A Posteriori Multi-Probe Locality Sensitive Hashing

Alexis Joly
INRIA Rocquencourt
Le Chesnay, 78153, France
alexis.joly@inria.fr

Olivier Buisson
INA, France
Bry-sur-Marne, 94360
obuisson@ina.fr

ABSTRACT

Efficient high-dimensional similarity search structures are essential for building scalable content-based search systems on feature-rich multimedia data. In the last decade, Locality Sensitive Hashing (LSH) has been proposed as indexing technique for approximate similarity search. Among the most recent variations of LSH, multi-probe LSH techniques have been proved to overcome the overlinear space cost drawback of common LSH. Multi-probe LSH is built on the well-known LSH technique, but it intelligently probes multiple buckets that are likely to contain query results in a hash table. Our method is inspired by previous work on probabilistic similarity search structures and improves upon recent theoretical work on multi-probe and query adaptive LSH. Whereas these methods are based on likelihood criteria that a given bucket contains query results, we define a more reliable a posteriori model taking account some prior about the queries and the searched objects. This prior knowledge allows a better quality control of the search and a more accurate selection of the most probable buckets. We implemented a nearest neighbors search based on this paradigm and performed experiments on different real visual features datasets. We show that our a posteriori scheme outperforms other multi-probe LSH while offering a better quality control. Comparisons to the basic LSH technique show that our method allows consistent improvements both in space and time efficiency.

Categories and Subject Descriptors

H.3.3 [**Information Storgage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Performance, Theory

1. INTRODUCTION AND RELATED

Efficient high-dimensional similarity search structures are essential for building scalable content-based multimedia systems including multimedia search engines as well as browsing, summarization or content enrichment technologies. Indeed, multimedia contents are typically represented by high dimensional feature vectors that are frequently processed by algorithms involving nearest neighbors search, e.g. ranking, matching, quantizing, clustering or learning.

Early proposed tree-based indexing methods for Nearest Neighbors (NN) search such as R-tree [8], SR-tree [13], M-tree [5] or more recently cover-tree [2] return accurate results, but they are not time efficient for data with high (intrinsic) dimensionalities. It has been shown in [21] that when the dimensionality exceeds about 10, existing indexing data structures based on space partitioning are slower than the brute-force, linear-scan approach.

Approximate nearest-neighbor algorithms have been shown to be an interesting way of dramatically improving the search speed, and are often a necessity. The principle is to speed-up the search by returning only an approximation of the exact query results, according to an accuracy measure. Some of the first proposed approximate solutions were simply extensions of exact methods to the search of ϵ-NN [22, 4]; a ϵ-NN being an object whose distance to the query is lower than $(1+\epsilon)$ times the distance of the true k-th nearest neighbor. In [22] e.g., Zezula et al. deal with ϵ-NN in a M-tree. The performance gain is around 20 for a recall of 50% compared to exact results.

Clustering-based approximate methods have also been proposed to achieve substantial speed-ups over sequential scan [7, 15]. These algorithms partition the data into clusters and rank them at query time according to their similarity with the query vector. Cluster preprocessing is however very time consuming and prevents from practical operations such as insertions or deletions. In [9], Houle et al. developed a practical index called $SASH$ for approximate similarity queries in extremely high-dimensional data. $SASH$ is a multi-level structure of random samples connected to some of their neighbors. Queries are processed by first locating approximate neighbors within the sample, and then using the pre-established connections to discover neighbors within the remainder of the data set.

Overall, one of the most popular approximate nearest neighbor search algorithms used in multimedia applications is the Locality-Sensitive Hashing (LSH) [6]. The basic method uses a family of locality-sensitive hash functions composed of linear projections over randomly selected directions in the feature space. The principle is that nearby objects are hashed into the same hash bucket with a high probability, for at least one of the used hash function. LSH has been proved to achieve very good time efficiency for high dimensional features and has been successfully applied in several

MM'08, October 26–31, 2008, Vancouver, British Columbia, Canada.

multimedia applications including visual local features indexing [14], songs intersection [3] or 3D object indexing [18]. Time efficiency improvements of the basic LSH method have been proposed recently: In [1], Andoni and Indyk propose a near-optimal LSH that uses a Leech lattice for the geometric hashing instead of one-dimensional random projections. The idea is that lattices offer better quantization properties for the mean square error dissimilarity measure used in Euclidean spaces. In [10], Jegou et al. also use a lattice instead of random projections, but improves the search time efficiency by performing an on-line selection of the most appropriate hash functions from the whole pool of functions.
To achieve high search accuracy, LSH methods needs however to use a large number of hash tables and its main drawback is that it requires very large amount of available memory. To solve this problem, Multi-probe LSH methods have been proposed recently [17, 19]. Such methods are built on the well-known LSH technique, but instead probing only the bucket containing the query in each hash table, they probe multiple buckets that are likely to contain query results. The first multi-probe LSH strategy, denoted entropy-based LSH, was proposed by Panigrahy [19]. The principle was to sample multiple buckets by randomly generating "perturbated" objects near the query object, resulting in several query objects whose results are merged in the end. The intention of the method was clearly to trade time for space requirements. In [17], Lv et al. propose a more efficient Multi-probe LSH method that generates directly perturbated hash buckets instead of perturbated query objects, thanks to an efficient algorithm producing optimal probing sequences of hash buckets that are likely to contain similar objects to the query. This paper presents a new Multi-probe LSH method that generalizes and improves upon these previous techniques. Whereas they based on a simple likelihood criterion that a given bucket contains query results, we define a more reliable a posteriori probabilistic model taking account some prior about the queries and the searched objects. This prior knowledge allows a more accurate selection of the most probable buckets, improving time efficiency and offering a better quality control of the search.
Our new Multi-probe LSH method is somehow inspired by previous works of the authors on Probabilistic similarity search structures [20, 12]. Such methods can also be considered as hashing algorithms but contrary to LSH techniques they are based on a single multidimensional hash function induced by a space filling curve or an adaptive grid in the orginal feature space. The principle of the search is then to select the most probable hash buckets according to a probabilistic model of the searched objects, learned on query samples. Such techniques have been proved to achieve very good time efficiency for the search of distorted features in huge datasets and has been successfully applied in scalable content-based copy detection applications [20, 12]. However these techniques fail to index high dimensional features whose intrinsic dimensionality exceeds about 30 to 40 features, mainly because they use a single multidimensional grid hash table whereas LSH methods solve this problem by using several radomly selected hash functions.
The paper is organized as follows: Section 2 reminds general principles of Locality-Sensitive Hashing methods. Section 3 describes the proposed a posteriori Multi-probe Locality Sensitive Hashing method. Section 4 reports experimental results of the proposed method on real datasets.

2. LOCALITY SENSITIVE HASHING

In this section, we remind the general Euclidean Locality-Sensitive Hashing algorithm as described in [6], since we use the same indexing scheme. The basic idea of LSH is to use a set of hash functions that map similar objects into the same hash bucket with a probability higher than non-similar objects. At indexing time, all the feature vectors of the dataset are inserted in L hash tables corresponding to L randomly selected hash functions. At query time, the query vector is also mapped onto the L hash tables and the correponding L hash buckets are selected as candidates to contain objects similar to the query. A final step is then performed to filter the candidate objects by computing their distance to the query.
More formally, let $\mathcal{V}$ be a dataset of N d-dimensional feature vectors in $\mathbb{R}^d$ under the l_2 norm. For any point $\mathbf{v} \in \mathbb{R}^d$, the notation $\|\mathbf{v}\|_2$ represents the l_2 norm of the vector $\mathbf{v}$.

Now let $\mathcal{G} = \left\{\mathbf{g} : \mathbb{R}^d \to \mathbb{Z}^k\right\}$ be a family of hash functions such as:

$$\mathbf{g}(\mathbf{v}) = (h_1(\mathbf{v}), \ldots, h_k(\mathbf{v}))$$

where the functions h_i for $i \in [1, k]$ belongs to a locality sensitive hashing function family $\mathcal{H} = \left\{h : \mathbb{R}^d \to \mathbb{Z}\right\}$ [6]. We remind that a function family $\mathcal{H} = \left\{h : \mathbb{R}^d \to \mathbb{Z}\right\}$ is called (R, cR, p_1, p_2)-sensitive for l_2 if for any $\mathbf{q}, \mathbf{v} \in \mathbb{R}^d$:

$$Pr(h(\mathbf{q}) = h(\mathbf{v})) \geq p_1 \quad when \quad \|\mathbf{q} - \mathbf{v}\|_2 \leq R \tag{1}$$

$$Pr(h(\mathbf{q}) = h(\mathbf{v})) \leq p_2 \quad when \quad \|\mathbf{q} - \mathbf{v}\|_2 \geq cR \tag{2}$$

where $c > 1$ and $p_1 > p_2$. Intuitively, that means that nearby objects within distance R have a greater chance of being hashed to the same value than objects that are far away (distance greater than cR).
For the l_2 metric, the typically used LSH functions $h \in \mathcal{H}$ are defined as:

$$h(\mathbf{v}) = \left\lfloor \frac{\mathbf{a}.\mathbf{v} + b}{w} \right\rfloor \tag{3}$$

where $\mathbf{a} \in \mathbb{R}^d$ is a random vector with entries chosen independently from a Gaussian distribution and b a real number chosen uniformly from the range $[0, w]$.
Now, the LSH indexing method works as follows:

1. Choose L hash functions $\mathbf{g}_1, \ldots, \mathbf{g}_L$ from $\mathcal{G}$, independently and uniformly at random (each hash function $\mathbf{g}_j = (h_{j,1}(\mathbf{v}), \ldots, h_{j,k}(\mathbf{v}))$ is the concatenation of k LSH functions randomly generated from $\mathcal{H}$).

2. Use each of the L hash functions to construct one hash table (resulting in L hash tables).

3. Insert all points $\mathbf{v} \in \mathcal{V}$ in each of the L hash tables by computing the corresponding L hash values.

At query time, the L hash values of a given query vector $\mathbf{q}$ are computed in order to generate a set of L candidate hash buckets (one in each hash table). The candidate objects are then filtered by computing their distance to the query according to the query objective (typically the K-nearest neighbors or the neighbors in a given range).
In multi-probe LSH methods [17, 19], the main difference is that the set of candidate hash buckets is extended to more than one bucket in each hash table, by selecting neighboring

hash buckets to the query one. The idea is to increase the probability to find a relevant neighbor in a single hash table and consequently reduce the number L of required hash tables. The different approaches mainly differ in terms of how they select the multiple buckets per hash table and we will come again to this point in the next section.

3. A POSTERIORI MULTI-PROBE LOCALITY SENSITIVE HASHING

This section describes the proposed method. Sub-section 3.1 introduces our new success probability criterion to find neighbors in a given bucket. Sub-section 3.2 describes the derived Probabilistic Query-directed Probing Sequence algorithm. Sub-section 3.3 focuses on the implementation details for approximate nearest neighbor search. Finally, sub-section 3.4 analyses more precisely the advantages of our method compared to other Multi-probe LSH methods.

To make our argumentation easier, let us first reformulate the definition of a hash function $\mathbf{g} \in \mathcal{G}$ as:

$$\mathbf{g} = \lfloor \mathbf{g}^r \rfloor$$

with

$$\mathbf{g}^r(\mathbf{v}) = \mathbf{A}\ \mathbf{v} + \mathbf{B} \tag{4}$$

and where

$$\mathbf{A} = \left(\frac{{}^t\mathbf{a}_1}{w}, \ldots, \frac{{}^t\mathbf{a}_k}{w} \right)$$

and

$$\mathbf{B} = \left(\frac{b_1}{w}, \ldots, \frac{b_k}{w} \right)$$

Thus, a hash function $\mathbf{g}^r$ can be uniquely defined by a set of parameters $\theta = \{\mathbf{A}, \mathbf{B}\}$ and we denote $\mathbf{g}^r_\theta(\mathbf{v})$ a hash function parametrized by a fixed θ.
Secondly, let $n(\mathbf{q})$ be the set of relevant neighbors of a given query $\mathbf{q}$. This relevant set of neighbors depends on the targeted similar objects, e.g. the K-nearest neighbors of $\mathbf{q}$ or the results of a R-range query around $\mathbf{q}$ or any other relevant vectors.

3.1 Success Probability Estimation

Although all multi-probe LSH approaches visit multiple buckets for each hash table, they are very different in terms of how they probe multiple buckets. In this section, we will introduce the criterion used by our technique to estimate the success probability that a given hash bucket in a given hash table (among L) contains a relevant object $\mathbf{v} \in n(\mathbf{q})$. We first start by introducing the criterion used by others multi-probe [17, 19] and query-adaptive [10] LSH methods.

Locality sensitive hashing theory is based on the probability distribution of the hash values of two given points $\mathbf{q}$ and $\mathbf{v}$, over the random choices of the hash functions, e.g. over random choices of parameters set θ. In other words, θ is considered as a random variable whereas $\mathbf{v}$ and $\mathbf{q}$ are considered as constants. More formally, let $\mathbf{g}^r_v(\theta)$ denote the hash function $\mathbf{g}$ for a constant $\mathbf{v}$ and a variable θ. And let

$$\delta_{q,v}(\theta) = \mathbf{g}^r_v(\theta) - \mathbf{g}^r_q(\theta) \tag{5}$$

be the difference of the hash values of $\mathbf{v}$ and $\mathbf{q}$ as a function of θ. Due to the property of p-stable distributions [6], to which the Gaussian distribution used to generate the LSH functions belongs, it is possible to show that the probability distribution of $\delta_{q,v}(\theta)$ is also a normal distribution with independent Gaussian components and with variances σ_R proportional to $R = \|\mathbf{v} - \mathbf{q}\|_2$:

$$p_{\delta_{q,v}(\theta)|(q,v)} = \begin{pmatrix} \mathcal{N}(0, \sigma_R^2) \\ \vdots \\ \mathcal{N}(0, \sigma_R^2) \end{pmatrix} \tag{6}$$

From this statement, LSH theory derives the probability that two given points $\mathbf{q}$ and $\mathbf{v}$ collide in the same bucket over the randomly picked hash functions as done in [6]. In a more general case, it is also possible to derive the probability that $\mathbf{q}$ and $\mathbf{v}$ belongs to adjacent buckets over the randomly picked hash functions. Such probability can be used to estimate the overall search quality of a step-wise multi-probe LSH approach (as the one mentioned in [17]), which would consist in probing the same bucket neighborhood in all hash tables (e.g. all the neighboring hash buckets which differ in at most c coordinates from the hash bucket of the query).

Now, basic multi-probe LSH method [17] and some query-adaptive LSH approach [10] are based on a likelihood interpretation of the probability distribution $p_{\delta_{q,v}(\theta)|(q,v)}$. The density probability of $\delta_{q,v}$ over random values of θ can indeed also be interprated as the likelihood of $\delta_\theta = \mathbf{g}^r_\theta(\mathbf{v}) - \mathbf{g}^r_\theta(\mathbf{q})$ over random choices of $\mathbf{q}$ and $\mathbf{v}$, for fixed values of θ (i.e. for a fixed hash table). Formally, let

$$l_{\delta_\theta(\mathbf{q},\mathbf{v})|\theta} = p_{\delta_{q,v}(\theta)|(q,v)} \tag{7}$$

denote this likelihood. From Equation 6, we can derive:

$$l_{\delta_\theta(\mathbf{q},\mathbf{v})|(\theta)} = K\ \exp^{-\frac{\left\|\mathbf{g}^r_\theta(\mathbf{v}) - \mathbf{g}^r_\theta(\mathbf{q})\right\|_2}{\sigma_R^2}} \tag{8}$$

As this likelihood mostly depends on $\|\mathbf{g}^r_\theta(\mathbf{v}) - \mathbf{g}^r_\theta(\mathbf{q})\|_2$, the authors of [17] and [10] suggest to use as success criterion of a given hash bucket $\mathbf{u} = \mathbf{g}_\theta(\mathbf{q}) + \Delta,\ \Delta \in \mathbb{Z}^d$, the distance between the boundaries of this hash bucket and the real query hash $\mathbf{g}^r_\theta(\mathbf{q})$.

At this point, it is important to remind that the likelihood $l_{\delta_\theta(\mathbf{q},\mathbf{v})|(\theta)}$ does not model the real probability to find a neighbor $\mathbf{v}$ of $\mathbf{q}$ in the hash bucket $\mathbf{u} = \mathbf{g}_\theta(\mathbf{q}) + \Delta$. Considering this likelihood as a probability density would be in fact a case of prosecutor's fallacy since the real density depends on the prior distribution of $\mathbf{v} \in n(\mathbf{q})$.
Our method rather estimates the success probability of a given hash bucket in a given hash table a posteriori, i.e. for an observed hash function g^r_θ parametrized by a known θ. For a given query $\mathbf{q}$, in the absence of evidence, a point $v \in n(\mathbf{q})$ is indeed a random variable to which we associate a prior probability distribution $p_{v|q}(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^d$. For a given query $\mathbf{q}$ and a given hash function g_θ, our success criterion is then based on the distribution of $g^r_\theta(\mathbf{v})$ over random choices of $\mathbf{v} \in n(\mathbf{q})$, denoted as $p_{g^r_\theta(\mathbf{v})|(q,\theta)}$. We will see later, in section 3.3, how we can derive this posterior distribution from prior distributions $p_{v|q}$. For now, we suppose that this distribution is known.
After scalar quantization of the components of g^r_θ, the prob-

ability to find a relevant neighbor in a bucket characterized by its key $\mathbf{u} = (u_1, \ldots, u_k) \in \mathbb{Z}^k$ is:

$$\begin{aligned} P_{g_\theta|(q,\theta)}(\mathbf{u}) &= Pr_v\left(g_\theta\left(\mathbf{x}\right) = \mathbf{u} : \mathbf{x} \in n\left(\mathbf{q}\right)\right) \\ &= \int_{u_1}^{u_1+1} \cdots \int_{u_k}^{u_k+1} p_{g_\theta^r|(q,\theta)}(\mathbf{y})\mathbf{dy} \end{aligned}$$

Now, the principle of our probabilistic multi-probe LSH method is to visit the most probable hash buckets of a given hash function $\mathbf{g}_\theta$ according to their posterior probabilities $P_{g_\theta|(q,\theta)}(\mathbf{u})$. More precisely, our algorithm selects the minimal set of hash buckets, such that the global probability is higher than a quality control parameter α. Formally, let U be a set of hash keys $\mathbf{u} \in \mathbb{Z}^k$ and let

$$\mathcal{U}_\alpha = \left\{ U : \sum_{\mathbf{u} \in U} P_{g_\theta|(q,\theta)}(\mathbf{u}) \geq \alpha \right\}$$

then we wish to find the set of keys $U_{min}(\alpha)$ such as:

$$U_{min}(\alpha) = \underset{U \in \mathcal{U}_\alpha}{\operatorname{argmin}}(|U|) \tag{9}$$

A naive way to construct $U_{min}(\alpha)$ would be to compute the success probability of all possible keys and sort them, but it is of course practically impossible. A more efficient but approximate way would be to first use an s-step-wise probing algorithm that selects all the hash buckets which differ in at most s coordinates from the hash bucket of the query and then to sort them according to their posterior probability. This method has the advantage to be generic but is still not very efficient since the number of hash buckets probabilities to estimate remains $\sum_{n=1}^{s} 2^n \times \begin{pmatrix} k \\ n \end{pmatrix}$. If we tolerate an independence hypothesis of the components of $P_{g_\theta|(q,\theta)}(\mathbf{u})$, it is however possible to use a drastically more efficient algorithm, similar to the Query-Directed Probing Sequence algorithm defined in [17] and which is described in section 3.2.
At this point, we can remark that if we consider the following discrete distribution

$$P_{g_\theta|(q,\theta)}(\mathbf{u}) = \begin{cases} 1 \; if \; \mathbf{u} = g_\theta(q) \\ 0 \; if \; \mathbf{u} \neq g_\theta(q) \end{cases} \tag{10}$$

our method is equivalent to the basic LSH method. Also, if the prior $p_{v|q}$ is modeled by an isotropic normal distribution around $\mathbf{q}$, $p_{g_\theta^r|(q,\theta)}$ would be also an isotropic normal distribution and our method would be somehow equivalent to the common likelihood-based multi-probe LSH method.

3.2 Probabilistic Query-directed Probing Sequence algorithm

As mentioned in previous section, under the independence hypothesis of the components of g_θ^r, it is possible to define an efficient probabilistic probing sequence algorithm. Note that this is in the general not the case since $g_\theta^r(\mathbf{v})$ is a function of the random variable $\mathbf{v} \in n(q)$ which generally does not have independent components. However, we will see in the experiments that it seems to be an acceptable hypothesis.

From this assumption, follows:

$$p_{g_\theta^r|(q,\theta)}(\mathbf{y}) = \prod_{i=1}^{k} p_{h_i^r|(q,\theta_i)}(y_i) \;\; \mathbf{y} \in \mathbb{R}^k \; y_i \in \mathbb{R} \tag{11}$$

where $\theta_i = \{\mathbf{a}_i, b_i\}$ and after quantization

$$P_{g_\theta|(q,\theta)}(\mathbf{u}) = \prod_{i=1}^{k} P_{h_i|(q,\theta_i)}(u_i) \;\; \mathbf{u} \in \mathbb{Z}^k \; u_i \in \mathbb{Z} \tag{12}$$

Note that in practice, the domain of u_i is not infinite and u_i belongs to a a very short range of integer values bounded by:

$$u_i^{min} = \min_{\mathbf{x} \in \mathcal{V}} h_i(\mathbf{x})$$

$$u_i^{max} = \max_{\mathbf{x} \in \mathcal{V}} h_i(\mathbf{x})$$

where $\mathcal{V}$ is the dataset we wish to index.
Now, to achieve the objective of Equation 9, our Probabilistic Query-directed Probing algorithm works in a similar way than the one defined in [17], so we let the reader refer to it for full details and illustrations. The main differences of our algorithm are (1) that the relevance criterion of a hash bucket is not a sum of squared distances but a product of probabilities and (2) the ending condition is not the number of iterations (i.e. the number of probes) but the estimated success probability over all generated probes (3) the probing is not limited to the hash buckets adjacent to the query bucket $\mathbf{u}_q = g_\theta(\mathbf{q})$.
The principle of the algorithm is to generate a list of hash buckets in decreasing order of their probability $P_{g_\theta|(q,\theta)}(\mathbf{u})$ and to stop when the sum of their probabilities is larger than α.
Given the query object q and the hash functions h_i for $i = 1, \ldots, k$ corresponding to a single hash table, we first compute $P_{h_i|(q,\theta_i)}(u_i)$ for $i = 1, \ldots, k$ and $u_i = u_i^{min}, \ldots, u_i^{max}$. This generates an array of k lists of n_i probabilities ($n_i = u_i^{max} - u_i^{min} + 1$). Then, the n_i values of each of the k lists are sorted in decreasing order, and finally the k lists are also sorted in decreasing order of their first element. Let $p_j[z_j]$ denote the $(z_j - 1)$-th element in the j-th list of the sorted array. A hash bucket with key $\mathbf{u}$, can now be uniquely represented by a *sorted key* $\mathbf{z} = (z_1, \ldots, z_j, \ldots, z_k) \in \mathbb{Z}^k$. According to equation 12, the probability of a hash bucket characterized by its *sorted key* $\mathbf{z}$ remains the product of independent probabilities:

$$Pr(\mathbf{z}) = \prod_{j=1}^{k} p_j[z_j]$$

The problem of generating hash buckets in decreasing order of their probability now reduces to the problem of generating *sorted keys* $\mathbf{z}$ in decreasing order of their probability. To do that, three operations generating child *sorted keys* from a parent one are defined. The two first ones (*shift* and *expand*) are similar to the ones used in [17], except that our representation is not the same. The last one (*extend*) is added to explore hash buckets not adjacent to the query bucket:

- *shift*($\mathbf{z}$): This operation shifts to the right the last non zero component of $\mathbf{z}$ if it is equal to one and if it is not the last one (e.g. *shift*$((1,2,1,0,0))$=$(1,2,0,1,0)$). Otherwise, returns nothing (e.g. *shift*$((1,2,1,0,1))$=$\emptyset$, *shift*$((0,2,0,0,0))$=$\emptyset$).

- *expand*($\mathbf{z}$): This operation sets to one the component following the last non zero component of $\mathbf{z}$ if it is not the last one $\mathbf{z}$ (e.g. *expand*$((1,2,1,0,0))=(1,2,1,1,0)$). Otherwise, it returns nothing (e.g. *expand*$((1,2,1,0,1))=\emptyset$).

- *extend*($\mathbf{z}$): This operation adds one to the last non zero component. (e.g. *extend*$((1,2,1,0,0))=(1,2,2,0,0)$.

The important property of these three operations is that they generate child hash buckets with probabilities lower than the parent one ($Pr\left(shift\left(\mathbf{z}\right)\right) < Pr\left(\mathbf{z}\right)$, $Pr\left(expand\left(\mathbf{z}\right)\right) < Pr\left(\mathbf{z}\right)$ and $Pr\left(extend\left(\mathbf{z}\right)\right) < Pr\left(\mathbf{z}\right)$). The other important property is that, for any hash bucket characterized by its *sorted key* $\mathbf{z}$, there is a unique sequence of *shift*, *expand* and *extend* operations which will generate $\mathbf{z}$ from the starting bucket $z_0 = (0, \ldots, 0)$.
Now, the algorithm used to generate the hash buckets in decreasing order of their probability and achieve the objective of equation 9 is the following:

```
maxHeap=∅;
OutputKeyList=∅;
z_0 = (0, ..., 0);
maxHeap→Insert(z_0,Pr(z_0));
l = 1;
P_t = 0;
while P_t < α do
    z_l=maxHeap→ExtractMax();
    OutputKeyList→Add(z_l);
    P_t = P_t + Pr(z_l);
    z_shift=shift(z_l);
    maxHeap→Insert(z_shift,Pr(z_shift));
    z_exp=expand(z_l);
    maxHeap→Insert(z_expand,Pr(z_expand));
    z_ext=extend(z_l);
    maxHeap→Insert(z_extend,Pr(z_extend));
    l = l + 1;
end
return OutputKeyList;
```

A max-heap is used to maintain the collection of candidate hash buckets and output the top node at each iteration. The number of elements in the heap at any point of time is less than two times the number of iterations (i.e. the number of generated probes).

3.3 Approximate nearest neighbor search implementation

3.3.1 A posteriori probabilities estimation

We implemented a nearest neighbors search technique based on the proposed method. This section describes how we compute the a posteriori probabilities $P_{h_i|(q,\theta_i)}(u_i)$ required by the Probabilistic Query-directed Probing algorithm (cf. Equation 12).
Our estimation is based on a training set of N_s sampled query objects $\mathbf{q}_s$ and corresponding retrieved objects $\mathbf{v} \in n(\mathbf{q}_s)$, typically obtained by randomly picking N_s sample objects in the dataset and searching their exact nearest neighbors thanks to an exhaustive scan of the dataset.
Now, let us model the prior distribution $p_{v|q}$ by a multivariate Normal distribution with conditional mean $\mu(q)$ and conditional covariance matrix $\Sigma(q)$:

$$p_{v|q} = \mathcal{N}(\mu(\mathbf{q}), \Sigma(\mathbf{q}))$$

The hash function $g^r_\theta(\mathbf{v})$ being a linear application of $\mathbf{v}$ (cf. Equation 4), it also produces a Gaussian random variable with conditional covariance matrix $\mathbf{A}\Sigma(\mathbf{q})\mathbf{A}^T$ and conditional mean $g^r_\theta(\mu(\mathbf{q}))$. The distribution of the independent components h^r_i is then:

$$p_{h^r_i|(q,\theta_i)} = \mathcal{N}(h^r_i(\mu(\mathbf{q})), \mathbf{a}_i^T \Sigma(\mathbf{q}) \mathbf{a}_i) \tag{13}$$

At this point, the conditional functions $\mu(\mathbf{q})$ and $\Sigma(\mathbf{q})$ for any query $\mathbf{q}$ could be estimated by a Parzen window over the N_s samples of the training set. However, this process would be to expensive at query time. In our current implementation, we simplified the problem by considering a simpler model where the mean and variance of $p_{h^r_i|(q,\theta_i)}$ are conditioned only by $h^r_i(\mathbf{q})$ and not $\mathbf{q}$ itself. The main fundation of this assumption is to consider $h^r_i(\mathbf{v})$ independent from $h^r_{i'\neq i}(\mathbf{q})$ which is quite realistic due to the independence of the randomly selected projection vectors $\mathbf{a}_i$. In practice, using the datasets described in section 4, the normalized mutual information between pairs of such variables is very law, around 0.04 on average with a maximum at 0.15 for the most dependent pair of hash functions of the HSV dataset.
For each sample query $\mathbf{q}_s$, we estimate the sample mean μ_s and sample covariance matrix Σ_s over the retrieved neighbors $\mathbf{v} \in n(\mathbf{q}_s)$. For all hash functions h^r_i, we then compute the N_s hashing values $h^r_i(\mathbf{q}_s)$ and associate them a sample variance (cf. Equation 13):

$$\sigma^2_i(h^r_i(\mathbf{q}_s)) = \mathbf{a}_i^T \Sigma_s \mathbf{a}_i$$

and a sample mean:

$$\mu_i(h^r_i(\mathbf{q}_s)) = h^r_i(\mu_s)$$

The conditional mean $\mu_i(h^r_i(\mathbf{q}))$ and conditional variance $\sigma^2_i(h^r_i(\mathbf{q}))$ for any $\mathbf{q}$ are then interpolated by a Gaussian kernel over the N_s sample:

$$\mu_i(h^r_i(\mathbf{q})) = \frac{\sum_{s=1}^{n_s} \mathcal{K}(h^r_i(\mathbf{q}), h^r_i(\mathbf{q}_s))\ h^r_i(\mu_s)}{\sum_{s=1}^{n_s} \mathcal{K}(h^r_i(\mathbf{q}), h^r_i(\mathbf{q}_s))} \tag{14}$$

$$\sigma^2_i(h^r_i(\mathbf{q})) = \frac{\sum_{s=1}^{n_s} \mathcal{K}(h^r_i(\mathbf{q}), h^r_i(\mathbf{q}_s))\ \sigma^2_i(h^r_i(\mathbf{q}_s))}{\sum_{s=1}^{n_s} \mathcal{K}(h^r_i(\mathbf{q}), h^r_i(\mathbf{q}_s))} \tag{15}$$

where $\mathcal{K}(x, y)$ is the Gaussian kernel function. At this point, the discrete distribution of the hash values after quantization can be computed as:

$$P_{h_i|(q,\theta_i)}(h^r_i(\mathbf{q}), u_i) = \int_{y=u_i}^{u_i+1} \mathcal{N}(\mu_i(h^r_i(\mathbf{q})), \sigma^2_i(h^r_i(\mathbf{q})))dy \tag{16}$$

In our experiments, we typically use $N_s = 1000$ query samples and $\sigma_K = \frac{w}{5}$ for the Gaussian kernel parameter.

3.3.2 Pre-computation of probabilities in Look-up Tables

In practice, to speed up the Probabilistic Query-directed Probing algorithm at query time, we pre-compute, at indexing time, the discrete distributions $P_{h_i|(q,\theta_i)}(h^r_i(\mathbf{q}), u_i)$ for quantized values of $h^r_i(\mathbf{q})$. Let $h^z_i(\mathbf{q}) \in [0, N_z]$ be the quantized value of $h^r_i(\mathbf{q})$:

$$h^z_i(\mathbf{q}) = \left\lfloor \left(h^r_i(\mathbf{q}) - u^{min}_i\right) \ \frac{N_z}{u^{max}_i + 1 - u^{min}_i} \right\rfloor \tag{17}$$

At indexing time, we pre-compute the discrete distributions $P_{h_i|(q,\theta_i)}(u_i)$ for all possible quantized values $h_i^z(\mathbf{q}) \in 0,\ldots,N_z$ according to Equations 14, 15 and 16. Over all hash functions h_i^j, this process generates a set of $L \times k \times N_z$ Look-up tables of size $u_i^{max} + 1 - u_i^{min}$. In the experiments, we used $N_z = 2500$ and the resulting space requirement for the Look-up tables did not exceed 5 Mb.
At query time, the probabilities required by the Probabilistic Query-directed Probing algorithm (cf. Equation 12) are computed only by quantizing $h_i^r(\mathbf{q})$ according to Equation 17 and reading the corresponding values in the Look-up tables.

3.3.3 *Refinement step*

At query time, once the most probable hash buckets are selected, the distance to the query is computed for all the objects they contain and only the objects satisfying the query objective are output ($\mathbf{v} \in n(\mathbf{q})$).

3.3.4 *Parameters settings*

The main parameters of our technique are the common LSH parameters L, k and w (cf. section 2) and the single hash table quality control parameter α (cf. section 3.1).
A suggested in [6] and [1], we choose common settings for the LSH techniques: $k = \ln(N)$ and $w = 4R$ where R is the average distance of the searched objects $\mathbf{v} \in n(\mathbf{q})$ to the query $\mathbf{q}$. The total quality of the search α_T is set by the user. In the basic LSH technique, it is equal to

$$\alpha_T = 1 - (1 - p_0^k)^L \tag{18}$$

where p_0 is the probability that a query q and a neighbor v collide in the same bucket for a single hash function. In our technique, the probability to retrieve a neighbor v in one of the L multidimensional indexes is estimated by α and thus we get:

$$\alpha_T = 1 - (1 - \alpha)^L \tag{19}$$

To guaranty that the global probability is higher than α_T we thus choose L as:

$$L = \left\lfloor \frac{\ln(1-\alpha_T)}{\ln(1-\alpha)} + 1 \right\rfloor \tag{20}$$

The estimation of the total search time T_t can be expressed as a function of α:

$$T_t(\alpha) = L \times T(\alpha) = \left(\frac{\ln(1-\alpha_T)}{\ln(1-\alpha)} + 1 \right) \times T(\alpha) \tag{21}$$

where $T(\alpha)$ is the search time in one of the L indexes. If it exists, this function has a minimum when its derivative is null, leading to the following equality:

$$\frac{T'(\alpha)}{T(\alpha)} = \frac{\ln(1-\alpha_T)}{\ln(1-\alpha)(1-\alpha)\left(\ln(1-\alpha) + \ln(1-\alpha_T)\right)} \tag{22}$$

The second term is a strictly decreasing function that tends to infinity when α tends to zero and to zero when α tends to one. The first term is the logarithmic derivative of $T(\alpha)$. It is always higher than zero due to the growth of $T(\alpha)$ with α and it is usually increasing with α due to the exponential growth of $T(\alpha)$. Thus, $T_t(\alpha)$ has usually a unique minimum at $\alpha = \alpha_{min}$ that can be determined experimentally by searching a single index with varying values of α, measuring $T(\alpha)$ and minimizing $T_t(\alpha)$ according to equation 21. Such estimation is illustrated on Figure 1 for the three datasets described in the experimental section 4 and α_T=0.95. The minimum of $T_t(\alpha)$ is achieved respectively at α_{min}=0.44 for HSV dataset, at α_{min}=0.57 for SIFT dataset and at α_{min}=0.78 for Dipole dataset. From these values, we can derive the optimal value of L thanks to equation 20, which gives respectively L=5, L=4 and L=2.

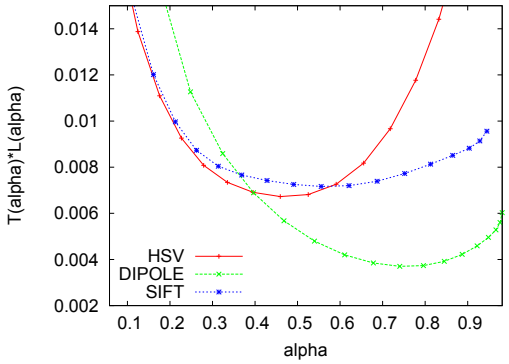

Figure 1: Theoretical global search time vs success probability control parameter *alpha* (global control quality parameter set to $\alpha_T = 0.95$): Mininima are achieved respectively at α_{min}=0.44 for HSV dataset, at α_{min}=0.57 for SIFT dataset and at α_{min}=0.78 for Dipole dataset

3.4 Advantages over other multi-probe LSH schemes

We summarize here the advantages of our probabilistic multi-probe LSH technique compared to other multi-probe LSH techniques:

1. More efficient filtering: taking account the prior distribution of the searched objects allows to reduce significantly the required number of probes to achieve similar recall. High recall can even be obtained efficiently with a single hash table.

2. Search quality control and parameters estimation: The relevance criterion of a hash bucket being a probability and not a likelihood score, it allows to have a coarse estimation of the probability to find relevant objects without tuning. Having an estimation of the probability also allows to estimate automatically the required number of hash tables L without tuning.

3. Genericity and Query adaptivity: Our probabilistic filtering algorithm is fully independent of the query type. It just requires query samples and corresponding relevant objects sets, not necessarily nearest neighbors. Examples of other relevant objects are distorted features obtained after transformation of a multimedia content or nearest neighbors of the query in another dataset (e.g. a category specific dataset or a training dataset). The search can also be easily adapted to different objectives by pre-computing different prior models and corresponding probabilities Look-up tables for the same index structure. A typical application is to achieve class dependent queries.

4. EXPERIMENTS

4.1 Experimental setup

This section describes the configurations of our experiments, including the evaluation datasets, benchmarks, metrics, and some implementation details.

4.1.1 Evaluation datasets

All the experiments are based on the three following visual features datasets. The dataset sizes are chosen such that the index data structure of the basic LSH method can fit in main memory.

- **HSV** dataset: A set of common 120-dimensional hsv histograms extracted from a collection of 512,927 images collected from the web in a design-oriented perspective (European project TRENDS).
- **SIFT** dataset: A set of common SIFT local features [16] extracted in a collection of 350 images randomly built from the **ImagEval**[1] benchmark corpus. The particularity of such features is that the generated vectors are very sparse with a large amount of null components.
- **DIPOLE** dataset: A set of 5,405,324 20-dimensional local features based on oriented dissociated dipoles extracted around multi-resolution Harris interest points [11]. Contrary, to the two previous datasets, such features are not histograms but differential operators with dense distributions. The source image collection includes 5,474 images built from the **ImagEval** benchmark corpus.

Table 1 summarizes the dimension and size of the three datasets.

dataset	size	dimension
HSV	512,927	120
SIFT	523,338	128
DIPOLE	5,405,324	20

Table 1: Experimental datasets summary

4.1.2 Evaluation benchmark

For each dataset, we randomly picked a set Q of 1000 objects as query objects. Depending on the experiment, the ideal answer of each query (ground truth) is defined either by the K nearest neighbors or by all the objects in a given range R (not including the query itself). We used the Euclidean distance for all datasets. Unless otherwise specified, we use a K-nearest neighbor search with $K = 100$. The performances are evaluated in two main aspects: search quality and speed. Search quality is measured by *recall*:

$$recall = \frac{|I(Q) \cap A(Q)|}{|I(Q)|} \tag{23}$$

where $I(Q)$ is the ideal set of answers over all queries and $A(Q)$ the set of actual answers over all queries. Note that we do not need to consider precision here, since all candidate objects found in checked hash buckets are filtered at query time according to the query parameters (Top K candidates or all candidates whose distance is below R for range queries).
Search speed is measured by averaging the query time over the 1000 queries.

[1]http://www.imageval.org/

4.1.3 Hardware

The evaluation is done on a PC with one 64-bit 2GHz CPU, 1024Kb L2 cache size and 6GB RAM.

4.2 Experimental results

4.2.1 Success probability criterion comparison

Although all multi-probe LSH approaches visit multiple buckets for each hash table, they are very different in terms of how they probe multiple buckets. In [17], Lv et al. already showed that their query directed probing algorithm based on a likelihood relevance criterion did require substantially fewer number of probes than the entropy-based method of [19] or than a simple step-wise probing. We thus only compare their likelihood relevance criterion to our probabilistic filtering algorithm within our own implementation of the technique.
To compare the two approaches in detail, we measure the number of visited hash buckets for varying recall values. As the likelihood-based method does not enable any control of the search quality, we vary directly the number of visited buckets (since it is the main parameter of this technique) and measure the resulting recall. For our method, we vary the value of the search quality parameter α and measure the resulting average number of visited buckets and the resulting recall.
We first did this experiment for a single hash function ($L = 1$). The results (number of visited buckets and recall) are then averaged over 10 randomly picked hash functions g_j. Figure 2 plots the results obtained for the HSV dataset. It shows that our method requires substantially fewer number of probes to achieve the same recall. The ratio is increasing with the recall value and is about 5 for a typical recall equal to 0.44 (related to the optimal theoretical value α_{min}=0.44, cf. section 3.3.4). Similar curves are obtained for the two other datasets: For SIFT dataset, the gain is about 6 for a recall equal to 0.57 (related to α_{min}=0.57); For DIPOLE dataset, the ratio is about 18 for a recall equal to 0.78 (related to α_{min}=0.78).

In a second step, we did the same experiment using multiple hashing functions with the values of L derived from the optimization procedure described in section 3.3.4. Table 2 summarizes the results obtained using $\alpha_T = 0.95$ for our technique and at similar recall for the likelihood-based method. It shows that our a posteriori success probability criterion requires substantially fewer number of probes to achieve similar recall. The reduction ratio is equal to 6.17 for HSV dataset, 2.38 for SIFT dataset and 8.9 for DIPOLE dataset.

To illustrate why our a posteriori method allows a more accurate selection of the probes, we did compute some statistics on the key values of the nearest neighbours of 60,000 sample queries. Figure 3 plots the experimental distribution

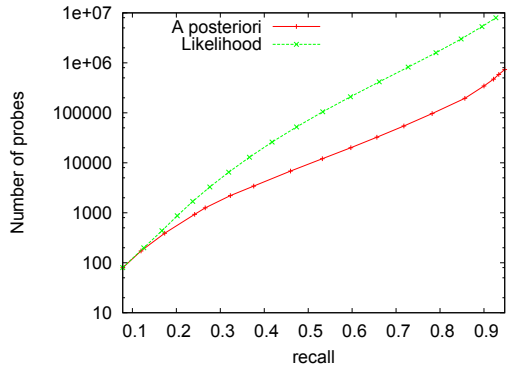

Figure 2: Number of probes required by likelihood multi-probe LSH and a posteriori multi-probe LSH to achieve certain search quality (HSV dataset, L=1)

dataset	method	L	recall	nb of probes
HSV	a posteriori	5	0.94	31,813
	likelihood	5	0.92	196,500
SIFT	a posteriori	4	0.92	2,689
	likelihood	4	0.92	6,400
DIPOLE	a posteriori	2	0.96	5,752
	likelihood	2	0.95	51,200

Table 2: Search performance comparison of a posteriori probabilistic probing vs. likelihood probing

of key differences between a query and its nearest neighbours for two different hash functions, on the SIFT dataset. It first shows that the neighbours of a query are not systematically contained in hash buckets adjacent to the query bucket, justifying the extension of our query-directed probing algorithm to non-adjacent buckets. It also illustrates the variability of the neighbours distribution between different hash functions.

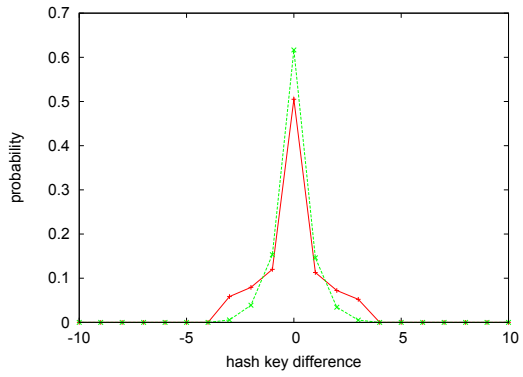

Figure 3: Hash key difference distribution (between query hash key and neighbours hash keys) for two hash functions on the SIFT dataset

4.2.2 Search quality control

To evaluate the search quality control of our method, we vary the control quality parameter α_T for the three datasets and measure the resulting recall. The number of hash tables L is set to the default settings ($L = 5$ for HSV, $L = 4$ for SIFT and $L = 2$ for DIPOLE). The parameter α of the Probabilistic Query-directed probing algorithm in each hash table is deduced from α_T through:

$$\alpha = 1 - (1 - \alpha_T)^{\frac{1}{L}} \quad (24)$$

The results are summarized in Table 3. They show that despite the independence hypothesis of the a posteriori probability, the quality control is fairly good and might be accurate enough for most applications. Note that the other multi-probe techniques do not allow such control at all and that the quality control of the basic LSH technique gives similar errors while using range queries.

	dataset	α_T			
		0.30	0.50	0.70	0.80
Recall	HSV	0.4634	0.6108	0.7684	0.8320
	SIFT	0.3581	0.4953	0.6426	0.7493
	DIPOLE	0.5233	0.6914	0.8115	0.8932

α_T					
0.85	0.90	0.95	0.97	0.99	0.999
0.8736	0.9098	0.9392	0.9512	0.9679	0.9880
0.8023	0.8554	0.9226	0.9494	0.9775	0.9965
0.9016	0.9403	0.9568	0.9694	0.9864	0.9947

Table 3: Experimental recall of 100-NN search for different values of the search quality control parameter α_T

4.2.3 Comparison to LSH

We compared our method to the Euclidean LSH method described in [6]. The code source of this method is kindly provided by the authors in the E^2LSH package [2]. Since this method is dedicated to range queries, we used this kind of queries. The radius R of the query for each dataset is set to the average distance of the exact 100-nearest neighbors. It was estimated on a set of 1000 queries sampled from the datasets using an exhaustive scan.
Note that the provided LSH code includes a script that computes automatically the main parameters of LSH in the first stage of data structure construction, for a given dataset, a given set of queries, a given range R and a given quality control parameter α_T. The parameters are chosen so that to optimize the estimated query time. Since the E^2LSH method requires a large amount of memory, the optimal parameters for large datasets might require an amount of memory which is greater than the available physical memory. Therefore, when choosing the optimal parameters, E^2LSH takes into consideration the upper bound on memory it can use.
Results obtained at constant recall on the three datasets are given in Table 4. They show that our method allows to drastically reduce the space requirement (L is 18 to 63 times smaller) while reducing significantly the search time. The time efficiency of LSH on the DIPOLE dataset is very bad due to its larger size and memory limitation. The links between space requirement and time efficiency are discussed further through two more detailed experiments.

[2]http://www.mit.edu/ andoni/LSH/

dataset	method	L	recall	query time (s)
HSV	exh scan		1.00	0.1590
	LSH	253	0.98	0.0041
	a posteriori LSH	5	0.98	0.0020
SIFT	exh scan		1.00	0.2000
	LSH	253	0.98	0.0046
	a posteriori LSH	4	0.98	0.0026
DIPOLE	exh scan		1.00	0.4350
	LSH	36	0.95	0.0512
	a posteriori LSH	2	0.95	0.0016

Table 4: Search performance comparison between our a posteriori LSH method, basic LSH and Exhaustive scan

Space requirement vs Time efficiency Since the main objective of multi-probe LSH methods is to drastically reduce the large space requirements of LSH, it is interesting to compare the time efficiency of both techniques according to the space requirements. To do that, we artificially vary the amount of available memory passed to the LSH optimization script and recompute the LSH parameters and structures for each upper bound on memory. We then re-apply our benchmarking procedure on each derived structure. For our technique, we only consider the single result obtained with default parameters.
Space requirement of LSH is measured by the ratio between the index size and the data size. Comparative time efficiency is measured by the ratio between LSH query time and the query time of our method. Figure 4 plots this time ratio according to the space requirement of LSH, for the HSV dataset. Note that the space ratio of our technique for this dataset is equal to 0.125, which means that the index is almost 10 times smaller than the data itself. The results show that our method is always faster than LSH since the Time ratio is always larger than 1. For a reasonable space requirement of 1 (index size equal to data size), our method is about 15 times faster than LSH. Since the curve is converging for large space ratio, we can also estimate that our method is about 2 times faster for unlimited memory space.

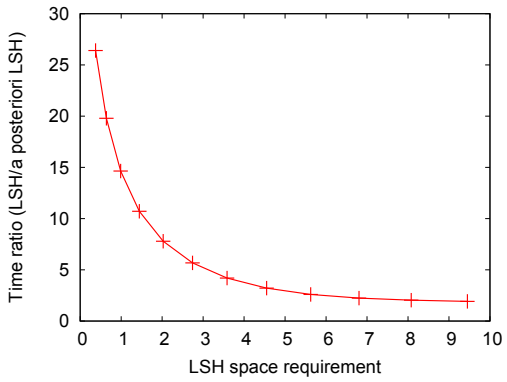

Figure 4: Search time ratio (LSH / our method) according to LSH space requirement (normalized by dataset space)

Influence of dataset size
To evaluate the influence of dataset size, we vary the size of the DIPOLE dataset which is the larger one. Each sub-dataset is built by randomly picking objects in the full dataset and is then indexed by . The quality control parameter of both techniques was set to $\alpha_T = 0.95$. The script optimizing LSH parameters was applied to each sub-dataset. The parameters of our method for each dataset were computed according to the procedure described in section 3.3.4.
We also applied an exhaustive scan on each sub-dataset to have a baseline linear reference.
The results are plotted on Figure 5 in normal and logarithmic coordinates. It shows that at constant available memory, the increase of LSH search speed over dataset size is supra-linear whereas the one of our method is sub-linear. Note that the number of hash tables used by LSH (optimized by LSH script according to available memory) decreases from $L = 378$ to $L = 36$ when the dataset sizeincreasess from $N = 100,000$ objects to $N = 5,405,324$ objects. This is due to the fact that the ideal dimension k of the hash functions is increasing with the size of the dataset and since L increases with k, an upper limit on memory imposes an upper limit on k and L.

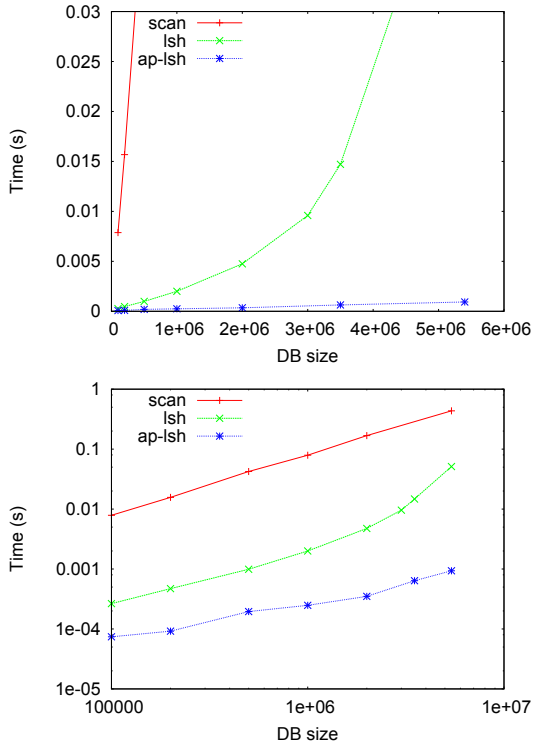

Figure 5: Search time efficiency comparison when varying the size of the dataset - the bottom graph represents the same curves in logarithmic coordinates

5. CONCLUSION AND FUTURE WORKS

In this paper, we presented a new similarity search technique that can be used to build efficient content-based search

systems on feature-rich multimedia data. The technique is inspired by previous theoretical works on multi-probe Locality Sensitive Hashing and improve them by taking account a prior knowledge about the searched objects through an efficient probabilistic query directed probing algorithm. This technique allows a better quality control of the search and a more accurate selection of the most probable buckets. We did show in the experiments that the number of required probes can be reduced significantly compared to commonly used likelihood based success criterion. Comparisons to the basic LSH technique show that our method allows consistent improvements both in space and time efficiency. Furtermore, we think that this technique has a high potential regarding new multimedia systems involving context-aware or personalized retrieval mechanisms. The prior knowledge used by our filtering algorithm can indeed be easily adapted to different contextual or personalized knowledge. The retrieval will be therefore automatically focused on the context-aware or personalized targeted objects while making the search more efficient. Future works could address two improvements of the proposed technique. The first one is to model more reliable prior knowledge, e.g. by deriving more reliable conditional prior distribution $p_{v|q}$ in the original feature space. The second one is to use the prior knowledge not only for the similarity search but also for the index construction. The hash functions of common LSH techniques are indeed randomly selected independently of the dataset and the targeted objects. Consistent improvements could be achieved by generating the hash functions according to data dependent distributions.

6. ACKNOWLEDGMENTS

This work was funded by the European Comission within VITALAS project, http://vitalas.ercim.org/.

7. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of ACM*, 51(1), 2008.

[2] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proc. of conf. on Machine learning*, pages 97–104, New York, NY, USA, 2006.

[3] M. Casey and M. Slaney. Song intersection by approximate nearest neighbour search. In *In Proc. Int. Symp. on Music Information Retrieval*, pages 2161–2168, 2006.

[4] P. Ciaccia and M. Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proc. of Int. Conf. on Data Engineering*, pages 244–255, 2000.

[5] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 426–435, 1997.

[6] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. of Symposium on Computational geometry*, pages 253–262, 2004.

[7] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. Abbadi. Approximate nearest neighbor searching in multimedia databases. In *ICDE*, pages 503–511, 2001.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD Conf. of Management of Data*, pages 47–57, 1984.

[9] M. E. Houle and J. Sakuma. Fast approximate similarity search in extremely high-dimensional data sets. In *Proc. of the Int. Conf. on Data Engineering*, pages 619–630, 2005.

[10] H. Jegou, L. Amsaleg, C. Schmid, and P. Gros. Query-adaptative locality sensitive hashing. In *International Conference on Acoustics, Speech, and Signal Processing*. IEEE, 2008. to appear.

[11] A. Joly. New local descriptors based on dissociated dipoles. In *CIVR '07: Proceedings of the 6th ACM international conference on Image and video retrieval*, pages 573–580, 2007.

[12] A. Joly, O. Buisson, and C. Frélicot. Content-based copy retrieval using distortion-based probabilistic similarity search. *IEEE Trans. on Multimedia*, 9(2):293–306, 2007.

[13] N. Katayama and S. Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 369–380, 1997.

[14] Y. Ke, R. Sukthankar, and L. Huston. Efficient near-duplicate detection and sub-image retrieval. In *Proc. of ACM Int. Conf. on Multimedia*, 2004.

[15] C. Li, E. Chang, M. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *IEEE Trans. on Knowledge and Data Engineering*, 14(4):792–808, 2002.

[16] D. G. Lowe. Object recognition from local scale-invariant features. In *Proc. of Int. Conf. on Computer Vision*, pages 1150–1157, 1999.

[17] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proc. of Conf. on Very Large Data Bases*, pages 253–262, 2007.

[18] M.-B. Matei, S. M.-Y. Shan, M.-H. S. Sawhney, S. M.-Y. Tan, M.-R. Kumar, M.-D. Huber, and M.-M. Hebert. Rapid object indexing using locality sensitive hashing and joint 3d-signature space estimation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(7):1111–1126, 2006.

[19] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *Proc. of annual ACM-SIAM symposium on Discrete algorithm*, pages 1186–1195, 2006.

[20] S. Poullot, O. Buisson, and M. Crucianu. Z-grid-based probabilistic retrieval for scaling up content-based copy detection. In *CIVR '07: Proceedings of the 6th ACM international conference on Image and video retrieval*, pages 348–355, 2007.

[21] R. Weber, H. J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 194–205, 1998.

[22] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate similarity retrieval with m-trees. *Very Large Data Bases Journal*, 7(4):275–293, 1998.