LOCALITY-SENSITIVE HASHING FOR LARGE SCALE IMAGE RETRIEVAL

Adrián Sánchez Álvarez
asanca00@estudiantes.unileon.es

Enrique Alegre
enrique.alegre@unileon.es

Víctor González Castro
victor.gonzalez@emse.fr

Diego García Ordás
dgaro@unileon.es

Abstract

In this article we give an overview of the Locality-sensitive Hashing (LSH) used for large scale image retrieval and to solve the nearest neighbor problem. The goal is to experiment with different parameters configurations of LSH over a set of video fingerprints to see which is the best in terms of precision and indexing so that later, given a query object, return the objects (nearest neighbors) that are most similar to the query. The configuration parameters that offered best results were a bucket size of 24 with 15 tables and hash sizes of 15 bits, obtaining about 50% of correct matches with P@4 and P@6 retrieval methods.

Key Words: High Scale Image Retrieval, Indexing Algorithms, Locality-sensitive Hashing, LSH, Nearest Neighbor.

1 INTRODUCTION

Nowadays, the use of image processing is hugely increasing in many domains such as web clustering [11], computational biology [7] [6], computer vision [11], computational drug design [9], computational linguistics [14], etc. because computer vision is very helpful in many issues. One of the problems that can be solved by means of image processing, which is also an active research field, is image retrieval (i.e. finding a specific object in a large collection of images).

Image retrieval methods are present in many applications, such as finding DVD covers, book covers, buildings, faces, etc. Other examples are systems like Google Street View or Bing Street Side, which are available for users and make extensive use of large-scale image databases. The user can query the system using a captured image or textual description to find similar images in the databases.

Image retrieval is a challenging task because, most times, the databases the search algorithms are working with hundreds of millions or billions of images and, given a query image, the system must retrieve matching images in the database as quickly and accurately as possible. Thus, the key in this case is carrying out a good indexing process of the images in the database. In this article the indexing and retrieval processes are based on nearest neighbor problem that will be explained later, but a basic definition is: given a collection of n points, we build a data structure that, given a query point, it returns the data points that are closer to the query point.

There are many multidimensional data structures [15], but we distinguish two principal kinds of indexing methods: Bag of Words methods and Full Representation Methods. In Bag of Words methods, the features of the image are extracted, quantified and converted into histograms of visual words. Afterwards, these histograms are stored in a data structure, such as *Inverted File* or *Min-Hash* tables, to search the query image with the most similar histogram, i.e. using the nearest-neighbor method. The reader interested in further details is addressed to [3].

The Full Representation Methods include *Kd-Trees* and *Locality-Sensitive Hashing*. Kd-Trees were introduced in 1975 by Jon Bentley [5]. In this data structure, one (or more) randomized Kd-trees are built for the database features to allow for approximate nearest-neighbor matching in logarithmic time. There are two parameters that affect Kd-Trees recognition performance, storage, and run time: (i) The number of trees and (ii) the number of backtracking steps. Increasing the former increases the accuracy with no high impact on the speed, but increasing however, the required storage. According to the latter, it poses a tradeoff between accuracy and speed: More steps will yield more accuracy but also make the query slower.

On the other hand, the Locality Sensitive Hashing (LSH) algorithm depends on the existence of a number of hash functions extracted from the feature database [2]. The key idea is to hash the features to ensure that all features with the same hash value should go to the same bucket, so a bucket should have closest objects. This article is focused on LSH method because it is better than their counterparts in terms of search time. It is one of the most popular algorithms for performing approximate search in high dimensionality.

In this paper a study of the Locally Sensitive Hashing (LSH) algorithm applied to video descriptors is presented. In Section 2, we introduce the LSH method and the nearest neighbor. This section also contains an overview of the LSH method and its algorithm. In Section 3, we explain in detail the performed experiment. The Section 4 contains an explication of the

ISBN-13: 978-84-697-0589-6

experiment's results. Finally in section 5, we present the conclusions we have reached.

2 LOCALITY-SENSITIVE HASHING APPROACH

2.1 INTRODUCTION TO LSH

The LSH technique was introduced by Indyck and Motwani [12] with the purpose of creating algorithms to solve the ε-Nearest Neighbor Search. The LSH method and its variants have been applied to computational problems in some areas like web clustering or computer vision [11].

The principal goal of the algorithm [1] is to index a collection of n points using hash functions that ensure that the collision (i.e. similarity between objects) is higher to objects that are close between each other than for points that are far away. Then, the nearest neighbors can be determined hashing the query point and returning the elements of the bucket which would contain that point.

2.1.1 The Nearest Neighbor Problem

The nearest neighbor problem is an optimization problem: the goal is to find a point that minimizes a certain objective function [1]. There are several approaches to the problem: *nearest-neighbor*, *R-near neighbor* y *c-approximate r-near neighbor* with their *randomized* variations.

Let the nearest neighbor of a given point 'q' be the nearest point to 'q'. The R-near neighbor refers to the points which are within a circle of range *R* centered on *q*, i.e., a R-near neighbor returns points *p* such that | (| *p* - *q* |) | ≤ *R*. A c-approximate R-near neighbor returns a point p such that | (| *p* - *q* |) | ≤ *cR*, see Figure 1 to a better understanding. Randomized means that a point p is returned with a 90% probability of being the nearest.

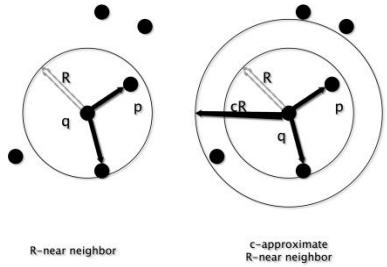


Figure 1: Example of the difference between R-near neighbor and its c-approximate variation

2.2 LSH

The LSH algorithm depends on the existence of a number of hash functions extracted from the feature database. Let *H* be a family of hash functions mapping $\mathbb{R}^{\boldsymbol{d}}$ for any universe *U*. Given two points, *p* and *q*, it is considered a process in which a function *h* is chosen from *H* uniformly and randomly, and the probability that *h* (*p*) = *h* (*q*) is analyzed. The family *H* is called locally sensitive if the following condition is satisfied:

***Definition (Locality-sensitive hashing)*:** A family *H* is called (R, cR, P_1, P_2)-sensitive if for any two points $q, p \in \mathbb{R}^{\boldsymbol{d}}$:

- If $||p - q|| \leq R$ then $Pr_H[h(q) = h(p)] \geq P_1$, (1)
- If $||p - q|| \geq cR$ then $Pr_H[h(q) = h(p)] \leq P_2$

Meaning; 1) c: c-approximate, 2) R: the range of search. 3) P_1, P_2, Pr_H are probabilities. The reader can see Figure 2 to see an illustration of the ranges cited in 1) and 2). In order to have a useful locality-sensitive hash family, it has to satisfy: $P_1 > P_2$ and $R > cR$.

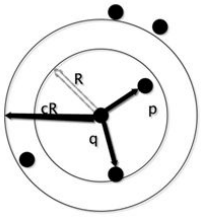


Figure 2: Locality-sensitive family

For a random hash function $h(\,)$ in $\mathbb{R}^{\boldsymbol{d}}$, for any points *p*, *q*: the probability P_1 is the probability that the distance between *p* and *q* is lower or equal than *R* (the points are near) and P_1 is high and the probability P_2 is the probability that the distance between *p* and *q* is higher or equal than R (the points are far) and P_2 is low.

2.3 THE ALGORITHM

A LSH hash functions family *H* can be used to elaborate an algorithm to solve the approximate search of the nearest-neighbor. The algorithm is formed by two stages: The pre-processing step and the query algorithm. This is detailed in Figure 3.

In the pre-processing step the data structure is built from the input dataset. *L* hash functions are extracted [2] $h_j, j = 1,2,3 \ldots L$ and a different hash table is created per function. The j^{th} table (with $j = 1,2,3 \ldots L$) contains the characteristics of the data set hashed using the function h_j to get the index of the bucket where the feature should go. Within each hash table *j*, all features with the same hash value should go to the same bucket.

ISBN-13: 978-84-697-0589-6

There are two parameters for any LSH [2] method: *L* and *T*. *L* is the total number of hash functions used, if there are many functions the runtime is increased but there are more probabilities of finding similar elements and, therefore, the bucket size is smaller. The parameter *T* is the number of hash tables used. A large number of tables increase the performance at the expense of increasing the runtime.

Using multiple hash tables allows us to improve the performance in terms of search, i.e., finding all possible nearest neighbors that would have not been found using a single table.

Table 1: Example of using multiple tables.

	Hash Tables' Buckets				
Hash table 1		P_1	P_2P_3	P_4	
Hash table 2	P_1		P_2P_3 P_4		
Hash table 3	P_1P_3		P_4		P_2

For Example, illustrated in Table 1, if only Hash table 1 was used, the nearest neighbor to P_3 would be only P_2, whereas if we use all the tables we find that the points P_1P_2 and $P_4$are also near P_3.

In the query step of the algorithm, the point q is searched in the database through the buckets $h_j(q), \dots, h_L(q)$ and the points stored in them are returned. For the sake of clarity: for hash table 1 (whose hash function is $h_1()$), it operates the hash function with the query point $h_1(q)$ to find out to which bucket of the hash table would go the query point. The stored items in that bucket, which would be the closest to q, are returned.

To determine the *R* nearest points or the nearest point we can measure the distance between the returned points and the query point q, by means of the Hamming distance as follows (see [10] to view in depth):

If the set of points returned is *P*, we assign to *C* the farthest coordinate to all points in *P*. We can introduce *P* in the Hamming cube with $d' = Cd$, transforming each point $p = (x_1 \dots x_d)$ in a binary vector

$$v(p) = Unary_C(x_1) \dots Unary_C(x_d) \qquad (2)$$

Where $Unary_C(x)$ denotes the unary representation of *x*, i.e., is a sequence of *x* ones followed by $C - x$ zeros. Now, for any two points *p* and q with coordinates in the set $\{1 \dots C\}$, the distance between them is given by the Hamming distance

$$d_1(p, q) = d_H(v(p), v(q)) \qquad (3)$$

With this, we can focus on solving the nearest neighbor problem in Hamming Space $H^{d'}$.

Preprocesing:

1. Choose *L* functions $g_j, j = 1, \dots L$ by setting $g_j = (h_{1,j}, h_{2,j}, \dots h_{k,j})$, whrere $h_{1,j}, \dots h_{k,j}$ are chosen at random from the LSH family *H*.
2. Construct L hash tables, where for each $j = 1, \dots L$ the j^{th} hash table contains the dataset points hashed using the function g_j.

Query algorithm for a query point q:

1. For each $j = 1, \dots L$
   - I. Retrieve the points from the bucket $g_j(q)$in the j^{th} hash table.
   - II. For each of the retrieved point, compute the distance from q to it, and report the point if it is a correct answer (near neighbor).
   - III. (optional) Stop as soon as the number of reported points is more than L'.

Figure 3: LSH algorithm

ISBN-13: 978-84-697-0589-6

3 EXPERIMENT

The goal of the experiment that we have carried out is to compare different configurations of LSH parameters to assess which is the best one. We used 4 different configurations of LSH: configuration 1, 2, 3 and 4 having buckets sizes 6, 12, 18 and 24 respectively. Then, each configuration is divided into six different combinations of numbers of tables and number of bits of the Hash Key. A configuration example is showed in Table 2, the number of tables and the numbers of bits of the hash key are the same for the others configurations, only changes the bucket size.

Table 2: Example of configuration.

Bucket Size	# Tables	# Bits of HashKey	Configuration
6	10	20	Configuration 1.1
6	15	20	Configuration 1.2
6	20	20	Configuration 1.3
6	15	10	Configuration 1.4
6	15	15	Configuration 1.5
6	15	25	Configuration 1.6

The LSH code used in the experiment was developed by Alex Andoni in 2004-2005 and can be found in [4]. This code is written in MATLAB and contains a simple implementation of Locality-Sensitive Hashing algorithm by Indyck. This code has two types of LSH: LSH and e2LSH.

The LSH type contains the older algorithm described in Gionis paper [10]. In this algorithm, for each hash function a single dimension of the data is chosen uniformly at random, and a single threshold value is drawn uniformly over the data range in that dimension.

The e2LSH is a more recent algorithm described in [8]. In this algorithm, for each hash function, a random line with independent, normally distributed coefficients is generated, and the data are projected to the line and shifted. Then, the range is divided into “cells” of a specific width; the hash value is determined by the cell into which a projected and shifted value falls.

We chose to work with the basic implementation of LSH: LSH type. The code needs to keep around the original data, in order to go back from indices to actual points because LSH will only index the data. There are three basic parameters to play with: the bucket’s size, the number of tables and the number of bits of the hash functions.

The final descriptors are a set of 1170 fingerprints
The extraction process of the fingerprints using SaKe has the following steps:

1. FPS Reduction: First of all, the size of the video (temporal, not spatial) is reduced by reducing the frames per second. That is because in a video, two consecutive frames are often very similar and provide redundant information.
2. Resizing and conversion to luminance: After reducing the fps in the previous step, the video is resized to a fixed size and converted to YCbCr color space. In this color space, we keep with the channel 'Y', which contains the luminance. This fixed size luminance video will be used in the next step.
3. Keyframe Detection: In this case, we define as keyframe a frame in which it takes place a significant change in luminance with respect to its contiguous frame. The keyframes calculated in this way are indicators of possible candidates to scene changes in a video.
4. HESRIs Generation: A HESRI is an image generated from the temporal and spatial information of a video block. Each keyframe obtained in the previous step determines the start of a new HESRI. Thus, a HERSI is a condensation of a video block in a picture.
5. Fingerprint Extraction: This step divides each video HESRI in p overlapped square pieces. On each piece we calculate a Discrete Cosine Transform (DCT) and we keep the first vertical and first horizontal coefficients of each DCT. This form we obtain $2p$ coefficients of each HESRI. The fingerprint is generated by concatenating in a vector all these coefficients along of all HESRIs of the Video.
6. Fingerprint Binarization: Finally, the vector of coefficients is transformed into a binary string (final fingerprint) by calculating the median of the vector and by setting to 1 the elements higher or equal to the median and the rest of elements to 0.

We have carried out two experiments to test the accuracy and the precision of the indexing method. The first one is a self-test code. In it, a number of descriptors and 2 neighbors to each one can be selected.

ISBN-13: 978-84-697-0589-6

The second test is a test of matching based on precision recall; see [13] for more information about these metric. Precision is the probability that a returned document (selected randomly) is relevant (% of elements selected that are correct). Recall is the probability that a relevant document (selected randomly) is returned in a search (% correct elements that are selected). We use Precision at n (P@n) to obtain the precision percentage. Precision at n is the precision evaluated given a list of ranges; the top-n documents are the first in the range. Therefore, precision at n is the proportion of the top-n documents that are relevant.

$$Precision\ at\ n = \frac{r}{n} \qquad (5)$$

r relevant documents have been returned of the range of n relevant documents. In our descriptors, there are six relevant documents, the descriptor and its five transformations.

In summary, the second test consists of extracting ten descriptors randomly and asking for six neighbors of every each. Then the test checks if the returned neighbors are similar or not and calculate the P@1, P@2, P@4 and P@6.

4 EXPERIMENTAL RESULTS

We have run the code with the different configuration parameters ten times and observed the average of these executions to avoid possible random effects on the results**,** so, the results we show here are an average of all the executions we did.

With respect to the runtime average, we have only checked the runtime for the P@n executions, and we always obtain a despicable runtime under 0,0186508s. So, we think the time here is not an important factor.

Then, regarding the results, on one hand the self-test code yields an idea of the occupancy of the buckets. Its results are presented in Figure 4. As it is shown, the buckets occupancy is related to the number of bits of the hash key. For a low number of bits, the buckets tend to be full, but as we will see in P@n results, a low number of bits give us full buckets but with a bad indexing. That is because the buckets are full of very different elements compared with each other. We note that the ideal case is that the LSH indexing method must store very similar elements in the same buckets. If we have a high number of bits, the elements in the buckets tend to be very similar at the expense of a very low bucket's occupancy (tend to be only one element per bucket).

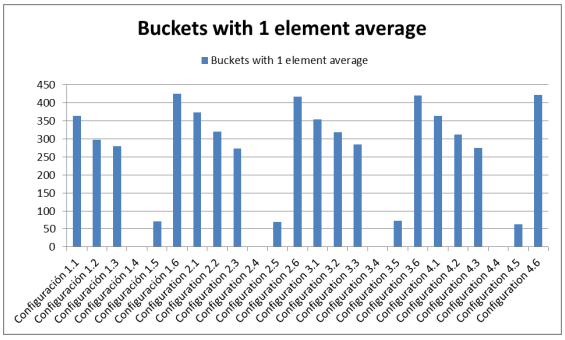


Figure 4: Buckets' occupancy

Finally, we talk about the matching test results based on P@n. The experiment contains four P@n (n=1, 2, 4 and 6). The relevant elements are the descriptor and its five transformations. The results are shown in Figure 5.

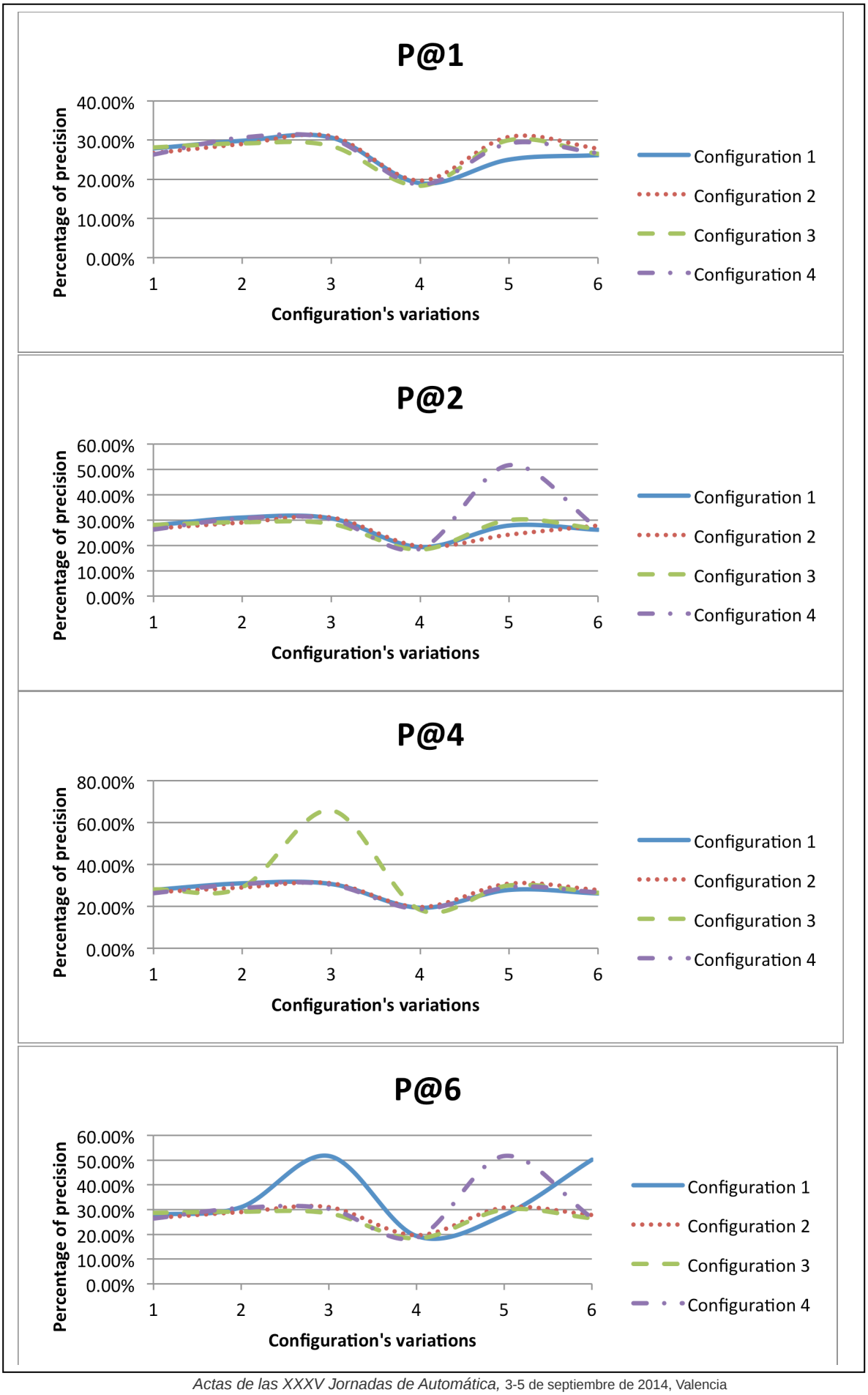
Using P@1 we can see that the configurations give us a percentage between 25% and 35% of correct matches (% successes), being better for the third sub configuration (20 tables and 20 bits) and worst for the fourth sub configuration (as we expect).

For P@2, an increase over 60% is produced by the Configuration 3.3 (Bucket size 18, #tables 20, #of hash key bits 20). This is the highest percentage in all the graphics.

If we continue increasing the parameter *n*, for P@4 we can see an increasing of precision for Configuration 4.5 (Bucket size 24, #tables 15, #of hash key bits 15), over 50%. The rest of configurations, as always, around the 30%.

Finally, for P@6 we can see an acceptable percentage over 50% for Configuration 1.3 (Bucket size 6, #tables 20, #of hash key bits 20), 1.6 (Bucket size 6, #tables 15, #of hash key bits 25) and for Configuration 4.5 (Bucket size 24, #tables 15, #of hash key bits 15). The rest give us a percentage around 30%. We need to find six relevant documents (remember, the descriptor and its five variants), so, this is a good measure for precision and recall.

ISBN-13: 978-84-697-0589-6



ISBN-13: 978-84-697-0589-6

5 CONCLUSIONS AND FUTURE DIRECTIONS

In this article we did an overview about the LSH method and the nearest neighbor problem. We also tested the LSH algorithm with many parameter configurations of the algorithm for indexing and retrieval video fingerprints to probe which is the best configuration.

As we see in experimental results section. The precision increases for a high number of bits of hash key. The configuration parameters are clear if we compare the data of Figure 4 and Figure 5. We try to keep the configuration with an acceptable percentage of precisions and bucket's occupation.

So, the final goal is to have the buckets with an acceptable number of elements (the ideal number is six, the descriptor and its variations), and to avoid buckets with just one element. If we compare the data of Figure 4 and Figure 5 we extract the following conclusions: for P@1 the best is the sub configuration 5 for all configurations but the precision percentage is lower (about 30%). For P@2 clearly, the configuration 3.3 with a percentage over 60% but have a high number of buckets with one element. For P@4 the configuration 4.5 is the best, have a precision percentage over 50% and a low number of buckets with one element. The same occurs for P@6.

In conclusion, we kept with P@6 because we have 6 relevant documents. The best configuration here is the configuration 4.5 (for P@4 and P@6) because give us a percentage over 50% and have the minimum number of buckets with one element, so this means that indexing was good and the buckets have an acceptable number of elements similar to each other. Then, it is better having a large bucket size (24 elements for this configuration) and a medium number of tables and hash key bits: 15 in both.

As future directions, we will test the configurations with SIFT descriptors in order to determine the best LSH configuration for those descriptors.

Acknowledgement
This work was supported by the DPI2012-36166 grant by the Spanish Government.

References

[1] A. Andoni and P. Indyck. (2008). Near-Optimal Hashing Algorithms for Apporximate Nearest Neighbor in High Dimensions. *Communications of the ACM, 51*(1), 117-122.

[2] Alaa El-Dien Mahmoud Hussein, M. A. (2011). *Searching Large-Scale Image Collections.* California Institute of Technology, Pasadena, California.

[3] Aly, M., Munich, M., & Perona, P. (s.f.). Bag Of Words For Large Scale Object Recognition: Properties and Benchmark. *Computational Vision Lab, Caltech, Pasadena, CA, USA; Evolution Robotics, Pasadena, CA, USA*.

[4] Andoni, A., & Indyck, P. (2006). *E2lsh: Exact Euclidean locality sensitive hashing*. Recuperado el 30 de 05 de 2014, de E2lsh: Exact Euclidean locality sensitive hashing: http://web.mit.edu/andoni/www/LSH/

[5] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Comm. ACM 18*, 509-517.

[6] Buhler, J. (2001). Efficient large-scale sequence comparison by localyty-sensitive hashing. *Bioinform*(17), 419-428.

[7] Buhler, J., & Tompa, M. (2001). Finding motifs using random projections. *Proceedings of the Annual Meeting of the Association Of Computational Lingüistics*.

[8] Datar, M., Immorlic, N., Indyk, P., & Mirrokni, V. (2006.). Locality Sensitive Hashing using stable distributions. En T. Darrell, P. Indyck, & G. Shakhnarovich, *Nearest Neighbor Methods in Learning and Vision: Theory and Practice.* MIT Press.

[9] Dutta, D., Guha, R., & Chen, T. (2006). Scalable Partitioning and exploration of chemical spaces using geometric hashing. *J.Chem. Inf. Model. 46*.

[10] Gionis, A., Indyck, P., & Motwani, R. (1999). Similarity Search inHigh Dimensions via Hashing. *Proceedings of the 25th VLDB*. Edinburgh, Scotland.

[11] Haveliwata, T., Gionis, A., & Indyck, P. (2000). Scalable Techniques for clustering the web. *WebDB Workshop*.

[12] Indyck, P., & Motwani, R. (1998). Approximate Nearest Neighbor - Towards Removing the Curse of Dimensionality. *In Proceedings of the 30th Symposium on Theory of Computing*, (págs. 604-613).

[13] M. W. Powers, D. (2007). Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. *Journal of Machine Learning Technologies*.

[14] Ravichandran, D., Pantel, P., & Hovy, E. (2005). Randomized algorithms and nlp: Using locality sensitive hash functions for high speed non clustering. *Proceedings of the Annual Meeting of the Association Of Computational Lingüistics*.

[15] Samet, H. (2006). Foundations of Multidimensional and Metric Data Structures. *Elsevier*.

ISBN-13: 978-84-697-0589-6