Indexing Multidimensional Data with Hashing

1st Jorge Fiestas
dept. of Computer Science
UTEC
Lima, Peru
jorge.fiestas@utec.edu.pe

2nd María Lovatón
dept. of Computer Science
UTEC
Lima, Peru
maria.lovaton@utec.edu.pe

3rd César Salcedo
dept. of Computer Science
UTEC
Lima, Peru
cesar.salcedo@utec.edu.pe

4th Roosevelt Ubaldo
dept. of Computer Science
UTEC
Lima, Peru
roosevelt.ubaldo@utec.edu.pe

I. Introduction

Hashing is a tool that can be found in most fields of computing, as it allows to map data from a key to a value. For simple data types, such as integers or characters, it is simple to generate Hashing functions that are fast and have low rates of collision. However, sometimes keys have a more complex structure and depend on multiple values or *dimensions*. This leads to an overall low performance by using the same hashing functions.

The effects of slow hashing methods are specially significant in one of it's main applications: *Indexing*. In the context of databases and data management, indexing is the process that creates ordered structures that allow for fast querying. Because of this, better performing hashing algorithms can translate to better performing databases.

In this work we will analyze *Locality-Sensitive Hashing (LSH)*, a probabilistic hashing algorithm that is useful to find similar data points in any number of dimensions. We will analyze its time and space complexity, as well as explain the key ideas behind this algorithm. Then, we will evaluate a LSH index for similarity search and compare its performance with a KD-tree index.

II. Background

A. The Nearest-Neighbor Search Problem

One of the ways to approach the indexing problem is to reduce it to the nearest neighbor problem. Given a set P of n points in a d-dimensional space $\mathbb{R}^d$, the goal is to construct a data structure which given any query point q finds the point in P that is closest to q.

When the points are in 1-dimensional space the problem can be solved in $O(\log n)$ time with $O(n)$ space. However, when scaling to higher dimensions, the time complexity of known solutions increases at least linearly in d.

A naive brute-force approach to solve this problem would be to compute the distances from q to all other points in P and return the point with minimum distance. It is simple to see that this approach has a time complexity $\Theta(dn)$, as n comparisons are needed, each taking d operations. This performance shows useful when working with small values of n and d, although this algorithm doesn't scale up nicely when working with millions of points and hundreds of dimensions.

Allowing the answers to be approximations rather than certain nearest neighbors reduces the dependency on dimensions, falling from linear to sub-linear algorithms. This idea can be formalized as the approximate nearest neighbor search problem, which is defined as follows.

Definition 1.1 (c-Approximate nearest neighbor) Given a set P of points in a d-dimensional space $\mathbb{R}^d$, construct a data structure which given any query point q, reports any point within distance at most c times the distance from q to p, where p is the point in P closest to q.

B. Locality-Sensitive Hashing (LSH)

To compute the approximate nearest-neighbor, an LSH algorithm can be used as it ensures the probability of collision of nearby points is greater than the one of distant points. This property gives us a way of obtaining close enough points to compute the nearest-neighbor problem in sublinear time, by choosing specific LSH function families. We describe some algorithms in the next paragraphs.

Indyk and Motwani developed a randomized algorithm in 2004 to solve the approximate nearest-neighbor problem under l_p norm, based on p-stable distributions [1]. The main idea is to create a hash function by selecting a random group of k *hyperplanes*. Each hyperplane would divide the d-dimensional space in two halves, therefore generating at most 2^k regions such that every point in P lies within one of these regions. It can be intuitively seen that the closer two points are to each other, the more likely it is they are in the same region. As a result, this hash functions would have the properties needed for *LSH* as it maps a point in space to its respective region.

The main idea of the hash function is that every region can be represented as a k-bit string in which the ith bit is 0 if the region is to the left of the hyperplane and 1 if it is to the right. In the same way, the region to which any point in the plane belongs can be expressed as this k-bit string. Therefore, by using a hash-table any point in the plane can be mapped to a binary string and grouped with other nearby points in P.

Following this approach, the algorithm can now just compare the point to other points in the same region, instead of all points in P. This allows for an overall lower time complexity and a more efficient algorithm.

A formal definition of an LSH function family comes as follows. Let D be a distance function of elements from a set

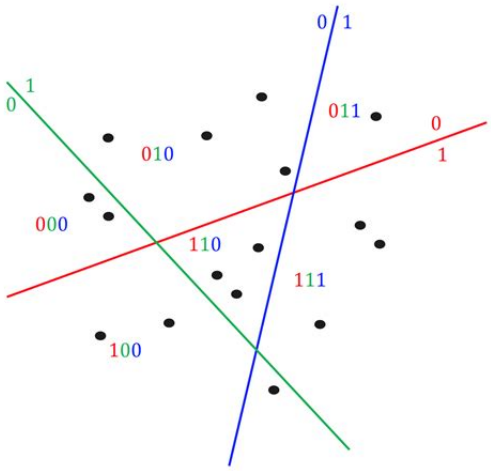

Fig. 1. Example of hyperplanes

S in the universe U, and for any $p \in S$ let $B(q, r)$ denote the set of elements from S within the distance r from p.

Definition 1.2 (Locality-Sensitive Hashing) 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 $Pr_{\mathcal{H}}[h(q) = h(v)] \leq p_2$

For an LSH family to be useful it has to satisfy $p_1 > p_2$ and $r_1 < r_2$.

Alternatively, we can define the (R, c)-NN problem, which sets $r_1 = R$ and $r_2 = cR$. In both problems, the need for *amplification* arises since applying the hash function once may result in multiple points in the same region, or none at all. In this context, given a family $\mathcal{H}(r_1, r_2, p_1, p_2)$ of hash functions, the gap between "high" probability p_1 and "low" probability p_2 is amplified by concatenating several functions. For a parameter k, a function family $\mathcal{G} = \{g : S \to U^k\}$ is defined, such that $g(v) = (h_1(v), \ldots, h_k(v))$, where $h_i \in \mathcal{H}$. Then, for an integer L, functions $g_1, \ldots, g_L$ are chosen from $\mathcal{G}$, independently and uniformly at random. As each point $p \in P$ has been stored in the bucket $g_j(p)$ for $j = 1, \ldots, L$ in the pre-processing step, and the total number of buckets may be large, only the non-empty buckets are retained by resorting to hashing.

To process a query point q, we search for every point in buckets $g_1(q), \ldots, g_L(q)$ until we get the first $3L$ points (including duplicates). If $v_j \in B(q, r_2)$ for any point $v_1, \ldots, v_t$ obtained in the previous step, we return $true$ and v_j, otherwise we return $false$.

The parameters k and L are chosen to ensure that the following properties hold with a constant probability.

1) If there exists $v^* \in B(q, r_1)$ then $g_j(v^*) = g_j(q)$ for some $j = 1 \ldots L$
2) The total number of collisions of q with points from $P - B(q, r_2)$ is less than $3L$ $\sum_{j=1}^{L} |(P - B(q, r_2)) \cup g_j^{-1}(g_j(q))| < 3L$

If items (1) and (2) hold, then the algorithm is correct.

III. Bibliographic revision

Multiple LSH families have been proposed to solve the approximate near neighbor problem. A family of LSH functions discovered by [1] achieves near optimal query time for the algorithm. It first performs a random projection of points in $\mathbb{R}^n$ to $\mathbb{R}^t$ with $t = o(\log n)$. Then, the resulting space is divided in cells using the "ball partitioning" method, in which a sequence of balls $B_1, B_2, ...$ with radius w and centers chosen independently at random is selected. By placing those balls in a repeating grid, one can fill up all space and assign a query point to a ball B_i such that $B_i \setminus \cup_{j<i} B_j$. For a carefully chosen value of w, this construction yields a query time of $\Theta(n^\rho)$ with $\rho = \frac{1}{c^2}$, which is close to optimal by a constant factor [2]. Using this family, indexing data structures for similarity search can be built efficiently. The similarity search problem is closely related to the nearest neighbor search problem.

The basic LSH method uses a family of locality-sensitive hash functions to hash nearby objects in the high-dimensional space into the same bucket. However, this method needs to create over a hundred hash tables to find a good candidate set [3]. At the same time, this basic approach doesn't handle space efficiently because the size of each hash table is proportional to the number of data objects. As a result, other approaches are preferred.

An entropy-based LSH method was proposed to trade time for space requirements [4]. It only uses one or a few hash tables and hash several randomly chosen points in the neighborhood of the query point showing that at least one of them will hash to the bucket containing its nearest neighbor. The selection of points depends on the entropy of their hash values at a distance r from q and using the locality preserving hash function. In their study, this method computes the c-nearest neighbor in time n^p with $p \approx 2.06/c$ and space $O(n)$.

A multi-probe LSH method was proposed to improve the exploration of hash buckets [5]. It uses a derived probing sequence to check multiple buckets that are likely to contain the nearest neighbors of q. A hash perturbation vector is applied to the hash values of q to probe multiple buckets and find the ones that are close by. This method was proved to be more time efficient than the entropy-based LSH method because it doesn't depend on the nearest neighbor distance.

IV. Analysis and Design

In this project, the basic approach of the LSH method is developed.

1) Generate random hyperplanes: h_1, h_2, h_3
2) Hash-code for point a: $H(a^T h_1, a^T h_2, a^T h_3) = 100$
3) Compare a with the other points with the same hash-code
4) Repeat with different set of hyperplanes

A. Time and Space Complexity Analysis

The LSH algorithm has two parts: pre-processing and querying. The pre-processing consists in building the L hash-tables in which all the points in P are grouped. It is simple to see that this has a complexity of $\Theta(Lndk)$, as hashing a point has a complexity of $\Theta(d)$ and this has to be done n times for each of the L tables.

Querying, on the other hand, has a complexity of $\Theta(Lkd + Ld\frac{n}{2^k})$. This is because generating the hash of a point has a complexity of kd as for each hyperplane the algorithm needs to check if it is to its left or right. Then, the distance to each point in the region needs to be calculated. Assuming that points are uniformly distributed among all regions, this would mean there are $\frac{n}{2^k}$ points in the region, and it takes $\Theta(d)$ to calculate the distance between two points. This has to be done for each hash-table, yielding a final complexity of $\Theta(Lkd + Ld\frac{n}{2^k})$.

It is important to realize that if the value of k is too small, then $\Theta(\frac{n}{2^k}) \approx \Theta(n)$, which is not optimal, since that *LSH* would have the same complexity as the brute-force method. For this reason, k is usually around $\log(n)$. Taking this into consideration and, with constant d and L, we have a pre-processing complexity of $\Theta(n \log n)$ and a querying complexity of $\Theta(\log n)$. This is substantially better than the brute force approach.

However, in the worst case scenario all the points are very close together, so even though there are a lot of regions, most points will still lie on the same region, meaning that the number of comparisons will still be in the order of $\Theta(n)$. However, as it is assumed that points follow a distribution, the probability of this happening is abysmally small. As this algorithm is probably fast and probably correct it lies under the *Atlantic City* family of probabilistic algorithms.

Memory wise, this algorithm occupies the memory needed to store L hash-tables with n entries each. It is known that each hash-table has a space complexity of $\Theta(n)$, therefore the overall space complexity of this solution is $\Theta(Ln)$. This is very expensive, as it is expected that n and L take large values. Some of the improved implementations of *LSH* manage to get more efficient memory usage.

B. Probability of Misses

As *LSH* is a probabilistic algorithm, there are some cases in which the result it yield is not totally correct. Because of the nature of the algorithm, the most common error is when the algorithm misses some points that should be identified as near neighbors.

1) False Negative Error: This occurs when the points a, b should collide but they don't. This can be interpreted in the following way.

$$
\begin{aligned}
&P(\text{false negative error}) \\
&= P(\text{none of the L hash-codes for a,b match}) \\
&= \prod_{i=1}^{L} P(i^{th}\text{hash-code for a is different from b}) \\
&= (P(hash - code(a) \neq hash - code(b))^L) \\
&= (P(\text{some of the K bits in the hash-code are different})^L) \\
&= (1 - P(\text{all K bits in the hash-code are the same})^L) \\
&= (1 - P(\text{a, b have the same bit})^K)^L
\end{aligned}
$$

2) False Positive Error: This occurs when the points a, b shouldn't collide but they do. This can be interpreted in the following way.

$$
\begin{aligned}
&P(\text{false positive error}) \\
&= P(\text{a,b match on at least one hash-code}) \\
&= 1 - (1 - P(\text{a, b have the same bit})^K)^L
\end{aligned}
$$

It's worth mentioning that different conditions are being assumed to calculate the probability that a, b have the same bit. In the false negative error, the fact that a, b are near duplicates is assumed. Whilst in the false positive error, the opposite is assumed.

The figure 3 gives an intuition of how to find the probability that a, b have the same bit.

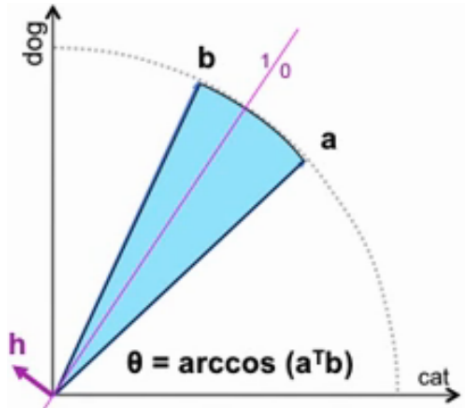

Fig. 2. A hyperplane h separating a,b

$$
\begin{aligned}
&P(\text{a, b have the same bit}) = \theta/(\pi/2) \\
&= 1 - \frac{2}{\pi} \arccos a^T b
\end{aligned}
$$

V. Methodology

An LSH based index in C++ was implemented in the context of solving the kNN problem and studied its performance for similarity search in a MNIST dataset. The algorithm was compared with a KD-tree indexing implementation. In this

section, the set-up of our methodology is described in terms of the datasets, main-memory implementation and parallelization technique.

A. Datasets

For our indexing tests, 70000 images were downloaded from from the MNIST dataset of handwritten digits. A sample of 60000 images were taken to initialize (train) LSH, while the remaining 10000 were used for querying (testing). Both LSH and KD-tree were trained and tested with the same datasets.

B. Implementation

First of all, a class *LSH* was created with the following attributes: *numOfPlanes*, *regions* and *hashes*. In *numOfPlanes* the number of hyperplanes to generate is defined, in *regions* the hash points are added to the hyperplane and in *hashes* we hash the points given. There also is a method *kNNValue* which finds the approximate k nearest points and verifies their values. The most frequent value among this points is returned as an estimate for the value of the query. To do this, we firstly hash the point to obtain a *hashedPoint* that is going to be used to find the it region in the hyperplane. Then the distance of the point to the other is computed and added to a bounded priority queue. Finally, this queue is used to find the most frequent value. Another method of the class *LSH* is *contains* it is used to find if the point is already part of the list of points.

C. Parallelization technique

Both the main-memory and disk-bask LSH indexes can be parallelized to work over a large number of servers. In our implementation, we decided to partition the data and calculate the kNN value of a range of points in different threads.

VI. Evaluation

In this section, results from our evaluation of LSH were presented. The focus of this study was on the accuracy of the query results obtained from the similarity search and the comparison of performance between the LSH and KD-tree indexes. The simulation developed show the following results:

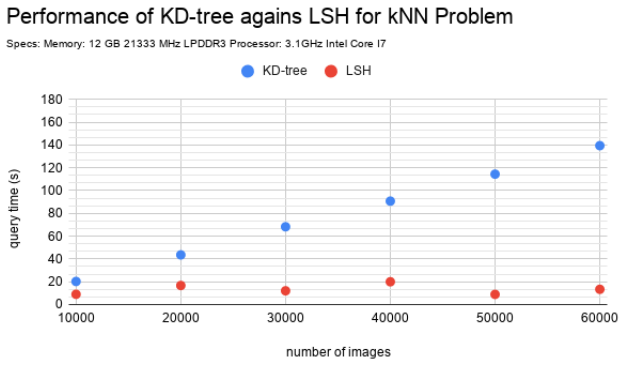

Fig. 3. Comparison between KD-tree and LSH query time for kNN Problem

A. Similarity search results

The accuracy of the results was measured using the test set labels that were included in the database we used. On average the KD-tree had an accuracy of 0.97, while the LSH algorithm had an accuracy of 0.95. In general, when considering the difference in running time, this decrease of accuracy is not very significant. It is important to highlight that the accuracy of the LSH can be improved, but it would imply an increase in running time, making its performance similar to the one of the KD-tree.

VII. Conclusions

In conclusion, we have presented an implementation of a basic LSH index and used it for a similarity search. In addition, we compared its performance with a KD-tree index. Our results demonstrated that the LSH index is faster than the KD-tree one. Both methods partition the space, either deterministically (KD-tree) or randomly (LSH) in order to allow for quick query time. Also, we tested its application for similarity search in a MNIST dataset, consisting of handwriting digit images.

Indexing is a process used in many areas of computation that allow for fast querying. The existence of a fast *dimensionless* algorithm, even if it is probabilistic, has very meaningfully effects on the performance of a lot of large data systems. Most of the experts agree that state-of-the-art improvements are every time getting less significant, probably implying that we are reaching a lower bound for complexity.

References

[1] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *2006 47th annual IEEE symposium on foundations of computer science (FOCS'06)*. IEEE, 2006, pp. 459–468.

[2] R. Motwani, A. Naor, and R. Panigrahi, "Lower bounds on locality sensitive hashing," in *Proceedings of the twenty-second annual symposium on Computational geometry*. ACM, 2006, pp. 154–157.

[3] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. of 25thIntl. Conf. on Very Large Data Bases(VLDB)*, 1999, pp. 518–529.

[4] R. Panigrahy, "Entropy based nearest neighbor search in high dimensions," in *Proc. of ACM-SIAMSymposium on Discrete Algorithms(SODA)*, 2006.

[5] Q. Lv, W. Josephson, Z. Wang, and M. Charikar, "Multi-probe lsh: Efficient indexing forhigh-dimensional similarity search," in *Proc. of the 33rd International Conference on Very Large Data Bases*, 2007.