Metric Space Searching based on Random Bisectors and Binary Fingerprints *

José María Andrade, César A. Astudillo, and Rodrigo Paredes

Departamento de Ciencias de la Computación, Universidad de Talca, Chile.
jandrade@alumnos.utalca.cl, {castudillo,raparede}@utalca.cl

Abstract. We present a novel index for approximate searching in metric spaces based on random bisectors and binary fingerprints. The aim is to deal with scenarios where the main memory available is small. The method was tested on synthetic and real-world metric spaces. Our results show that our scheme outperforms the standard permutant-based index in scenarios where memory is scarce.

Keywords: Similarity search, Random bisectors, Binary signatures.

1 Introduction

Similarity search is an extension of exact searching, motivated by data types that cannot be queried by exact matching. This problem consists in finding elements within a given dataset that are similar to a given query according to a similarity criterion. There is a wide range of applications where the exact comparison is of little use. For instance, consider the case when a person is asked to scan its fingerprint so as to retrieve medical records. The system will obtain a different version of the fingerprint depending on the amount of pressure the person places on the sensor. In these situations, the only way of retrieving relevant objects — that is, objects that are similar to the query— is by tolerating small variations between objects. Other applications include multimedia databases containing images, audio, video, documents, and so on [4].

Proximity queries can be formalized using the metric space model [4, 8, 10, 11]. Essentially, this model considers a pair $(\mathbb{X}, d)$, where $\mathbb{X}$ is a universe of objects and $d : \mathbb{X} \times \mathbb{X} \to R^+ \cup \{0\}$ is a nonnegative distance function defined among them. Objects in $\mathbb{X}$ do not necessarily have coordinates (think, for instance, in strings). On the other hand, the function d provides a dissimilarity criterion to compare objects from $\mathbb{X}$. In general, the smaller the distance between two objects, the more "similar" they are. The function d satisfies the metric properties, namely: *positiveness* $d(x, y) \geq 0$, *symmetry* $d(x, y) = d(y, x)$, *reflexivity* $d(x, x) = 0$, and *triangle inequality* $d(x, z) \leq d(x, y) + d(y, z)$, for every $x, y, z \in \mathbb{X}$.

The standard scenario of proximity searching considers a finite database of interest $\mathbb{U} \subset \mathbb{X}$, of size n. Later, when a new query object $q \in \mathbb{X} \setminus \mathbb{U}$ arrives, its proximity query consists in retrieving relevant objects from $\mathbb{U}$.

* This work is partially funded by Fondecyt grants 11121350 and 1131044, Chile.

There are two basic queries, namely, *range* and *k-nearest neighbor* ones. The range query (q, r) retrieves all the elements in $\mathbb{U}$ within distance r to q. The k-nearest neighbor query $NN_k(q)$ retrieves the k elements in $\mathbb{U}$ that are closest to q. Both queries can be trivially answered by exhaustively scanning the database, requiring n distance evaluations. However, as the distance function is assumed to be expensive to compute (e.g., when comparing two fingerprints), frequently the complexity of the search is defined in terms of the total number of distance evaluations performed, instead of using other indicators such as CPU or I/O time. Thus, the ultimate goal is to build an offline index that, hopefully, will accelerate the process of solving online queries.

In this paper, we show a novel metric space index based on random bisectors and binary fingerprints to approximately solve the similarity search problem. An advantage of our index is that only requires a marginal amount of space. As we detail in the experimental section, when solving the $NN_1(q)$ in the hard metric space of uniformly distributed vectors in $\mathbb{R}^{128}$ under Euclidean distance, our method is able to retrieve 98% of the true answer by analyzing only 10% of the dataset, and this is achieved by using only 288 bits per element in the index. In the same experimental setup, our index overcomes the state-of-the-art Permutation Based Index (PBI) [3], as the later only retrieves 77% of the answer.

2 Related Work

In this section we briefly explain the *compact-partition based algorithms* and the PBI. Then, we describe two concepts that are central for the present study, namely, the Hamming distance and locality-sensitive hashing.

Compact-Partition Based Indices These methods split the space into zones as compact as possible. For each partition, they store a representative object and extra information that permits the exclusion of that partition at query time.

This family of methods can be divided into the *Voronoi partition* and *covering radius* schemes. A *Voronoi Partition* method selects a subset of representative objects, called centers, denoted as $\{c_1, \ldots, c_m\}$, associating the remainder of the objects according to their proximity to its closest center. At query time, the distances $(d(q, c_1), \ldots, d(q, c_m))$ are evaluated and the closest center c is identified. Those regions satisfying the inequality $d(q, c_i) > d(q, c) + 2r$ can be safely discarded because they never intersects the query ball. On the other hand, the *covering radius*, $cr(c_i)$, corresponds to the distance between its center and the farthest element in its respective zone. So, at query time, when $d(q, c_i) - r > cr(c_i)$, then the zone i can be safely discarded.

The Permutation Based Index Let $\mathbb{P} \subset \mathbb{U}$ be a subset of permutants. Each element $u \in \mathbb{U}$ computes the distance towards all the permutants $p_1, \ldots, p_{|\mathbb{P}|} \in \mathbb{P}$. The PBI *does not store distances*. Instead, for each $u \in \mathbb{U}$, it stores a sequence of permutant identifiers $\Pi_u = i_1, i_2, \ldots, i_{|\mathbb{P}|}$, called the permutation of

u. Each permutation Π_u stores the identifiers in increasing order of distance, so $d(u, \mathbb{P}_{i_j}) \leq d(u, \mathbb{P}_{i_{j+1}})$. Permutants at the same distance take an arbitrary but consistent order. Thus, a simple implementation needs $n|\mathbb{P}|$ space. Observe that it is possible to compact several permutant identifiers in a single machine word.

The crux of the PBI is that two equal objects are associated to the same permutation, while similar objects are, hopefully, related to similar permutations. In this sense, when Π_u is similar to Π_q one expects that u is close to q. The similarity between the permutations can be measured by Kendall Tau K_τ, Spearman Footrule S_F, or Spearman Rho S_ρ metric [5], among others. As these three distances have similar retrieval performance [3], for simplicity we use S_F, defined as $S_F(\Pi_u, \Pi_q) = \sum_{j=[1,|\mathbb{P}|]} |\Pi_u^{-1}(i_j) - \Pi_q^{-1}(i_j)|$, where $\Pi_u^{-1}(i_j)$ denotes the position of permutant p_{i_j} in the permutation Π_u. For example, if we have two permutations $\Pi_u = (42153)$ and $\Pi_v = (32154)$, then $S_F(\Pi_u, \Pi_v) = 8$.

Finally, at query time, we compute Π_q and compare it with all the permutations stored in the PBI. Next, $\mathbb{U}$ is traversed in increasing permutation dissimilarity. If we limit the number of distance computations, we obtain a probabilistic search algorithm that is able to find the right answer with some probability.

Hamming Distance Given two binary sequences of equal length, the Hamming distance is the number of positions at which the corresponding symbols differ [7].

Locality-Sensitive Hashing The Locality-Sensitive Hashing (LSH) is a family of techniques that map the input data into a set of buckets using several hash functions. The overall goal is that, with high probability, similar objects are mapped to the same bucket, and simultaneously, different objects are assigned to different buckets. This concept differs from the usual approach of hash functions, instead of avoiding collisions between similar objects, LSH encourages them.

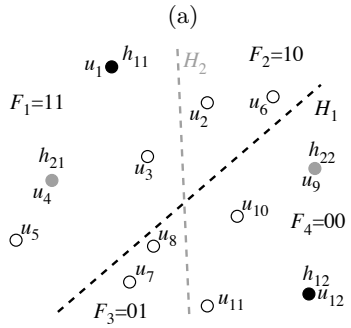
The key point of LSH is to define the hash function family. The authors of [1] survey several alternatives for the vector space. One of those alternatives is related to our bisector approach. That idea is to pick random unit-length vectors $u \in \mathbb{R}^D$ and then define $h_u(v) = sign(u \cdot v)$. Using many random vectors, it is possible to build a binary sequence for each object. This hash function family was devised to approximate the cosine distance between two vectors in $\mathbb{R}^D$.

As far as we know, there is only one application of LSH to metric spaces. In [9], the authors use LSH to avoid the sequential scanning of the PBI.

3 Random Bisectors and Binary Fingerprints

In this section, we detail our proposed index for approximated similarity searching in metric spaces and the corresponding algorithms for solving similarity queries. The index uses the concepts of *virtual* random bisectors and binary fingerprints (RBBF) to build the data structure. We use *virtual* bisectors, since in general metric spaces, objects do not necessarily possess Cartesian coordinates.

Our index, called the Random Bisectors and Binary Fingerprints 1 (RBBF1), represents the objects using binary fingerprints. RBBF1 can be classified as a



	H_1	H_2		H_1	H_2
u_1	1	1	u_7	0	1
u_2	1	0	u_8	0	1
u_3	1	1	u_9	0	0
u_4	1	1	u_{10}	0	0
u_5	1	1	u_{11}	0	0
u_6	1	0	u_{12}	0	0

Fig. 1: Randomly generated data points used to show the behavior of our index. In (a), we see the objects partitioned using two virtual bisector hyperplanes H_1 and H_2. In (b), we see the fingerprint associated to each object.

new LSH approach for metric space searching, where each bisector hyperplane is a member in the hash function family.

For lack of space, in the following we only sketch the algorithms. A longer explanation, including pseudo-codes can be found in [2] (in Spanish).

Construction RBBF1 only stores the binary fingerprint F_j for each object $u_j \in \mathbb{U}$. One of the main advantages of this philosophy is that it demands very little memory. To compute the fingerprint of an object, we need to simulate several bisectors. In this context, a bisector is understood as a *virtual* hyperplane which is orthogonal to the imaginary segment connecting the two endpoints and which intersects the midpoint of the segment.

If we were considering the vector space, each bisector hyperplane can be actually computed. However, in general metric spaces, objects do not necessarily have coordinates. Instead, what is available is the dissimilarity function d. Hence, when randomly picking two objects $h_{i1}, h_{i2} \in \mathbb{U}$, we can implicitly separate the space into two regions (by closeness to h_{i1} or h_{i2}) in an analogous manner as the above-mentioned bisector. Therefore, the method computes the distance between each object $u \in \mathbb{U}$ to the objects h_{i1} and h_{i2}. The index then determines which of two object is the closest, identifying the corresponding region with a bit.

Fig. 1a illustrates the RBBF1. It shows a scatter plot with 12 random objects and two random bisectors. The first bisector, H_1, is induced by the objects $h_{11} = u_1$ and $h_{12} = u_{12}$. The first component of the fingerprint refers to H_1, and is set to one for those elements closer to u_1 and set to zero when they are closer to u_{12}. Analogously, the second bisector, H_2, is generated from the objects $h_{21} = u_4$ and $h_{22} = u_9$. Fig. 1b shows resultant fingerprints for each object.

As we manage several hyperplanes, we store all the location information in a binary matrix F as follows. For all the objects $u_j \in \mathbb{U}$, and for all the λ hyperplanes, if $d(u_j, h_{i1}) \leq d(u_j, h_{i2})$ then $F_{ji} \leftarrow 1$, otherwise, $F_{ji} \leftarrow 0$. The j-th row of the matrix F is called the *fingerprint* for instance j, and contains

λ bits, one for each bisector, respectively. Naturally, the construction cost of RBBF1 is $O(n\lambda)$ both in evaluations of the distance function and CPU time.

Solving Similarity Queries with RBBF1 We use the RBBF1 index to speed up both k-nearest neighbor and range search queries, as explained below.

Our assumption is that two objects that are equivalent (i.e., with distance equal to zero) possess the same binary fingerprint, and that similar objects should be associated to fingerprints that differ in few bits. More in detail, if the fingerprint F_q is similar to the F_u, we expect that the object u is close to q. Note that two neighboring regions in the Voronoi diagram differ in just one bit in their respective fingerprints. We decided to follow the intuitive idea of traversing the dataset following the order induced by the increasing Hamming distance between the query fingerprint F_q and the fingerprint of every object in $\mathbb{U}$.

RBBF1 does not allow the exclusion of objects at query time, which in our opinion is not an inconvenient, because in high-dimensional metric spaces almost all the exact algorithms resort to sequential scanning. Fortunately, as the order induced by the Hamming distance is so promissory, we can stop the searching after reviewing a fraction αn of the objects in the dataset, as a workload, and obtain a really good answer. Naturally, the bigger the workload, the fewer relevant elements are lost by the technique. This desirable property is verified in Section 4 with strings and vectors.

The search mechanism starts by computing the fingerprint F_q of the query object q. Next, the method ranks all the elements within $\mathbb{U}$ by increasing Hamming distance with respect to F_q so as to compute the promissory review order. Subsequently, we use the workload to compare the best ranked objects with the query using the real distance of the metric space.

When solving range queries, the method reports any object in the workload within a distance r with respect to the query object q. On the other hand, in the case of k-nearest neighbors, the k closets objects in the workload are reported.

4 Experimental Evaluation

We tested our method on strings using the edit distance, and also uniformly distributed vectors in $\mathbb{R}^D$, for dimensions $D = 32$, 64, and 128 using Euclidean distance. The experiments were run on an Intel i5 of 2.6 GHz (two physical and four virtual cores), with 2GB of RAM, local drive and MS Windows 8.1 Professional of 64 bits, using JDK version 1.7.0_45. In the construction experiment, we only measure the CPU as RBBF1 needs $2n\lambda$ distance evaluations to build the index. The results shown correspond to averages after 10 constructions.

To test the search method, we measure the percentage of query retrieval varying λ and the workload, and also the percentage of retrieval for a fixed workload varying λ. We compare our approach with the standard PBI in the compact version, that is, we pack several permutant identifiers in the same machine word. The permutations are compared using the Spearman Footrule (see Section 2). Additionally, we do not measure distance evaluations as they are

limited by the workload. The results include the average of 100 NN_k queries. A longer experimental evaluation, including range query results is available in [2].

4.1 String under Edit Distance

We tested RBBF1 using a dictionary called `Dutch.dic`, obtained from the Metric Space Library [6], which contains an unsorted set of 229,328 words belonging to the Dutch language.

Construction RBBF1 pre-calculates the distances between each object of the dataset to the set of λ pair of objects, thus demanding $O(n\lambda)$ time, with a correlation coefficient $R^2 \geq 0.966$.

Searching Fig. 2a shows a summary of the best retrieval results for NN_1 and NN_{20} queries using the optimal value of λ for this dataset. RBBF1 presents a good retrieval performance. For instance, reviewing just a 5% of the dataset, RBBF1 retrieves 96% of the true answers in NN_1 queries, and 82% in NN_{20} queries. This is achieved by requiring *only* a single integer per object. For space constrains, we omit range query plots. However, we verify that the performance is similar. For instance, a query $(q, 1)$ retrieves 1.38 objects in average, and the optimal value of λ is also 24, retrieving a 91.3% of the true answers.

Because 8 identifiers can be packed using 32 bits, it is fair to compare RBBF1 with the standard PBI using 8 permutants. We observed that RBBF1 outperforms the PBI index with respect to retrieval ratio using the same space. We allowed more permutants in the PBI until its performance matches the one obtained by RBBF1 (these curves are omitted in the plot). This occurred when PBI employed 14 permutants, necessitating 63% more memory than RBBF1.

4.2 Uniformly Distributed Vectors under Euclidean Distance

We also performed tests by generating 30,000 random vectors from a uniform distribution in the range $[0, 1]^D$. We randomly selected 100 query items and computed the respective percentage of retrieval. Our aim is to observe the performance of RBBF1 in different dimensional spaces.

Searching Figs. 2b, 2c and 2d summarize the best retrieval results for NN_1 and NN_{20} queries for D equals to 32, 64, and 128, respectively. RBBF1 and PBI are compared using the same memory requirements (i.e., 8 permutants). We also allow that the PBI uses more memory to match the performance of RBBF. This occurs when the PBI uses 63% to 100% more memory than the RBBF.

We observe that for the case of $\mathbb{R}^{32}$, the RBBF1 index with a workload of reviewing 10% of the dataset reaches to 82% and 74% of retrieval for NN_1 and NN_{20}, respectively. Since RBBF1 consistently improves with the available space, we run experiments using longer signatures.

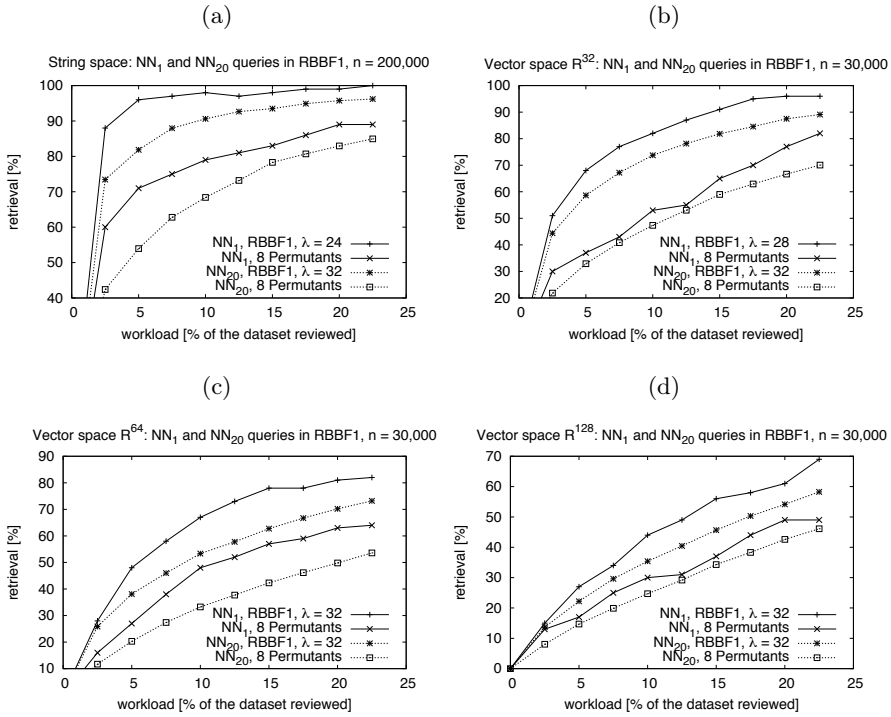


Fig. 2: Comparison of RBBF1 and PBI using NN_k queries on strings and vectors.

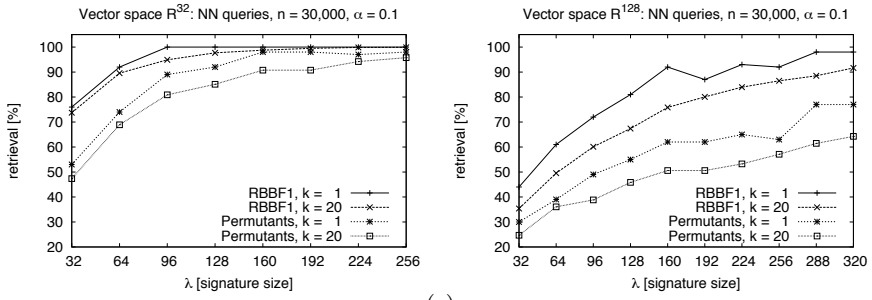
Table 3c shows the size of a permutant set that uses the same memory than the allowed for the long signatures. Figure 3 shows NN_k retrieval for $k = 1$ and 20 reviewing 10% of the database (that is, with $\alpha = 0.1$). We note that RBBF1 effectively uses the extra space in order to improve the retrieval, and also use the space more efficiently than standard PBI.

5 Conclusions and Future Work

The paper presented a novel index for approximate searching that relies on random bisectors and binary fingerprints. The method was tested on strings and vectors, comparing the respective performances with the Permutant-Based Index (PBI). The experimental results show that our index outperformed the PBI scheme. We believe that this occurs because of the sorting mechanism, based on neighboring regions between fingerprints. Remarkably, a marginal amount of memory is required for storing the index. In fact, we use a single integer per object for the experiments with strings and nine for the case of vectors.

Avenues to be explored include the pattern recognition applications of the RBBF and the concept of maintaining a graph of neighborhood between signatures that, hopefully, would improve the results.

(a) (b)



(c)

Signature Size	32	64	96	128	160	192	224	256	288	320
Permutant Set Size	8	16	20	26	32	32	38	43	48	54

Fig. 3: NN_k queries of uniformly distributed vectors in $\mathbb{R}^D$, for $D = \{32, 128\}$, using an increasing number of fingerprints. In (c) the equivalence between the number of fingerprints and the cardinality of the permutant set.

References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun. ACM 51(1), 117–122 (Jan 2008)
2. Andrade, J.M.: Diseño y desarrollo de un índice basado en hiperplanos para búsqueda en espacios métricos (Mayo 2014), memoria de Título del Departamento de Ciencias de la Computación, Universidad de Talca. In spanish.
3. Chávez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI) 30(9), 1647–1658 (2009)
4. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (Sep 2001)
5. Fagin, R., Kumar, R., Sivakumar, D.: Comparing top k lists. SIAM J. Discrete Math. 17(1), 134–160 (2003)
6. Figueroa, K., Navarro, G., Chávez, E.: Metric spaces library (2007), available at `http://www.sisap.org/Metric_Space_Library.html`
7. Hamming, R.W.: Error detecting and error correcting codes. The Bell System Technical Journal 29(2), 147–160 (1950)
8. Hjaltason, G., Samet, H.: Index-driven similarity search in metric spaces. ACM Transactions Database Systems 28(4), 517–580 (2003)
9. Sadit Tellez, E., Chavez, E.: On locality sensitive hashing in metric spaces. In: Proc. 3rd Intl. Conf. on Similarity Search and Applications (SISAP'10). pp. 67–74. ACM Press (2010)
10. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann (2006)
11. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search – The Metric Space Approach, Advances in Database System, vol. 32. Springer (2006)