4 Locality-sensitive hashing using stable distributions

4.1 The LSH scheme based on s-stable distributions

In this chapter, we introduce and analyze a novel locality-sensitive hashing family. The family is defined for the case where the distances are measured according to the l_s norm, for any $s \in [0, 2]$. The hash functions are particularly simple for the case $s = 2$, i.e., the Euclidean norm. The new family provides an efficient solution to (approximate or exact) randomized near neighbor problem.

Part of this work appeared earlier in [DIIM04].

4.1.1 s-stable distributions

Stable distributions [Zol86] are defined as limits of normalized sums of independent identically distributed variables (an alternate definition follows). The most well-known example of a stable distribution is Gaussian (or normal) distribution. However, the class is much wider; for example, it includes heavy-tailed distributions.

Definition 4.1 *A distribution $\mathcal{D}$ over $\Re$ is called s-*stable*, if there exists $p \geq 0$ such that for any n real numbers $v_1 \dots v_n$ and i.i.d. variables $X_1 \dots X_n$ with distribution $\mathcal{D}$, the random variable $\sum_i v_i X_i$ has the same distribution as the variable $(\sum_i |v_i|^p)^{1/p} X$, where X is a random variable with distribution $\mathcal{D}$.*

It is known [Zol86] that stable distributions exist for any $p \in (0, 2]$. In particular:

- a *Cauchy distribution* $\mathcal{D}_C$, defined by the density function $c(x) = \frac{1}{\pi} \frac{1}{1+x^2}$, is 1-stable
- a *Gaussian (normal) distribution* $\mathcal{D}_G$, defined by the density function $g(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$, is 2-stable

We note from a practical point of view, despite the lack of closed form density and distribution functions, it is known [CMS76] that one can generate s-stable random variables essentially from two independent variables distributed uniformly over $[0, 1]$.

Stable distribution have found numerous applications in various fields (see the survey [Nol] for more details). In computer science, stable distributions were used for "sketching" of high dimensional vectors by Indyk ([Ind00]) and since have found use in various applications. The main property of s-stable distributions mentioned in the definition above directly translates into a sketching technique for high dimensional vectors. The idea is to generate a random vector a of dimension d whose each entry is chosen independently from a s-stable distribution. Given a vector v of dimension d, the dot product $a.v$ is a random variable which is distributed as $(\sum_i |v_i|^s)^{1/s}X$ (i.e., $||v||_s X$), where X is a random variable with s-stable distribution. A small collection of such dot products $(a.v)$, corresponding to different a's, is termed as the sketch of the vector v and can be used to estimate $||v||_s$ (see [Ind00] for details). It is easy to see that such a sketch is linearly composable, i.e., for any $p, q \in \Re^d$, $a.(p-q) = a.p - a.q$.

4.1.2 Hash family based on s-stable distributions

We use s-stable distributions in the following manner. Instead of using the dot products $(a.v)$ to estimate the l_s norm we use them to assign a hash value to each vector v. Intuitively, the hash function family should be locality sensitive, i.e. if two points (p, q) are close (small $||p-q||_s$) then they should collide (hash to the same value) with high probability and if they are far they should collide with small probability. The dot product $a.v$ projects each vector to the real line; It follows from s-stability that for two vectors (p, q) the distance between their projections $(a.p - a.q)$ is distributed as $||p-q||_s X$ where X is a s-stable distribution. If we "chop" the real line into equi-width segments of appropriate size w and assign hash values to vectors based on which segment they project onto, then it is intuitively clear that this hash function will be locality preserving in the sense described above.

Formally, each hash function $h_{a,b}(v) : \mathcal{R}^d \rightarrow \mathcal{N}$ maps a d dimensional vector v onto the set of integers. Each hash function in the family is indexed by a choice of random a and b where a is, as before, a d dimensional vector with entries chosen independently from a s-stable distribution and b is a real number chosen uniformly from the range $[0, w]$. For a fixed a, b the hash function $h_{a,b}$ is given by $h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{w} \rfloor$

4.1.2.1 Collision probability

We compute the probability that two vectors p, q collide under a hash function drawn uniformly at random from this family. Let $f_s(t)$ denote the probability density function of the **absolute value** of the s-stable distribution. We may drop the subscript s whenever it is clear from the context.

For the two vectors p, q, let $u = ||p-q||_s$ and let $p(u)$ denote the probability (as a function of u) that p, q collide for a hash function uniformly chosen from the family $\mathcal{H}$ described above. For a random vector a whose entries are drawn from a s-stable distribution, $a.p - a.q$ is distributed as cX where X is a random variable drawn from a s-stable distribution. Since b is drawn uniformly from $[0, w]$ it is easy to see that

$$p(u) = Pr_{a,b}[h_{a,b}(p) = h_{a,b}(q)] = \int_0^w \frac{1}{u} f_s(\frac{t}{u})(1 - \frac{t}{w})dt$$

For a fixed parameter w the probability of collision decreases monotonically with $u = ||p - q||_s$. Thus, as per the definition, the family of hash functions above is (R, cR, P_1, P_2)-sensitive for $P_1 = p(1)$ and $P_2 = p(c)$.

4.2 Approximate near neighbor

In what follows we will bound the ratio $\rho = \frac{\ln 1/P_1}{\ln 1/P_2}$, which as discussed earlier is critical to the performance when this hash family is used to solve the c-approximate near neighbor problem.

Note that we have not specified the parameter w, for it depends on the value of c and s. For every c we would like to choose a finite w that makes ρ as small as possible.

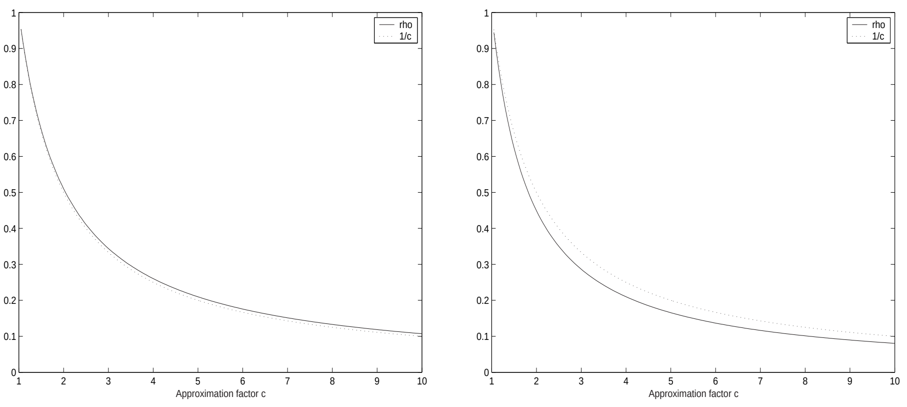
We focus on the cases of $s = 1, 2$. In these cases the ratio ρ can be explicitly evaluated. We compute and plot this ratio and compare it with $1/c$. Note, $1/c$ is the best (smallest) known exponent for n in the space requirement and query time that is achieved in [IM98] for these cases.

For $s = 1, 2$ we can compute the probabilities P_1, P_2, using the density functions mentioned before. A simple calculation shows that $P_2 = 2\frac{tan^{-1}(w/c)}{\pi} - \frac{1}{\pi(w/c)} \ln(1 + (w/c)^2)$ for $s = 1$ (Cauchy) and $P_2 = 1 - 2norm(-w/c) - \frac{2}{\sqrt{2\pi}w/c}(1 - e^{-(w^2/2c^2)})$ for $s = 2$ (Gaussian), where $norm(\cdot)$ is the cumulative distribution function (cdf) for a random variable that is distributed as $N(0, 1)$. The value of P_1 can be obtained by substituting $c = 1$ in the formulas above.

For c values in the range $[1, 10]$ (in increments of 0.05) we compute the minimum value of ρ, $\rho(c) = min_w \log(1/P_1)/\log(1/P_2)$, using *Matlab*. The plot of c versus $\rho(c)$ is shown in Figure 4.1. The crucial observation for the case $s = 2$ is that the curve corresponding to optimal ratio ρ ($\rho(c)$) lies strictly below the curve $1/c$. As mentioned earlier, this is a strict improvement over the previous best known exponent $1/c$ from [IM98]. While we have computed here $\rho(c)$ for c in the range $[1, 10]$, we believe that $\rho(c)$ is strictly less than $1/c$ for all values of c.

For the case $s = 1$, we observe that $\rho(c)$ curve is very close to $1/c$, although it lies above it. The optimal $\rho(c)$ was computed using *Matlab* as mentioned before. The *Matlab* program has a limit on the number of iterations it performs to compute the minimum of a function. We reached this limit during the computations. If we compute the true minimum, then we suspect that it will be very close to $1/c$, possibly equal to $1/c$, and that this minimum might be reached at $w = \infty$.

If one were to implement our LSH scheme, ideally they would want to know the optimal value of w for every c. For $s = 2$, for a given value of c, we can compute the value of w that gives the optimal value of $\rho(c)$. This can be done using programs like *Matlab*. However, we observe that for a fixed c the value of ρ as a function of



(a) Optimal ρ for l_1 (b) Optimal ρ for l_2

Figure 4.1 Optimal ρ vs c

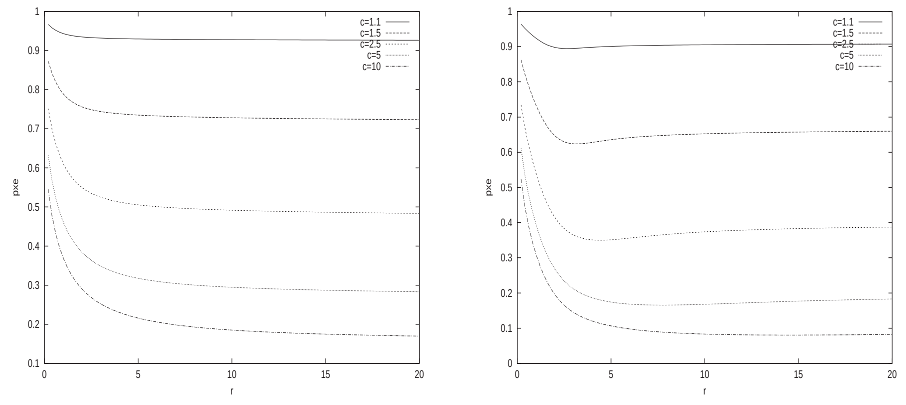
w is more or less stable after a certain point (see Figure 4.2). Thus, we observe that ρ is not very sensitive to w beyond a certain point and as long we choose w "sufficiently" away from 0, the ρ value will be close to optimal. Note, however that we should not choose an w value that is too large. As w increases, both P_1 and P_2 get closer to 1. This increases the query time, since k increases as $\log_{1/P_2} n$.

We mention that for the l_2 norm, the optimal value of w appears to be a (finite) function of c.

We also plot ρ as a function of c for a few fixed w values(See Figure 4.3). For $s = 2$, we observe that for moderate w values the ρ curve "beats" the $1/c$ curve over a large range of c that is of practical interest. For $s = 1$, we observe that as w increases the ρ curve drops lower and gets closer and closer to the $1/c$ curve.

4.3 Exact Near Neighbor

LSH can also be used to solve the randomized version of the exact near neighbor problem. To use it for the exact near neighbor, we use the "Strategy 2" of the basic LSH scheme, and keep only the R-near neighbors of q. Thus, the running time depends on the data set $\mathcal{P}$. In particular, the running time is slower for "bad" data sets, e.g., when for a query q, there are many points from $\mathcal{P}$ clustered right outside the ball of radius R centered at q (i.e., when there are many approximate near neighbors).



(a) ρ vs w for l_1 (b) ρ vs w for l_2

Figure 4.2 ρ vs w

4.3.1 Parameters k and L of the LSH scheme

There are two steps for choosing the parameters k and L that are optimal for a data set. First, we need to determine the bounds on k and L that guarantee the correctness of the algorithm. Second, within those bounds, we choose the values k and L that would achieve the best expected query running time.

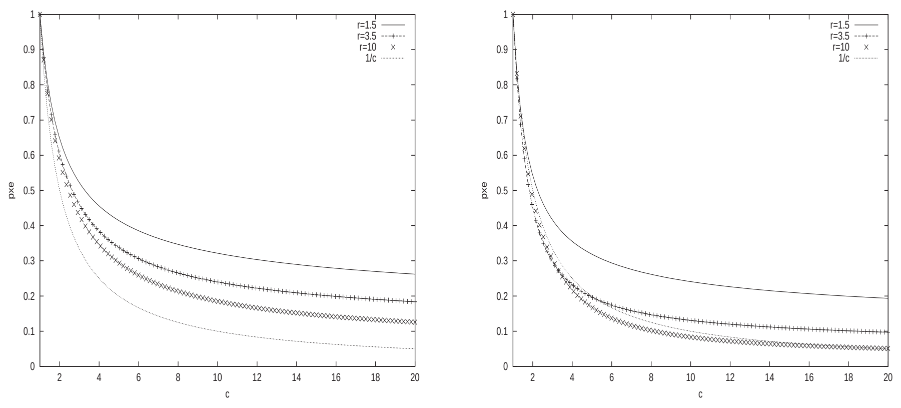
Next, we derive the bounds that need to be satisfied by k and L to guarantee the correctness of the algorithm. We need to ensure that our data structure reports a R-near neighbor with a probability at least $1 - \delta$. To analyse what condition this implies, consider a query point q and an R-near neighbor p of q. Let $P_1 = p(R)$. Then, $Pr_{g\in\mathcal{G}}[g(q) = g(p)] \geq P_1^k$. Thus, q and p fail to collide for all L functions g_i with probability at most $(1 - P_1^k)^L$. Requiring that the point q collides with p on some function g_i is equivalent to saying $1 - (1 - P_1^k)^L \geq 1 - \delta$, which implies that

$$L \geq \frac{\log 1/\delta}{-\log(1 - P_1^k)} \tag{4.1}$$

Since we want to choose L as small as possible (for a fixed k), the best value for L is $L = \left\lceil \frac{\log 1/\delta}{-\log(1-P_1^k)} \right\rceil$.

Thus, one is free to choose only k since it is the only remaining degree of freedom in choosing parameters k and L.

To understand better how the choice of k affects the query running time, we decompose the running time into two terms, T_g and T_c. T_g is the time necessary for computing L functions g_i for the query point q as well as for retrieving the buckets $g_i(q)$ from hash tables; the expression for T_g is $T_g = O(dkL)$.



(a) ρ vs c for l_1 (b) ρ vs c for l_2

Figure 4.3 ρ vs c

The second term, T_c, represents the time for computing the distance to all points encountered in the retrieved buckets; T_c is equal to $O(d \cdot \#collisions)$, where $\#collisions$ is the number of points encountered in the buckets $g_1(q), \ldots g_L(q)$ for a query point q. The expected value of T_c is

$$E[T_c] = O(d \cdot E[\#collisions]) = O(dL \cdot \sum_{p \in \mathcal{P}} p^k(\|q - p\|)) \tag{4.2}$$

Intuitively, T_g increases as a function of k, while T_c decreases as a function of k. The latter is due to the fact that higher values of k magnify the gap between the collision probabilities of “close” and “far” points, which (for proper values of L) decreases the probability of collision of far points. Thus, typically there exists an optimal value of k that minimizes the sum $T_g + T_c$ (for a given query point q). Note that there might be different optimal k's for different query points, therefore the goal would be optimize the mean query time for all query points.

4.4 LSH in practice: E^{2}LSH

In this section we present a practitioner's view on how to implement the LSH scheme for solving the R-near neighbor reporting problem in practice. Specifically, we describe a concrete method for choosing algorithm's parameters, as well as present some implementation details that both clarify steps of the scheme and demonstrate how to optimize the scheme in practice.

The section is based on the package E^{2}LSH (*Exact Euclidean LSH*), version 0.1, which is authors' current implementation of LSH scheme [AI04]. E^{2}LSH solves the exact near neighbor reporting problem.

Note that E^{2}LSH uses a few addition optimizations to improve the search performance, in addition to what is described below. Please refer to the manual [AI04] for more information.

4.4.1 Data structure construction

Before constructing the data structure, E^{2}LSH first computes the parameters k, L as a function of the data set $\mathcal{P}$, the radius R, and the probability $1-\delta$ as outlined in earlier sections. In what follows, we consider L as a function of k, and the question remains only of how to choose k.

For choosing the value k, the algorithm experimentally estimates the times T_g and T_c as a function of k. Remember that the time T_c is dependent on the query point q, and, therefore, for estimating T_c we need to use a set S of sample query points (the estimation of T_c is then the mean of the times T_c for points from S). The sample set S is a set of several points chosen at random from the query set. (The package also provides the option of choosing S to be a subset of the data set $\mathcal{P}$.)

Note that to estimate T_g and T_c precisely, we need to know the constants hidden by the $O(\cdot)$ notation in the expressions for T_g and T_c. To compute these constants, the implementation constructs a sample data structure and runs several queries on that sample data structure, measuring the actual times T_g and T_c. Note that T_g and T_c depend on k. Thus, k is chosen such that $T_g + \tilde{T}_c$ is minimal (while the data structure space requirement is within the memory bounds), where $\tilde{T}_c$ is the mean of the times T_c for all points in the sample query set S: $\tilde{T}_c = \frac{\sum_{q \in S} T_c(q)}{|S|}$.

Once the parameters k, m, L are computed, the algorithm constructs the data structure containing the points from $\mathcal{P}$.

4.4.2 Bucket hashing

Recall that the domain of each function g_i is too large to store all possible buckets explicitly, and only non-empty buckets are stored. To this end, for each point p, the buckets $g_1(p), \ldots g_L(p)$ are hashed using the universal hash functions. For each function g_i, $i = 1 \ldots L$, there is a hash table H_i containing the buckets $\{g_i(p) \mid v \in \mathcal{P}\}$. For this purpose, there are 2 associated hash functions $t_1 : \mathbb{Z}^k \to \{0, \ldots, \mathit{tableSize} - 1\}$ and $t_2 : \mathbb{Z}^k \to \{0, \ldots, C\}$. The function t_1 determines for a LSH bucket the index of the point in the hash table. The second hash function identifies the buckets in chains.

The collisions within each index in the hash table are resolved by chaining. When storing a bucket $g_i(p) = (x_1, \ldots x_k)$ in its chain, instead of storing the entire vector $(x_1, \ldots x_k)$ for bucket identification, we store only $t_2(x_1, \ldots x_k)$. Thus, a bucket $g_i(p) = (x_1, \ldots x_k)$ has only the following associated information stored in its chain: the identifier $t_2(x_1, \ldots, x_k)$, and the points in the bucket, which are $g_i^{-1}(x_1, \ldots x_k) \cap \mathcal{P}$.

The reasons for using the second hash function t_2 instead of storing the value $g_i(p) = (x_1, \ldots x_k)$ are twofold. Firstly, by using a fingerprint $t_2(x_1, \ldots x_k)$, we decrease the amount of memory for bucket identification from $O(k)$ to $O(1)$. Secondly, with the fingerprint it is faster to look up a LSH bucket in the chain containing it. The domain of the function t_2 is chosen big enough to ensure with a high probability that any two different buckets in the same chain have different t_2 values.

All L hash tables use the same primary hash function t_1 (used to dermine the index in the hash table) and the same secondary hash function t_2. These two hash functions have the form

$$\begin{aligned} t_1(a_1, a_2, \ldots, a_k) &= \left(\left(\sum_{i=1}^k r'_i a_i\right) \mod P\right) \bmod \textit{tableSize} \\ t_2(a_1, a_2, \ldots, a_k) &= \left(\sum_{i=1}^k r''_i a_i\right) \mod P \end{aligned}$$

where r'_i and r''_i are random integers, *tableSize* is the size of the hash tables, and P is a prime.

In the current implementation, $\textit{tableSize} = |\mathcal{P}|$, a_i are represented by 32-bit integers, and the prime P is equal to $2^{32} - 5$. This value of the prime allows fast hash function computation without using modulo operations. Specifically, without loss of generality, consider computing $t_2(a_1)$ for $k = 1$. We have that:

$$t_2(a_1) = (r''_1 a_1) \bmod \left(2^{32} - 5\right) = (\textit{low}\,[r''_1 a_1] + 5 \cdot \textit{high}\,[r''_1 a_1]) \bmod (2^{32} - 5)$$

where $\textit{low}[r''_1 a_1]$ are the low-order 32 bits of $r''_1 a_1$ (a 64-bit number), and $\textit{high}[r''_1 a_1]$ are the high-order 32 bits of $r''_1 a_1$. If we choose r''_i from the range $\{1, \ldots 2^{29}\}$, we will always have that $\alpha = \textit{low}\,[r''_1 a_1] + 5 \cdot \textit{high}\,[r''_1 a_1] < 2 \cdot \left(2^{32} - 5\right)$. This means that

$$t_2(a_1) = \begin{cases} \alpha & , \text{if } \alpha < 2^{32} - 5 \\ \alpha - \left(2^{32} - 5\right) & , \text{if } \alpha \geq 2^{32} - 5 \end{cases}$$

For $k > 1$, we compute progressively the sum $\left(\sum_{i=1}^k r''_i a_i\right) \bmod P$ keeping always the partial sum modulo $\left(2^{32} - 5\right)$ using the same principle as the one above. Note that the range of the function t_2 thus is $\{1, \ldots 2^{32} - 6\}$.

4.4.3 Memory Requirement for LSH

The data structure described above requires $O(nL)$ memory (for each function g_i, we store the n points from $\mathcal{P}$). Since, L increases as k increases, the memory requirement could be large for a large data set, or for moderate data set for which optimal time is achived with higher values of k. Therefore, an upper limit on memory imposes an upper limit on k.

Because the memory requirement is big, the constant in front of $O(nL)$ is very important. In E^{2}LSH, with the best variant of the hash tables, this constant is 12 bytes. Note that it is the structure and layout of the L hash tables that dictates memory usage.

Below we show two variants of the layout of the hash tables that we deployed. We assume that:

- the number of points is $n \leq 2^{20}$;
- each pointer is 4 bytes long;
- $tableSize = n$ for each hash table.

One of the most straightforward layouts of a hash table H_i is the following. For each index l of the hash table, we store a pointer to a singly-linked list of buckets in the chain l. For each bucket, we store its value $h_2(\cdot)$, and a pointer to a singly-linked list of points in the bucket. The memory requirement per hash table is $4 \cdot tableSize + 8 \cdot \#buckets + 8 \cdot n \leq 20n$, yielding a constant of 20.

To reduce this constant to 12 bytes, we do the following. Firstly, we index all points in $\mathcal{P}$, such that we can refer to points by index (this index is constant across all hash tables). Refering to a point thus takes only 20 bits (and not 32 as in the case of a pointer). Consider now a hash table H_i. For this hash table, we deploy a table Y of 32-bit unsigned integers that store all buckets (with values $h_2(\cdot)$) and points in the buckets (thus, Y is a hybrid storage table since it stores both buckets' and points' description). The table has a length of $\#buckets + n$ and is used as follows. In the hash table H_i, at index l, we store the pointer to some index e_l of Y; e_l is the start of the description of the chain l. A chain is stored as follows: $h_2(\cdot)$ value of the first bucket in chain (at position e_l in Y) followed by the indices of the points in this bucket (positions $e_l + 1, \ldots e_l + n_1$); $h_2(\cdot)$ value of the second bucket in the chain (position $e_l + n_1 + 1$) followed by the indices of the points in this second bucket (positions $e_l + n_1 + 2, \ldots e_l + n_1 + 1 + n_2$); and so forth.

Note that we need also to store the number of buckets in each chain as well as the number of points in each bucket. Instead of storing the chain length, we store for each bucket a bit that says whether that bucket is the last one in the chain or not; this bit is one of the unused bits of the 4-byte integer storing the index of the first point in the corresponding bucket (i.e., if the $h_2(\cdot)$ value of the bucket is stored at position e in Y, then we use a high-order bit of the integer at position $e + 1$ in Y). For storing the length of the bucket, we use the remaining unused bits of the first point in the bucket. When the remaining bits are not enough (there are more than $2^{32-20-1} - 1 = 2^{11} - 1$ points in the bucket), we store a special value for the length (0), which means that there are more than $2^{11} - 1$ points in the bucket, and there are some additional points (that do not fit in the $2^{11} - 1$ integers alloted in Y after the $h_2(\cdot)$ value of the bucket). These additional points are also stored in Y but at a different position; their start index and number are stored in the unused bits of the remaining $2^{11}-2$ points that follow the $h_2(\cdot)$ value of the bucket and the first point of the bucket (i.e., unused bits of the integers at positions $e+2, \dots e+2^{11}-1$).

4.5 Experimental Results

In this section we present some preliminary experimental results on the performance of E^{2}LSH.

For the comparison, we used the MNIST data set [Cun]. It contains 60,000 points, each having dimension $28\times 28 = 784$. The points were normalized so that each point has its l_2 norm equal to 1.

We compared the performance of E^{2}LSH and ANN [AM]. The latter provides an efficient implementation of a variant of the kd-tree data structure. It supports both exact and approximate nearest neighbor search (we used the former).

To compare the running times of ANN and E^{2}LSH, we need to have E^{2}LSH find the *nearest neighbor*, as opposed to the *near neighbor*. We achieve this by solving the near neighbor problem for one value of R. We chose this value to ensure that all but, say, 3% of the data points have their nearest neighbor within distance R. To find such R, it suffices to find, say, the 97%-percentile of the distances from points to their nearest neighbor (this can be approximated fast by sampling). In our case, we chose $R = 0.65$. Then, to find the nearest neighbor, we find the R-near neighbors and report the closest point.

We note that, in general, the value of R obtained using the above method might not lead to an efficient algorithm. This is because, for some data sets, the number of R-near neighbors of an average query point could be very large, and sifting through all of them during the query time would be inefficient. For such data sets one needs to build data structures for *several* values of R. During the query time, the data structures are queried in the increasing order of R. The process is stopped when a data structure reports an answer.

Another parameter that is required by E^{2}LSH is the probablity of error δ. We set it to 10%. Lower probability of error would increase the running times, although not very substantially. E.g., using two separate data structures in parallel (or, alternatively, doubling the number of hash functions L), would reduce the error from 10% to at most $(10\%)^2 = 1\%$.

To perform the running time comparisons, we ran the algorithms on random subsets of the original data sets of size 10000, 30000 and 50000. The actual times per query are reported in Figure 4.4.

As can be observed, the running times of E^{2}LSH are much lower than the times of ANN. Additional experiments (not reported here) indicate that the times do not decrease substantially if ANN is allowed to report c-approximate nearest neighbor for small values of c (say, $c < 1.5$). On the other hand, setting c to a large value (say, $c = 10$) reduces running times of ANN drastically, since the search procedure is stopped at a very early stage; the resulting running times become comparable to E^{2}LSH. At the same time, the *actual* error of ANN is remarkably low: it reports

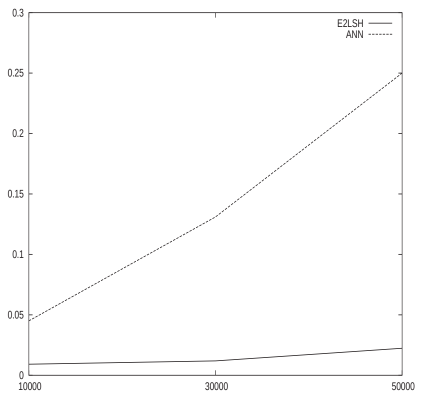


Figure 4.4 Experiments: LSH vs ANN.

the exact nearest neighbor for about 2/3 of the query points. The fact that kd-trees search procedure (using priority queues) reports “good” nearest neighbors, even if the search is interrupted very early, has been observed earlier in the literature (e.g., see [Low04]). Note, however, that any guarantees for this method are only empirical, while, for the R-near neighbor search problem, E^{2}LSH provides rigorous guarantees on the probability of missing a near neighbor.

References

[AI04] A. Andoni and P. Indyk. E2lsh: Exact euclidean locality-sensitive hashing. *Implementation available at* `http://web.mit.edu/andoni/www/LSH/index.html`, 2004.

[AM] S. Arya and D. Mount. Ann: Library for approximate nearest neighbor searching. *available at* `http://www.cs.umd.edu/~mount/ANN/`.

[CMS76] J. M. Chambers, C. L. Mallows, and B. W. Stuck. A method for simulating stable random variables. *J. Amer. Statist. Assoc.*, 71:340–344, 1976.

[Cun] Y. Le Cunn. The mnist database of handwritten digits. *Available at* `http://yann.lecun.com/exdb/mnist/`.

[DIIM04] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. *Proceedings of the ACM Symposium on Computational Geometry*, 2004.

[IM98] P. Indyk and R. Motwani. Approximate nearest neighbor: towards removing the curse of dimensionality. *Proceedings of the Symposium on Theory of Computing*, 1998.

[Ind00] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. *Annual Symposium on Foundations of Computer Science*, 2000.

[Low04] D. Lowe. Fast high-dimensional feature indexing for object recognition. *Slides available at* `http://www.cs.ubc.ca/~nando/nipsfast/slides/fast04.pdf`, 2004.

[Nol] J. P. Nolan. An introduction to stable distributions. *available at* `http://www.cas.american.edu/~jpnolan/chap1.ps`.

[Zol86] V.M. Zolotarev. *One-Dimensional Stable Distributions.* Vol. 65 of Translations of Mathematical Monographs, American Mathematical Society, 1986.