An Improved Algorithm Finding Nearest Neighbor Using Kd-trees

Rina Panigrahy

Microsoft Research, Mountain View CA, USA
rina@microsoft.com

Abstract. We suggest a simple modification to the Kd-tree search algorithm for nearest neighbor search resulting in an improved performance. The Kd-tree data structure seems to work well in finding nearest neighbors in low dimensions but its performance degrades even if the number of dimensions increases to more than two. Since the exact nearest neighbor search problem suffers from the curse of dimensionality we focus on approximate solutions; a c-approximate nearest neighbor is any neighbor within distance at most c times the distance to the nearest neighbor. We show that for a randomly constructed database of points if the query point is chosen close to one of the points in the data base, the traditional Kd-tree search algorithm has a very low probability of finding an approximate nearest neighbor; the probability of success drops exponentially in the number of dimensions d as $e^{-\Omega(d/c)}$. However, a simple change to the search algorithm results in a much higher chance of success. Instead of searching for the query point in the Kd-tree we search for a random set of points in the neighborhood of the query point. It turns out that searching for $e^{\Omega(d/c)}$ such points can find the c-approximate nearest neighbor with a much higher chance of success.

1 Introduction

In this paper we study the problem of finding the nearest neighbor of a query point in a high dimensional (at least three) space focusing mainly on the Euclidean space: given a database of n points in a d dimensional space, find the nearest neighbor of a query point. This fundamental problem arises in several applications including data mining, information retrieval, and image search where distinctive features of the objects are represented as points in $\mathbb{R}^d$ [28, 30, 6, 7, 12, 11, 27, 10].

One of the earliest data structures proposed for this problem that is still the most commonly used is the Kd-tree [3] that is essentially a hierarchical decomposition of space along different dimensions. For low dimensions this structure can be used for answering nearest neighbor queries in logarithmic time and linear space. However the performance seems to degrade as a number of dimensions becomes larger than two. For high dimensions, the exact problem of nearest neighbor search seems to suffer from the curse of dimensionality; that is, either the running time or the space requirement grows exponentially

E.S. Laber et al. (Eds.): LATIN 2008, LNCS 4957, pp. 387–398, 2008.
© Springer-Verlag Berlin Heidelberg 2008

in d. For instance Clarkson [5] makes use of $O(n^{\lceil d/2 \rceil (1+\delta)})$ space and achieves $O(2^{O(d \log d)} \log n)$ time. Meiser [24] obtains a query time of $O(d^5 \log n)$ but with $O(n^{d+\delta})$ space.

The situation is much of a better for finding an approximate solution whose distance from the query point is at most $1 + \epsilon$ times its distance from the nearest neighbor [2, 21, 18, 22]. Arya et. al. [2] use a variant of Kd-trees that they call BDD-trees (Balanced Box-Decomposition trees) that performs $(1 + \epsilon)$-approximate nearest neighbor queries in time $O(d\lceil 1 + 6d/\epsilon \rceil^d \log n)$ and linear space. For arbitrarily high dimensions, Kushilevitz et. al. [22] provide an algorithm for finding an $(1 + \epsilon)$-approximate nearest neighbor of a query point in time $\tilde{O}(d \log n)$ using a data structure of size $(nd)^{O(1/\epsilon^2)}$. Since the exponent of the space requirement grows as $1/\epsilon^2$, in practice this may be prohibitively expensive for small ϵ. Indeed, since even a space complexity of $(nd)^2$ may be too large, perhaps it makes more sense to interpret these results as efficient, practical algorithms for c-approximate nearest neighbor where c is a constant greater than one. Note that if the gap between the distance to the nearest neighbor and to any other point is more than a factor of c then the c-approximate nearest neighbor is same as the nearest neighbor. So in such cases – which may very well hold in practice – these algorithms can indeed find the nearest neighbor.

Indyk and Motwani [18] provide results similar to those in [22] but use hashing to perform approximate nearest neighbor search. They provide an algorithm for finding the c-approximate nearest neighbor in time $\tilde{O}(d + n^{1/c})$ using an index of size $\tilde{O}(n^{1+1/c})$ (while their paper states a query time of $\tilde{O}(dn^{1/c})$, if d is large this can easily be converted to $\tilde{O}(d + n^{1/c})$ by dimension reduction). In their formulation, they use a locality sensitive hash function that maps points in the space to a discrete space where nearby points out likely to get hashed to the same value and far off points out likely to get hashed to different values. Precisely, given a parameter m that denotes the probability that two points at most r apart hash to the same bucket and g the probability that two points more than cr apart hash to the same bucket, they show that such a family of hash functions can find a c-approximate nearest neighbor in $\tilde{O}(d + n^{\rho})$ time using a data structure of size $\tilde{O}(n^{1+\rho})$ where $\rho = log(1/m)/log(1/g)$. Recently, Andoni and Indyk [1] obtained and improved locality sensitive hash function for the Euclidean norm resulting in a ρ value of $O(1/c^2)$ matching the lower bounds for locality sensitive hashing method from [25]. An information theoretic formulation of locality sensitive hashing was studied in [26] resulting in a data structure of linear size; the idea there was to perturb the query point before searching for it in the hash table and do this for a few iterations.

However, to the best of our knowledge the most commonly used method in practice is the old Kd-tree. We show that a simple modification to the search algorithm on a Kd-tree can be used to find the nearest neighbor in high dimensions more efficiently. The modification consists of simply perturbing the query point before traversing the tree, and repeating this for a few iterations. This is essentially the same idea from [26] on locality sensitive hash functions applied to Kd-trees. For a certain database of random points if we choose a query point

close to one of the points in the database, we show that the traditional Kd-tree search algorithm has a very low probability of finding the nearest neighbor – $e^{-\Omega(d/c)}$ where c is a parameter that denotes how much closer the query point is to the nearest neighbor than to other points in the database. Essentially c is the inverse ratio of the distance of the nearest query point to the nearest and the second nearest neighbor; so one can think of the nearest neighbor as a c-approximate nearest neighbor. Next we show that the modified algorithm significantly improves the probability of finding the c-approximate nearest neighbor by performing $e^{O(d/c)}$ iterations. One may be tempted to think that if the traditional algortithm has a success probability of $e^{-\Omega(d/c)}$, perhaps we could simply repeat it $e^{O(d/c)}$ times to boost the probability to a constant. However, this doesn't work in our situation since repeating the same query in a tree will always give the same result. One can use a different (randomly constructed) tree each time but this will blow up the space requirement to that of $e^{O(d/c)}$ trees. Our result essentially shows how to boost the probability while using one tree but by perturbing the query point each time. The intuition behind this approach is that when we perturb the query point and then search the tree, we end up looking not only at one leaf region but also the neighboring leaf regions that are close to the query point thus increasing the probability of success. This is similar in spirit to some of the variants of Kd-trees such as [2] that maintain explicit pointers from each leaf to near by leaves; our method on the other hand performs this implicity without maintaining pointers which keep the data structure simple. We also provide empirical evidence through simulations to show that the simple modification results in high probability of success in finding nearest neighbor search in high dimensions.

We apply the search algorithms on a planted instance of the nearest neighbor problem on a database of n points chosen randomly in a unit d-dimensional cube. We then plant a query point close of one of the database points p (chosen randomly) and then ask the search algorithm for the nearest neighbor of our planted query point. The query point is chosen randomly on a ball of a certain radius around p so that it is much closer to p than to any other point in the database – by a factor of c. We measure the efficacy of an algorithm by looking at the probability that it returns the nearest neighbor p on query q. It is for this distribution that we will show a low success probability for the traditional Kd-tree search algorithm and a high probability of success for our modified algorithm.

In our experiments we observed that our modified algorithm indeed boosts the probability of success. For instance in a database of million points in 3 dimensions we plant a query point close to a random database point; the query point is closer to its nearest neighbor than any other point by a factor of $c = 2$. For this instance we find that the Kd-tree search algorithm succeeds with probability 74%; whereas, this can be boosted to about 90% byrunning our modified algorithm with only 5 iterations. The success probability increases with more iterations.

2 Preliminaries

2.1 Problem Statement

Given a set S of n points in d-dimensions, our objective is to construct a data structure that given a query finds the nearest (or a c-approximate) neighbor. A c-approximate near neighbor is a point at distance at most c times the distance to the nearest neighbor. Alternatively it can be viewed as finding the exact nearest neighbor when the second nearest neighbor is more than c times the distance to the nearest neighbor.

We also work with the following decision version of the c-approximate nearest neighbor problem: given a query point and a parameter r indicating the distance to its nearest neighbor, find any neighbor of the query point that is that distance at most cr. We will refer to this decision version as the (r, cr)-nearest neighbor problem and a solution to this as a (r, cr)-nearest neighbor. It is well known that the reduction to the decision version adds only a logarithmic factor in the time and space complexity [18,13]. We will be working with the euclidian norm in $\mathbb{R}^d$ space.

2.2 Kd-trees

Although many different flavors of Kd-trees have been devised, their essential strategy is to hierarchically decompose space into a relatively small number of cells such that no cell contains too many input objects. This provides a fast way to access any input object by position. We traverse down the hierarchy until we find the cell containing the object. Typical algorithms construct Kd-trees by partitioning point sets recursively along with different dimensions. Each node in the tree is defined by a plane through one of the dimensions that partitions the set of points into left/right (or up/down) sets, each with half the points of the parent node. These children are again partitioned into equal halves, using planes through a different dimension. Partitioning stops after $\log n$ levels, with each point in its own leaf cell. The partitioning loops through the different dimensions for the different levels of the tree, using the median point for the partition. Kd-trees are known to work well in low dimensions but seem to fail as the number of dimensions increase beyond three.

Notations:

- $B(p, r)$: Let $B(p, r)$ denote the sphere of radius r centered at p a point in $\mathbb{R}^d$; that is the set of points at distance r from p.
- $I(X)$: For a discrete random variable X, let $I(X)$ denote its information-entropy. For example if X takes N possible values with probabilities $w_1, w_2, ..., w_N$ then $I(X) = I(w_1, w_2, .., w_N) = \sum I(w_i) = \sum -w_i \log w_i$. For a probability value p, we will use the overloaded notation $I(p, 1-p)$ to denote $-p \log p - (1-p) \log(1-p)$
- $N(\mu, r), \eta(x)$: Let $N(\mu, r)$ denote the normal distribution with mean μ and variance r^2 with probability density function given by $\frac{1}{r\sqrt{2\pi}} e^{-(x-\mu)^2/(2r^2)}$. Let $\eta(x)$ denote the function $\frac{1}{\sqrt{2\pi}} e^{-x^2/2}$.

- $N^d(p,r)$: For the d-dimensional Euclidean space, for a point $p = (p_1, p_2, ..., p_d) \in \mathbb{R}^d$ let $N^d(p,r)$ denote the normal distribution in $\mathbb{R}^d$ around the point p where the ith coordinate is randomly chosen from the normal distribution $N(p_i, r/\sqrt{d})$ with mean p_i and variance r^2/d. It is well known that this distribution is spherically symmetric around p. A point from this distribution is expected to be at root-mean squared distance r from p; in fact, for large d its distance from p is close to r with high probability (see for example lemma 6 in [18])
- $erf(x), \Phi(x)$: The well-known error function $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-x^2}\, dx$, is equal to the probability that a random variable from $N(0, 1/\sqrt{2})$ lies between $-x$ and x. Let $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-x^2/2}\, dx = \frac{1-erf(x/\sqrt{2})}{2}$. For $x \geq 0$, $\Phi(x)$ is the probability that a random variable from the distribution $N(0,1)$ is greater than x.
- $Pinv(k,r)$: Let $Pinv(k,r)$ denote the distribution of the time it takes to see k events in a poisson process of rate r. $Pinv(1,r)$ is the exponential distribution with rate r.

3 An Improved Search Algorithm on Kd-trees

We will study the nearest neighbor search problem on the following planted instance in a random database. Consider a database of points chosen randomly and uniformly from the unit cube $[0,1]^d$. We will then choose a random database point p and implant query point q that is about c times closer to p than its distance to any other database point. Success of an algorithm is measured by the probability of finding the nearest neighbor. Let r denote the distance of a random point in $[0,1]^d$ to the nearest database point. We will show that if the query point is chosen to be a random point at distance about r/c around p (precisely, q is chosen from $N^d(p, r/c)$) then the probability that a Kd-tree search algorithm reports the nearest neighbor p is about $e^{-\Omega(d/c)}$.

We then propose a minor modification to the search algorithm so as to boost this probability to a large value. The modification is simple: Instead of searching for the query point q in the Kd-tree, perturb it randomly by distance r/c (precisely, the perturbed point is chosen from $N^d(q, r/c)$) and search for this perturbed point in the Kd-tree using the regular Kd-tree search algorithm. Repeat this for $e^{O(d/c)}$ random perturbations and report the nearest neighbor found using the Kd tree. We will show that this simple modification results in an a much higher chance of finding the nearest neighbor p.

4 Traditional Kd-tree Algorithms Fails with High Probability

In this section we show that the traditional Kd-tree algorithm fails in solving the planted instance of the nearest neighbor problem with high probability; the

success probability is at most $e^{-\Omega(d/c)}$. The following lemma estimates r, the distance from a random point in the unit cube to the nearest database point.

Lemma 1. *With high probability of $1 - O(1/2^d)$, the distance r of a random point in the unit cube to the nearest point in the database is $\Theta(\sqrt{d}/n^{1/d})$*

Proof. The volume of a sphere of radius r in d dimensions is $\frac{2\pi^{d/2}r^d}{d\Gamma(d/2)} = \Theta(1)\frac{1}{d^{3/2}}(\frac{2e\pi}{d})^{d/2}r^d$. Since there are n random points in the database, the expected number of points in a ball of radius r is $\Theta(1)n\frac{1}{d^{3/2}}(\frac{2e\pi}{d})^{d/2}r^d$. So the radius for which the expected number of points in this ball is one is given by $r = \Theta(\frac{\sqrt{d}}{n^{1/d}})$ (We have used the Stirlings approximation for the Γ function which is valid only for integral arguments. Although, $d/2$ may not be integral it lies between two consecutive integers and this proves the Θ bound on r. Note that even if the point is near the boundary of the cube, at least $1/2^d$ fraction of the sphere is inside the cube which does not change the value of r by more than a constant factor). The value of r is sharply concentrated around this value as the volume of the sphere in high dimensions changes dramatically with a change in the radius; by a factor of 2^d with a factor 2 change in the radius. High concentration bounds can be used to show that there must be at least one point for larger radii and almost no point for smaller radii. For this value of r, the probability that there is a database point at distance $r/2$ is at most $n1/(2^d n) = 1/2^d$. And the probability that there is no point within distance $2r$ is atmost $(1 - 2^d/n)^n = exp(-2^d)$.

Lemma 2. *If we pick a random query point q at distance r/c from a random point in the database then the probability that a Kd-tree search returns the nearest neighbor is at most $e^{-\Omega(d/c)}$.*

Proof. Let us focus on a leaf cell containing the point p. We need to compute the probability that a random query point q chosen from $N(p, r/c)$ lies within the same cell.

If we project the cell along any one dimension we get an interval. Since the Kd-tree has depth $\log n$, the number of branches along any one dimension is $\frac{\log n}{d}$. And since we are picking a random cell, the expected value of the length l of this interval containing p is $E[l] = 1/2^{\frac{\log n}{d}} = 1/n^{1/d}$. So with probability at least $1/2$, $l < 2/n^{1/d}$. Conditioned on the event that $l < 2/n^{1/d}$, we will argue that the probability that the query point q lies outside the interval is $\Omega(1/c)$. To see this note that if x denotes the distance of p from one of the end points of the interval then x is distributed uniformly in the range $[0, l]$ since p is a random point in its cell. Since along one dimension q and p are separated by a distance of $N(0, \frac{r}{c\sqrt{d}}) = N(0, \frac{1}{cn^{1/d}})$ (ignoring the Θ in the expression for r for simpler notation), the probability that q does not line the interval is $\Phi(xcn^{1/d})$. By a standard change of variable to $z = xcn^{1/d}$ this amounts to the expected value of $\Phi(z)$ where z is uniformly distributed in the range $[0, 2c]$. Looking at integral values of z, observe that $\Phi(z) = e^{-\Omega(z^2)}$ drops at least geometrically with each increment of z. So the expected value of $\Phi(z)$ is $\int_0^{2c} \frac{1}{2c} e^{-\Omega(z^2)}\, dz = \Omega(1/c)$.

This implies that along any one dimension the probability that q lies within the projection along that dimension of the cell containing p is at most $1 - \Omega(1/c)$. Since values along all dimensions are independent, the probability that q lies within the cell of p is at most $(1 - \Omega(1/c))^d = e^{-\Omega(d/c)}$.

5 Modified Algorithm has a Higher Probability of Success

We will show that the modified algorithm has a much higher probability of success. Although our theoretical guarantee only provides a success probability of $\Omega(c/d)$, we believe this is simply an artifact of our analysis. Our experimental results show that the success probability is very high.

Theorem 1. *With probability at least $\Omega(c/d)$, the new search algorithm finds the nearest neighbor in $e^{O(d/c)}$ iterations for the random database and the planted query point.*

The proof of this theorem follows from the following two lemmas.

Guessing the value of a random variable: We first present a lemma that states the number of times one needs to guess a random variable with a given distribution so as to guess its correct value. If a random variable takes one of N discrete values with equal probability then a simple coupon collection based argument shows that if we guess N random values at least one of them should hit the correct value with constant probability. The following lemma taken from [26] states the required number of samples for arbitrary random variables so as to 'hit' a given random value of the variable.

Lemma 3. *[26] Given an random instance x of a discrete random variable with a certain distribution D with entropy I, if $O(2^I)$ random samples are chosen from this distribution at least one of them is equal to x with probability at least $\Omega(1/I)$.*

Proof. Assume that the distribution D takes N values with probabilites $w_1, w_2, ..., w_N$. x is equal to the ith value with probability w_i. If s samples are randomly chosen from the distribution, the probability that at least one of them takes the ith value is $1 - (1 - w_i)^s$. After $s = 4.(2^I + 1)$ samples the probability that x is chosen is $\sum_i w_i[1 - (1 - w_i)^s]$. If $w_i \geq 1/s$ then the term in the summation is at least $w_i(1 - 1/e)$. Divide the probability values $w_1, w_2, ..., w_N$ into two parts – those at least $1/s$ and the others less than $1/s$. So if all the $w_i's$ that are at least $1/s$ add up to at least $1/I$ then the above sum is at least $\Omega(1/I)$. Otherwise we have a collection of $w_i's$ each of which is at most $1/s$ and they together add up to more than $1 - 1/I$.

But then by paying attention to these probabilities we see that the entropy $I = \sum_i w_i \log(1/w_i) \geq \sum_i w_i \log s \geq (1 - 1/I) \log s \geq (1 - 1/I)(I + 2) = I + 1 - 2/I$. For $I \geq 4$, this is strictly greater than I, which is a contradiction. If $I < 4$ then the largest w_i must be at least $1/16$ as otherwise a similar argument shows that

$I = \sum_i w_i \log(1/w_i) > w_i \log 16 = 4$, a contradiction; so in this case even one sample guesses x with constant probability.

Distribution of Kd-tree partition sizes: Let us now estimate the distribution of the Kd-tree partition sizes as we walk down the tree. Consider a random point p in the database and the Kd-tree traversal while searching p. As we walk down the tree the cell containing p shrinks from the entire unit cube to the single leaf cell containing p. The leaf cell of p is specified by $\log n$ choices of left or right while traversing the Kd-tree. Let us track the length of the cell along a dimension as we walk down the tree. The number of decisions along each dimension is $\frac{\log n}{d}$. Focusing on the first dimension (say x-axis), look at the interval along the x-axis containing p. The interval gets shorter as we partition along the x-axis. Let l_i denote the length of this interval after the i^{th} branch along the x-axis; note that two successive branches along the x-axis are separated by $d-1$ branches along the other dimensions. The initial cell length $l_0 = 1$. $E[l_i] = 1/2^i$; let us look at the distribution of l_i. The database points projected along the x-axis gives us n points randomly distributed in $[0,1]$. For large n, any two successive values are separated by an exponential distribution with rate $1/n$, and the distance between k consecutive points is given by $Pinv(k, 1/n)$, the inverse poisson distribution at rate $1/n$ for k arrivals. The median point is the first partition point dividing the interval into two parts containing $n/2$ points each. p lies in one of these intervals of length l_1 distributed as $Pinv(n/2, 1/n)$.

To find the distribution of l_2, note that there are $d-1$ branches along other dimensions between the first and the second partition along the x-axis, and each branch eliminates $1/2$ the points from p's cell. Since coordinate values along different dimensions are independent, this corresponds to randomly sampling $1/2^{d-1}$ fraction of the points from the interval after the first branch. From the points that are left, we partition again along the median and take one of the two parts; each part contains $n/2^d$ points. Since the original n points are chosen randomly and we sample at rate $1/2^{d-1}$, after the second branch successive points are separated by an exponential distribution with rate $1/2^{d-1}$. So after the second branch along x-axis the side of the interval l_2 is distributed as $Pinv(n/2^d, 1/2^{d-1})$. Note that this is not entirely accurate since the points are not sampled independently but instead exactly $1/2^{d-1}$ fraction of the points are chosen in each part; however thinking of n and 2^d as large we allow ourselves this simplification.

Continuing in this fashion, we get that the interval corresponds to l_i contains $n/2^{di}$ points and the distance between successive points is distributed as the exponential distribution with rate $2^{(d-1)i}/n$. So l_i is distributed as $Pinv(n/2^{di}, 2^{(d-1)i}/n)$. Since p is a random point in its cell, the distance x_i between p and the dividing plane at the i^{th} level is a random value between 0 and l_i. At the leaf level $l_{\log n/d}$ is distributed as $Pinv(1, 2^{\frac{(d-1)\log n}{d}}/n)$ which is same as the exponential distribution with rate $2^{\frac{(d-1)\log n}{d}}/n = 1/n^{1/d}$. Extending this argument to other dimensions tells us that at the leaf level the length of the cell along any dimension is given by an exponential distribution with rate $O(1/n^{1/d})$.

Now p is a random database point and q is a random point with distribution $N^d(p, r/c)$. Let $L(p)$ denote the leaf of the Kd-tree (denoted by T) where p resides. For a fixed Kd-tree, look at the distribution of $L(p)$ given q. We will estimate $I[L(p)|q, T]$ – the information required to guess the cell containing p given the query point q and the tree structure T (the tree structure T includes positions of the different partition planes in the tree).

Lemma 4. $I[L(p)|q, T] = O(d/c)$

Proof. The cell of p is specified by $\log n$ choices of left or right while traversing the Kd-tree. The number of decisions along each dimension is $\frac{\log n}{d}$. Let b_{ij} ($i \in 1 \cdot\cdot \log n/d, j \in 0 \cdot\cdot d-1$) denote this choice in the i^{th} branch along dimension j. Let B_{ij} denote the set of all the previous branches on the path to the branch point b_{ij}. Then since the leaf cell $L(p)$ is completely specified by the branch choices b_{ij}, $I[L(p)|q, T] \leq \sum_{i,j} I[b_{ij}|B_{ij}, q, T]$. Let us now bound $I[b_{ij}|B_{ij}, q, T]$. Focusing on the first dimension for simplicity, for the i^{th} choice in the first dimension, the distance x_i between p and the partition plane is uniform in the range $[0, l_i]$ where l_i has mean $1/2^i$ (and distribution $Pinv(n/2^{di}, 2^{(d-1)i}/n)$). The distance between q and p along the dimension is given by $N(0, \frac{r}{c\sqrt{d}}) = N(0, \frac{1}{cn^{1/d}})$ (again ignoring the Θ in the expression for r for simpler notations). So it is easy to verify that the distribution of the distance y_i of q from the partition plane is close to uniform (up to constant factors) in the range $[0, 1/2^i]$ (essentially y_i is obtained by first picking a random value in the interval $[0, l_i]$ and then perturbing it by a small value drawn from $N(0, \frac{1}{cn^{1/d}})$).

If the distance y_i of q from the partition plane in the i-th branch is equal to y, the probability that p and q are on different sides is $\Phi(ycn^{1/d})$. Since y_i is completely specified by the path B_{i0}, q and the tree structure T, $I[b_{i0}|B_{i0}, q, T] \leq I[b_{i0}|y_i = y] = E_y[I(\Phi(ycn^{1/d}), 1 - \Phi(ycn^{1/d}))]$. So we need to bound the expected value of $I(\Phi(ycn^{1/d}), 1 - \Phi(ycn^{1/d}))$. Again by a change of variable to $z = ycn^{1/d}$ this is equal to $E_z[I(\Phi(z), 1 - \Phi(z))]$.

We will argue that this is $O(\frac{2^i}{cn^{1/d}})$. Looking at integral values of z, observe that $I(\Phi(z), 1 - \Phi(z)) = e^{-\Omega(z^2)}$ drops faster than geometrically with each increment of z. It is $o(\frac{2^i}{cn^{1/d}})$ for $z > \frac{cn^{1/d}}{2^i}$. Further since the distribution of y in the range $[0, 1/2^i]$ is uniform (up to constant factor) so is the distribution of z in the range $[0, \frac{cn^{1/d}}{2^i}]$. So the probability that z lies in any unit interval in this range is $O(\frac{2^i}{cn^{1/d}})$. So the expected value of $I(\Phi(z), 1 - \Phi(z))$ is $O(\frac{2^i}{cn^{1/d}})$. So $I[b_{i0}|B_{ij}, q, T] = O(\frac{2^i}{cn^{1/d}})$. Similarly bounding and summing over all dimensions, $I[L(p)|q, T] \leq \sum_{i,j} I[b_{ij}|B_{ij}, q, T] \leq d \sum_i O(\frac{2^i}{cn^{1/d}}) = O(d/c)$

We are now ready to complete the proof of Theorem 1.

Proof. **[of Theorem 1].** The proof follows essentially from lemmas 3 and 4. Lemma 4 states that $I[L(p)|q, T] = O(d/c)$. But $I[L(p)|q, T]$ is the expected value of $I[L(p)]$ for a random fixed choice of q and T. So by Markov's inequality for a random fixed choice of q and T, with probability at least $1/2$, $I[L(p)] \leq$

$2I[L(p)|q,T] = O(d/c)$. So again with probability at least $1/2$ for a random instance of the problem, $I[L(p)] = O(d/c)$. Now lemma 3 states that $2^{O(c/d)}$ samples from the distribution of $L(p)$ given q must hit upon the correct cell containing p, completing the proof.

6 Experiments

We perform experiments on the planted instance of the problem with both the standard Kd-tree algorithm and our modified algorithm on a database of $n = 1$ million points chosen randomly in the unit cube in d dimensions. We then picked a random point p in the database and measured its distance r from the nearest other database point. We then planted a query point q with the distribution $N^d(p, r/c)$ so that it is at distance about r/c from p. We tried four different values for $d : 3, 5, 10$ and 20; and three different values for $c : 4/3, 2$ and 4. For each combination of values we performed $10,000$ search operations using both the traditional Kd-tree search algorithm and our modified algorithm. In the modified algorithm the query point was perturbed before the search and this was repeated for a few iterations; the number of iterations was varied from 5 to 30.

The success rates of finding the nearest neighbor are summarized in table 1. The third column shows the success rate of the traditional kd-tree search algortihm and the remaining columns state the success rate for the modified algorithm for different number of iterations. As can be seen, in 3 dimensions for $c = 4$, Kd-trees report the nearest neighbor 84% of the time whereas even with 5 iterations of the new algorithm this goes up to 96%. In 5 dimensions for $c = 4$ our algorithm boosts the success rate from 73% to 91% in 5 iterations and to 97.5% in 15

Table 1. Simulation Results: The entries indicate the percentage of times the nearest neighbor is found. As the number of iterations k is increased the success rate increases. The third column gives the success rate for the standard Kd-tree search algorithm. The latter columns report the performance of the modified algorithm for different number of iterations of searching for perturbed points.

d	c	Kd-tree	5 iter	15 iter	20 iter	25 iter	30 iter
3	4	84	96.1	98.8	99.3	99.3	99.8
3	2	73.9	89.5	97.4	98.4	99.0	98.7
3	4/3	73	88.5	96	96.6	98.7	98.7
5	4	73.6	91	97.5	98.1	98.5	99.3
5	2	54	78	92.1	94.9	94.4	96.2
5	4/3	50.7	71.3	87	91.2	92.3	94
10	4	60.7	80.5	94.8	96.6	96.7	96.8
10	2	36	56.4	77.6	84.3	86.6	88.4
10	4/3	25	43.7	61	70	73.4	75.6
20	4/3	13	25	28	41	42	46
20	2	22	42	67	68	70	72

iterations. In 10 dimensions for $c = 4$, 15 iterations raises the success rate from 60% to about 95%. In 20 dimensions for $c = 2$, Kd-trees succeed only 22% of the time, where as the new algorithm succeeds 67% of the time with 15 iterations.

References

1. Andoni, A., Indyk, P.: Near-Optimal Hashing Algorithms for Near Neighbor Problem in High Dimensions. In: Proceedings of the Symposium on Foundations of Computer Science (FOCS 2006) (2006)
2. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.: An optimal algorithm for approximate nearest neighbor searching. In: Proc. 5th ACM-SIAM Sympos. Discrete Algorithms, pp. 573–582 (1994)
3. Bentley, J.L., Friedman, J.H., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software 3(3), 209–226 (1977)
4. Borodin, A., Ostrovsky, R., Rabani, Y.: Lower bounds for high dimensional nearest neighbor search and related problems. In: Proceedings of the 31st ACM Symposium on Theory of Computing, pp. 312–321 (1999)
5. Clarkson, K.L.: Nearest neighbor queries in metric spaces. In: Proceedings of the 29th Annual ACM Symposium on Theory of Computing, May 1997, pp. 609–617 (1997)
6. Cover, T., Hart, P.: Nearest neighbor pattern classification. IEEE Trans. Information Theory IT-13, 21–27 (1967)
7. Deerwester, S., Dumais, S.T., Landauer, T.K., Furnas, G.W., Harshman, R.A.: Indexing by latent semantic analysis. Journal of the Society for Information Science 41(6), 391–407 (1990)
8. Dolev, D., Harari, Y., Parnas, M.: Finding the neighborhood of a query in a dictionary. In: Proc. 2nd Israel Symposium on Theory of Computing and Systems, pp. 33–42 (1993)
9. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.: Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In: Proceedings of the Symposium on Computational Geometry (2004), Talk is available at: `http://theory.lcs.mit.edu/ indyk/brown.ps`
10. Devroye, L., Wagner, T.J.: Nearest neighbor methods in discrimination. In: Krishnaiah, P.R., Kanal, L.N. (eds.) Handbook of Statistics, vol. 2, North-Holland, Amsterdam (1982)
11. Fagin, R.: Fuzzy Queries in Multimedia Database Systems. In: Proc. ACM Symposium on Principles of Database Systems, pp. 1–10 (1998)
12. Flickner, M., Sawhney, H., Niblack, W., Ashley, J., Huang, Q., Dom, B., Gorkani, M., Hafner, J., Lee, D., Petkovic, D., Steele, D., Yanker, P.: Query by image and video content: The QBIC system. Computer 28, 23–32 (1995)
13. Har-Peled, S.: A replacement for voronoi diagrams of near linear size. In: Proceedings of the Symposium on Foundations of Computer Science (2001)
14. Indyk, P.: High-dimensional computational geometry, Dept. of Comput. Sci., Stanford Univ. (2001)
15. Indyk, P.: Approximate Nearest Neighbor under Frechet Distance via Product Metrics. In: ACM Symposium on Computational Geometry (2002)
16. Indyk, P.: Nearest neighbors in high-dimensional spaces. In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, ch. 39, 2nd edn., CRC Press, Boca Raton (2004)

17. Indyk, P., Motwani, R., Raghavan, P., Vempala, S.: Locality-preserving hashing in multidimensional spaces. In: Proceedings of the 29th ACM Symposium on Theory of Computing, pp. 618–625 (1997)
18. Indyk, P., Motwani, R.: Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In: Proc. 30th Symposium on Theory of Computing, pp. 604–613 (1998)
19. Indyk, P., Thaper, N.: Embedding Earth-Mover Distance into the Euclidean space (manuscript, 2001)
20. Jayram, T.S., Khot, S., Kumar, R., Rabani, Y.: Cell-probe lower bounds for the partial match problem. In: Proc. 35th Annu. ACM Symp. Theory Comput., pp. 667–672 (2003)
21. Kleinberg, J.: Two algorithms for nearest-neighbor search in high dimension. In: Proc. 29th Annu. ACM Sympos. Theory Comput., pp. 599–608 (1997)
22. Kushilevitz, E., Ostrovsky, R., Rabani, Y.: Efficient search for approximate nearest neighbor in high dimensional spaces. In: Proc. of 30th STOC, pp. 614–623 (1998)
23. Linial, N., Sasson, O.: Non-Expansive Hashing. In: Proc. 28th STOC, pp. 509–517 (1996)
24. Meiser, S.: Point location in arrangements of hyperplanes. Information and Computation 106(2), 286–303 (1993)
25. Motwani, R., Naor, A., Panigrahy, R.: Lower Bounds on Locality Sensitive Hashing. In: Proceedings of the 22nd Annual ACM Symposium on Computational Geometry (2006)
26. Panigrahy, R.: Entropy based nearest neighbor search in high dimensions. In: SODA 2006: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, Miami, FL, pp. 1186–1195. ACM Press, New York (2006)
27. Pentland, A., Picard, R.W., Sclaroff, S.: Photobook: Tools for content-based manipulation of image databases. In: Proceedings of the SPIE Conference On Storage and Retrieval of Video and Image Databases, February 1994, vol. 2185, pp. 34–47 (1994)
28. van Rijsbergen, C.J.: Information Retrieval, Butterworths, London, United Kingdom (1990)
29. Vapnik, V.N., Chervonenkis, A.Y.: On the uniform convergence of relative frequencies of events to their probabilities. Theory Probab. Appl. 16, 264–280 (1971)
30. Salton, G.: Automatic Text Processing. Addison-Wesley, Reading (1989)