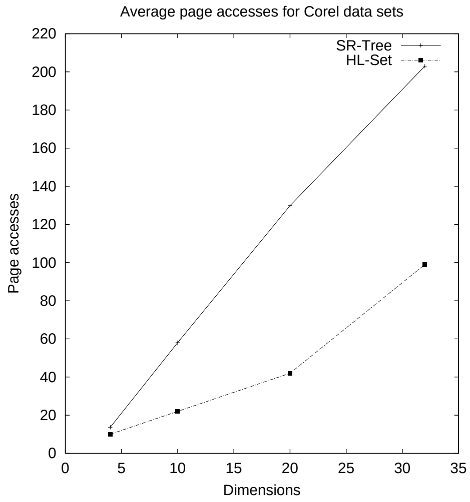

Figure 2: Number of page accesses as a function of number of dimensions, color histogram data.

data set. Specifically, we consider the color histogram data sets using the first 4, 10, 20, and 32 dimensions, as described above. In Figure 2 we plot the average number of page accesses per query versus the number of dimensions. Clearly, as d increases, the number of page accesses grows much faster for the SR-tree than for the approximate solution. Note that the steeper slope of the curve for the approximation algorithm between 20 and 32 dimensions is due to the B-trees growing one in height. Since height is $\log_p$, where p is the branching factor, the slope would return to linear again as the number of dimensions was increased.

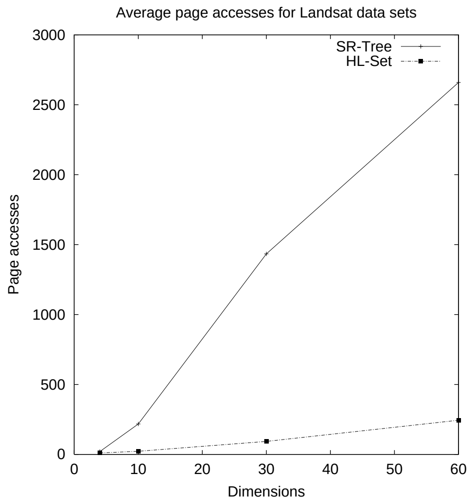

Figure 3: Number of page accesses as a function of number of dimensions, Landsat data.

Our second experiment also studies the effect of increasing the number of dimensions but this time for the Landsat data set. Specifically we used the Landsat data with d equal to 4, 10, 30, and 60. In Figure 3 we plot the average number of page accesses per query versus the number of dimensions. Again, the rate of growth in number of page accesses is faster for the SR-tree code than for the approximate solution. At 60 dimensions the SR-tree requires 14 times more page accesses.

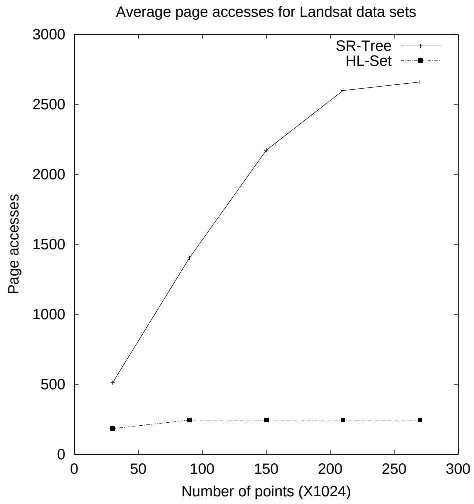

Figure 4: Number of page Accesses as a function of input size (n), Landsat data.

Our third experiment considers the effect of increasing the number of points while holding the number of dimensions constant. In Figure 4 we plot the average number of page accesses per query versus the number of points in the data set for the Landsat data set. We obtained the data sets by using only N random points from the Landsat data set. Once again, the approximate solution exhibits a much smaller rate of growth as the input size increases.

The previous three experiments suggest that the approximate algorithm is substantially faster than the exact SR-tree NN algorithm. The surprising result is that our heuristic usually finds the exact nearest neighbor!

In Tables 1 and 2 we present the average error ratio and frequency with which the approximate algorithm finds the exact NN. The error ratio is calculated as the ratio of the distance from the query point to the data point found by the approximate algorithm over the distance between the query point and the exact NN. The average error ratio presented is just the average of this ratio over all 1000 query points. For the Landsat data with a large number of points the approximate algorithm reports the exact NN 86.7% of

Data set	d	Approx/exact ratio	Found exact NN
Moments	9	1.00090	99.9%
Texture	16	1.00023	99.6%
Histogram	32	1.00021	99.5%
Landsat	4	1.00000	100.0%
	10	1.00000	100.0%
	30	1.00333	92.5%
	60	1.00580	86.7%

Table 1: Approximation error as a function of number of dimensions.

n	Approx/exact ratio	Found exact NN
30K	1.00068	98.8%
90K	1.001	96.8%
150K	1.002	94.6%
270K	1.0058	86.7%

Table 2: Approximation error as a function of input size (n), Landsat data ($d = 60$).

the time. When missing the *NN* it reported the second nearest 8.6% of the time, the third nearest 3% of the time, the fourth nearest 1.1% of the time, and the seventh nearest 0.5% of the time. For only one of 1000 queries it returned a point farther than the 7th nearest neighbor. This one time it returned the 15th nearest neighbor. The approximation ratio was less than 1.01 when averaged over the 1000 queries. Even for the single case where the approximation returned was the 15th nearest neighbor, the approximation ratio was 1.21.

For the Corel Color Histogram data the results were even more encouraging. For the 1000 queries examined our approximation algorithm returned the nearest neighbor 99.5% of the time, the second nearest 0.4% of the time and the third nearest 0.1% of the time. It never returned a point that was farther than the third nearest.

The space and time requirement can be reduced for the approximate algorithm if we are willing to sacrifice accuracy. This is achieved by reducing the number of indexes built on the data set. In Tables 3 and 4 we present the error ratio and percent of time the exact NN is found as a function of the number of indices built for the color histogram and Landsat data sets respectively. As can be seen, we can reduce the number of indexes to about 30% of the original size (10 indexes for the Corel data, 19 indexes for the Landsat data) with little reduction in accuracy. Even for a small number of indexes, the error ratio is small, though the percentage of time the exact NN is found drops significantly.

# Lists	Approx/exact ratio	Found exact NN
1	1.1475	45.6%
3	1.0300	78.8%
6	1.0110	90.5%
10	1.0037	96.1%
33	1.0002	99.5%

Table 3: Degradation in accuracy by reducing the number of lists. Color histogram data

# Lists	Approx/exact ratio	Found exact NN
1	1.270	11.0%
3	1.134	25.1%
6	1.080	38.5%
10	1.055	47.5%
19	1.027	65.6%
61	1.006	85.9%

Table 4: Degradation in accuracy by reducing the number of lists, Landsat data

An easy way to greatly improve the approximation accuracy is to consider larger page (node) sizes. All previous results assume a page size of 4K. If we increase page size to 8K then twice as many candidates are considered. For the Landsat data set with $n = 270$K and 61 indexes, the error ratio improves to 1.0031 and the percent exact improves to 91.6%. Furthermore, larger page sizes marginally increase the page access time. Additional experiments are need to more fully explore the tradeoff on page size.

We can compare our performance and accuracy to the LSH method by considering Figure 9 in [6]. In that figure the authors show that to achieve an error of 13% approximately 100 page accesses are needed. To achieve this number they assume a reduced Landsat data set of 200,000 points. They also assume 1 byte coordinates and 8K page sizes. Our method returns an error of less than 1270,000 point data set. Our results were obtained with a 4K page assuming 4 byte coordinates. If we make the number of points per page equal, we would require only 30 page accesses to acquire this much better degree of approximation

In our final experiment we consider how buffer size affects the relative performance of the algorithms. In Figure 5 we plot the total number of disk reads (buffer misses) versus the buffer size for the Corel color histogram data. We plot the buffer size as the percentage of the the actual data set size. Both methods require more space than the data set size. If there was sufficient memory to store the entire data in memory there would be no need to index the data, thus we set an upper bound of the buffer size equal to 95% of the data set size. The top line is for the SR tree. The next

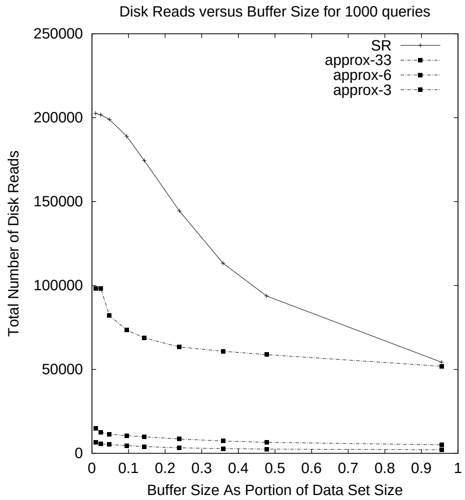

Figure 5: Disk accesses as a function of buffer size, color histogram data.

three lines from top to bottom assume the approximate method with 33, 6, and 3 B^+-trees respectively. As long as the buffer available is small relative to the data set size, the approximate method using 33 B^+-trees requires substantially less I/O. If willing to give up some accuracy, the indexes built on 3 and 6 B^+-trees require an order of magnitude less I/O.

5 Conclusions

We have presented a new approach for nearest neighbor queries on high dimensional data. Our approach is an approximation, but in practice, on real data sets, we have found it finds the *exact* nearest neighbor between 87% and 99% of the time and a point no farther than the third nearest neighbor between 98% and 100% of the time.

Our algorithm works under any L_t-metric, $t = 1, \ldots, \infty$, and is efficient and simple to implement. Furthermore, our solution is dynamic, allowing insertion or deletion of points in $O(d \log_p n)$ page accesses and generalizes easily to find approximate k-nearest neighbors.

The algorithm uses multiple shifted copies of the data points and stores them in up to $(d+1)$ B-trees where d is the dimensionality of the data, sorted according to their position along a space filling curve. This is done in a way that allows us to provide a theoretical guarantee that a neighbor within an $O(d^{1+1/t})$ factor of the exact nearest, can be returned with at most $(d+1)\log_p n$ page accesses, where p is the branching factor of the B-trees. In practice, we found that the approximation is much more accurate than the theoretically derived bound.

In the future we intend to continue exploring accuracy versus time and space tradeoffs by varying the number of indexes. We also intend to explore the performance of our algorithms on k-nearest neighbor searches.

References

[1] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoretic. Comp. Sci.*, 181:3–15, 1997.

[2] S. Berchtold and D. Keim. High dimensional indexing structures database support for next decade's applications. In *Proc. ACM SIGMOD*, page 501, 1998.

[3] T. Chan. Approximate nearest neighbor queries revisited. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 352–358, 1997.

[4] S. Craver, B. Yeo, and M. Yeung. Multi-linearization data structure for image browsing. *SPIE*, 3656:155–166, 1999.

[5] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30:170–231, 1998.

[6] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. Int. Conf. on Very Large DataBases (VLDB)*, pages 518–529, 1999.

[7] N. Katayama and S. Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD*, pages 369–380, 1997.

[8] B.S. Manjunath and W.Y. Ma. Texture features for browsing and retrieval of large image data. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 18(8):837–842, 1996.

[9] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:989–1003, 1997.

[10] A.L. Rosenberg and L. Snyder. Time and space optimality in b-trees. *ACM Transactions on Database Systems*, 6(1), 1981.

[11] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. ACM SIGMOD*, pages 71–79, 1995.

[12] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[13] T. Sellis, N. Roussopoulos, and C. Faloutsos. Multidimensional access methods: Trees have grown everywhere. In *Proc. of 23th International Conference on Very Large Databases (VLDB)*, pages 13–15, 1997.

[14] J. Shepherd, X. Zhu, and N. Megiddo. A fast indexing method for multidimensional nearest neighbor search. In *Proc. of SPIE Vol. 3656, Storage and retrieval for image and video databases*, pages 350–355, 1998.