An Investigation of Practical Approximate Nearest Neighbor Algorithms

Ting Liu
Dept. Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
tingliu@cs.cmu.edu

Andrew W. Moore
Dept. Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
awm@cs.cmu.edu

Alex Gray
Dept. Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
agray@cs.cmu.edu

Ke Yang
Dept. Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
yangke@cs.cmu.edu

Abstract

This paper concerns approximate nearest neighbor searching algorithms, which have become increasingly important, especially in high dimensional perception areas such as computer vision, with dozens of publications in recent years. Much of this enthusiasm is due to a successful new approximate nearest neighbor approach called Locality Sensitive Hashing (LSH). In this paper we ask the question: can earlier spatial data structure approaches to *exact* nearest neighbor, such as metric trees, be altered to provide approximate answers to proximity queries and if so, how? We introduce a new kind of metric tree that allows overlap: certain datapoints may appear in both the children of a parent. We also introduce new approximate k-NN search algorithms on this structure. We show why these structures should be able to exploit the same random-projection-based approximations that LSH enjoys, but with a simpler algorithm and perhaps with greater efficiency. We then provide a detailed empirical evaluation on five large, high dimensional datasets which show accelerations one to three orders of magnitude over LSH. This result holds true throughout the spectrum of approximation levels.

1 Introduction

The *k-nearest-neighbor searching problem* is to find the k nearest points in a dataset $X \subset R^D$ containing n points to a query point $\mathsf{q} \in R^D$, usually under the Euclidean distance. It has applications in a wide range of real-world settings, in particular pattern recognition, machine learning [7] and database querying [11]. Several effective methods exist for this problem when the dimension D is small (e.g. 1 or 2), such as Voronoi diagrams [22], or when the dimension is moderate (i.g. up to the 10's), such as *kd*-trees [8] and metric trees. Metric trees [24], or ball-trees [20], so far represent the practical state of the art for achieving efficiency in the largest dimensionalities possible [19, 6]. However, many real-world problems are posed with very large dimensionalities which are beyond the capability of such search structures to achieve sub-linear efficiency, for example in computer vision, in which each pixel of an image represents a dimension. Thus, the high-dimensional case is the long-standing frontier of the nearest-neighbor problem.

Approximate searching. One approach to dealing with this apparent intractability has been to define a different problem, the $(1 + \epsilon)$ *approximate k-nearest-neighbor searching problem*, which returns points whose distance from the query is no more than $(1 + \epsilon)$ times the distance of the true k^{th} nearest-neighbor. Further, the problem is often relaxed to only do this with high probability, and without a certificate property telling the user when it has failed to do so, nor any guarantee on the actual *rank* of the distance of the points returned, which may be arbitrarily far from k [4]. Another commonly used modification to the problem is to perform the search under the L_1 norm rather than L_2.

1.1 Locality Sensitive Hashing

Roughly speaking, a locality sensitive hashing function has the property that if two points are "close," then they hash to same bucket with "high" probability; if they are "far apart," then they hash to same bucket with "low" probability. Formally, a function family $\mathcal{H} = \{h : S \to U\}$ is (r_1, r_2, p_1, p_2)-*sensitive*, where $r_1 < r_2$, $p_1 > p2$, for distance function D if for any two points $\mathsf{p}, \mathsf{q} \in S$, the following properties hold:

1. if $\mathsf{p} \in \mathbb{B}(\mathsf{q}, r_1)$, then $\Pr_{h\in\mathcal{H}}[h(\mathsf{q}) = h(\mathsf{p})] \geq p_1$, and
2. if $\mathsf{p} \notin \mathbb{B}(\mathsf{q}, r_2)$, then $\Pr_{h\in\mathcal{H}}[h(\mathsf{q}) = h(\mathsf{p})] \leq p_2$,

where $\mathbb{B}(\mathsf{q}, r)$ denotes a hypersphere of radius r centered at q.

By defining a LSH scheme, namely a $(r, r(1+\epsilon), p_1, p_2)$-sensitive hash family, the $(1+\epsilon)$-NN problem can be solved by performing a series of hashing and searching within the buckets. See [12, 14] for detailed discussions.

Here we only focus on a very simple hash function used in [10].

From a very high level, the hash function does the following. First, we assume that all the datapoints are integers and the range in each coordinate is between 0 and C. Then, for a d-dimensional data point $x = (x^1, x^2, ...x^d)$, one first express it in *unary*. In other words, x^i is converted to a C-dimensional vector containing x^i 1's followed by $(C - x^i)$ 0's. In this way we obtain a Cd-dimension binary vector y. Then, a *random sampling* is performed, where K out of Cd entries of y are randomly chosen to form a new K-dimensional binary vector z, and z is the hash value of x.

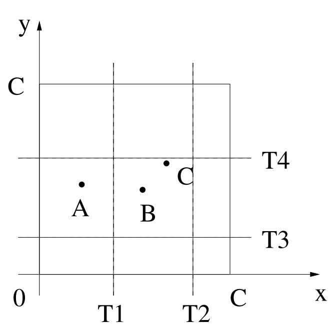

Figure 1: the "bucketing" view of LSH.

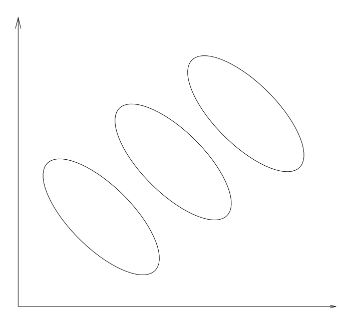

Figure 2: a tilted distribution.

Alternatively, we can interpret the hash function from a geometric point of view (see also [9] for a similar discussion). We call it the "bucketing view." Figure 1 shows an example where $d = 2$. The hash simply partitions the space $[0, C]^d$ into sub-rectangle using K *randomly* generated partition planes. As for the example in Figure 1, there are four partition planes: T_1 and T_2 are along the y-axis; T_3 and T_4 are along the x-axis. These planes split the space $[0, C]^d$ into $3 \times 3 = 9$ "buckets." Since these partition planes are randomly chosen, if two points are "close" (in L_1 norm), then the probability that they are mapped into the same bucket is "large." As in the example, point B is closer to point C than to point A, and thus it is more likely that B and C are mapped into the same bucket than that A and B are mapped into the same bucket.

Equipped with such a hash function, one simply needs to search within the bucket a query point q falls into in order to find a neighbor that is "close enough" to q. At a higher level, the LSH algorithm repeats the process L times. Each time a random hash function is independently generated and all the points within the bucket q falls into are searched. By increasing L, we can easily boost the probability that the LSH algorithm finds a $(1+\epsilon)$-NN successfully. On the other hand, if too many points fall in the same bucket as q, then LSH stops after it encounters certain points. In this way, LSH remains fast.

We stress that there exists a large family of LSH algorithms, with different hash functions. We only present the one with simplest hash function, since it is very efficient compared with others and has practical applications.

One very attractive feature of the LSH algorithm is that it enjoys a rigorous, theoretical performance guarantee. Indyk and Motwani [12] prove that even in the worst case, the LSH algorithm find an $(1+\epsilon)$-NN of any query point with high probability in a reasonable amount of time. It is also demonstrated [10, 13] that LSH can be useful in practice.

On the other hand, we observe that LSH has its own limitations. First, it was originally designed to work only in the L_1 norm, rather than the (more popularly used) L_2 norm. Although some later work has extended LSH to other norms, the hashing function needs to be changed and thus becomes more complicated.

Second, since LSH is designed to guarantee the worst-case behavior, it might not be as efficient on real-world data, which normally exhibit a rather "benign" behavior. For example, the data points typically form clusters, rather than uniformly distributed in the space. But since the LSH algorithm partitions the space uniformly, it does not exploit the clustering property of the data. In other words, since LSH does not adapt to the *local* resolution of the data points, in the case where the distribution of the datapoints are far from uniform, one necessarily needs to design a *family* of LSH functions, one for each level of resolution. In fact, the theoretical correctness of LSH stipulates one to "guess" a correct asymptotic value of $d(\mathsf{q}, X)$, the nearest distance between q and points in X. Therefore, in the worst case, many instances of the LSH algorithm needs to be run in order to guarantee the correctness.

Naturally, the real-world data are typically not in the worst case. In fact, Gionis et al. [10] observes that in practice, it suffices to use a *single* estimate r as the "expected" distance between a query point q and points in X. Notice that this r can be obtained by sampling a small random subset of X. However, it is unlikely that the theoretical correctness result will be maintained by this process.

From another direction, some work has been done in attempt to adapt LSH to the local resolution of the data points. For example, Georgescu et al. [9] proposed a modification to the random partition process using marginal distributions. Their modification enables regions of higher density of datapoints (i.e., clusters) to have a higher resolution of partitions. In [9], Georgescu et al. reported empirical efficiency improvement over the original LSH algorithm for a specific application. Nonetheless, this modification is largely heuristic and could be application-specific.

AWM Comments: We should discuss the following. I think the LSH folks could validly argue that it doesn't matter because if some buckets are empty we can afford to increase the resolution for the non-empty sections while overall using the same amount of memory.

Finally, the partition planes used in LSH are all aligned with the coordinates. This limitation on the orientation of the partition can hurt efficiency. Imagine a "tilted" distribution where data points form clusters that are distributed along the line $x = y$ (see Figure 2) Naturally, partitions along the anti-diagonal($x = -y$) direction are more efficient than ones along x or y axis. By insisting the partition direction, LSH may need to adopt a very high resolution.

2 Metric Trees and Spill Trees

We review the metric tree data structure and then introduce a variant, known as *spill trees*.

2.1 Metric Trees

The *metric tree* [24, 21, 5] is a data structure that supports efficient nearest neighbor search. We briefly A metric tree organizes a set of points in a spatial hierarchical manner. It is a binary tree whose nodes represent a set of points. The root node represents all points, and the points represented by an internal node v is partitioned into two subsets, represented by its two children. Formally, if we use $N(\mathsf{v})$ to denote the set of points represented by node v, and use $\mathsf{v.lc}$ and $\mathsf{v.rc}$ to denote the left child and the right child of node v, then we have

$$N(\mathsf{v}) = N(\mathsf{v.lc}) \cup N(\mathsf{v.rc}) \quad (1)$$
$$\emptyset = N(\mathsf{v.lc}) \cap N(\mathsf{v.rc}) \quad (2)$$

for all the non-leaf nodes. At the lowest level, each leaf node contains very few points.

Partitioning. The key to building a metric-tree is how to partition a node v. A typical way is as follows. We first choose two *pivot* points from $N(\mathsf{v})$, denoted as $\mathsf{v.lpv}$ and $\mathsf{v.rpv}$. Ideally, $\mathsf{v.lpv}$ and $\mathsf{v.rpv}$ are chosen so that the distance between them is the largest of all-pair distances within $N(\mathsf{v})$. More specifically, $||\mathsf{v.lpv} - \mathsf{v.rpv}|| = \max_{p1,p2 \in N(\mathsf{v})} ||p1 - p2||$. However, it takes $O(n^2)$ time to find the optimal $\mathsf{v.lpv}$ and $\mathsf{v.rpv}$. In practice, we resort to a linear-time heuristic that is still able to find reasonable pivot points.[1] After $\mathsf{v.lpv}$ and $\mathsf{v.rpv}$ are found, we can go ahead to partition node v.

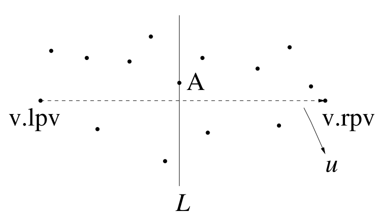

Figure 3: partitioning in a metric tree.

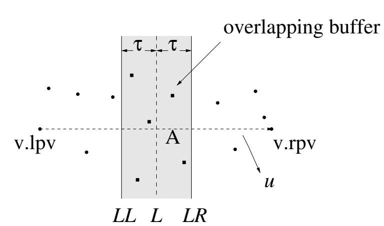

Figure 4: partitioning in a spill tree.

Here is one possible strategy for partitioning. We first project all the points down to the vector $\vec{u} = \vec{\mathsf{v.rpv}} - \vec{\mathsf{v.lpv}}$, and then find the median point A along $\vec{u}$. Next, we assign all the points projected to the left of A to $\mathsf{v.lc}$, and all the points projected to the right of A

[1]Basically, we first randomly pick a point p from v. Then we search for the point that is the farthest to p and set it to be $\mathsf{v.lpv}$. Next we find a third point that is farthest to $\mathsf{v.lpv}$ and set it as $\mathsf{v.rpv}$.

to v.rc. We use L to denote the d-1 dimensional plane orthogonal to $\vec{u}$ and goes through A. It is known as the *decision boundary* since all points to the left of L belong to v.lc and all points to the right of L belong to v.rc (see Figure 3). By using a median point to split the datapoints, we can ensure that the depth of a metric-tree is $\log n$. However, in our implementation, we use a mid point (i.e. the point at $\frac{1}{2}(\mathsf{v}.\vec{\mathsf{lpv}} + \mathsf{v}.\vec{\mathsf{rpv}}))$ instead, since it is more efficient to compute, and in practice, we can still have a metric tree of depth $O(\log n)$.

Each node v also has a hypersphere $\mathbb{B}$, such that all points represented by v fall in the ball centered at v.center with radius v.r, *i.e.* we have

$$N(\mathsf{v}) \subseteq \mathbb{B}(\mathsf{v.center}, \mathsf{v.r}). \tag{3}$$

Notice that the balls of the two children nodes are not necessarily disjoint.

In fact, v.r is chosen to be the minimal value satisfying (3). As a consequence, we know that the radius of any node is always greater than the radius of any of its children nodes. The leaf nodes have very small radii.

Searching. A search on a metric-tree is simply a guided DFS (for simplicity, we assume that $k = 1$). The decision boundary L is used to decide which child node to search first. If the query q is on the left of L, then v.lc is searched first, otherwise, v.rc is searched first. At all times, the algorithm maintains a "candidate NN", which is the nearest neighbor it finds so far while traversing the tree. We call this point x, and denote the distance between q and x by r. If DFS is about to exploit a node v, but discovers that no member of v can be within distance r of q, then it *prunes* this node (i.e., skip searching on this node, along with all its descendants). This happens whenever $\|\mathsf{v.center} - \mathsf{q}\| - \mathsf{v.r} \geq r$. We call this DFS search algorithm MT-DFS thereafter.

Compared to LSH, metric-trees have some advantages. First, metric trees can easily capture the clustering property the dataset and automatically adapt to the local resolution of the data points. Second, since the partitioning plane (i.e. decision boundary L) of a metric-tree can be along any direction, and they are generated according to the distribution of datapoints.

Third, metric trees automatically decide their degree of partitioning.

In practice, the MT-DFS algorithm is very efficient for NN search, and particularly when the dimension of a dataset is low (say, less than 30). Typically for MT-DFS, we observe an order of magnitude speed-up over naïve linear scan and other popular data structures such as SR-trees. However, MT-DFS starts to slow down as the dimension of the datasets increases. We have found that in practice, metric tree search typically finds a very good NN candidate quickly, and then spends up to 95% of the time verifying that it is in fact the true NN. This motivated our new proposed structure, the *spill-tree*, which is designed to avoid the cost of exact NN verification.

2.2 Spill-Trees

A *spill-tree* (sp-tree) is a variant of metric-trees in which the children of a node can "spill over" onto each other, and contain shared datapoints. The partition procedure of a metric-tree implies that point-sets of v.lc and v.rc are disjoint: these two sets are separated by the decision boundary L. In a sp-tree, we change the splitting criteria to allow overlaps between two children. In other words, some datapoints may belong to both v.lc and v.rc.

We first explain how to split an internal node v. See Figure 4 as an example. Like a metric-tree, we first choose two pivots v.lpv and v.rpv, and find the decision boundary L that goes through the mid point A, Next, we define two new separating planes, LL and LR, both of which are parallel to L and at distance τ from L. Then, all the points to the *right* of plane LL belong to the child v.rc, and all the points to the *left* of plane LR belong to the child v.lc. Mathematically, we have

$$N(\mathsf{v.lc}) = \{x \mid x \in N(\mathsf{v}), d(x, LR) + 2\tau > d(x, LL)\} \quad (4)$$
$$N(\mathsf{v.rc}) = \{x \mid x \in N(\mathsf{v}), d(x, LL) + 2\tau > d(x, LR)\} \quad (5)$$

Notice that points fall in the region between LL and LR are shared by v.lc and v.rc. We call this region the *overlapping buffer*, and we call τ the *overlapping size*. For v.lc and v.rc, we can repeat the splitting procedure, until the number of points within a node is less than a specific threshold, at which point we stop.

3 Approximate Spill-tree-based Nearest Neighbor Search

It may seem strange that we allow overlapping in sp-trees. The overlapping obviously makes both the construction and the MT-DFS less efficient than regular metric-trees, since the points in the overlapping buffer may be searched twice. Nonetheless, the advantage of sp-trees over metric-trees becomes clear when we perform the *defeatist search*, an $(1+\epsilon)$-NN search algorithm based on sp-trees.

3.1 Defeatist Search

As we have stated, the MT-DFS algorithm typically spends a large fraction of time back-tracking to prove a candidate point is the true NN. Based on this observation, a quick revision would be to descends the metric tree using the decision boundaries at each level without backtracking, and then output the point x in the first leaf node it visits as the NN of query q. We call this the *defeatist* search on a metric-tree. Since the depth of a metric-tree is $O(\log n)$, the complexity of defeatist search is $O(\log n)$ per query.

The problem with this approach is very low accuracy. Consider the case where q is very close to a decision boundary L, then it is almost equally likely that the NN of q is on the same side of L as on the opposite side of L, and the defeatist search can make a mistake with probability close to $1/2$. In practice, we observe that there exists a non-negligible fraction of the query points that are close to one of the decision boundaries. Thus the average accuracy of the defeatist search algorithm is typically unacceptably low, even for approximate NN search.

This is precisely the place where sp-trees can help: the defeatist search on sp-trees has much higher accuracy and remains very fast. We first describe the algorithm. For simplicity, we continue to use the example shown in Figure 4. As before, the *decision boundary* at node v is plane L. If a query q is to the left of L, we decide that its nearest neighbor is in v.lc. In this case, we only search points within $N(\mathsf{v.lc})$, i.e., the points to the left of LR. Conversely, if q is to the right of L, we only search node v.rc, i.e. points to the right of LL. Notice that in either case, points in the overlapping buffer are always searched. By introducing this buffer of size τ, we can greatly reduce the probability of making a wrong decision. To see this, suppose that q is to the left of L, then the only points eliminated are the one to the right of plane LR, all of which are at least distance τ away from q. So immediately, if τ is greater than the distance between q and its nearest neighbor, then we *never* make a mistake. In practice, however, τ can be much smaller and the defeatist search still have a very high accuracy.

3.2 Hybrid Sp-Tree Search

One problem with spill-trees is that their depth varies considerably depending on the over-lapping size τ. If $\tau = 0$, a sp-tree turns back to a metric tree with depth $O(\log n)$. On the other hand, if $\tau \geq ||\mathsf{v.rpv} - \mathsf{v.lpv}||/2$, then $N(\mathsf{v.lc}) = N(\mathsf{v.rc}) = N(\mathsf{v})$. In other words, both children of node v contain *all* points of v. In this case, the construction of a sp-tree does not even terminate and the depth of the sp-tree is ∞.

To solve this problem, we introduce *hybrid sp-trees* and actually use them in practice. First we define a *balance threshold* $\rho < 1$, which is usually set to 70%. The constructions of a hybrid sp-tree is similar to that of a sp-tree except the following. For each node v, we first split the points using the overlapping buffer. However, if either of its children contains more than ρ fraction of the total points in v, we undo the overlapping splitting. Instead, a conventional metric-tree partition (without overlapping) is used, and we mark v as a *non-overlapping* node. In contrast, all other nodes are marked as *overlapping nodes*. In this way, we can ensure that each split reduces the number of points of a node by at least a constant factor and thus we can maintain the logarithmic depth of the tree.

The NN search on a hybrid sp-tree also becomes a hybrid of the MT-DFS search and the defeatist search. We only do defeatist search on overlapping nodes, for non-overlapping nodes, we still do backtracking as MT-DFS search.

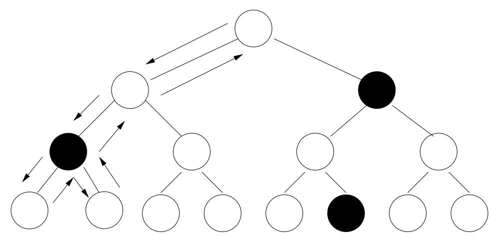

Figure 5: search on a hybrid Sp-tree.

An example of the hybrid search is shown in Figure 5. The white nodes represent *overlapping* nodes, while black nodes are *non-overlapping* nodes. When the hybrid search algorithm hits a white node, it will either go left or right, depending on which side the query point is at, but will not backtrack. On the other hand, when the algorithm hits a black node, no matter which child node it searches first, it will also backtrack to search the other child. In Figure 5, the arrows indicates one possible route the hybrid search algorithm takes. As we can see, if the nodes at the top-levels are overlapping nodes, the hybrid search remains very efficient compared to MT-DFS. Notice that we can control the hybrid by varying τ. If $\tau = 0$, we have a pure sp-tree with defeatist search — very efficient but not accurate enough; if $\tau \geq ||\mathsf{v.rpv} - \mathsf{v.lpv}||/2$, then every node is a non-overlapping node (due to the balance threshold mechanism) — in this way we get back to the traditional metric-tree with MT-DFS, which is perfectly accurate but inefficient. By setting τ to be somewhere in between, we can achieve a balance of efficiency and accuracy. As a general rule, the greater τ is, the more accurate and the slower the search algorithm becomes.

3.3 Further Efficiency Improvement Using Random Projection

The hybrid sp-tree search algorithm is much more efficient than the traditional MT-DFS algorithm. However, this speed-up becomes less pronounced when the dimension of a dataset becomes high (say, over 30). In some sense, the hybrid sp-tree search algorithm also suffer from the curse of dimensionality, only much less severely than MT-DFS.

However, a well-known technique, namely, *random projection* is readily available to deal with the high-dimensional datasets. In particular, the Johnson-Lindenstrauss Lemma [15] states that one can embed a dataset of n points in a subspace of dimension $O(\log n)$ with little distortion on the pair-wise distances. Furthermore, the embedding is extremely simple: one simply picks a *random subspace* S and project all points to S.

In our $(1+\epsilon)$-NN search algorithm, we use random projection as a *pre-processing* step: project the datapoints to a subspace of lower dimension, and then do the hybrid sp-tree search. Both the construction of sp-tree and the search are conducted in the low-dimensional subspace. Naturally, by doing random projection, we will lose some accuracy. But we can easily fix this problem by doing *multiple rounds* of random projections and doing one hybrid sp-tree search for each round. Assume the failure probability of each round is δ, then by doing L rounds, we drive down this probability to δ^L.

The core idea of the hash function used in [10] can be viewed as a variant of random projection.[2] Random projection can be used as a pre-processing step in conjunction with other techniques such as conventional MT-DFSẆe conducted a series of experiments which show that a modest speed-up is obtained by using random projection with MT-DFS (about 4-fold), but greater (up to 700-fold) speed-up when used with sp-tree search. We discuss these results in Section 4.

3.4 More Optimizations

We discuss more optimizations we use in our $(1+\epsilon)$-NN algorithm, namely, *aggressive pruning* and *redundant search*.

Aggressive Pruning This is used to speed up the $(1+\epsilon)$-NN search. It is based on the observation that the traditional MT-DFS algorithm (and part of the hybrid Sp-tree search on non-overlapping nodes) spends about 95% of the time *validating* that the NN candidate is the correct one. For the aggressive pruning, we introduce an additional parameter ϵ, known as the *aggressive factor*. During the search, suppose that the algorithm already has an NN candidate that is distance r from the query point q. When the algorithm is about to explore a new node v, it can estimate a lower bound on the distance between q and the points within v as $r' = \|\mathsf{v.center} - \mathsf{q}\| - \mathsf{v.r}$. We know that if $r' < r$, then there is no need to search node v, i.e., we *prune* node v. In *aggressive pruning*, we change the criterion so that we prune node v whenever $(1+\epsilon)r' < r$. The aggressive pruning is valid since in the worst case, we miss an NN whose distance to q is at least $(1+\epsilon)^{-1}$ of that of the current candidate. So we immediately have an $(1+\epsilon)$-approximate NN search algorithm. In practice, however, we can afford to have a much greater relax factor ϵ without losing too much on the accuracy. On the other hand, the aggressive pruning technique can improve the efficiency by up to a factor of 4.

Redundant Search This is used to improve the accuracy of the search. Suppose we want to find the k-NN of a query point q. After the random projection process, we find k'-NN of q in a new subspace S using Sp-tree search. Clearly there is redundancy when $k' > k$. After finding the k' NNs in the subspace S, we then inspect their preimages in the original space, and find the k-NN within them. Empirically, the running-time penalty for searching for more NNs is quite small (less than 2-fold for $k = 1$, $k' = 15$), but it improves the accuracy by a large margin. In our applications, we typically set k' to be a value between 10 and 30 for $k = 1$.

4 Experimental Results

We report our experimental results based on hybrid sp-tree search on a variety of real-world datasets, with the number of datapoints ranging from 20,000 to 275,465, and dimensions from 60 to 3,838. The first two datasets are same as the ones used in [10], where it is demonstrated that LSH can have a significant speedup over SR-trees.

[2] The Johnson-Lindenstrauss Lemma only works for L_2 norm. The "random sampling" done in the LSH in [10] roughly corresponds to the L_1 version of the Johnson-Lindenstrauss Lemma.

Aerial Texture feature data contain 275,465 feature vectors of 60 dimensions representing texture information of large aerial photographs [18, 17].

Corel_hist 20,000 histograms (64-dimensional) of color thumbnail-sized images taken from the COREL STOCK PHOTO library. However, of the 64 dimensions, only 44 of them contain non-zero entries. See [23] for more discussions. We are unable to obtain the original dataset used in [10] from the authors, and so we reproduced our own version, following their description. We expect that the two datasets to be almost identical.

Corel_uci 68,040 histograms (64-dimensional) of color images from the COREL library. This dataset differs significantly from Corel_hist and is available from the UCI repository [1].

Disk_trace 40,000 content traces of disk-write operations, each being a 1 Kilo-Byte block (therefore having dimension 1,024). The traces are generated from a desktop computer running SuSe Linux during daily operation.

Galaxy Spectra of 40,000 galaxies from the Sloan Digital Sky Survey, with 4000 dimensions.

We also summarize the datasets in Table 4.

Table 1: The datasets

Dataset	**Num.Data**	**Num.Dim**
Aerial	275,465	60
Corel_small	20,000	64
Corel_uci	68,040	64
Disk	40,000	1024
Galaxy	40,000	3838

Besides the sp-tree search algorithm, we also run a number of other algorithms:

LSH The original LSH implementation used in [10] is not public and we were unable to obtain it from the authors. So we used our own efficient implementation. Experiments (described later) show that ours is comparable to the one in [10].

Naïve The naïve linear-scan algorithm.

SR-tree We use the implementation of SR-trees by Katayama and Satoh [16].

Metric-Tree This is highly optimized k-NN search based on metric trees [24, 19], and code is publicly available [2].

The experiments are run on a dual-processor AMD Opteron machine of 1.60 GHz with 8 GB RAM. We perform 10-fold cross-validation on all the datasets. We measure the *CPU time* and *accuracy* of each algorithm. Since all the experiments are memory-based (all the data fit into memory completely), there is no disk access during our experiments. To measure accuracy, we use the *effective distance error* [3, 10], which is defined as $E = \frac{1}{Q}\sum_{q\in Q}\left(\frac{d_{alg}}{d^*} - 1\right)$, where d_{alg} is the distance from a query q to the NN found by the algorithm, and d^* is the distance from q to the true NN. The sum is taken over all queries. For the k-NN case where $(k > 1)$, we measure separately the distance ratios between the closest points found to the nearest neighbor, the 2nd closest one to the 2nd nearest neighbor and so on, and then take the average. Obviously, for all exact k-NN algorithms, $E = 0$, for all approximate algorithms, $E \geq 0$. We also measure the *rank* of all the algorithms.

For every query q, we find its exact 10-NNs. For an $(1 + \epsilon)$-NN algorithm, we check if the approximate NN this algorithm finds is within the exact 10-NN, and we record the (empirical) probability that this happens to a random query. This rank measure turns out to be rather consistent with the effective distance error measure across all algorithms.

4.1 The Experiments

The experiments consist of two parts. The first part, concentrating on the Aerial dataset, aims at understanding the behavior of sp-Trees. The second part, using all five datasets, compares the performance of various algorithms.

4.1.1 Understanding sp-trees Using the Aerial Dataset

Random Projection The first batch of experiments are aimed at understanding the effectiveness of the random projection, used as a pre-processing step of our algorithm. We fix the overlapping size τ to be ∞, effectively turning the algorithm into the traditional MT-DFS algorithm. We fix the number of loops L to be 1. We vary the *projected dimension* d from 5 to 60, and then measure the distance error E, the CPU time, and the number of distance computations (which accounts for most of the CPU time). The results are shown in Figure 6.

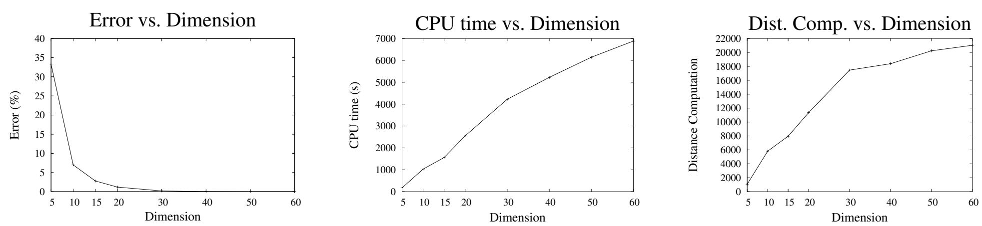

Figure 6: influence of the random projection (Aerial, $D = 60$, $n = 275,476$, $k = 1$).

As expected, the error E is large when the dimension is low: it exceeds 30% when $d = 5$. However, E decreases very fast as d increases: when $d = 30$, E is less than 0.2%, which is more than acceptable for most applications. This also suggests that the *intrinsic dimension* of Aerial can be quite small (at least smaller than 30).

The CPU time decreases almost linearly with d. However, the slope becomes slightly steeper when $d < 30$, indicating sublinear dependence of CPU time on d. This is more clearly illustrated in the number of distance computations: a sharp drop after $d < 30$. Interestingly, we observe the same behavior(that the CPU time becomes sublinear and the number of distance computation drops dramatically when $d < 30$) in all five datasets. This observation suggests that for metric-trees, the "curse of dimensionality" occurs when d exceeds 30. As a consequence, searching algorithms based on metric-trees (including the hybrid sp-tree search) performs much better when $d < 30$.[3] Empirically, for all five datasets we tested, our algorithm achieves optimal performance almost always when $d < 30$.

Loops We investigate the influence of number of loops L. We fix $\tau = \infty$ and we test three projected dimensions: $d = 5, 10, 15$. See Figure 7.

As expected, all cases, the CPU time scales linearly with L, and the distance error E decreases with L. However, the rate that E decreases slows down as L increases. Therefore, for the trade-off between L and d, it is typically more economical to have a single loop

[3]sp-trees still suffer from the curse of dimensionality, but the degree is much less severe than metric-trees

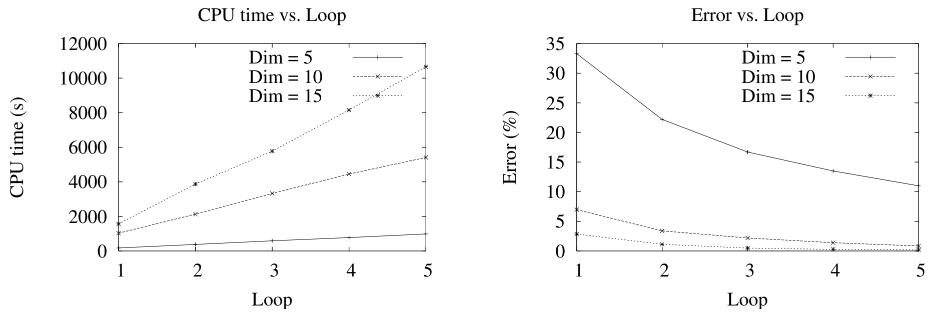

Figure 7: influence of the loop L (Aerial, $D = 60, n = 275,476, k = 1$).

with a large d than having a larger L and a smaller d, unless d is very small. Very roughly speaking, one would expect about the same accuracy if we keep $L \cdot d$ to be a constant — Notice that when $d > 30$, the CPU time scales linearly with both L and d, but E decreases much faster with d increasing than with L. However, for very low dimensions ($d < 10$), since CPU time scales sub-linearly with d, it might be worthwhile to choose a small d and $L > 1$.

The Overlapping Size τ We test the influence of the overlapping size τ. Again, we fix $L = 1$ and test three projected dimension: $d = 5, 10, 15$. The results are shown in Figure 8.

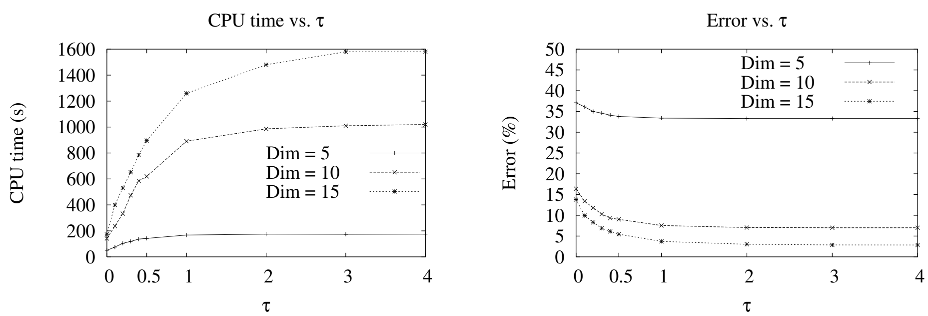

Figure 8: influence of the overlapping size τ (Aerial, $D = 60, n = 275,476, k = 1$).

As we have mentioned before, both the accuracy and the CPU time increases with τ. When $\tau = 0$, we have pure defeatist search; when τ is large enough (in this case, when $\tau > 3$), the search becomes MT-DFS, which is slow but perfectly accurate.[4] In the "interesting" range where $0 < \tau < 1$, we see dramatic decrease in CPU time and modest increase in E.

As an specific example, the MT-DFS on the original dataset runs in 402s with perfect accuracy. By projecting to a 10-dimensional space, the error becomes 7%, and MT-DFS runs in 103s. This is a 3.9-fold speed-up. Then by setting $\tau = 0$, we switch defeatist search, and the error increases to 16%, while the CPU time decreases to 14.1s, a 7.3-fold speed-up. By employing the aggressive pruning technique (setting the aggressive factor $\epsilon = 3.6$), the CPU time decreases to 3.33s. We have an additional 4.2-fold speed-up at the expense that E further increases to 20%. Overall, these techniques combined give us a 121-fold speed-up over the tradition MT-DFS with error being 20%.

[4]The "residue" errors in Figure 8 in the case $\tau = 4$ comes from random projection (c.f. Figure 6).

A word about choosing τ. We need to set τ to a "right" value (at least in the right order of magnitude) in order to utilize the full power of sp-tree search. Notice that if τ is greater than the distance between query q and its NN, then the sp-tree search is perfectly accurate, but at the same time it slows down the search. Empirically, the optimal value of τ is found near the average NN-distance (the average distance between a random point and its NN). In practice, we suggest the following heuristic: take a small sample of the datapoints and estimate the average NN-distance r. Then the optimal τ is typically within the range of $[0.1r, r]$ and we can find it by doing a sequence of experiments on small subsets. All these can be done at the training phase and do not affect the search time.

4.1.2 Comparison of Algorithms

First, as a benchmark, we run the Naïve, SR-tree, and the Metric-tree algorithms. All of them find exact NN. The results are summarized in Table 2.

Table 2: the CPU time of exact SR-tree, Metric-tree, and Naïve search

Algorithm (%)	Aerial	Corel_hist ($k=1$)	Corel_hist ($k=10$)	Corel_uci	Disk_trace	Galaxy
Naive	43620	462	465	5460	27050	46760
SR-tree	23450	184	330	3230	n/a	n/a
Metric-tree	3650	58.4	91.2	791	19860	6600

All the datasets are rather large, and the metric-tree is consistently the fastest. On the other hand, the SR-tree implement only has limited speedup over the Naïve algorithm, and it fails to run on Disk_trace and Galaxy, both of which have very high dimensions.

Then, for approximate NN search, we compare sp-tree with three other algorithms: LSH, traditional Metric-tree and SR-tree. For each algorithm, we measure the CPU time needed for the error E to be 1%, 2%, 5%, 10% and 20%, respectively. Since Metric-tree and SR-tree are both designed for exact NN search, we also run them on randomly chosen subsets of the whole dataset to produce approximate answers. The comparison results are summarized in Figure 9. We also examine the speed-up of sp-tree over other algorithms. In particular, the CPU time and the speedup of sp-tree search over LSH is summarized in Table 3.

Table 3: the CPU time (s) of Sp-tree and its speedup (in parentheses) over LSH

Error (%)	Aerial	Corel_hist ($k=1$)	Corel_hist ($k=10$)	Corel_uci	Disk_trace	Galaxy
20	33.5 (31)	1.67 (8.3)	3.27 (6.3)	8.7 (8.0)	13.2 (5.3)	24.9 (5.5)
10	73.2 (27)	2.79 (9.8)	5.83 (7.0)	19.1 (4.9)	43.1 (2.9)	44.2 (7.8)
5	138 (31)	4.5 (11)	9.58 (6.8)	33.1 (4.8)	123 (3.7)	76.9 (11)
2	286 (26)	8 (9.5)	20.6 (4.2)	61.9 (4.4)	502 (2.5)	110 (14)
1	426 (23)	13.5 (6.4)	27.9 (4.1)	105 (4.1)	1590 (3.2)	170 (12)

Since we used our own implementation of LSH, we first need to verify that it has comparable performance as the one used in [10]. We do so by examining the speedup of both implementations over SR-tree on the Aerial and Corel_hist datasets, with both $k = 1$ and $k = 10$ for the latter.[5] For the Aerial dataset, in the case where E varies from 10% to 20%,

[5] The comparison in [10] is on disk access while we compare CPU time. So strictly speaking, these results are not comparable. Nonetheless we expect them to be more or less consistent.

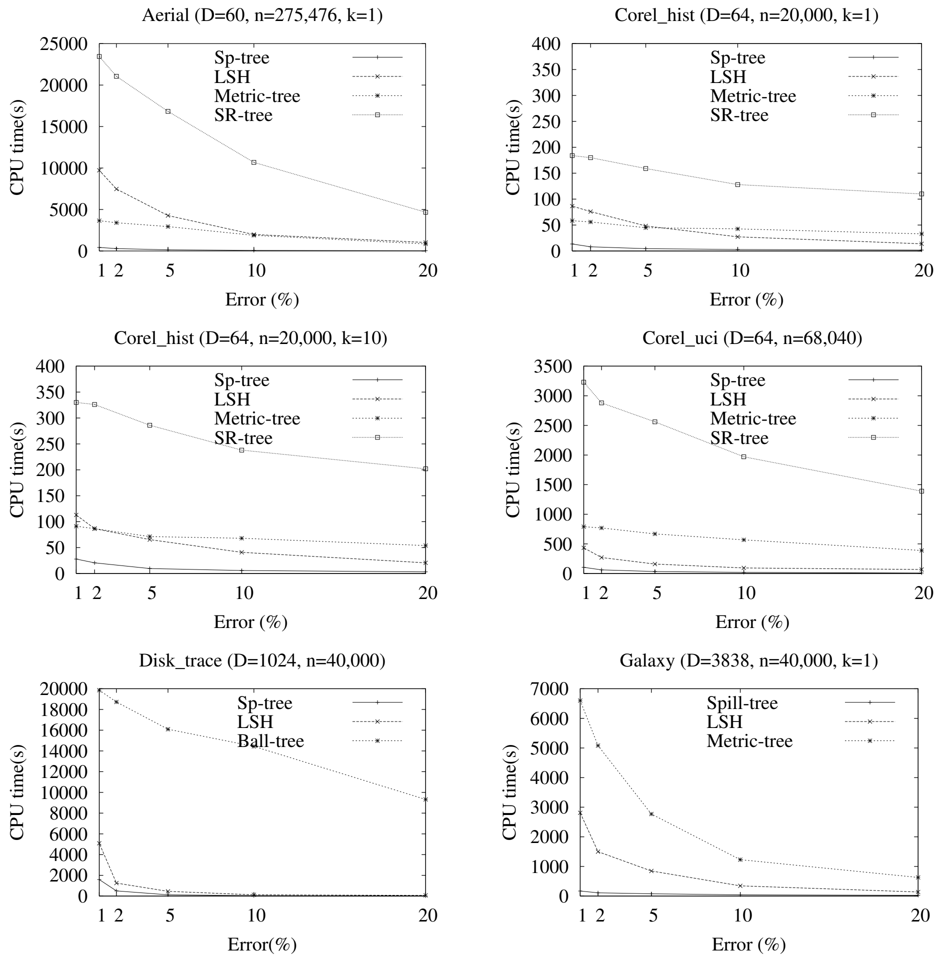

Figure 9: CPU time (s) vs. Error (%) for all datasets.

the speedup of LSH in [10] over SR-tree varies from 4 to 6, and as for our implementation, the speedup varies from 4.5 to 5.4. For Corel_hist, when E ranges from 2% to 20%, in the case $k = 1$, the speedups of LSH in [10] ranges from 2 to 7, ours from 2 to 13. In the case $k = 10$, the speedup in [10] is from 3 to 12, and ours from 4 to 16. So overall, our implementation is comparable to, and often outperforms the one in [10].

Perhaps a little surprisingly, the Metric-tree search algorithm (MT-DFS) performs very well on Aerial and Corel_hist datasets. In both cases, when the E is small (1%), MT-DFS outperforms LSH by a factor of up to 2.7, even though it aims at finding the *exact* NN, while LSH only finds an *approximate* NN. Furthermore, the *approximate* MT-DFS algorithm (conventional metric-tree based search using a random subset of the training data) consistently outperforms LSH across the entire error spectrum on Aerial. We believe that it is because that in both datasets, the *intrinsic* dimensions are quite low and thus the Metric-tree does not suffer from the curse of dimensionality. Indeed, Aerial has dimension 60, but a random projection to a subspace of dimension 30 virtually does not hurt the accuracy as all (c.f. Figure 6); Corel_hist has dimension 64, but only 44 of them contain non-zero entries.

For the rest of the datasets, namely, Corel_uci, Disk_trace, and Galaxy, metric-tree becomes rather inefficient because of the curse of dimensionality, and LSH becomes competitive. But in all cases, sp-tree search remains the fastest among all algorithms, frequently achieving 2 or 3 orders of magnitude in speed-up. Space does not permit a lengthy conclusion, but the summary of this paper is that there is empirical evidence that with appropriate redesign of the data structures and search algorithms, spatial data structures remain a useful tool in the realm of approximate k-NN search.

We also compare the sp-tree search with approximate MT-DFS (see Table 4). In this case, we observe a speedup of from 3.3 to 706. Almost always, the speedup is more pronounced when E is large, and as E decreases, the benefit of sp-tree becomes less significant. This is expected, since the techniques sp-tree uses (random projection, defeatist search) introduce certain errors. By insisting on a very small error, we have to limit the extent these techniques are used and at the same time, constraining their powers.

Table 4: the CPU time speedup of Sp-tree over Metric-tree

Error (%)	Aerial	Corel_hist ($k=1$)	Corel_hist ($k=10$)	Corel_uci	Disk_trace	Galaxy
20	25	20	16	45	706	25
10	26	15.3	12	30	335	28
5	21	10	7.4	20	131	36
2	12	7.0	4.2	12	37.2	46
1	8.6	4.3	3.3	7.5	12.4	39

References

[1] http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html.

[2] http://www.autonlab.org/autonweb/showsoftware/154/.

[3] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[4] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? *Lecture Notes in Computer Science*, 1540:217–235, 1999.

[5] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB International Conference*, September 1997.

[6] K. Clarkson. Nearest Neighbor Searching in Metric Spaces: Experimental Results for sb(S). , 2002.

[7] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons, 1973.

[8] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[9] B. Georgescu, I. Shimshoni, and P. Meer. Mean shift based clustering in high dimensions: A texture classification example. In *Proceedings of the 9th IEEE International Conferences on Computer Vision (ICCV 2003)*, 2003.

[10] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proc 25th VLDB Conference*, 1999.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. Assn for Computing Machinery, April 1984.

[12] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[13] P. Indyk and N. Thaper. Fast image retrieval via embeddings. In *the 3rd International Workshop on Statistical and Computational Theories of Vision (SCTV 2003)*, 2003.

[14] Piotr Indyk. *High Dimensional Computational Geometry*. PhD. Thesis, 2000.

[15] W. Johnson and J. Lindenstrauss. Extensions of lipschitz maps into a hilbert space. *Contemp. Math.*, 26:189–206, 1984.

[16] Norio Katayama and Shin'ichi Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. pages 369–380, 1997.

[17] B. S. Manjunath. Airphoto dataset, http://vivaldi.ece.ucsb.edu/Manjunath/research.htm.

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

[19] A. W. Moore. The Anchors Hierarchy: Using the Triangle Inequality to Survive High-Dimensional Data. In *Twelfth Conference on Uncertainty in Artificial Intelligence*. AAAI Press, 2000.

[20] S. M. Omohundro. Efficient Algorithms with Neural Network Behaviour. *Journal of Complex Systems*, 1(2):273–347, 1987.

[21] S. M. Omohundro. Bumptrees for Efficient Function, Constraint, and Classification Learning. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*. Morgan Kaufmann, 1991.

[22] F. P. Preparata and M. Shamos. *Computational Geometry*. Springer-Verlag, 1985.

[23] Y. Rubnet, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000.

[24] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.