Locality-Sensitive Hashing Scheme Based on p-Stable Distributions

Mayur Datar
Department of Computer Science,
Stanford University
datar@cs.stanford.edu

Nicole Immorlica
Laboratory for Computer Science, MIT
nickle@theory.lcs.mit.edu

Piotr Indyk
Laboratory for Computer Science, MIT
indyk@theory.lcs.mit.edu

Vahab S. Mirrokni
Laboratory for Computer Science, MIT
mirrokni@theory.lcs.mit.edu

ABSTRACT

We present a novel Locality-Sensitive Hashing scheme for the Approximate Nearest Neighbor Problem under l_p norm, based on p-stable distributions.

Our scheme improves the running time of the earlier algorithm for the case of the l_2 norm. It also yields the first known provably efficient approximate NN algorithm for the case $p < 1$. We also show that the algorithm finds the *exact* near neigbhor in $O(\log n)$ time for data satisfying certain "bounded growth" condition.

Unlike earlier schemes, our LSH scheme works directly on points in the Euclidean space without embeddings. Consequently, the resulting query time bound is free of large factors and is simple and easy to implement. Our experiments (on synthetic data sets) show that the our data structure is up to 40 times faster than kd-tree.

1. INTRODUCTION

A similarity search problem involves a collection of objects (documents, images, etc.) that are characterized by a collection of relevant features and represented as points in a high-dimensional attribute space; given queries in the form of points in this space, we are required to find the nearest (most similar) object to the query. A particularly interesting and well-studied instance is d-dimensional Euclidean space. This problem is of major importance to a variety of applications; some examples are: data compression, databases and data mining, information retrieval, image and video databases, machine learning, pattern recognition, statistics and data analysis. Typically, the features of the objects of interest (documents, images, etc) are represented as points in $\Re^d$ and a distance metric is used to measure similarity of objects. The basic problem then is to perform indexing or similarity searching for query objects. The number of features (i.e., the dimensionality) ranges anywhere from tens to thousands.

The low-dimensional case (say, for the dimensionality d equal to 2 or 3) is well-solved, so the main issue is that of dealing with a large number of dimensions, the so-called "curse of dimensionality". Despite decades of intensive effort, the current solutions are not entirely satisfactory; in fact, for large enough d, in theory or in practice, they often provide little improvement over a linear algorithm which compares a query to each point from the database. In particular, it was shown in [28] (both empirically and theoretically) that *all* current indexing techniques (based on space partitioning) degrade to linear search for sufficiently high dimensions.

In recent years, several researchers proposed to avoid the running time bottleneck by using *approximation* (e.g., [3, 22, 19, 24, 15], see also [12]). This is due to the fact that, in many cases, approximate nearest neighbor is almost as good as the exact one; in particular, if the distance measure accurately captures the notion of user quality, then small differences in the distance should not matter. In fact, in situations when the quality of the approximate nearest neighbor is much worse than the quality of the actual nearest neighbor, then the nearest neighbor problem is *unstable*, and it is not clear if solving it is at all meaningful [4, 17].

In [19, 14], the authors introduced an approximate high-dimensional similarity search scheme with provably sublinear dependence on the data size. Instead of using tree-like space partitioning, it relied on a new method called *locality-sensitive hashing (LSH)*. The key idea is to hash the points using several hash functions so as to ensure that, for each function, the probability of collision is much higher for objects which are close to each other than for those which are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point. In [19, 14] the authors provided such locality-sensitive hash functions for the case when the points live in binary Hamming space $\{0, 1\}^d$. They showed experimentally that the data structure achieves large speedup over several tree-based data structures when the data is stored on disk. In addition, since the LSH is a hashing-based scheme, it can be naturally extended to the *dynamic* setting, i.e., when insertion and deletion operations also need to be supported. This avoids the complexity of dealing with tree structures when the data is dynamic.

The LSH algorithm has been since used in numerous applied settings, e.g., see [14, 10, 16, 27, 5, 7, 29, 6, 26, 13]. However, it suffers from a fundamental drawback: it is fast and simple only when the input points live in the Hamming space (indeed, almost all of the above applications involved binary data). As mentioned in [19, 14], it is possible to extend the algorithm to the l_2 norm, by embedding l_2 space into l_1 space, and then l_1 space into Hamming space. However, it increases the query time and/or error by a large factor and complicates the algorithm.

In this paper we present a novel version of the LSH algorithm. As the previous schemes, it works for the (R, c)-Near Neighbor (NN) problem, where the goal is to report a point within distance cR from a query q, if there is a point in the data set P within distance R from q. Unlike the earlier algorithm, our algorithm works directly on points in Euclidean space without embeddings. As a consequence, it has the following advantages over the previous algorithm:

- For the l_2 norm, its query time is $O(dn^{\rho(c)} \log n)$, where $\rho(c) < 1/c$ for $c \in (1, 10]$ (the inequality is strict, see Figure 1(b)). Thus, for large range of values of c, the query time exponent is better than the one in [19, 14].

- It is simple and quite easy to implement.

- It works for any l_p norm, as long as $p \in (0, 2]$. Specifically, we show that for any $p \in (0, 2]$ and $\gamma > 0$ there exists an algorithm for (R, c)-NN under l_d^p which uses $O(dn + n^{1+\rho})$ space, with query time $O(n^\rho \log_{1/\gamma} n)$, where where $\rho \leq (1 + \gamma) \cdot \max\left(\frac{1}{c^p}, \frac{1}{c}\right)$. To our knowledge, this is the only known *provable* algorithm for the high-dimensional nearest neighbor problem for the case $p < 1$. Similarity search under such *fractional* norms have recently attracted interest [1, 11].

Our algorithm also inherits two very convenient properties of LSH schemes. The first one is that it works well on data that is extremely high-dimensional but sparse. Specifically, the running time bound remains unchanged if d denotes the maximum number of non-zero elements in vectors. To our knowledge, this property is not shared by other known spatial data structures. Thanks to this property, we were able to use our new LSH scheme (specifically, the l_1 norm version) for fast color-based image similarity search [20]. In that context, each image was represented by a point in roughly 100^3-dimensional space, but only about 100 dimensions were non-zero per point. The use of our LSH scheme enabled achieving order(s) of magnitude speed-up over the linear scan.

The second property is that our algorithm provably reports the *exact* near neighbor very quickly, if the data satisfies certain *bounded growth* property. Specifically, for a query point q, and $c \geq 1$, let $N(q, c)$ be the number of c-approximate nearest neighbors of q in P. If $N(q, c)$ grows "sub-exponentially" as a function of c, then the LSH algorithm reports p with constant probability within time $O(d \log n)$, assuming it is given a constant factor approximation to the distance from q to its nearest neighbor. In particular, we show that if $N(q, c) = O(c^b)$, then the running time is $O(\log n + 2^{O(b)})$. Efficient nearest neighbor algorithms for data sets with polynomial growth properties in general metrics have been recently a focus of several papers [9, 21, 23]. LSH solves an easier problem (near neighbor under l_2 norm), while working under weaker assumptions about the growth function. It is also somewhat faster, due to the fact that the $\log n$ factor in the query time of the earlier schemes is *multiplied* by a function of b, while in our case this factor is additive.

We complement our theoretical analysis with experimental evaluation of the algorithm on data with wide range of parameters. In particular, we compare our algorithm to an approximate version of the kd-tree algorithm [2]. We performed the experiments on synthetic data sets containing "planted" near neighbor (see section 5 for more details); similar model was earlier used in [30]. Our experiments indicate that the new LSH scheme achieves query time of up to 40 times better than the query time of the kd-tree algorithm.

1.1 Notations and problem definitions

We use l_p^d to denote the space $\Re^d$ under the l_p norm. For any point $\boldsymbol{v} \in \Re^d$, we denote by $||\vec{\boldsymbol{v}}||_p$ the l_p norm of the vector $\vec{\boldsymbol{v}}$. Let $\mathcal{M} = (X, d)$ be any metric space, and $v \in X$. The *ball* of radius r centered at v is defined as $B(v, r) = \{q \in X \mid d(v, q) \leq r\}$.

Let $c = 1 + \epsilon$. In this paper we focus on the (R, c)-NN problem. Observe that (R, c)-NN is simply a decision version of the Approximate Nearest Neighbor problem. Although in many applications solving the decision version is good enough, one can also reduce the approximate NN problem to approximate NN via binary-search-like approach. In particular, it is known [19, 15] that the c-approximate NN problem reduces to $O(\log(n/\epsilon))$ instances of (R, c)-NN. Then, the complexity of c-approximate NN is the same (within log factor) as that of the (R, c)-NN problem.

2. LOCALITY-SENSITIVE HASHING

An important technique from [19], to solve the (R, c)-NN problem is locality sensitive hashing or LSH. For a domain S of the points set with distance measure D, an LSH family is defined as:

DEFINITION 1. *A family $\mathcal{H} = \{h : S \to U\}$ is called (r_1, r_2, p_1, p_2)-sensitive for D if for any $v, q \in S$*

- *if $v \in B(q, r_1)$ then $\Pr_{\mathcal{H}}[h(q) = h(v)] \geq p_1$,*
- *if $v \notin B(q, r_2)$ then $\Pr_{\mathcal{H}}[h(q) = h(v)] \leq p_2$.*

In order for a locality-sensitive hash (LSH) family to be useful, it has to satisfy inequalities $p_1 > p_2$ and $r_1 < r_2$.

We will briefly describe, from [19], how a LSH family can be used to solve the (R, c)-NN problem: We choose $r_1 = R$ and $r_2 = c \cdot R$. Given a family $\mathcal{H}$ of hash functions with parameters (r_1, r_2, p_1, p_2) as in Definition 1, we amplify the gap between the "high" probability p_1 and "low" probability p_2 by concatenating several functions. In particular, for k specified later, define a function family $\mathcal{G} = \{g : S \to U^k\}$ such that $g(v) = (h_1(v), \ldots, h_k(v))$, where $h_i \in \mathcal{H}$. For an integer L we choose L functions $g_1, \ldots, g_L$ from $\mathcal{G}$, independently and uniformly at random. During preprocessing, we store each $v \in P$ (input point set) in the bucket $g_j(v)$, for $j = 1, \ldots, L$. Since the total number of buckets may be large, we retain only the non-empty buckets by resorting to hashing. To process a query q, we search all buckets $g_1(q), \ldots, g_L(q)$; as it is possible (though unlikely) that the total number of points stored in those buckets is large, we interrupt search after finding first $3L$ points (including duplicates). Let $v_1, \ldots, v_t$ be the points encountered therein. For each v_j, if $v_j \in B(q, r_2)$ then we return YES and v_j, else we return NO.

The parameters k and L are chosen so as to ensure that with a constant probability the following two properties hold:

1. If there exists $v^* \in B(q, r_1)$ then $g_j(v^*) = g_j(q)$ for some $j = 1 \ldots L$, and

2. The total number of collisions of q with points from $P - B(q, r_2)$ is less than $3L$, i.e.

$$\sum_{j=1}^{L} |(P - B(q, r_2)) \cap g_j^{-1}(g_j(q))| < 3L.$$

Observe that if (1) and (2) hold, then the algorithm is correct. It follows (see [19] Theorem 5 for details) that if we set $k = \log_{1/p_2} n$, and $L = n^\rho$ where $\rho = \frac{\ln 1/p_1}{\ln 1/p_2}$ then (1) and (2) hold with a constant probability. Thus, we get following theorem (slightly different version of Theorem 5 in [19]), which relates the efficiency of solving (R, c)-NN problem to the sensitivity parameters of the LSH.

THEOREM 1. *Suppose there is a (R, cR, p_1, p_2)-sensitive family $\mathcal{H}$ for a distance measure D. Then there exists an algorithm* *for (R,c)-NN under measure D which uses $O(dn + n^{1+\rho})$ space, with query time dominated by $O(n^\rho)$ distance computations, and $O(n^\rho \log_{1/p_2} n)$ evaluations of hash functions from $\mathcal{H}$, where $\rho = \frac{\ln 1/p_1}{\ln 1/p_2}$.*

3. OUR LSH SCHEME

In this section, we present a LSH family based on p-stable distributions, that works for all $p \in (0, 2]$.

Since we consider points in l_p^d, without loss of generality we can consider $R = 1$, which we assume from now on.

3.1 p-stable distributions

Stable distributions [31] 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.

Stable Distribution: A distribution $\mathcal{D}$ over $\Re$ is called *p-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 [31] 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 [8] that one can generate p-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 [25] for more details). In computer science, stable distributions were used for "sketching" of high dimensional vectors by Indyk ([18]) and since have found use in various applications. The main property of p-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 $\boldsymbol{a}$ of dimension d whose each entry is chosen independently from a p-stable distribution. Given a vector $\boldsymbol{v}$ of dimension d, the dot product $\boldsymbol{a}.\boldsymbol{v}$ is a random variable which is distributed as $(\sum_i |v_i|^p)^{1/p} X$ (i.e., $||\boldsymbol{v}||_p X$), where X is a random variable with p-stable distribution. A small collection of such dot products ($\boldsymbol{a}.\boldsymbol{v}$), corresponding to different $\boldsymbol{a}$'s, is termed as the sketch of the vector $\boldsymbol{v}$ and can be used to estimate $||\boldsymbol{v}||_p$ (see [18] for details). It is easy to see that such a sketch is linearly composable, i.e. $\boldsymbol{a}.(\boldsymbol{v_1} - \boldsymbol{v_2}) = \boldsymbol{a}.\boldsymbol{v_1} - \boldsymbol{a}.\boldsymbol{v_2}$.

3.2 Hash family

In this paper we use p-stable distributions in a slightly different manner. Instead of using the dot products ($\boldsymbol{a}.\boldsymbol{v}$) to estimate the l_p norm we use them to assign a hash value to each vector $\boldsymbol{v}$. Intuitively, the hash function family should be locality sensitive, i.e. if two vectors ($\boldsymbol{v_1}, \boldsymbol{v_2}$) are close (small $||\boldsymbol{v_1} - \boldsymbol{v_2}||_p$) 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 $\boldsymbol{a}.\boldsymbol{v}$ projects each vector to the real line; It follows from p-stability that for two vectors ($\boldsymbol{v_1}, \boldsymbol{v_2}$) the distance between their projections ($\boldsymbol{a}.\boldsymbol{v_1} - \boldsymbol{a}.\boldsymbol{v_2}$) is distributed as $||\boldsymbol{v_1} - \boldsymbol{v_2}||_p X$ where X is a p-stable distribution. If we "chop" the real line into equi-width segments of appropriate size r 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 $h_{\boldsymbol{a},b}(\boldsymbol{v}) : \mathcal{R}^d \to \mathcal{N}$ maps a d dimensional vector $\boldsymbol{v}$ onto the set of integers. Each hash function in the family is indexed by a choice of random $\boldsymbol{a}$ and b where $\boldsymbol{a}$ is, as before, a d dimensional vector with entries chosen independently from a p-stable distribution and b is a real number chosen uniformly from the range $[0, r]$. For a fixed $\boldsymbol{a}, b$ the hash function $h_{\boldsymbol{a},b}$ is given by $h_{\boldsymbol{a},b}(\boldsymbol{v}) = \lfloor \frac{\boldsymbol{a}\cdot\boldsymbol{v}+b}{r} \rfloor$

Next, we compute the probability that two vectors $\boldsymbol{v_1}, \boldsymbol{v_2}$ collide under a hash function drawn uniformly at random from this family. Let $f_p(t)$ denote the probability density function of the **absolute value** of the p-stable distribution. We may drop the subscript p whenever it is clear from the context. For the two vectors $\boldsymbol{v_1}, \boldsymbol{v_2}$, let $c = ||\boldsymbol{v_1} - \boldsymbol{v_2}||_p$. For a random vector $\boldsymbol{a}$ whose entries are drawn from a p-stable distribution, $\boldsymbol{a}.\boldsymbol{v_1} - \boldsymbol{a}.\boldsymbol{v_2}$ is distributed as cX where X is a random variable drawn from a p-stable distribution. Since b is drawn uniformly from $[0, r]$ it is easy to see that

$$p(c) = Pr_{\boldsymbol{a},b}[h_{\boldsymbol{a},b}(\boldsymbol{v_1}) = h_{\boldsymbol{a},b}(\boldsymbol{v_2})] = \int_0^r \frac{1}{c} f_p(\frac{t}{c})(1 - \frac{t}{r})dt$$

For a fixed parameter r the probability of collision decreases monotonically with $c = ||\boldsymbol{v_1} - \boldsymbol{v_2}||_p$. Thus, as per Definition 1 the family of hash functions above is (r_1, r_2, p_1, p_2)-sensitive for $p_1 = p(1)$ and $p_2 = p(c)$ for $r_2/r_1 = c$.

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 (R, c)-NN problem.

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

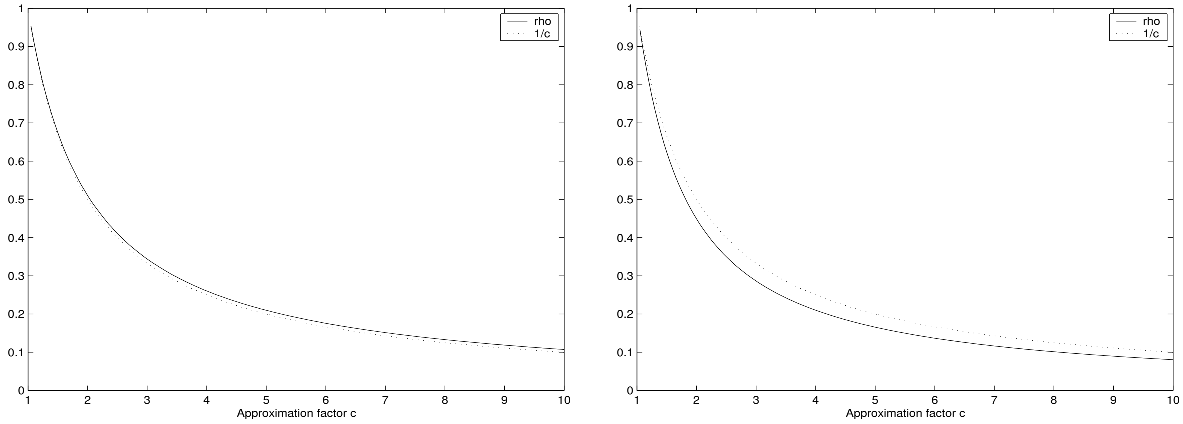
4. COMPUTATIONAL ANALYSIS OF THE RATIO $\rho = \frac{\ln 1/P_1}{\ln 1/P_2}$

In this section we focus on the cases of $p = 1, 2$. 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 [19] for these cases.

4.1 Computing the ratio ρ for special cases

For the special cases $p = 1, 2$ we can compute the probabilities p_1, p_2, using the density functions mentioned before. A simple calculation shows that $p_2 = 2\frac{tan^{-1}(r/c)}{\pi} - \frac{1}{\pi(r/c)}\ln(1 + (r/c)^2)$ for $p = 1$ (Cauchy) and $p_2 = 1 - 2norm(-r/c) - \frac{2}{\sqrt{2\pi}r/c}(1 - e^{-(r^2/2c^2)})$ for $p = 2$ (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, $\rho(c) = min_r \log(1/p_1)/\log(1/p_2)$, using *Matlab*. The plot of c versus $\rho(c)$ is shown in Figure 1. The crucial observation for the case $p = 2$ 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 [19]. 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.



(a) Optimal ρ for l_1

(b) Optimal ρ for l_2

Figure 1: Optimal ρ vs c

For the case $p = 1$, 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 $r = \infty$.

If one were to implement our LSH scheme, ideally they would want to know the optimal value of r for every c. For $p = 2$, for a given value of c, we can compute the value of r 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 r is more or less stable after a certain point (see Figure 2). Thus, we observe that ρ is not very sensitive to r beyond a certain point and as long we choose r "sufficiently" away from 0, the ρ value will be close to optimal. Note, however that we should not choose an r value that is too large. As r increases, both p_1 and p_2 get closer to 1. This increases the query time, since k, which is the "width" of each hash function (refer to Subsection 2), increases as $\log_{1/p_2} n$.

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

We also plot ρ as a function of c for a few fixed r values(See Figure 3). For $p = 2$, we observe that for moderate r values the ρ curve "beats" the $1/c$ curve over a large range of c that is of practical interest. For $p = 1$, we observe that as r increases the ρ curve drops lower and gets closer and closer to the $1/c$ curve.

5. EMPIRICAL EVALUATION OF OUR TECHNIQUE

In this section we present an experimental evaluation of our novel LSH scheme. We focus on the Euclidean norm case, since this occurs most frequently in practice. Our data structure is implemented for main memory.

In what follows, we briefly discuss some of the issues pertaining to the implementation of our technique. We then report some preliminary performance results based on an empirical comparison of our technique to the kd-tree data structure.

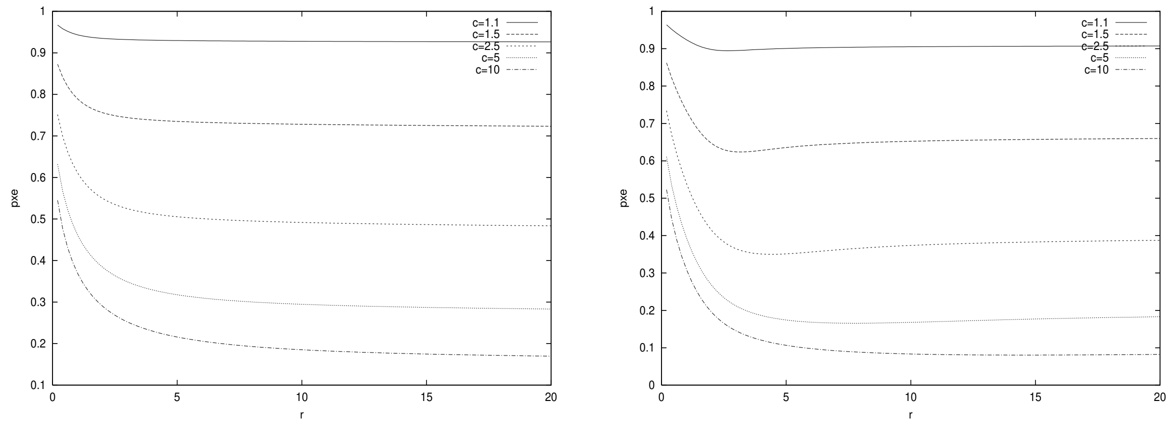
Parameters and Performance Tradeoffs: The three main parameters that affect the performance of our algorithm are: number of projections per hash value (k), number of hash tables (l) and the width of the projection (r). In general, one could also introduce another parameter (say T), such that the query procedure stops after retrieving T points. In our analysis, T was set to $3l$. In our experiments, however, the query procedure retrieved *all* points colliding with the query (i.e., we used $T = \infty$). This reduces the number of parameters and simplifies the choice of the optimal.

For a given value of k, it is easy to find the optimal value of l which will guarantee that the fraction of false negatives are no more than a user specified threshold. This process is exactly the same as in an earlier paper by Cohen et al. ([10]) that uses locality sensitive hashing to find similar column pairs in market-basket data, with the similarity exceeding a certain user specified threshold. In our experiments we tried a few values of k (between 1 and 10) and below we report the k that gives the best tradeoff for our scenario. The parameter k represents a tradeoff between the time spent in computing hash values and time spent in pruning false positives, i.e. computing distances between the query and candidates; a bigger k value increases the number of hash computations. In general we could do a binary search over a large range to find the optimal k value. This binary search can be avoided if we have a good model of the relative times of hash computations to distance computations for the application at hand.

Decreasing the width of the projection (r) decreases the probability of collision for any two points. Thus, it has the same effect as increasing k. As a result, we would like to set r as small as possible and in this way decrease the number of projections we need to make. However, decreasing r below a certain threshold increases the quantity ρ, thereby requiring us to increase l. Thus we cannot decrease r by too much. For the l_2 norm we found the optimal value of r using Matlab which we used in our experiments.

Before we report our performance numbers we will next describe the data set and query set that we used for testing.

Data Set: We used synthetically generated data sets and query points to test our algorithm. The dimensionality of the underlying l_2 space was varied between 20 and 500. We considered generating all the data and query points independently at random. Thus, for a data point (or query point) its coordinate along every dimen-



(a) ρ vs r for l_1 (b) ρ vs r for l_2

Figure 2: ρ **vs** r

sion would be chosen independently and uniformly at random from a certain range $[-a, a]$. However, if we did that, given a query point all the data points would be sharply concentrated at the same distance from the query point as we are operating in high dimensions. Therefore, the approximate nearest neighbor search would not make sense on such a data set. Testing approximate nearest neighbor requires that for every query point q, there are few data points within distance R from q and most of the points are at a distance no less than $(1+\epsilon)R$. We call this a "planted nearest neighbor model".

In order to ensure this property we generate our points as follows (a similar approach was used in [30]). We first generate the query points at random, as above. We then generate the data points in such a way that for every query point, we guarantee at least a single point within distance R and all other points are distance no less than $(1 + \epsilon)R$. This novel way of generating data sets ensures every query point has a few (in our case, just one) approximate nearest neighbors, while most points are far from the query.

The resulting data set has several interesting properties. Firstly, it constitutes the worst-case input to LSH (since there is only one correct nearest neighbor, and all other points are "almost" correct nearest neighbors). Moreover, it captures the typical situation occurring in real life similarity search applications, in which there are few points that are relatively close to the query point, and most of the database points lie quite far from the query point.

For our experiments the range $[-a, a]$ was set to $[-50, 50]$. The total number of data points was varied between 10^4 and 10^5. Both our algorithm and the kd-tree take as input the approximation factor $c = (1 + \epsilon)$. However, in addition to c our algorithm also requires as input the value of the distance R (upper bound) to the nearest neighbor. This can be avoided by guessing the value of R and doing a binary search. We feel that for most real life applications it is easy to guess a range for R that is not too large. As a result the additional multiplicative overhead of doing a binary search should not be much and will not cancel the gains that we report.

Experimental Results: We did three sets of experiments to evaluate the performance of our algorithm versus that of kd-tree: we increased the number n of data points, the dimensionality d of the data set, and the approximation factor $c = (1 + \epsilon)$. In each set of experiments we report the average query processing times for our algorithm and the kd-tree algorithm, and also the ratio of the two ((average query time for kd-tree)/(average query time for our algorithm)), i.e. the speedup achieved by our algorithm. We ran our experiments on a Sun workstation with 650 MHz UltraSPARC-IIi, 512KB L2 cache processor, having no special support for vector computations, with 512 MB of main memory.

For all our experiments we set the parameters $k = 10$ and $\ell = 30$. Moreover, we set the percentage of false negatives that we can tolerate up to 10% and indeed for all the experiments that we report below we did not get the more than 7.5% false negatives, in fact less in most cases.

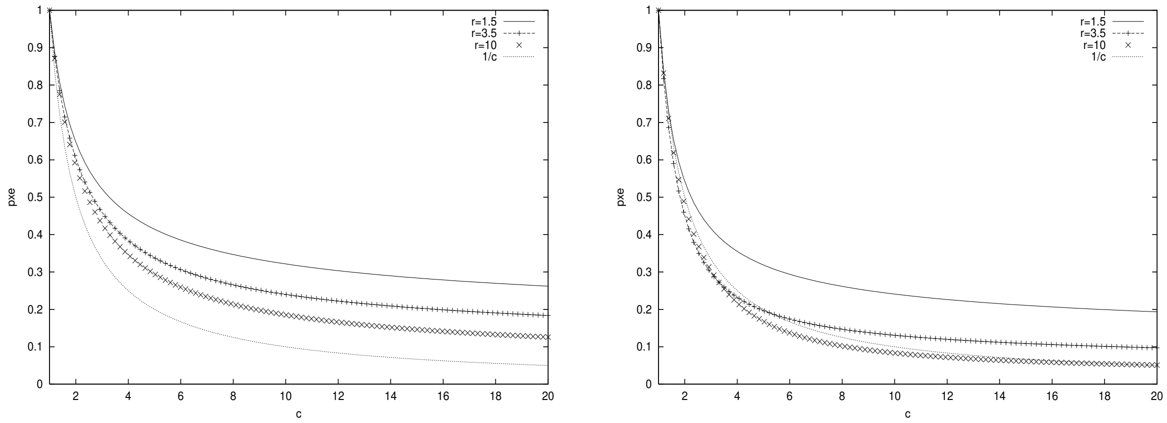
For all the query time graphs that we present, the curve that lies above is that of kd-tree and the one below is for our algorithm.

For the first experiment we fixed $\epsilon = 1$, $d = 100$ and $r = 4$. We varied the number of data points from 10^4 to 10^5. Figures 4(a) and 4(b) show the processing times and speedup respectively as n is varied. As we see from the Figures, the speedup seems to increase linearly with n.

For the second experiment we fixed $\epsilon = 1$, $n = 10^5$ and $r = 4$. We varied the dimensionality of the data set from 20 to 500. Figures 5(a) and 5(b) show the processing times and speedup respectively as d is varied. As we see from the Figures, the speedup seems to increase with the dimension.

For the third experiment we fixed $n = 10^5$ and $d = 100$. The approximation factor $(1 + \epsilon)$ was varied from 1.5 to 4. The width r was set appropriately as a function of ϵ. Figures 6(a) and 6(b) show the processing times and speedup respectively as ϵ is varied.

Memory Requirement: The memory requirement for our algorithm equals the memory to store the data points themselves and the memory required to store the hash tables. From our experiments, typical values of k and l are 10 and 30 respectively. If we insert each point in the hash tables along with their hash values and a pointer to the data point itself, it will require $l \cdot (k + 1)$ words (int) of memory, which for our typical k, l values evaluates to 330 words. We can reduce the memory requirement by not storing the hash value explicitly as concatenation of k projections, but instead hash these k values in turn to get a single word for the hash. This would reduce the memory requirement to $l \cdot 2$, i.e. 60 words per data point. If the data points belong to a high dimensional space (e.g., with 500 dimension or more), then the overhead of maintaining the



(a) ρ vs c for l_1

(b) ρ vs c for l_2

Figure 3: ρ **vs** c

hash table is not much (around 12% with the optimization above) as compared to storing the points themselves. Thus, the memory overhead of our algorithm is small.

6. CONCLUSIONS

In this paper we present a new LSH scheme for the similarity search in high-dimensional spaces. The algorithm is easy to implement, and generalizes to arbitrary l_p norm, for $p \in [0, 2]$. We provide theoretical, computational and experimental evaluations of the algorithm.

Although the experimental comparison of LSH and kd-tree-based algorithm suggests that the former outperforms the latter, there are several caveats that one needs to keep in mind:

- We used the kd-tree structure "as is". Tweaking its parameters would likely improve its performance.
- LSH solves the decision version of the nearest neighbor problem, while kd-tree solves the optimization version. Although the latter reduces to the former, the reduction overhead increases the running time.
- The approximate kd-tree algorithm can be ran with approximation parameter c that is much larger than the intended approximation. Although the resulting algorithm would provide very weak guarantee on the quality of the returned neighbor, typically the actual error is much smaller than the guarantee.

7. REFERENCES

[1] C. Aggarwal and D. Keim A. Hinneburg. On the surprising behavior of distance metrics in high dimensional spaces. *Proceedings of the International Conference on Database Theory*, pages 420–434, 2001.

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

[3] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, 1994.

[4] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbor meaningful? *Proceedings of the International Conference on Database Theory*, pages 217–235, 1999.

[5] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17:419–428, 2001.

[6] J. Buhler. Provably sensitive indexing strategies for biosequence similarity search. *Proceedings of the Annual International Conference on Computational Molecular Biology (RECOMB02)*, 2002.

[7] J. Buhler and M. Tompa. Finding motifs using random projections. *Proceedings of the Annual International Conference on Computational Molecular Biology (RECOMB01)*, 2001.

[8] 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.

[9] K. Clarkson. Nearest neighbor queries in metric spaces. *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 609–617, 1997.

[10] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding interesting associations without support prunning. *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, 2000.

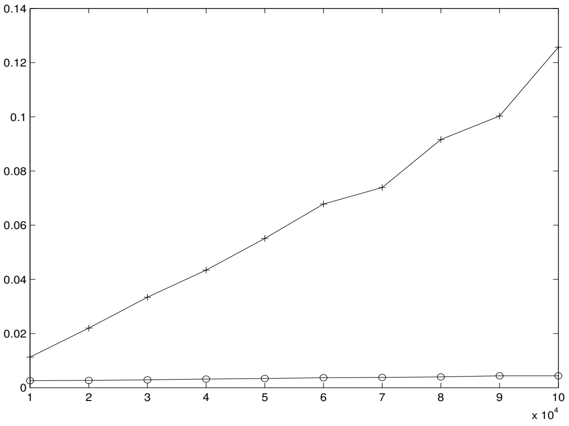
[11] G. Cormode, P. Indyk, N. Koudas, and S. Muthukrishnan. Fast mining of massive tabular data via approximate distance computations. *Proc. 18th International Conference on Data Engineering (ICDE)*, 2002.

[12] T. Darrell, P. Indyk, G. Shakhnarovich, and P. Viola. Approximate nearest neighbors methods for learning and vision. *NIPS Workshop at* http://www.ai.mit.edu/projects/vip/nips03ann, 2003.

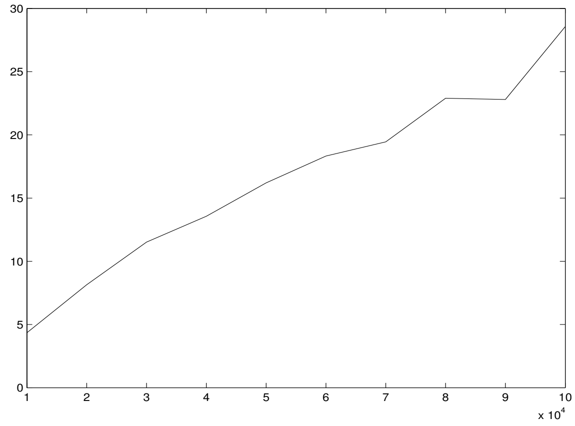
[13] B. Georgescu, I. Shimshoni, and P. Meer. Mean shift based clustering in high dimensions: A texture classificatio n example. *Proceedings of the 9th International Conference on Computer Vision*, 2003.

[14] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, 1999.

[15] S. Har-Peled. A replacement for voronoi diagrams of near linear size. *Proceedings of the Symposium on Foundations of*



(a) query time vs n



(b) speedup vs n

Figure 4: Gain as data size varies

Computer Science, 2001.

[16] T. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. *WebDB Workshop*, 2000.

[17] A. Hinneburg, C. C. Aggarwal, and D. A. Keim. What is the nearest neighbor in high dimensional spaces? *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 506–515, 2000.

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

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

[20] P. Indyk and N. Thaper. Fast color image retrieval via embeddings. *Workshop on Statistical and Computational Theories of Vision (at ICCV)*, 2003.

[21] D. Karger and M Ruhl. Finding nearest neighbors in growth-restricted metrics. *Proceedings of the Symposium on Theory of Computing*, 2002.

[22] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, 1997.

[23] R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[24] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *Proceedings of the Thirtieth ACM Symposium on Theory of Computing*, pages 614–623, 1998.

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

[26] Z. Ouyang, N. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of collections of files. *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, 2002.

[27] N. Shivakumar. *Detecting digital copyright violations on the Internet (Ph.D. thesis)*. Department of Computer Science, Stanford University, 2000.

[28] Roger Weber, Hans J. Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. *Proceedings of the 24th Int. Conf. Very Large Data Bases (VLDB)*, 1998.

[29] C. Yang. Macs: Music audio characteristic sequence indexing for similarity retrieval. *Proceedings of the Workshop on Applications of Signal Processing to Audio and Acoustics*, 2001.

[30] P.N. Yiannilos. Locally lifting the curse of dimensionality for nearest neighbor search. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2000.

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

APPENDIX

A. GROWTH-RESTRICTED DATA SETS

In this section we focus exclusively on data sets living in the l_2^d norm.

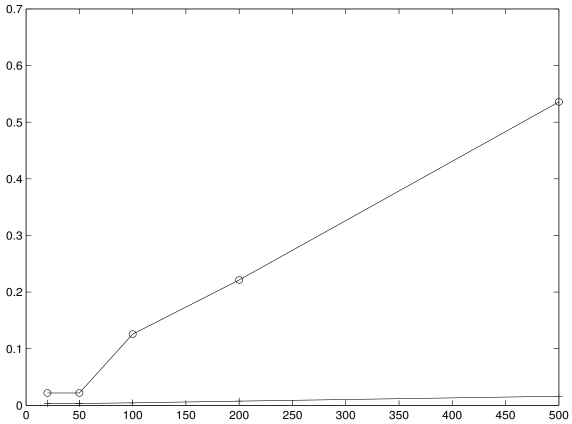
Consider a data set P, query q, and let p be the closest point in P to q. Assume we know the distance $\|p - q\|$, in which case we can assume that it is equal to 1, by scaling[1]. For $c \geq 1$, let $P(q, c) = P \cap B(q, c)$ and let $N(q, c) = |P(q, c)|$.

We consider a "single shot" LSH algorithm, i.e., that uses only $L = 1$ indices, but examines all points in the bucket containing q. We use the parameters $k = r = T \log n$, for some constant $T > 1$. This implies that the hash function can be evaluated in time $O(\log n)$.

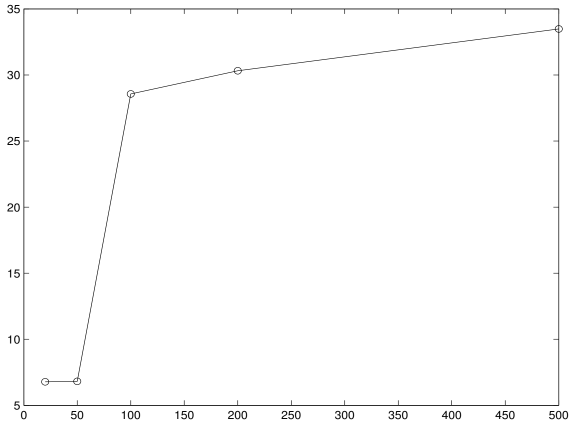
THEOREM 2. *If $N(q, c) = O(c^b)$ for some $b > 1$, then the "single shot" LSH algorithm finds p with constant probability in expected time $d(\log n + 2^{O(b)})$.*

Proof: For any point p' such that $\|p' - q\| = c$, the probability that $h(p') = h(q)$ is equal to $p(c) = \int_0^r \frac{1}{c} f_2(\frac{t}{c})(1 - \frac{t}{r})dt$, where

[1]Similar guarantees can be proved when we only know a constant approximation to the distance.



(a) query time vs dimension



(b) speedup vs dimension

Figure 5: Gain as dimension varies

$f_2(x) = \frac{2}{\sqrt{2\pi}} e^{-x^2/2}$. Therefore

$$\begin{aligned} p(c) &= 1 - \frac{2}{\sqrt{2\pi}} \int_0^r \frac{1}{c} e^{-(\frac{t}{c})^2/2} \frac{t}{r} dt \\ &= 1 - \frac{2}{\sqrt{2\pi}} \cdot \frac{c}{r} \int_0^r e^{-(\frac{t}{c})^2/2} \frac{t}{c^2} dt \\ &= 1 - \frac{2}{\sqrt{2\pi}} \cdot \frac{c}{r} \int_0^{r^2/(2c^2)} e^{-y} dy \\ &= 1 - \frac{2}{\sqrt{2\pi}} \frac{c}{r} [1 - e^{-r^2/(2c^2)}] \end{aligned}$$

If $c^2 \leq r^2/2$, then we have

$$1 - \frac{2}{\sqrt{2\pi}} \frac{c}{r} \leq p(c) \leq 1 - \frac{2}{\sqrt{2\pi}} \frac{c}{r} (1 - 1/e)$$

or equivalently $1 - Ac/r \leq p(c) \leq 1 - Bc/r$, for proper constants $A, B > 0$.

The probability that p collides with q is at least $(1 - A/r)^k \approx e^{-A}$, which is constant. Thus the algorithm is correct with constant probability.

Now consider the expected number of points colliding with q. Let C be a multiset containing all values of $c = \|p' - q\|/\|p - q\|$ over $p' \in P$. We have

$$\begin{aligned} E[|P \cap g^{-1}(q)|] &= \sum_{c \in C} p(c)^k \\ &= \sum_{c \in C, c \leq r/\sqrt{2}} p(c)^k + \sum_{c \in C, c > r/\sqrt{2}} p(c)^k \\ &\leq \sum_{c \in C, c \leq r/\sqrt{2}} (1 - Bc/r)^k + (1 - B)^r n \\ &\leq \int_1^{r/\sqrt{2}} (1 - Bc/r)^k N(q, c+1) dc + O(1) \\ &\leq \int_1^{r/\sqrt{2}} e^{-Bc} N(q, c+1) dc + O(1) \end{aligned}$$

If $N(q, t) = O(c^b)$, then we have

$$E[|P \cap g^{-1}(q)|] = O\left(\int_1^{r/\sqrt{2}} e^{-Bc} (c+1)^b dc \right) = 2^{O(b)}$$

B. ASYMPTOTIC ANALYSIS FOR THE GENERAL CASE

THEOREM 3. *For any $p \in (0, 2]$ there is a (r_1, r_2, p_1, p_2)-sensitive family $\mathcal{H}$ for l_p^d such that for any $\gamma > 0$,*

$$\rho = \frac{\ln 1/p_1}{\ln 1/p_2} \leq (1 + \gamma) \cdot \max\left(\frac{1}{c^p}, \frac{1}{c} \right).$$

We prove that for the general case ($p \in (0, 2]$) the ratio $\rho(c)$ gets arbitrarily close to $\max\left(\frac{1}{c^p}, \frac{1}{c}\right)$. For the case $p < 1$, our algorithm is the first algorithm to solve this problem, and so there is no existing ratio against which we can compare our result. However, we show that for this case ρ is arbitrarily close to $\frac{1}{c^p}$. The proof follows from the following two Lemmas, which together imply Theorem 3.

Let $l = \frac{1-p_2}{1-p_1}$, $x = 1 - p_1$. Then $\rho = \frac{\log(1-x)}{\log(1-lx)} \leq \frac{1-p_1}{1-p_2}$ by the following lemma.

LEMMA 1. *For $x \in [0, 1)$ and $l \geq 1$ such that $1 - lx > 0$,*

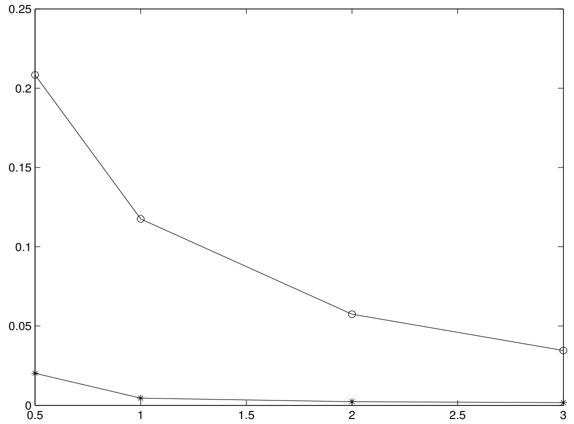
$$\frac{\log(1-x)}{\log(1-lx)} \leq \frac{1}{l}.$$

Proof: Noting $\log(1 - lx) < 0$, the claim is equivalent to $l \log(1 - x) \geq \log(1 - lx)$. This in turn is equivalent to

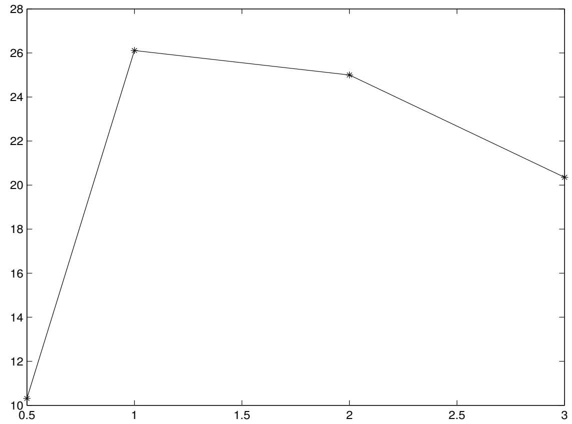
$$g(x) \equiv (1 - lx) - (1 - x)^l \leq 0.$$

This is trivially true for $x = 0$. Furthermore, taking the derivative, we see $g'(x) = -l + l(1 - x)^{l-1}$, which is non-positive for $x \in [0, 1)$ and $l \geq 1$. Therefore, g is non-increasing in the region in which we are interested, and so $g(x) \leq 0$ for all values in this region.

Now our goal is to upper bound $\frac{1-p_1}{1-p_2}$.



(a) query time vs ϵ



(b) speedup vs ϵ

Figure 6: Gain as ϵ varies

LEMMA 2. *For any $\gamma > 0$, there is $r = r(c, p, \gamma)$ such that*

$$\frac{1-p_1}{1-p_2} \le (1+\gamma)\cdot \max\left(\frac{1}{c^p}, \frac{1}{c}\right).$$

Proof: Using the values of p_1, p_2 calculated in Sub-section 3.2, followed by a change of variables, we get

$$\begin{aligned}\frac{1-p_1}{1-p_2} &= \frac{1-\int_0^r (1-\frac{t'}{r}) f(t')dt'}{1-\int_0^r (1-\frac{t'}{r})\frac{1}{c} f(\frac{t'}{c})dt'} \\ &= \frac{1-\int_0^r (1-\frac{t}{r}) f(t)dt}{1-\int_0^{r/c}(1-\frac{tc}{r}) f(t)dt} \\ &= \frac{(1-\int_0^r f(t)dt) + \frac{1}{r}\int_0^r tf(t)dt}{(1-\int_0^{r/c} f(t)dt) + \frac{c}{r}\int_0^{r/c} tf(t)dt}.\end{aligned}$$

Setting

$$F(x) = 1 - \int_0^x f(t)dt$$

and

$$G(x) = \frac{1}{x}\int_0^x tf(t)dt$$

we see

$$\begin{aligned}\frac{1-p_1}{1-p_2} &= \frac{F(r)+G(r)}{F(r/c)+G(r/c)} \\ &\le \max\left(\frac{F(r)}{F(r/c)}, \frac{G(r)}{G(r/c)}\right).\end{aligned}$$

First, we consider $p \in (0,2) - \{1\}$ and discuss the special cases $p = 1$ and $p = 2$ towards the end. We bound $F(r)/F(r/c)$. Notice $F(x) = \Pr_a[a > x]$ for a drawn according to the absolute value of a p-stable distribution with density function $f(\cdot)$. To estimate $F(x)$, we can use the Pareto estimation ([25]) for the cumulative distribution function, which holds for $0 < p < 2$,

$$\begin{aligned}&\forall \delta > 0\ \exists x_0 \text{ s.t. } \forall x \ge x_0, \\ &\quad C_p x^{-p}(1-\delta) \le F(x) \le C_p x^{-p}(1+\delta)\end{aligned}$$

where $C_p = \frac{2}{\pi}\Gamma(p)\sin(\pi p/2)$. Note that the extra factor 2 is due to the fact that the distribution function is for the absolute value of the p-stable distribution. Fix $\delta = \min(\gamma/4, 1/2)$. For this value of δ let r_0 be the x_0 in the equation above.

If we set $r > r_0$ we get

$$\begin{aligned}\frac{F(r)}{F(r/c)} &\le \frac{C_p r^{-p}(1+\delta)}{C_p (r/c)^{-p}(1-\delta)} \\ &\le \frac{r^{-p}(1+\delta)(1+2\delta)}{(r/c)^{-p}} \\ &\le \left(\frac{1}{c}\right)^p (1+4\delta) \\ &\le \left(\frac{1}{c}\right)^p (1+\gamma).\end{aligned}$$

Now we bound $G(r)/G(r/c)$. We break the proof down into two cases based on the value of p.

Case 1: $p > 1$. For these p-stable distributions, $\int_0^\infty tf(t)dt$ converges to, say, k_p (since the random variables drawn from those distributions have finite expectations). As $tf(t)$ is non-negative on $[0,\infty)$, $\int_0^x tf(t)dt$ is a monotonically increasing function of x which converges to k_p. Thus, for every $\delta' > 0$ there is some r_0' such that

$$(1-\delta')k_p \le \int_0^{r_0'} tf(t)dt.$$

Set $\delta' = \min(\gamma/2, 1/2)$ and choose $r' > cr_0'$. Then

$$\begin{aligned}\frac{G(r')}{G(r'/c)} &= \frac{\frac{1}{r'}\int_0^{r'} tf(t)dt}{\frac{c}{r'}\int_0^{r_0'} tf(t)dt + \frac{c}{r'}\int_{r_0'}^{r'/c} tf(t)dt} \\ &\le \frac{\frac{1}{r'}\int_0^{\infty} tf(t)dt}{\frac{c}{r'}\int_0^{r_0'} tf(t)dt} \\ &\le \frac{\frac{1}{r'}k_p}{\frac{c}{r'}(1-\delta')k_p} \\ &\le \frac{1}{c}(1+2\delta') \\ &\le \frac{1}{c}(1+\gamma). \qquad (1)\end{aligned}$$

Case 2: $p < 1$. For this case we will choose our parameters so that we can use the Pareto estimation for the density function. Choose x_0 large enough so that the Pareto estimation is accurate to within a factor of $(1 \pm \delta)$ for $x > x_0$. Then for $x > x_0$,

$$G(x) = \frac{1}{x}\int_0^{x_0} tf(t)dt + \frac{1}{x}\int_{x_0}^{x} tf(t)dt$$
$$< \frac{1}{x}\int_0^{x_0} tf(t)dt + \frac{1+\delta}{x}\int_{x_0}^{x} pC_p t^{-p}dt$$
$$= \frac{1}{x}\int_0^{x_0} tf(t)dt + (\frac{pC_p}{x(1-p)}x^{-p+1} - \frac{pC_p}{x(1-p)}x_0{}^{-p+1})(1+\delta)$$
$$= \frac{1}{x}(\int_0^{x_0} tf(t)dt - \frac{pC_p(1+\delta)}{(1-p)}x_0{}^{-p+1}) + \frac{1}{x^p}(\frac{pC_p}{(1-p)}(1+\delta)).$$

Since x_0 is a constant that depends on δ, the first term decreases as $1/x$ while the second term decreases as $1/x^p$ where $p < 1$. Thus for every δ' there is some x_1 such that for all $x > x_1$, the first term is at most δ' times the second term. We choose $\delta' = \delta$. Then for $x > \max(x_1, x_0)$,

$$G(x) < (1+\delta)^2 \left(\frac{pC_p}{(1-p)x^p}\right).$$

In the same way we obtain

$$G(x) > (1-\delta)^2 \left(\frac{pC_p}{(1-p)x^p}\right).$$

Using these two bounds, we see for $r > c\max(x_1, x_0)$,

$$\begin{aligned}\frac{G(r)}{G(r/c)} &< \frac{(1+\delta)^2\left(\frac{pC_p}{(1-p)r^p}\right)}{(1-\delta)^2\left(\frac{pC_p}{(1-p)(r/c)^p}\right)}\\ &\leq \frac{1}{c^p}(1+9\delta)\\ &\leq \frac{1}{c^p}(1+\gamma)\end{aligned}$$

for $\delta < \min(\gamma/9, 1/2)$.

We now consider the special cases of $p \in \{1, 2\}$. For the case of $p = 1$, we have the Cauchy distribution and we can compute directly $G(r) = \frac{\ln(r^2+1)}{\pi r}$ and $F(r) = 1 - \frac{2}{\pi}\tan^{-1}(r)$. In fact for the ratio $\frac{F(r)}{F(r/c)}$, the previous analysis for general p works here. As for the ratio $\frac{G(r)}{G(r/c)}$, we can prove the upper bound of $\frac{1}{c}$ using L'Hopital rule, as follows:

$$\begin{aligned}\lim_{r\to\infty}\frac{G(r)}{G(\frac{r}{c})} &= \lim_{r\to\infty}\frac{\ln(r^2+1)}{c\ln((r/c)^2+1)}\\ &= \lim_{r\to\infty}\frac{\frac{2r}{(r^2+1)}}{c(\frac{2r}{c^2(r^2/c^2+1)})}\\ &= \lim_{r\to\infty}\frac{c^2(r^2/c^2+1)}{c(r^2+1)}\\ &= \lim_{r\to\infty}\frac{c^2(2r/c^2)}{2cr}\\ &= \frac{1}{c}.\end{aligned}$$

Also for the case $p = 2$, i.e. the normal distribution, the computation is straightforward. We use the fact that for this case $F(r) \simeq f(r)/r$ and $G(r) = \frac{2}{\sqrt{2\pi}}\frac{1-e^{\frac{-r^2}{2}}}{r}$, where $f(r)$ is the normal density function. For large values of r, $G(r)$ clearly dominates $F(r)$, because $F(r)$ decreases exponentially ($e^{-r^2/2}$) while $G(r)$ decreases as $1/r$. Thus, we need to approximate $\frac{G(r)}{G(r/c)}$ as r tends to infinity, which is clearly $\frac{1}{c}$.

$$\lim_{r\to\infty}\frac{G(r)}{G(\frac{r}{c})} = \lim_{r\to\infty}\frac{1-e^{\frac{-r^2}{2}}}{c(1-e^{-\frac{r^2}{2c^2}})} = \frac{1}{c}$$

Notice that similar to the previous parts, we can find the appropriate $r(c, p, \gamma)$ such that $\frac{1-p_1}{1-p_2}$ is at most most $(1+\gamma)\frac{1}{c}$.