Information Systems **University of Zurich**
Seminar: Database Systems Alphonse Mariyagnanaseelan, 15–712–698

Locality-Sensitive Hashing Scheme Based on p-Stable Distributions [1]

1 Introduction

Generally, hashing algorithms, that are for example used for cryptographic purposes, are not locality sensitive. That means the probability for a collision is not dependent on the difference between the values that are being hashed. When we want to make a query using an approximate value, what we want to receive is the nearest neighbors of that query point. So what we want is a hashing algorithm that collides (or in other words: Maps to the same hash value) with a high probability, if the input values are close to each other (or *similar*), and collides only with a small probability if the input values are far apart. Hashing schemes that have these properties are called locality sensitive hashing schemes.

There already exist locality sensitive hashing schemes, one of them [2] uses mappings into the Hamming space. Roughly, there you split the space using hyperplanes and then map the points depending on which side of the hyperplanes they were located. In this paper [1] the authors propose a new method for a locality sensitive hashing scheme, that does not use mappings to the Hamming space, but is based on p-stable distributions.

We will start with an example. Assume we have phone numbers consisting of one digit. If we want to see how many phone calls were made from one phone number to another, we could make a matrix as shown in figure 1. The y-axis contains the caller's phone numbers and the x-axis contains the receiver's numbers.

Let's say we are searching for a person who made one phone call to the number 1 and six calls to the number 3. We could formulate this query as $q = (0, 1, 0, 6, 0, 0, 0, 0, 0, 0)$. As we can see in the matrix (figure 1), there is no such person, but the person with the number 5 had a similar behavior to what we were searching. So the problem we want to solve now is a nearest neighbor problem.

To formalize the conditions of our search, we first have to choose a distance metric. We can calculate the distance (or difference) between two points using norms. Norms that are commonly used as distance metrics are for example:

- The Euclidean norm $l_2(v)$ for a d dimensional vector v is $\sqrt{\sum_{i=1}^{d} |v_i|^2}$
- The Manhattan norm $l_1(v)$ for a d dimensional vector v is $\sum_{i=1}^{d} v_i$

	0	1	2	3	4	5	6	7	8	9
0		4							6	1
1	2									
2						6				
3						5				
4										2
5		1		5						
6										
7									9	1
8	2							8		
9	5				3					

Figure 1: Cellphone calling behavior - Caller on y-axis and receiver on x-axis

These norms can be formulated as:

$$l_p(v) = \left(\sum_{i=1}^{d} |v_i|^p \right)^{1/p}$$

First we will take a look at the different Near Neighbor problems to better define which points we expect as result to our query point.

2 Near Neighbor Problems

The classical Nearest Neighbor problem is to find the closest point v for a query point q. The distance between those two points we can measure using any norm $\|\cdot\|_p$, the p depends on the application.

The Near Neighbor problem is to find all (or most) points v for a query q around a fixed radius R. Thus a point v is a near neighbor of q if $\|q - v\|_p \leq R$.

Because the problem of finding the exact nearest neighbor is a difficult problem, we will look at the approximation versions of those problems. The Approximate Nearest Neighbor problem is defined as:

Definition 1 *If q is a query on the set V and $v_q \in V$ is the nearest neighbor of q, then a point $v \in V$ is an ϵ-Approximate Nearest Neighbor (ϵ-NN) if $\|q - v\|_p \leq (1 + \epsilon) \cdot \|q - v_q\|_p$.*

If the distance between the query point and the true nearest neighbor is c, then any point that has a distance of at most $(1 + \epsilon)c$ to the query point is called an approximate nearest neighbor. The problem with this definition is that we still first have to find the nearest neighbor. We can look at an easier problem, the Approximate Near Neighbor:

Definition 2 *If q is a query on the set V and there is any point $v \in V$ such that $\|q - v\|_p \leq R$, then any point within the radius $(1+\epsilon)R$ around q is called an (R, ϵ)-near neighbor of q.*

The difference here is that we search in a fixed radius R. We have to note though, that a (R, ϵ)-NN is only guaranteed to be an ϵ-NN if R equals to the distance between q and the true near neighbor ($R = \|q - v_q\|_p$).

In [2], it is shown that the ϵ-NN can be reduced to the (R, ϵ)-NN problem. So if we can find an efficient solution to the (R, ϵ)-NN problem, then we have an efficient solution to the ϵ-NN problem too. Therefore the paper [1] focuses on solving the (R, ϵ)-Near Neighbor problem.

3 Locality Sensitive Hashing

We have now decided that we will be solving the (R, ϵ)-Near Neighbor problem. We want to have a hashing algorithm that helps solving this problem efficiently. The requirements can so far be formulated as:

- If point v is an (R, ϵ)-NN of query q, then $h(q) = h(v)$
- If point v is not an (R, ϵ)-NN of query q, then $h(q) \neq h(v)$ (as much as possible)

As mentioned before, common hashing algorithms are not locality sensitive, so for our purpose a hashing algorithm has to fulfill some additional requirements. The (R, ϵ)-Near Neighbor definition can be nicely transformed to the definition of locality sensitive hashing. Let $r_1 = R$ and $r_2 = (1 + \epsilon)R$. We can formulate the requirements more concretely, by using probabilities:

- If two points are close (the distance is smaller than r_1) then they must collide with a high probability (at least p_1)
- If two points are far apart (the distance is greater than r_2) then they must collide with a low probability (at most p_2)

If a hashing algorithm fulfills the above conditions for given r_1, r_2, p_1 and p_2 then the hashing algorithm is part of the (r_1, r_2, p_1, p_2)-sensitive hashing family $\mathcal{H}$.

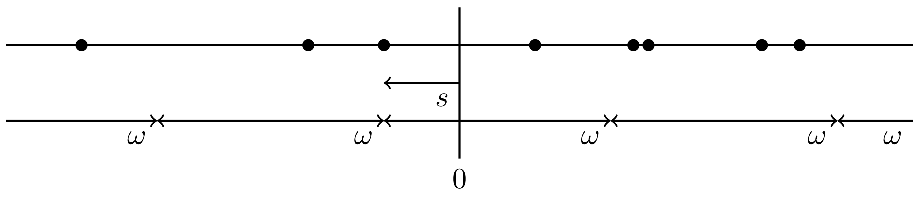

Figure 2: Split into buckets with a random shift

3.1 Locality Sensitive Hashing based on p-Stable Distributions

The proposed hashing algorithm [1] first projects the input data point onto the 1-dimensional line $\mathbb{R}$. We first create a vector a, where every element is chosen randomly. The vector a must have the same dimensionality as an input vector v. The dot product $a \cdot v$ will be the projection of v onto the 1-dimensional line.

Now the projection will be randomly distributed, depending on how we chose the elements of the vector a. We assume there exists a distribution, such that if we chose the elements of the vector a randomly from that distribution, then the projections of two vectors v_1 and v_2 will be close together if they are close together in reality, and the projections will be far apart if they are far apart in reality.

We cut the 1-dimensional line with the projections into equally sized segments of size ω. Each segment will define a bucket for the hashing algorithm, thus every vector that is projected into the same segment will be placed in the same bucket. We allow the buckets to be shifted, so that they don't necessarily have to begin at 0 (see figure 2).

As hash function, we will use $h(v) = \lfloor \frac{a \cdot v + s}{\omega} \rfloor$, where a is like before a vector with random elements. Like mentioned, we allow a random shift s, thus we pick s uniformly and at random from $[0, \omega]$.

We can amplify the gap between p_1 and p_2 by using multiple hash functions from the chosen LSH family $\mathcal{H}$. For that, we randomly select k such hash functions and concatenate them to a function $g(v)$:

$$g(v) = \begin{pmatrix} h_1(v) \\ h_2(v) \\ \vdots \\ h_k(v) \end{pmatrix}$$

For the final hashing scheme, we will use L different composed hashing functions $g_L(v)$. Now we can optimize bucket size ω, number of hash functions k and number of composed hash functions L depending on the data. We will take a look at some optimal values provided by [1] and [2] and explain why those values are valid choices.

Our index structure is ready now, and we can generate the index for a set of points V. For every point $v \in V$ we do:

1. Evaluate $g_1(v), \ldots, g_L(v)$
2. For every resulting bucket g_i, insert reference to v

After that, to process a query q we do:

1. Evaluate the hash functions $g_1(q), \ldots, g_L(q)$
2. For every bucket $g_i(q)$, retrieve all resulting points
3. Calculate distance between q and the retrieved points
4. If we find a near neighbor ($\|q - v\|_p \leq R$) we return it, otherwise we stop after trying $3L$ points

4 p-Stable Distributions

Before, we assumed that there exists a distribution $\mathcal{D}$, such that for a vector a where each element is chosen from $\mathcal{D}$:

- If two vectors v_1 and v_2 are close together, then their projection on the 1-dimensional line is close together
- If two vectors v_1 and v_2 are far apart, then their projection on the 1-dimensional line is far apart (as much as possible)

Stable distributions are a family of probability distributions that share a few characteristics, one of which is the probability density function, that looks like a bell curve. Thus values, that are near the peak of the probability density function are drawn with a higher probability than values that are far away from that peak. Two well-known examples of p-stable distributions are:

- The standard Cauchy distribution, that is 1-stable and has a probability density function of $f_1(x) = \frac{1}{\pi}\frac{1}{1+x^2}$
- The standard Gaussian distribution, that is 2-stable and has a probability density function of $f_2(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$

The most interesting characteristic for us is (from [1]):

Definition 3 *A distribution $\mathcal{D}$ is called p-stable, if there exists $p \geq 0$ such that for any n real numbers $b_1, \ldots, b_n$ and independent identically distributed variables $X_1, \ldots, X_n$ with distribution $\mathcal{D}$, the random variable $\sum_i b_i X_i$ has the same distribution as the random variable $(\sum_i |b_i|^p)^{1/p} X$, where X is a random variable with distribution $\mathcal{D}$.*

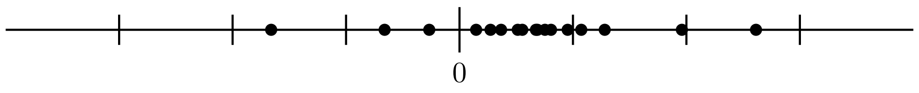

Figure 3: Multiple projections of a vector v

In our case, the numbers b_i can be associated with the elements of a vector v. For the random vector a we could chose the elements the way we chose the random variables X_i. The dot product $a \cdot v$ is then the sum of products of corresponding vector elements, like in definition 3.

Recall the definition of norms:

$$l_p(v) = \|v\|_p = \left(\sum_{i=1}^{d} |v_i|^p \right)^{1/p}$$

The distance between the projections of two vectors v_1 and v_2 is:

$$v_1 \cdot a - v_2 \cdot a = (v_1 - v_2) \cdot a \quad \sim \quad \|v_1 - v_2\|_p \cdot X$$

If we repeat this for a few random vectors a, we will receive some points on the 1-dimensional line. The projected points will have the same distribution as the distribution $\mathcal{D}$ where we chose the elements of a from, which was a p-stable distribution. Considering the probability density function, we can see intuitively that if we produce a few projections of a vector using different random vectors, the projections may look similar to figure 3.

The proposed method works for any p-stable distribution, since no other constraints were made regarding that aspect. It is known that p-stable distributions exist for all $p \in (0, 2]$ (see [3]). The probability density function cannot be formulated for every p-stable distribution, in fact only for a few special cases this is possible (a few of which we mentioned before). But it is still possible to generate p-stable random variables by sampling (see [4]).

5 Analysis

In this part, we try to verify some claims of the paper by using the proposed values for k and L. In the first part we check if the hashing functions itself are indeed locality sensitive. In the second part, we check if the composed, final hashing function holds the conditions given by locality sensitivity.

5.1 Single-Component Hashing Algorithm

For this part, we will consider a hashing function $g(v)$ consisting of only one component, thus $k = 1$. For that reason, in this section we will only look at functions $h(v)$ as we defined before.

The hashing algorithm was defined as $h(v) = \lfloor \frac{a \cdot v + s}{\omega} \rfloor$, with a random vector a and a random shift s. We want to calculate the probability that for a given random vector a, the query point q and a vector v will be mapped to the same bucket. Let $c = \|q - v\|_p$ be the distance between q and v, and $f(x)$ be the probability density function of the absolute value of the p-stable distribution. We consider the probability density function of the absolute value, because we're only interested in the distance, not the direction.

We can define the probability of a collision as follows:

$$\Pr = \left[\left\lfloor \frac{a \cdot q + s}{\omega} \right\rfloor = \left\lfloor \frac{a \cdot v + s}{\omega} \right\rfloor \right]$$

A collision can only occur if both of these conditions are hold:

1. The distance between q and v is smaller than the bucket size ($|q \cdot a - v \cdot a| < \omega$)
2. The points do not get inserted into neighboring buckets (thus there is no boundary between $q \cdot a$ and $v \cdot a$)

Let $c = \|q - v\|_p$. Due to p-stability, we can rewrite the first condition as

$$|(q - v) \cdot a| < \omega \quad \iff \quad |cX| < \omega$$

where X has the same distribution like the chosen p-stable distribution. Assume c is positive. We can formulate this as an integral:

$$X < \frac{\omega}{c} \quad \implies \quad p(c) = \int_0^{\omega/c} f(u)\, du$$

The probability that the boundary lies between $q \cdot a$ and $v \cdot a$ is

$$\frac{|(q - v) \cdot a|}{\omega} = \frac{|cX|}{\omega}$$

We can formulate this as integral again, using the upper bound given by the first condition:

$$1 - \frac{|cX|}{\omega} \quad \implies \quad p(c) = \int_0^{\omega/c} 1 - \frac{c \cdot u}{\omega}\, du$$

We can now combine those probabilities and we receive the probability of a collision

$$p(c) = \int_0^{\omega/c} f(u) \left(1 - \frac{c \cdot u}{\omega}\right) du$$

If we now substitute the equation with $t = c \cdot u$ we receive

$$p(c) = \int_0^{\omega} \frac{1}{c} f\left(\frac{t}{c}\right) \left(1 - \frac{t}{\omega}\right) dt$$

For a fixed bucket size ω, the probability only depends on the distance c between the points q and v. As we can see, this probability decreases monotonically with the distance. Thus, for any p-stable distribution with absolute probability density function $f(x)$ this hashing algorithm is (r_1, r_2, p_1, p_2)-sensitive, for any r_1 and r_2 where $r_1 < r_2$.

5.2 Composed Hashing Algorithm

We want to show that for the proposed values (see [1] and [2])

$$k = \log_{1/p_2}(n)$$

and

$$L = n^{\rho} \quad \text{with} \quad \rho = \frac{\ln(1/p_1)}{\ln(1/p_2)}$$

that the following two conditions hold:

- If there is a (R, ω)-Near Neighbor $v \in V$ for a query q (thus $\|q - v\|_p = c \leq R$), then with constant probability it must hold that for some $i \in 1, \ldots, L$ the hash values should collide ($g_i(q) = g_i(v)$).
- The total number of collisions with points $v \in V$ such that $\|q - v\|_p = c > (1 + \epsilon)R$ is less than $3L$.

We can provide a lower bound for the probability defined in the first condition. For a fixed i we get:

$$\Pr\left[g_i(q) = g_i(v)\right] \geq p_1^k$$

We use $k = \log_{1/p_2}(n)$:

$$p_1^k = p_1^{\log_{1/p_2}(n)} = n^{\log_{1/p_2}(p_1)} = n^{\frac{\log(p_1)}{\log(1/p_2)}} = n^{-\frac{\log(1/p_1)}{\log(1/p_2)}} = n^{-\frac{\ln(1/p_1)}{\ln(1/p_2)}}$$

We use $\rho = \frac{\ln(1/p_1)}{\ln(1/p_2)}$, where $0 < \rho < 1$ due to $p_1 > p_2$. The probability of a collision now for at least one i is:

$$\Pr\left[\exists i : g_i(q) = g_i(v)\right] \geq 1 - \left(1 - n^{-\rho}\right)^L$$

We use $L = n^\rho$ and $\lim_{m\to\infty}\left(1 - m^{-1}\right)^m = \frac{1}{e}$:

$$1 - \left(1 - n^{-\rho}\right)^{n^\rho} \geq 1 - \frac{1}{e} > 63\%$$

For the second condition, we assume there is $u \in V$ with $\|q - u\|_p > (1-\epsilon)R$:

$$\Pr\left[g_i(q) = g_i(u)\right] \leq p_2^k = p_2^{\log_{1/p_2}(n)} = n^{\log_{1/p_2}(p_2)} = n^{-\frac{\log(1/p_2)}{\log(1/p_2)}} = \frac{1}{n}$$

For a set V of size n, the probability of a "bad" collision is thus at most 1. For L different hashing functions the number of collisions is then at most L. Let C be the number of collisions where $\|q - u\|_p > (1-\epsilon)R$. By using Markov's inequality, we can show:

$$\Pr\left[C \geq 3L\right] \leq \frac{E(C)}{3L} = \frac{L}{3L} = \frac{1}{3}$$

6 Conclusion

The proofs mentioned in the paper [1] are replicable, but some of them are not carried out explicitly. However, the authors documented the empirical evaluation thoroughly and comprehensible.

The paper presented a new locality sensitive hashing scheme that is based on p-stable distributions and works for all corresponding distance metrics l_p, where $0 < p \leq 2$, and it is also the first algorithm for $p < 1$.

The algorithm is easy to implement, as it only requires dot product calculations, drawing random values and trivial operations. Since it is a hashing index, it can be efficiently used in dynamic setups with lots of insert and update operations.

References

[1] M. Datar, et al., *Locality-sensitive hashing scheme based on p-stable distributions*, Proceedings of the twentieth annual symposium on Computational geometry, ACM, 2004.

[2] P. Indyk and R. Motwani., *Approximate nearest neighbor: towards removing the curse of dimensionality*, Proceedings of the thirtieth annual ACM symposium on Theory of computing, ACM, 1998.

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

[4] J. M. Chambers, C. L. Mallows and B. W. Stuck, *A method for simulating stable random variables*, Journal of the American Statistical Association, *71*(354), 340-344, JSTOR, 1976.