On Locality Sensitive Hashing in Metric Spaces

Eric Sadit Tellez
sadit@lsc.fie.umich.mx
Universidad Michoacana, México

Edgar Chavez
elchavez@umich.mx
Universidad Michoacana / CICESE, México

ABSTRACT

Modeling proximity search problems as a metric space provides a general framework usable in many areas, like pattern recognition, web search, clustering, data mining, knowledge management, textual and multimedia information retrieval, to name a few. Metric indexes have been improved over the years and many instances of the problem can be solved efficiently. However, when very large/high dimensional metric databases are indexed exact approaches are not yet capable of solving efficiently the problem, the performance in these circumstances is degraded to almost sequential search.

To overcome the above limitation, non-exact proximity searching algorithms can be used to give answers that either in probability or in an approximation factor are close to the exact result. Approximation is acceptable in many contexts, specially when human judgement about closeness is involved.

In vector spaces, on the other hand, there is a very successful approach dubbed Locality Sensitive Hashing which consist in making a succinct representation of the objects. This succinct representation is relatively insensitive to small variations of the locality. Unfortunately, the hashing function have to be carefully designed, very close to the data model, and different functions are used when objects come from different domains.

In this paper we give a new schema to encode objects in a general metric space with a uniform framework, independent from the data model. Finally, we provide experimental support to our claims using several real life databases with different data models and distance functions obtaining excellent results in both the speed and the recall sense, specially for large databases.

Categories and Subject Descriptors

E.2 [**Data**]: DATA STORAGE REPRESENTATIONS—*Hash-table representations*; H.3.3 [**Information Systems**]: Information search and retrieval—*Search process*

SISAP 2010, September 18-19, 2010, Istanbul, Turkey.

1. INTRODUCTION

Proximity searching appears in a large number of applications ranging from pattern recognition, web searching, clustering, data mining, knowledge management and textual and multimedia information retrieval, to name a few.

The most general setup to state the proximity searching problem is the (dis)similarity space model. A (dis)similarity space is a pair $(\mathbb{X}, d)$ with $\mathbb{X}$ a set and $d(\cdot,\cdot)$ a positive real valued function between objects denoted by $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$. A finite subset $\mathbb{U} \subseteq \mathbb{X}$ is the database. The proximity searching problem of interest for us is to obtain the k-nearest neighbors of a query $q \in \mathbb{X}$ defined as $KNN(q) = \{x \in \mathbb{U} | d(x,q) \leq d(y,q) \forall y \in \mathbb{U}\}$ and $|KNN(q)| = k$. If $d(\cdot,\cdot)$ obeys the metric properties as follows: for any $x, y, z \in \mathbb{X}$, $d(x,y) > 0$, $d(x,y) = 0 \iff x = y$, $d(x,y) = d(y,x)$ (symmetry), and the triangle inequality: $d(x,z) + d(z,y) \geq d(x,y)$ then the pair $(\mathbb{X}, d)$ is called a *metric space*. A more restrictive setup consist in demanding the objects to have coordinates and the distance to be an ℓ_p metric, in this case the objects will be either vectors or a vector representation of the objects.

Proximity queries can be solved trivially by measuring the distance from the query to every database element. If the database is large and/or the distance function is expensive this sequential scan is no acceptable and an index should be used to obtain sublinear query times. As reported in many places [11, 7, 3], sublinear query times are only achievable in either low-dimensional vector spaces or intrinsically low dimensional metric spaces.

To alleviate the above described problem, authors have resorted to approximate or probabilistic solutions to the proximity problem as surveyed in [10]. In the probabilistic setup the query time can be speedup if instead of searching for the correct solution set $KNN(q)$ we accept a set $\widehat{KNN}(q)$ such that $\frac{KNN(q) \cap \widehat{KNN}(q)}{k} \geq \delta$ for a given $\delta < 1$. In other words, we accept up to a small number of false positives in exchange of speeding up the query process.

Probably the most well known probabilistic solution for the proximity searching problem in the context of high dimensional vector spaces is the *locality sensitive hashing* (LSH) (described below in 1.1) which consist in projecting the original high dimensional space to a lower dimensional space obtained as a random sample of the original coordinates. For the metric space searching model, a successful probabilistic approach is to use *permutations* (described below in 1.2) as object descriptors and to search in the representation space instead of searching in the original database, provided a distance between permutations is defined. The key difference

between both approaches is that a proximity searching is converted on the one hand, into a set of *exact* searches in the LSH approach, and on the other hand into a simpler proximity search in the permutations space.

A number of adhoc indexes have been designed to speedup the searching in the permutations space without a sequential scan. The pp-index approach consist in building a compact trie with the prefixes of the permutations, and obtaining as candidate answer to the query the subtree matching the prefix of the query, as described in [8]. The metric inverted index compute an approximation to the distance between permutations using an inverted index [1].

1.1 Locality Sensitive Hashing

The idea behind LSH is to pack a proximity searching problem into an exact searching problem using the following guideline:

Definition A family $\mathcal{H}$ of functions $h : R^d \to U$ is called $(P1, P2, r, cr)$-sensitive, if for any p, q:

- if $||p - q|| < r$ then $Pr[h(p) = h(q)] > P1$
- if $||p - q|| > cr$ then $Pr[h(p) = h(q)] < P2$

The above implies using exact matching only (comparing $h(q)$ and $h(p)$ with q the query and p the database elements), which can be accomplished in constant time, to solve a proximity searching. The most well studied case for LSH is the Hamming space which boils down to using random subsampling of bit-strings as described in [2]. A downside of LSH is the need to design a hashing function $\mathcal{H}$ with particular properties for each data model. In particular, for a general metric space, finding the bounding probabilities can be very hard.

To fix ideas, lets see an example with the binary case. Let $g_{xyz}(s)$ be a function copying and concatenating the x, y, z-th bits of s, and $\mathcal{H} = \{g_{152}, g_{743}\}$. Consider u, v, w in $\mathbb{U}$ in a d dimensional binary hamming space, the $Pr[g_x(u) = g_x(v)] = 1 - D(u, v)/d$ where D is the hamming distance. Then $Pr[g_{xyz}(u) = g_{xyz}(q)] = 1 - (D(u, v)/d)^3$ must follows the properties of definition 1.1. Let $u = 10010101$, $v = 11010000$, $w = 01001101$. Exemplifying, $D(u, v) = 3$, $D(u, w) = 4$, $D(v, w) = 5$, our hashing functions gives $g_{152}(u) = 100$, $g_{152}(v) = 101$, $g_{152}(w) = 011$, $g_{743}(u) = 010$, $g_{743}(v) = 010$, $g_{743}(w) = 000$. We can see that $g_{743}(u) = g_{743}(v) = 010$ whose are in fact the closer pairs in the triplet.

For vector spaces in L_p we can assume integer coordinates without loss of generality. This case reduces to the Hamming space by concatenating the coordinates in fixed sized unary representation per coordinate, and then use Hamming distance to compute the distance. The L_2 norm can be embedded into L_1 using projections. In general L_p can be projected in any other L_s with $s < p$ using random projections, as depicted in [2].

1.2 Overview of the Permutations Index

The motivation behind this indexing method [6] is to shift the problem of comparing directly the query object against every object in the database to comparing the *perspective* in which a set of elements is perceived. Each database element has an unique perspective of the *permutants* (defined below) and the query is only compared to those elements having similar perspective of the permutants.

Let $\mathbb{U}$ be the database of objects, and $\mathbb{P} \subseteq \mathbb{U}$ be a set of distinguished objects from the database, called *permutants*. Each $u \in \mathbb{U}$ defines a *permutation* Π_u, where the elements of $\mathbb{P}$ are written in increasing order of distance to u. Ties are broken using any consistent order, for example, the order of the elements in $\mathbb{P}$.

Definition Let $\mathbb{P} = \{p_1, p_2, \ldots, p_k\}$ and $u \in \mathbb{S}$. Then we define Π_u as a permutation of $(1 \ldots k)$ so that, for all $1 \leq i < k$ it holds either $d(p_{\Pi_u(i)}, x) < d(p_{\Pi_u(i+1)}, x)$, or $d(p_{\Pi_u(i)}, x) = d(p_{\Pi_u(i+1)}, x)$ and $\Pi_u(i) < \Pi_u(i + 1)$.

Each database element u is represented by a permutation Π_u. The query is represented by Π_q using the same definition. Elements that are close have similar permutations. We define what we mean by similar permutations as follows.

Sum the squares of differences in the relative positions of each element in both permutations. That is, for each $p_i \in \mathbb{P}$ we compute its position in Π_u and Π_q, namely $\Pi_u^{-1}(i)$ and $\Pi_q^{-1}(i)$, and sum up the squares of the differences in the positions [6].

Definition Given permutations Π_u and Π_q of $(1 \ldots k)$, Spearman Rho is defined as

$$S_\rho(\Pi_u, \Pi_q) = \sum_{1 \leq i \leq k} \left(\Pi_u^{-1}(i) - \Pi_q^{-1}(i)\right)^2.$$

Note that we can compute $S_\rho(\Pi_q, \Pi_u)$ by obtaining the inverse of both permutations and then computing the Euclidean distance of the inverse. It is also shown in [6] that we can use the sum of the absolute of the differences, without the squares, without noticeable penalization in the index recall.

The result is a table of n rows (one per database element) and k columns (one per permutant). Each cell needs $\lceil \log_2 k \rceil$ bits to store one permutation at each row. The indexing cost is kn distance computations plus $O(nk \log k)$ CPU time to sort all the permutations.

The search has two phases. The first sorts the database according to the permutation distance and selects as candidates the first elements. The second phase is to check the list. The permutation index allows KNN searches in pseudo-metric spaces, because the triangle inequality is not used explicitly. Our technique inherits this property allowing faster searches and smaller indexes.

Certain approximations to searching in the permutations space can be done as explored in [8, 1]. We describe below another alternative leading directly to a binary reduction of the problem which is suitable for LSH indexing with the Hamming distance.

1.3 Brief Permutations

A permutation can be represented in a binary string, using half a bit for each permutant. The idea for this representation is to discretize the *displacement* of each permutant from the canonical position (the identity permutation). This idea is exploited in [12] and described in algorithm 1 where the *Encode* function pack the permutation into a bit string. Notice that for large enough m (e.g $m \geq \frac{|P|}{2}$) the permutants in the center will be rarely set to 1. In order to use more effectively the space, we can pack a double number of permutants by swapping the central part, as described in 2.

Algorithm 1 Bit-encoding of the permutation P under the module m

Encode(Permutation P, Positive Integer m)

```
1: Let P^{-1} be the inverse P.
2: C ← 0 {Bit string of size |P|, initialized to zeros}
3: for all i from 0 to |P| − 1 do
4:    if |i − P^{-1}[i]| > m then
5:       C[i] ← 1
6:    end if
7: end for
8: return C
```

Algorithm 2 Bit-encoding permutations with swapping the central part

EncodePC(Permutation P, Positive Integer m)

```
1: Let P^{-1} be the inverse P
2: C ← 0^{|P|} {Bit string of size |P|, initialized to zeros}
3: M = ⌊|P|/4⌋
4: for all i from 0 to |P| − 1 do
5:    I ← i
6:    if ⌊I/M⌋ mod 3 ≠ 0 then
7:       I ← I + M
8:    end if
9:    if |I − P^{-1}[i]| > m then
10:      C ← C|(1 << i)
11:   end if
12: end for
13: return C
```

As a forming example to fix ideas, let $m = 2$, $u = (3, 6, 2, 1, 5, 4)$, $r = (5, 3, 1, 6, 2, 4)$ and $q = (6, 2, 3, 1, 4, 5)$. After the inverse $u^{-1} = (4, 3, 1, 6, 5, 2)$, $r^{-1} = (3, 5, 2, 6, 1, 4)$ and $q^{-1} = (4, 2, 3, 5, 6, 1)$. Applying algorithm 1 we have $\hat{u} = (|1-4| > m, |2-3| > m, |3-1| > m, |4-6| > m, |5-5| > m, |6-2| > m) = (1, 0, 0, 0, 0, 1)$, supposing $|a-b| > m$ evaluates to 1 for true and 0 for false. Similarly, we obtain $\hat{r} = (0, 1, 0, 0, 1, 0)$ and $\hat{q} = (1, 0, 0, 0, 0, 1)$. If H is the hamming distance, $H(\hat{u}, \hat{q}) = 0$ and $H(\hat{r}, \hat{q}) = 2$. Clearly, q is to u, and this can be verified using S_ρ as $S_\rho(u, q) = 8$, and $S_\rho(r, q) = 46$.

The rationale behind the brief permutations, or the binary encoded permutations, is to capture the proximity between the representations. The binary differences account for displacements larger than m in the permutant positions, which is the same behavior we observe in S_ρ.

It is interesting that even computing sequentially the Hamming distance is way faster, because we can use XOR bit-operation ($\oplus$) using the bit parallelism inherent in the computer integer operations computing 32 or 64 operations per instruction instead of the most expensive operations difference and product used in the S_ρ. Bit count can also be precomputed using a table.

With the above assumptions, $0 \oplus 0 = 0$ accounts for a small movement and $0 \oplus 1 = 1$ denotes a large movement. Since $1 \oplus 1$ can be interpreted either as a large or an small displacement (because the two permutations are compared against the identity , in order to encode in just one bit each permutant we choose only one, a small difference, which also coincide with the Hamming distance, and hence $1 \oplus 1 = 0$. If we choose $1 \oplus 1 = 1$ can be efficiently computed using $\oplus$ as OR instead of XOR. The third alternative is to use two bits, encoding left and right displacements. Experimentally all the above alternatives behave the same, except when the number of permutants is small, which is not very interesting in our setup.

2. LOCALITY SENSITIVE HASHING FOR METRIC SPACES

Obtaining the LSH for the general metric space setup is now very natural. The first step is to compute the brief permutation of all the database elements, and then compute a family of hashes for each binary string representing the objects. LSH can also be used with the regular (full) permutation representation with an ℓ_p hash family. With either alternative we can encapsulate finding a suitable LSH for each data model. Our approach is more general.

It is worth notice the nice properties of the permutation-space representation are preserved in the brief representation, namely the ability to predict proximity in the non-metric case, as described in [6], since the triangle inequality is not used explicitly.

To satisfy a query we first map q to the permutants space, and then retrieve a set of candidates using a suitable hash function (either in the full or brief representation). Once the candidates are chosen, there are two possibilities:

- If the original similarity function is expensive, we must refine LSH candidates using the S_ρ.
- If the original similarity function is cheap, we verify the entire set of candidates.

Finally, all the candidates are ranked with the original similarity function and presented as result.

In the full representation we need an embedding for L_1 or L_2 (Spearman footrule and Spearman ρ respectively) as detailed in [9, 2], this alternative produces good results but increase the searching and indexing costs; and since the permutations are just an approximation and not the final result we lean to using LSH over Hamming directly. If the original distance is not expensive, we don't store the mapping, keeping just the corresponding hashes.

In our experiments, We study three parameters:

- Number of permutants or references (*numrefs*). The number of permutants, randomly chosen at the preprocessing step, fixes the size of the bit-string. We stress we don't need to save bit-strings for most applications. As a rule of thumb, larger numrefs implies better recall.
- Number of hashing functions (*numhashes*). The number of hashing functions to be computed by LSH, affecting the total number of candidates and the recall: the more candidates, the bigger the recall and higher the cpu cost for checking candidates.
- Size of the bit sample (*samplesize*). Another LSH parameter, it controls the probability of collision. Larger *samplesize* values produces sparser tables, resulting in faster searches but with few candidates (most of the times meaning low recall).

It is difficult to obtain bounding probabilities for the hashing. Nevertheless, the problem is suitable for an experimental tuning of the three parameters described above. A good

tradeoff between speed and recall can be obtained by varying the parameters in a specific setting. We will devote the rest of the paper to show the experimental results in a number of example databases.

3. EXPERIMENTS

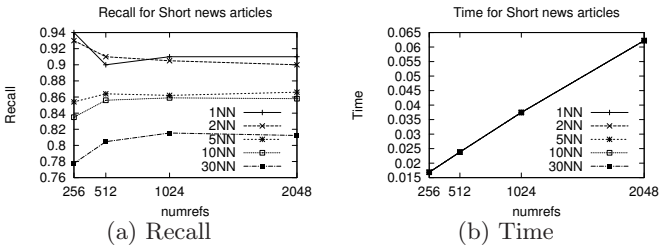
The test databases were taken from the *metric space* library[1] and the CoPhIR project [4]. Our implementation is available as open source from `www.natix.org`. All indexes share the same distance function's implementation. We must notice that all the test indexes were developed in C# under the mono 2.6.1 framework and linux 2.6.27 under Ubuntu/Linux 8.10 in a Xeon 2.33GHz with 8GiB of ram workstation. The indexes run in main memory and without parallelization. We choose four databases: documents and dictionaries for textual information retrieval, color's histogram and MPEG7 vectors for images in two different databases for multimedia information retrieval. Please note that the brief representation of the permutant space allows keeping the index in main memory, as well as the LSH tables.

The experiments were performed with brief indexes using modulus of 0.5 times the number of permutants, applying the central permutation technique as described in Algorithm 2. In our two level index we use several configurations, providing information to optimize in both recall and time axes. We provide a simple comparison against the sequential scanning to give an standard point of reference with other approaches. Additionally, the comparison against the sequential brief index is shown. The behavior of the BKT for the same set of queries is described too. The experimental results covers both execution time and number of distance computations, trying to give a complete idea of the performance from distinct points of view.

Our implementation uses a table of precomputed Hamming distances for two bytes integers. We considered all the distances as expensive, to give an idea of the worst case scenarios, and we computed the Hamming distance and sorted the candidates obtained from the LSH phase.

3.1 Documents

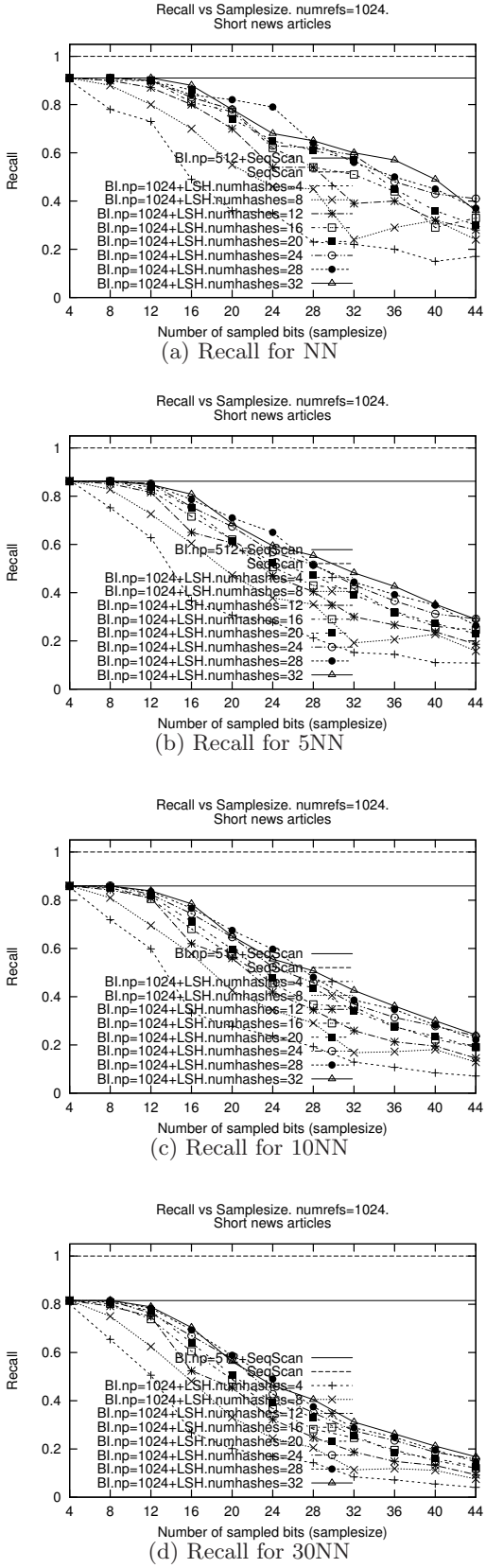
We used a collection of 25157 short news articles in $TF \times IDF$ format from Wall Street Journal $1987 - 1989$ files, specifically from TREC-3 collection. We use the standard metric of the angle between vectors as distance.



(a) Recall (b) Time

Figure 1: Recall and time behavior for brief index for different configurations of the database of news documents

[1]Metric space library `www.sisap.org`.



(a) Recall for NN

(b) Recall for 5NN

(c) Recall for 10NN

(d) Recall for 30NN

Figure 2: Experiment results for brief index against the two layer index using the documents $TF \times IDF$ collection and vector's angle as distance.

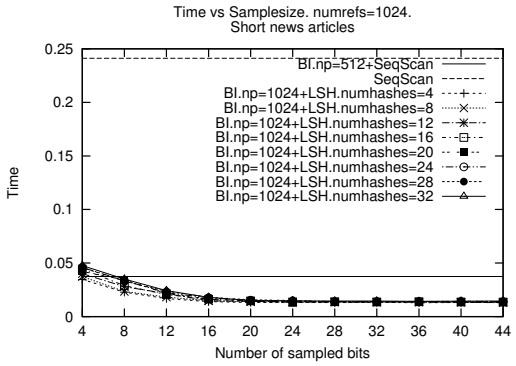


Figure 3: Experiment results for brief index against our two layer index using the documents $TF \times IDF$ collection and vector's angle as distance.

We extracted 100 random documents as queries, these documents were not indexed. We produce queries for 1, 5, 10, and 30 nearest neighbors (a metric index, like BKT using a ring width of 0.001 [7] needs to check up to 98% of the database for 30NN). We fix the number of distance computations to 1000 + the number of references, see Figure 2. Then for $30NN$ with recall bigger than 0.82 we need to review only the 8% of the database instead of the 98% in the alternative metric index (not shown for space constrains).

The recall and time behavior for the original brief permutation algorithm are shown 1. We can configure our index for several number of references, i.e. *numrefs* values, and we can reach at maximum the recall listed in Figure 1(a). Looking for a fast configuration with a good recall we chose to use 1024 permutants. Figure 1(b) shows the time necessary to solve queries. These times set the allowed upper bounds for each configuration. All curves are drawn as a single line because all of them uses the same resources.

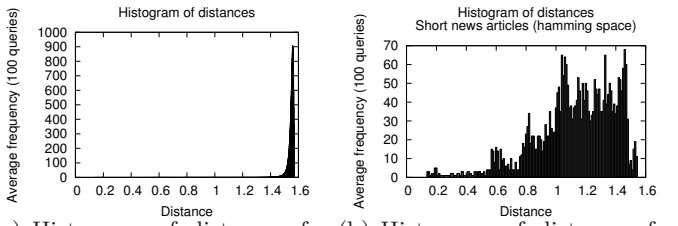
Figure 3 shows the average time per search for different configurations of permutants and LSH paramenters. For sampled bits bigger than 8, we can see that the two layer index is faster than the sequential brief index and always faster than the naïve solution of Figure 1(b), and in fact for $samplesize = 12$ we achieve faster times than the faster setup in the original algorithm. Configurations beyond these values are notoriously faster than any configuration in the original approach.

We claim that our mapped space is *easier* than the original one, since the high recall in the output (with a very small number of candidates) and the histogram of distances for every query set 4(b). The resulting space is more indexable by traditional metric indexes as shown in [5]. Using LSH as second layer, can be replaced by any other index, but LSH is faster and with a small overhead.

3.2 Dictionary

Searching in dictionaries for misspelled words, OCR errors, etc. is a common task in information retrieval or pattern recognition. We use the metric space library's English dictionary with 69069 words. English dictionary was selected to avoid encoding problems, but we expect the same behavior from other non-agglutinant languages. We used the plain edit distance.

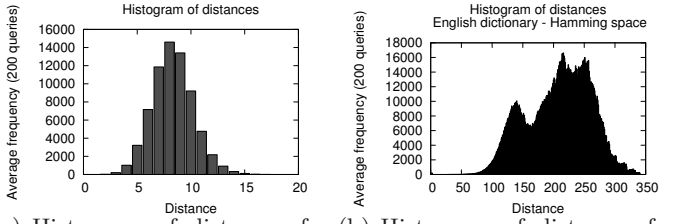
We took 200 randomly selected words from the database



(a) Histogram of distances for the original space (angle between vectors)

(b) Histogram of distances for the mapped space (Hamming using the brief index)

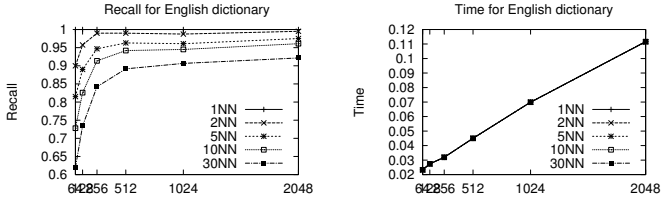
Figure 4: Histogram of distances for our query set, both in the original space and the mapped space, showing the behavior in the entire database.



(a) Histogram of distances for the original space

(b) Histogram of distances for the mapped space (Hamming using the brief index)

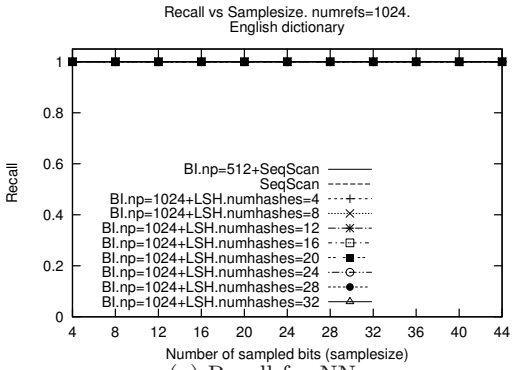
Figure 5: Histogram of distances for the English.dic dictionary, on the original space and the hamming space



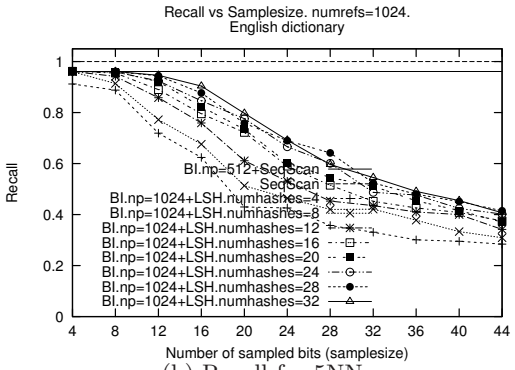
(a) Recall

(b) Time

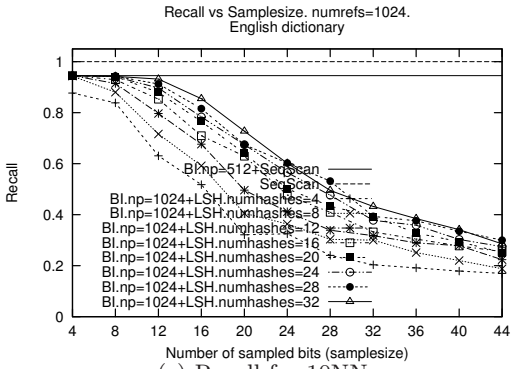
Figure 6: Recall and time behavior for brief index and different configurations over English dictionary



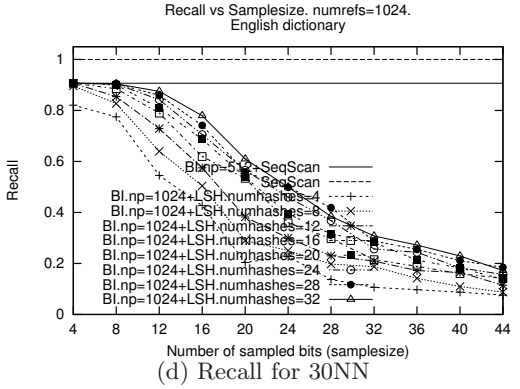
(a) Recall for NN



(b) Recall for 5NN



(c) Recall for 10NN



(d) Recall for 30NN

Figure 7: Results for brief index and our two layer index for the english dictionary, using edit distance.

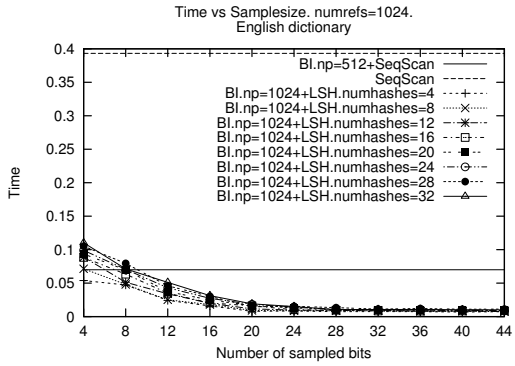


Figure 8: Time performance for brief index and the two layer index for color's histogram collection and L_2 distance.

as queries and searched for 30NN. For the NN (i.e the word as query) we have perfect recall of 1.0 using more than 128 permutants, and was close to one for 2NN also. To keep the comparison. in the same framework we completed the full 5, 10 and 30NN (Figure 7(d)). Please note that being a discrete distance for words, we have many ties in the KNN searches. The histogram of distances for our query set is shown in Figure 5, both the original distance and the mapped space (i.e. hamming) are depicted.

Just to compare, the BKT needs to review 56% of the database for the same task. The brief index needs to review 3% to get a recall of 0.97 for 2NN and 0.84 for 30NN. The average search time is shown in Figure 8, where our index is faster for any configuration with *samplesize* bigger than 8.

3.3 MPEG7 Vectors

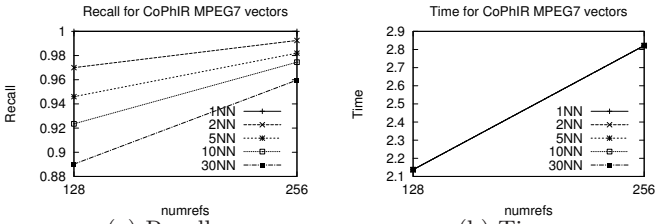
A database of 10 million 208-dimensional vectors from the CoPhIR project [4] were selected. It uses the L_1 distance. Every vector is one of five different MPEG7 vectors as described in [4]. We choose the first 200 vectors from the database as queries. Search for the 30NN were performed. Traditional metric indexes like BKT do not perform well with the this database size. This database is difficult because its size, since it avoids the use of traditional structures with high overhead.

Figure 10 shows the recall for different configurations, for a brief index with both sequential scanning and LSH layer. Here the step for *samplesize* is bigger than other databases because we want to discern between a larger number of objects. We fix the number of verified candidates to 50000, equivalent to review just the 0.5% of the database.

The brief index recall is shown in Figure 9(a), where the recall is close to 1. We only tested 128 and 256 permutants because of these good results. Figure 9(b) shows a very competitive time against the sequential scan of 18.6 seconds.

A dramatic improvement is observed in the time using the two layer index, see Figure 11. For index with $samplesize = 32$, queries are solved in close to 0.1 seconds. For $samplesize = 96$ over 0.002 seconds. These is achieved because the LSH layer never gives the entire set of requested candidates. For example $samplesize = 32$ achieves approximately 30000 candidates for the presented *numhashes*. Fixing $samplesize = 96$ we get close to 500, 1100 and 1600 candidates for 4, 8, 12 *numhashes* respectively. So, if we optimize the index for

performance it needs to review 0.016% of the database. A higher recall can be obtained reviewing 0.3% of the objects. The recall is affected by the small candidate set in the two layer index, as shown in Figure 10.



(a) Recall (b) Time

Figure 9: Recall and time behavior for brief index and different configurations over English dictionary

4. CONCLUSIONS AND FUTURE WORK

In this paper we presented a method to obtain a Locality Sensitive Hashing schema for general metric spaces and we showed experimentally how to obtain a good tradeoff between speed and recall. Our approach allows to create similarity indexes in a black box scheme without a deep knowledge about the data model and its similarity function.

The method is two layered. In the lower layer we have a permutation based index which can handle high dimensional spaces and can be used in similarity spaces as well. In the upper layer we use LSH allowing scalability. The entire setup reduces to a Hamming space to be searched for candidates and a subsequent candidate check with the original distance function.

The performance can be tuned by making large/small candidate lists to be verified and retrieved. Even with an small candidate set we found a competitive recall. In the storage side, we have presented two options. The first one for expensive similarity functions: we need to save 1 bit per permutant per each item in the database + hash tables, in the second (i.e. cheap similarity functions) we only need to save hash tables.

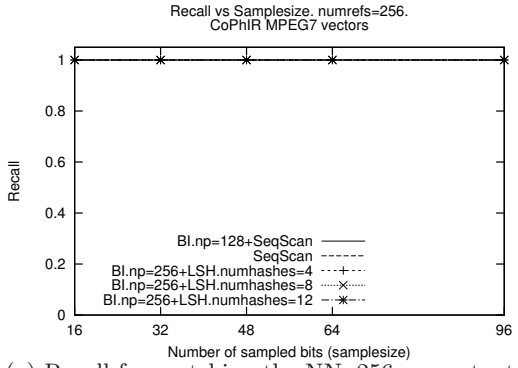
Very large databases show better the improvement in performance, as observed in the CoPhIR database. Smaller databases are improved but not significantly, they remain in the same order of magnitude even if the query time is improved two or three fold.

We are currently working in obtaining theoretical bounds on the distance from the exact proximity query and the approximated output we obtain with LSH and binary permutations.

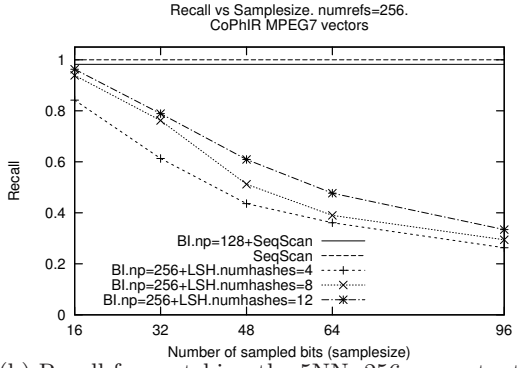
Future work includes secondary memory implementation and the inherent parallelism of the technique: the computation of permutations, searching in several hash functions, the verification stage, and the necessary sorting steps. Then a complete set of algorithms can be designed specially for multiple core processors and GPU architectures.

5. REFERENCES

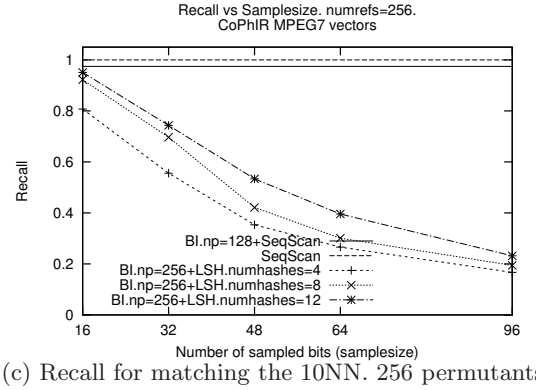
[1] Giuseppe Amato and Pasquale Savino. Approximate similarity search in metric spaces using inverted files. In *InfoScale '08: Proceedings of the 3rd international*



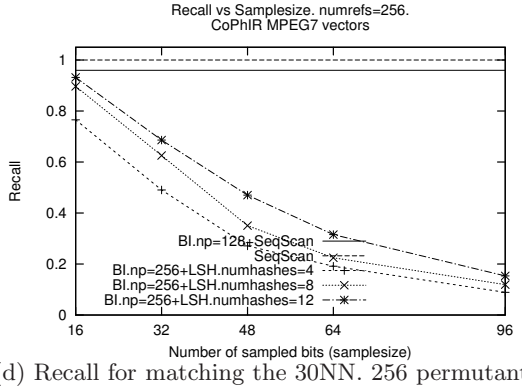
(a) Recall for matching the NN. 256 permutants



(b) Recall for matching the 5NN. 256 permutants



(c) Recall for matching the 10NN. 256 permutants



(d) Recall for matching the 30NN. 256 permutants

Figure 10: Results for brief index and our two layer index for CoPhIR 10M MPEG7 database.

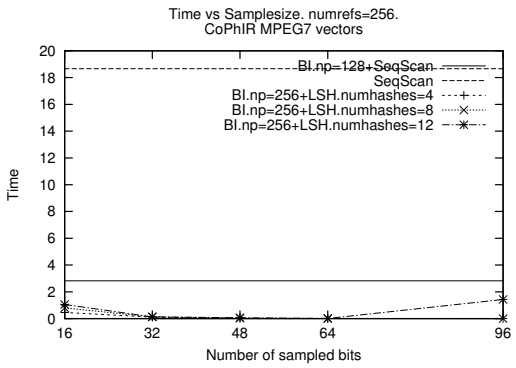


Figure 11: Time performance for brief index and the two layer index for MPEG7 vectors of 10M CoPhIR database

conference on Scalable information systems, pages 1–10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[2] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[3] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[4] Paolo Bolettieri, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, Tommaso Piccioli, and Fausto Rabitti. Cophir: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627v2, 2009.

[5] B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357–2366, 2003.

[6] Edgar Chavez, Karina Figueroa, and Gonzalo Navarro. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(9):1647–1658, September 2008.

[7] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001.

[8] Andrea Esuli. Pp-index: Using permutation prefixes for efficient and scalable approximate similarity search. In *LSDS-IR 2009 Workshop*, 2009.

[9] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[10] M. Patella and P. Ciaccia. Approximate similarity search: A multi-faceted problem. *Journal of Discrete Algorithms*, 7(1):36–48, 2009.

[11] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers, 2006.

[12] Eric Sadit Tellez, Edgar Chavez, and Antonio Camarena-Ibarrola. A brief index for proximity searching. In *Proceedings of 14th Iberoamerican Congress on Pattern Recognition CIARP 2009*, 2009.