ISSN 2007-9737

Permutation Based Algorithm Improved by Classes for Similarity Searching

Karina Figueroa[1], Antonio Camarena-Ibarrola[2], Luis Valero[1]

[1] Universidad Michoacana de San Nicolás de Hidalgo, Facultad de Ciencias Físico Matemáticas, Mexico [2] Universidad Michoacana de San Nicolás de Hidalgo, Facultad de Ingeniería Eléctrica, Mexico

karina.figueroa@umich.mx

Abstract. Similarity searching is the most important task in multimedia databases, It consists in retrieving the most similar elements to a given query from a database, knowing that an element identical to the query would not be found. Dissimilarity between objects is measured with a distance function (usually expensive to compute), this allows approaching this problem with a metric space. Many algorithms have been designed to address this problem, in particular, the Permutation Based index has shown an unbeatable performance. This technique uses reference objects to determine a string for each element in the database that is a permutation of the same string. However, Huge databases and the memory required for these indexes make this problem a real challenge. In this paper, we present an improvement to the first approach where classes of reference objects were used instead of single references. In this paper, a new way to choose these classes is proposed and a new way to evaluate similarity between permutations. Our experiments show that we can avoid distance evaluations up to 90% with respect to the original technique, and up to 80% to the first approach.

Keywords. Similarity searching, metric spaces, pattern recognition, nearest neighbor.

1 Introduction

Similarity searching consists in retrieving the most similar objects to a given query from a database. This task has become essential in different areas such as, pattern recognition, artificial intelligence, etc. Actually, it can be applied to any field with a set of objects and a similarity measure between any two objects of the set is defined.

Dissimilarity is a measure of how different two objects are and is preferably computed with a distance function. This measure is normally defined by an expert in the specific application domain and can be used as a black box. A distance function is frequently very expensive to compute, therefore our goal is to reduce the number of distance computations needed to solve each query.

Many important databases are huge and lack of structure. Perfect examples are the multimedia databases. These are challenging and should not be handled with traditional database manipulation systems but rather with metric-space indices. A metric space is defined by a collection of objects and a distance function, we describe it in Section 2.1. An important challenge is to keep the index in main memory, thus, as the size of the database increases more efficient techniques in terms of memory space are needed.

In general, the complete process is split in two parts: building an index, which is an offline process, and querying the index. The performance of a proximity index depends on the intrinsic dimensionality (IDim) of the data; in practice, when the IDim is too high the index performance might collapse [8, 19]. In this paper, we are proposing a novel improvement to one of the best algorithms in high dimension, with a very small index, that can be kept in main memory.

This paper is organized as follows. Section 2 gives details about the related work. Section 3 describes our proposal, some definitions about permutants, classes, etc.; its experimental evaluation is shown in Section 4. Section 5 provides some conclusions and future work. An early version of this work appeared in [13]. The main changes are We propose to use a recent metric between permutations, a new way for selecting the elements per class, and new distances from elements to classes..

2 Related Work

2.1 Basic Concepts

Let $\mathbb{X}$ be the universe of objects and d a distance function ($d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+ \cup \{0\}$). Function d is usually expensive to compute (think, for instance, in computing the distance between two images), d must satisfy these properties: reflexivity, $d(x, x) = 0$; strict positiveness, $x \neq y \Rightarrow d(x, y) > 0$; symmetry, $d(x, y) = d(y, x)$; and the triangle inequality, $d(x, z) \leq d(x, y) + d(y, z)$.

Formally, a metric space is a pair $(\mathbb{X}, d)$. The actual database is a finite set $\mathbb{U} \subseteq \mathbb{X}$ of size $n = |\mathbb{U}|$.

Basically, there are two kind of queries: *Range queries* and *K-Nearest Neighbor queries*. A *Range query* $R(q, r)$ retrieves those objects within a region centered on a given query object q; formally, $R(q, r) = \{u \in \mathbb{X}, d(u, q) \leq r\}$. A *K-Nearest Neighbor* query $NN_K(q)$ retrieves the K elements of $\mathbb{U}$ that are closest to q, that is, $NN_K(q)$ is a set such that for all $x \in NN_K(q)$ and $y \in \mathbb{U} \setminus NN_K(q)$, $d(q, x) \leq d(q, y)$, and $|NN_K(q)| = K$. To solve these queries, we resort to the use of indices.

2.2 Metric Space Indices

A complete survey of metric space searching can be seen in [8, 23, 21]. Metric spaces indices are classified in three categories: *pivot-based* indices, *partition-based* indices, and *permutation-based* indices.

A *pivot-based* index (PiBI) chooses a small set of elements called *pivots*, and every pivot computes and stores all the distances to the rest of the elements, these computed distances are stored as an index. There are several proposals about storing these distances in a data structure using PiBI ([2, 8, 15, 19] to mention a few).

At querying time, first query q is compared to every pivot, and using the triangle inequality property it is possible to approximate the distance between the query q and the rest the elements in the database. Those which are too far from the query can be safely discarded. However PiBIs work well only in low dimensional spaces.

The second family is a *partition-based* index (PaBI), which splits the space using some reference objects, called *centers*. Centers define partitions (by several criteria: closer ones, whithin a radio, etc), and the set of objects is partitioned. At query time, subsets that do not intersect with the query are discarded [17, 23, 7, 16, 9]. PaBIs work reasonable well in high dimension, but usually need $O(n^2)$ distance s to compute the index, however in [6] authors showed that PaBIs were defeated by Permutation-Based Indices.

2.2.1 Permutation-Based Indices

In [5, 6, 1], the authors introduced the *permutation-based* index (PeBI) as follows: Let $\mathbb{P} = \{p_1, p_2, \ldots, p_k\}$ be a subset of objects from $\mathbb{U}$, which are called *permutants*. Each element u of the database computes and stores a *permutation* Π_u of $\{1, \ldots, k\}$ which contains all the permutants in increasing order of distance to u. Formally, for $1 \leq i < k$, $d(p_{\Pi_u(i)}, u) \leq d(p_{\Pi_u(i+1)}, u)$, where $\Pi_u(i)$ means a permutant in position i.

Ties are broken using any consistent order. The hypothesis is *Similar objects are expected to have similar permutations*. To find relevant objects to a given query, permutations similar to the query permutation should be reviewed. The new problem is to find these similar permutations.

There are many similarity measures between permutations and in [6] the authors showed the performance of some. The simplest one (and with a competitive performance) was *Spearman Footrule* (S_f) similarity, defined as:

$$S_f(\Pi_q, \Pi_u) = \sum_{1 \leq i \leq k} \left| \Pi_u^{-1}(i) - \Pi_q^{-1}(i) \right|. \quad (1)$$

where $\Pi^{-1}(i)$ represents the position of the i-th permutant in the permutation.

For example, let $\Pi_q = [p_1, p_2, p_3, p_4, p_5]$ be the query's permutation and let $\Pi_u = [p_2, p_5, p_3, p_1, p_4]$

be the permutation of an element of the database; in this example $\Pi_u^{-1}(5)$ is 2, because the permutant p_5 is at position 2 in Π_u.

Note that permutant p_1 in Π_q is 3 positions away with respect to its position in Π_u. As the absolute values of position differences for each permutant are 3,1,0,1, 3, then in this example, $S_f(\Pi_q, \Pi_u) = 8$.

PeBIs have an excellent performance in high dimension and large permutations work better than short ones. Authors in [10, 20] did no use all but just a few permutants (the prefix of the closest ones to the object for each permutation). The prefixes are structured in an index in RAM. This way the authors achieved better compression, saving the space used by the index at the expense of loosing precision in the retrieval stage [11, 18]. There is another kind of PeBI using *nSimplex* projected vectors, however, not all databases can be processed with this technique [22].

An exact answer for a similarity query retrieves all the objects that satisfy it. As the IDim of the dataset increases the cost of computing the answer exponentially grows, phenomenon known as *the curse of dimensionality*. In some cases we can trade precision in the results for computing time. This is known as approximate retrieval, and is very useful in metric spaces of high IDim.

PiBIs work well for exact retrieval in low IDim, while PaBIs perform reasonable well for exact and approximated retrieval in high IDim. Finally, PeBIs work very well in high IDim for approximated retrieval.

On the other hand, in [14] the authors introduced another metric to measure similarity between permutations. The authors considered penalizing harder when in two permutations one specific permutant appears in different positions. This metric takes advantage of the fact that at solving queries near positions of a specific permutant in two permutations gives you more valuable information than the oposite. The metric proposed in [14] was defined as follows: ϕ_i, $1 \leq i \leq m$, for some Π_u and Π_q, as follows:

$$\phi_i = |\Pi_u^{-1}(i) - \Pi_q^{-1}(i)|, \qquad 1 \leq i \leq m. \quad (2)$$

and the new metric is:

$$T_\alpha(\Pi_u, \Pi_q) = \sum_{i \in k} \Phi_i. \quad (3)$$

where

$$\Phi_i = \begin{cases} \phi_i^\alpha & : \phi_i \geq \mu, \\ \phi_i & : \phi_i < \mu. \end{cases} \quad (4)$$

μ is a parameter that depends on the dimension of the space. Also, reagrding α, the authors claimed that with $\alpha = \log_{10}(\phi)$, they achieved a competitive performance.

3 Our Proposal

The main problem of PeBIs, causing loss of accuracy in retrieval stage, is depicted in Fig. 1.

Notice that query q has a different permutation than that of its nearest neighbor u_1. This is because they are both near the middle between permutants p_1 and p_2. On the other hand, even though u_2 is far from q, since they are both closer to p_1, they have the same permutation:

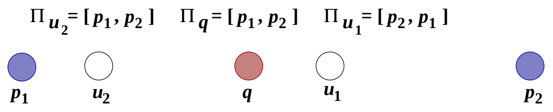


Fig. 1. The Main problem of the PeBI. In the example shown here u_2 is assumed to be q's closest object since they have the same permutation According to Eq. (1) . However, the nearest nearest neighbor of q is actually u_1

In this paper, we extend [13], which is a novel way to reduce the permutation size as compared to that needed for standard PeBI, without reducing the retrieval precision. Our proposal consists in having permutations of *classes of permutants* ($\Pi^{\mathbb{G}}$) instead of *isolated permutants*.

Formally, let $\mathbb{G} = \{G_1, G_2, \ldots, G_k\}$ be a partition of $\mathbb{P}$, where each class has exactly m elements, and $\mathbb{P} = G_1 \cup G_2 \ldots \cup G_k$. Also, let $D : \mathbb{U} \times \mathbb{G} \rightarrow \mathbb{R}$ be the function that computes the distance between an element $u \in \mathbb{U}$ of the database and a class $G_i \in \mathbb{G}$.

We compute $D(u, G_i), \forall\, u \in \mathbb{U}, i \in [1, k]$, and sort these distances by proximity to u.

In the next sections, we discuss criteria for selecting elements for each class and computing D.

Algorithm 1 C1e

```
Require: $\mathbb{U}$, $m$ , $k$
Ensure: $I$
1: for $i := 1$ to $k$ do
2:     /* select each first element of each class */
3:     $G_i[1] \leftarrow Random(\mathbb{U})$
4:     $\mathbb{U} \leftarrow \mathbb{U} - G_i[1]$
5: end for
6: for $i := 1$ to $k$ do
7:     let $NN_{m-1}(G_i[1])$ the closest elements to $G_i[1]$
8:     $G_i \leftarrow NN_{m-1}(G_i[1])$
9: end for
10: Return $I$
```

Algorithm 2 C2e

```
Require: $\mathbb{U}$, $m$ , $k$
Ensure: $I$
1: for $i := 1$ to $k$ do
2:     /* select each first element of each class */
3:     $v \leftarrow Random(\mathbb{U})$
4:     $w \leftarrow NN_1(v)$ /* except $v$*/
5:     $G_i \leftarrow \{v, w\}$
6:     $\mathbb{U} \leftarrow \mathbb{U} - \{v, w\}$
7: end for
8: for $i := 1$ to $k$ do
9:     let $v, w \in G_i$
10:    $G_i \leftarrow$ the closest $m - 2$ that $\min_{\forall u \in \mathbb{U}} (d(v, u) + d(w, u))$
11: end for
12: Return $I$
```

3.1 Criteria to Select Groups of Permutants

In order to explore this technique, we use several criteria to form classes of permutants:

1. *Rand* This criterion consists of choosing each class randomly.

2. *C1e* selects the first element of each class randomly and adds its $m - 1$ closest permutants to the class. See Algorithm 1.

3. *F1e* selects the first element of each class randomly and adds its $m - 1$ farthest permutants to the class.

Algorithm 3 SSS

```
Require: $\mathbb{U}$, $m$ , $k$
Ensure: $I$
1: Let $N_G \leftarrow 1$
2: Let $M$ the maximum distance between each par in $\mathbb{U}$
3: Let $G_{N_G}[1] \leftarrow Random(\mathbb{U})$
4: Let $\mathbb{U} \leftarrow \mathbb{U} - G_{N_G}[1]$
5: for all $u \in \mathbb{U}$ and $N_G < k$ do
6:     for all $v \in \mathbb{G}$ and $N_G < k$ do
7:         if $d(v, u) < 0.4 \times M$ then
8:             $N_G := N_G + 1$
9:             $G_{N_G} \leftarrow u$
10:            $\mathbb{U} \leftarrow \mathbb{U} - u$
11:        end if
12:    end for
13: end for
14: for $i := 1$ to $k$ do
15:    $G_i \leftarrow NN_{m-1}(G_i[1]))$
16: end for
17: Return $I$
```

4. *C2e* selects, for each class, a pair of mutual nearest neighbors and the $m - 2$ permutants that minimize the sum of distances to the previous ones in the class. See Algorithm 2.

5. *SSS* selects k head of classes following the method in [4] to chose objects scattered in the space.

   Let M be the maximum distance between all object pairs. According to the authors, the heads of classes are at least at a distance of $0.4 * M$ between them. Classes are completed with the closest element to each head of class. See Algorithm 3.

3.2 Distance to a Class

Since we have m elements in each class, we can define several criteria to compute the distance to a class. We consider these four options:

1. D_{min} is the lowest distance to all the objects in the class. Formally, $D_{min}(u, G_i) = min_{p \in G_i} d(p, u)$.

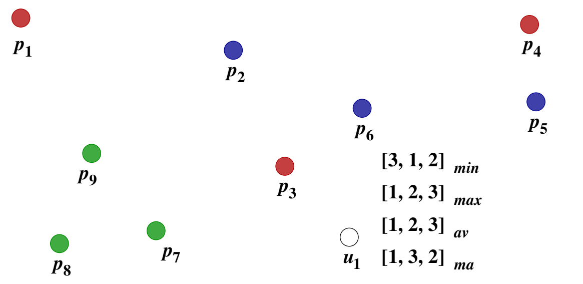


Fig. 2. Classes 1, 2 and 3 in blue, green and red respectively. We show the permutations by classes according to the four options of distances. That is, using D_{min}, p_3 of G_3 is closest that anyone, so, according to D_{min} the closest class is 3, then 1, and 2

2. D_{max} is the greatest distance to all the objects in the class. Formally, $D_{max}(u, G_i) = max_{p \in G_i} d(p, u)$.

3. D_{av} is the average of all the distances to objects in the class. Formally, $D_{av}(u, G_i) = \sum_{p \in G_i} d(p, u)/m$.

4. D_{am} is the sum of distances D_{av} and D_{min}. This is $D_{am}(u, G_i) = D_{av}(u, G_i) + D_{min}(u, G_i)$.

In order to illustrate our ideas, in Fig. 2 all classes were selected in random way. For object u_1, $\Pi_{u_1} = [3, 6, 7, 2, 9, 5, 8, 4, 1]$, while the permutations formed by classes are $\Pi^{\mathbb{G}}_{u_1} = [3, 1, 2]_{min}$ according to D_{min}; $\Pi^{\mathbb{G}}_{u_1} = [1, 2, 3]_{max}$ due to D_{max}; $\Pi^{\mathbb{G}}_{u_1} = [1, 2, 3]_{av}$ to D_{av}; and $\Pi^{\mathbb{G}}_{u_1} = [1, 3, 2]_{am}$ according to D_{am}.

Our proposal is to use a new metric for evaluating how similar classes of permutations are instead of using Spearman's metric.

4 Experimental Results

We ran the experiments in synthetic and real world datasets. Synthetic ones allow us to assess the strength of our technique while varying some parameters, such as dimensionality, dataset size, number of permutants and classes, distance to classes and the way to conform the classes.

On the other hand, real world datasets show the performance of our technique in practical situations.

The performance of our proposed technique is measured in terms of distance computations.

4.1 Synthetic Databases

Our proposal was tested using a synthetic database composed by vectors uniformly distributed in the unitary cube. Our dataset consists of 100,000 points in $\mathbb{R}^d$ with $d \in [16, 128]$, using Euclidean distance to measure how far they are to each other.

In Figure 3 the performance of our technique as the dimension increases is showed. As we expected, the number of distances rises as the dimension increases. Notice that the PeBI idea has a good performance when we use just 16 permutants, however, for $k = 64$ the idea of classes is better than PeBI, in particular, when the space is uniform we can use the Rand criterion and distance D_{av}.

It is important to notice that we have two phases of distance computations: when a query q is given, we compute $d(q, p)$ where $p \in \mathbb{P}$ (i.e. internal distances) in order to get the permutation of the query; the second phase is when we have a promissory order to compare against the query (i.e. external distances).

The internal distances are unavoidable, and external distances allow us to answer the queries quickly. All images reported are the total distances (internal + external distances).

4.2 Real Databases

In this section we show the performance of our similarity searching method in real-world metric spaces using the benchmark set for similarity searching community [12].

In this case, we have a non-uniform database, we can use the technique proposed in this paper: D_{am}, and *SSS* technique, of course we are comparing against the best previous works that used this database (D_{av} and *Random* selection).

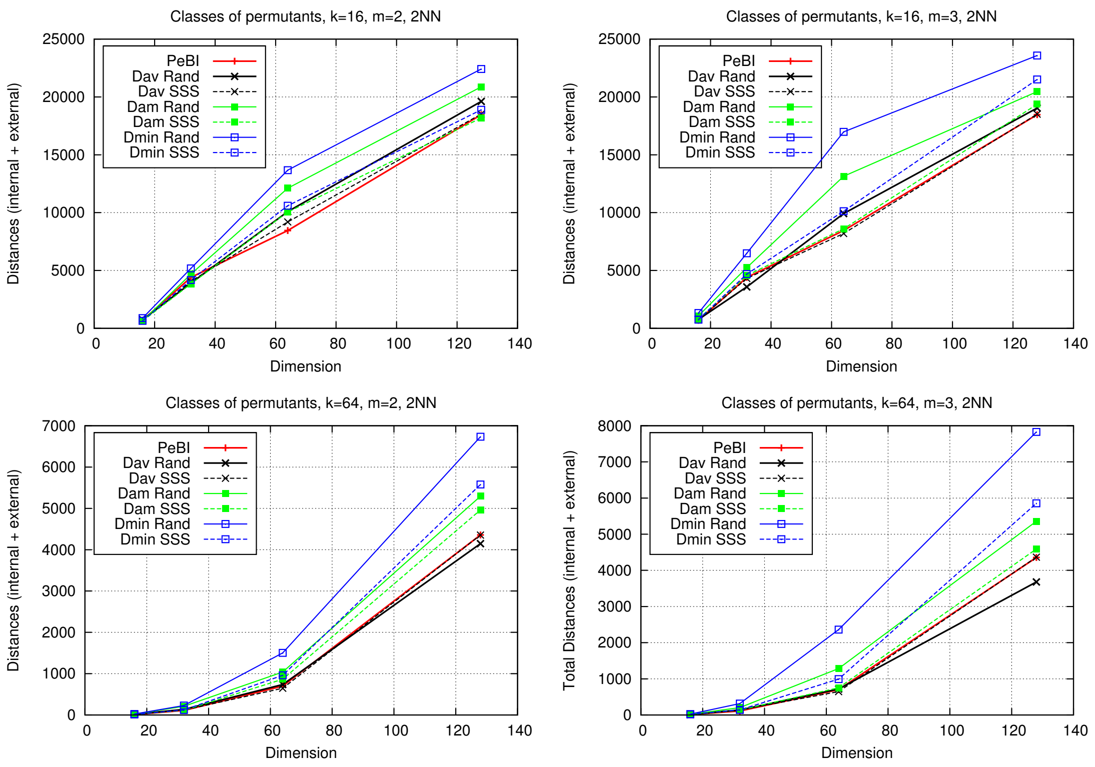


Fig. 3. Computed distances for solving $2NN$ queries for the synthetic database of uniformly distributed vectors in the unitary cube as the dimension increases to show the dependence on dimensionality for permutations. The figures at the top use 16 permutants, the figures at the bottom use 64 permutants, parameter $m = 2$ for the figures at the left and $m = 3$ for the figures at the right

4.2.1 NASA Images

This dataset consists of 40,150 feature vectors in $\mathbb{R}^{20}$. These 20-dimensional vectors were generated from images downloaded from NASA[1], duplicate vectors were eliminated.

We used the Euclidean distance to compare the feature vectors of this collection of images. We chose 500 histograms randomly as test set, for querying, and the rest as our database to be indexed.

In Figure 4, we are using two different size of classes $k = 16, 32$. Notice that as we are looking for more nearest neighbors, our proposal is working better than PeBI technique (red line). However, in this database with $k = 16$ top image the best performance is achieved with D_{am}, *SSS* $m = 3$, and another metric between permutations, with 3 elements per class ($\mu \geq 8$). But, when $k = 32$ (bottom image), the best performance is with $m = 2$, D_{am}, SSS.

[1] http://www.dimacs.rutgers.edu/Challenges/Sixth/software.-html

4.2.2 Colors

This database consists of 112,682 color histograms, represented as 112-dimensional feature vectors. This dataset was obtained from the SISAP project's metric space benchmark set [12]. We chose 500 histograms randomly as test set, for querying, and the rest as our database to be indexed.

In Figure 5 we show that our proposals have excellent results.

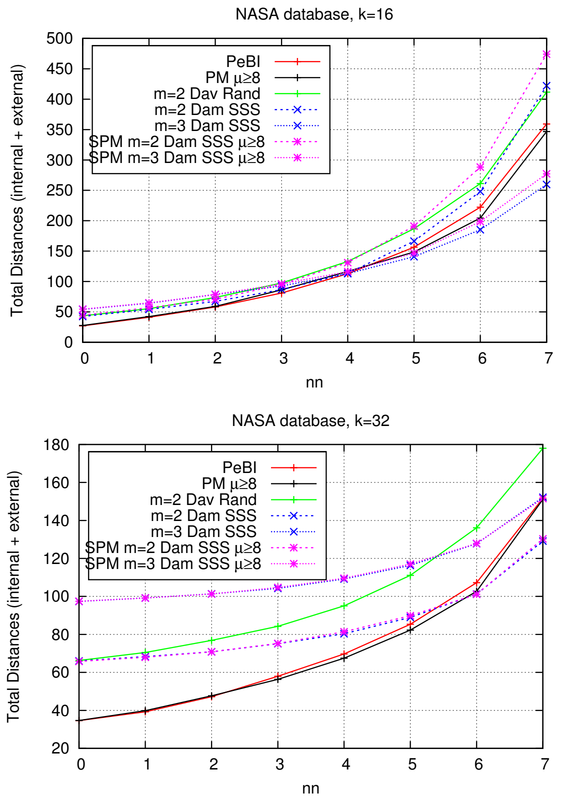


Fig. 4. Total distances computed for finding nearest neighbors as the number of nearest neighbors increase for the NASA database. The number of permutans is $k = 16$ for the figure at the top, and $k = 32$ for the figure at the bottom

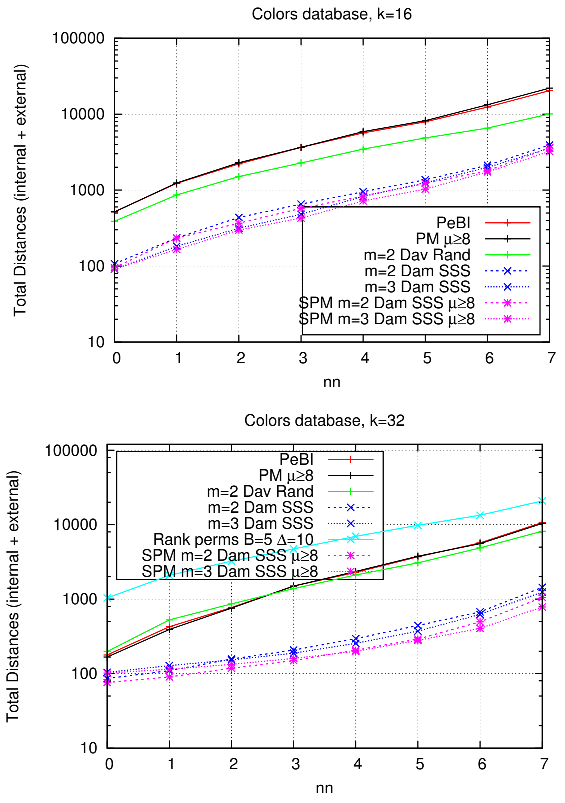


Fig. 5. Total distances computed for finding nearest neighbors as the number of nearest neighbors increase for the COLORS database. The number of permutans is $k = 16$ for the figure at the top, and $k = 32$ for the figure at the bottom

Notice that PeBIs technique (red line, using k permutants, that is $m = 1$) has the same performance of the PeBIs with the new metric (PM $\mu \geq 8$, black line, that is $m = 1$), and the proposal of permutations of classes is almost 90% less work.

However, our new proposal is better, in particular notice that using D_{am} distance and *SSS* has the best performance specially when we use 3 permutants per class and the new metric is used $\mu \geq 8$ (label with SPM), see top image.

Finally, using $k = 32$ (bottom image) the new metric with $m = 3$ (with $\mu \geq 8$) has the best performance. That is, in this kind of databases, using the new metric is helpful.

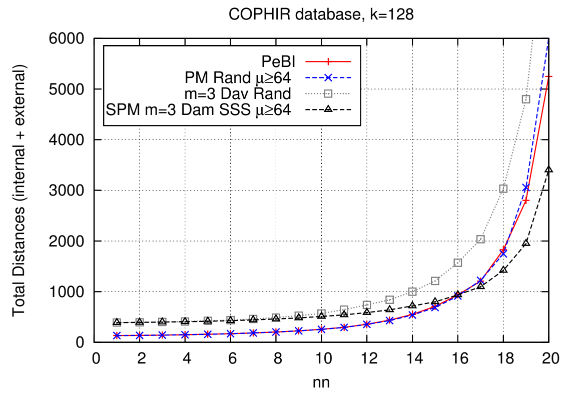
4.2.3 CoPhIR

This database consists of 10 million objects selected from the CoPhIR project [3]. Each object is a 208-dimensional vector and we use the L_1 distance.

Each vector was created in a linear combination of five different MPEG7 vectors as described in [3]. We chose the first 500 vectors from the database as queries. Each query consisted of searching for twenty nearest neighbors.

In Figure 6 we show two images, the top plot considers all computed distances (internal + external, where internal is $k * m$), while Figure in bottom shows the performance our new proposal. Notice we can get the nearest neighbor using up to

ISSN 2007-9737



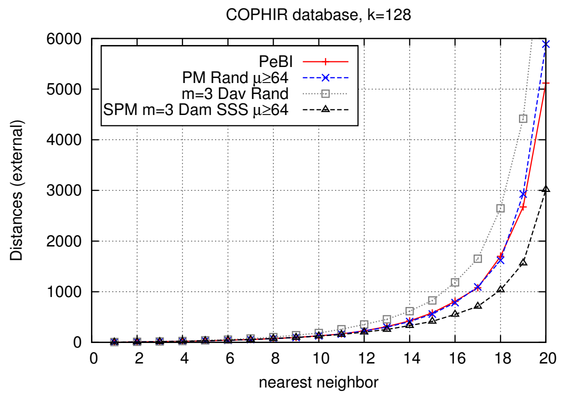


Fig. 6. Total distances computed for finding nearest neighbors as the number of nearest neighbors increase for the CoPhIR database using $k = 128$ permutants, at the top the sum of internal and external distances, at the bottom just the externals

42% less work than the original technique (PeBI). That is, using classes of permutants instead single ones, also, using the metric proposed in [14] and the criterion SSS to select elements in each class, and using D_{am} distance to define which class is nearest to an element.

5 Conclusion and Future Work

In this paper, we presented an extended version of [13]. The original idea of to use classes of permutants instead of single ones in the permutation based algorithm [6]. In this paper, we introduced a new distance between classes and experimented with different ways of selecting elements inside classes.

Our experimental results showed that we can improve the permutation based algorithm up to 90% in real databases.

As a future work, we would like to test how this technique works using different size of classes. Also, another interesting idea is to mix different criteria to choose a class, in this paper, all classes are selected using exactly the same criterion.

Acknowledgments

We thank Universidad Michoacana de San Nicolás de Hidalgo for funding this work through its research department *Coordinación de la Investigación Científica*.

References

1. **Amato, G., Savino, P. (2008).** Approximate similarity search in metric spaces using inverted files. Infoscale, pp. 28.
2. **Baeza-Yates, R., Cunto, W., Manber, U., Wu, S. (1994).** Proximity matching using fixed-queries trees. CPM, pp. 198–212.
3. **Bolettieri, P., Esuli, A., Falchi, F., Lucchese, C., Perego, R., Piccioli, T., Rabitti, F. (2009).** CoPhIR: a test collection for content-based image retrieval. CoRR, Vol. abs/0905.4627v2.
4. **Brisaboa, N. R., Fariña, A., Pedreira, O., Reyes, N. (2006).** Similarity search using sparse pivots for efficient multimedia information retrieval. Proc. 8th IEEE International Symposium on Multimedia (ISM'06), pp. 881–888.
5. **Chávez, E., Figueroa, K., Navarro, G. (2005).** Proximity searching in high dimensional spaces with a proximity preserving order. MICAI 2005: Advances in Artificial Intelligence, volume 3789 of Lecture Notes in Computer Science, pp. 405–414.
6. **Chávez, E., Figueroa, K., Navarro, G. (2008).** Effective proximity retrieval by ordering permutations. IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI), Vol. 30, No. 9, pp. 1647–1658.
7. **Chávez, E., Navarro, G. (2005).** A compact space decomposition for effective metric indexing. Pattern Recognit. Lett., Vol. 26, pp. 1363–1376.

8. **Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J. (2001).** Searching in metric spaces. ACM Computing Surveys, Vol. 33, No. 3, pp. 273–321.

9. **Das, A. K., Bhuyan, P. K. (2019).** Self-organizing tree algorithm (sota) clustering for defining level of service (los) criteria of urban streets. Periodica Polytechnica Transportation Engineering,, Vol. 47, No. 4, pp. 309–317.

10. **Esuli, A. (2009).** MiPai: using the PP-Index to build an efficient and scalable similarity search system. Proc. 2nd Intl. Wksp. on Similary Search and Applications (SISAP'09), IEEE Computer Society, pp. 146–148.

11. **Esuli, A. (2012).** Use of permutation prefixes for efficient and scalable approximate similarity search. Information Processing & Management, Vol. 48, No. 5, pp. 889–902. DOI: https://doi.org/10.1016/j.ipm.2010.11.011. Large-Scale and Distributed Systems for Information Retrieval.

12. **Figueroa, K., Navarro, G., Chávez, E. (2007).** Metric spaces library. Available at `http://www.-sisap.org/Metric_Space_Library.html`.

13. **Figueroa, K., Paredes, R., Rangel, R. (2011).** Efficient group of permutants for proximity searching. Proc. 3rd Mexican Conf. on Pattern Recognition (MCPR'11), LNCS 6718, Springer, pp. 42–49.

14. **Figueroa, K., Paredes, R., Reyes, N. (2018).** New permutation dissimilarity measures for proximity searching. LNCS, volume 11223, pp. 122–133.

15. **Hjaltason, G., Samet, H. (2003).** Index-driven similarity search in metric spaces. ACM Transactions Database Systems, Vol. 28, No. 4, pp. 517–580.

16. **Kalantari, I., McDonald, G. (1983).** A data structure and an algorithm for the nearest point problem. IEEE Transactions on Software Engineering, Vol. SE-9, pp. 631–634.

17. **Mamede, M. (2005).** Recursive lists of clusters: A dynamic data structure for range queries in metric spaces. LNCS, volume 3733, pp. 843–853.

18. **Mohamed, H., Marchand-Maillet, S. (2015).** Quantized ranking for permutation-based indexing. Information Systems, Vol. 52, pp. 163–175. DOI: https://doi.org/10.1016/j.is.2015.01.009. Special Issue on Selected Papers from SISAP 2013.

19. **Navarro, G., Paredes, R., Reyes, N., Bustos, C. (2017).** An empirical evaluation of intrinsic dimension estimators. Information Systems, Vol. 64, pp. 206–218.

20. **Novak, D., Zezula, P. (2016).** PPP-codes for large-scale similarity searching. TLDKS XXIV, pp. 61–87.

21. **Samet, H. (2006).** Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann.

22. **Vadicamo, L., Connor, R., Falchi, F., Gennaro, C., Rabitti, F. (2019).** Splx-perm: A novel permutation-based representation for approximate metric search. **LNCS**, editor, Similarity Search and Applications, volume 11807.

23. **Zezula, P., Amato, G., Dohnal, V., Batko, M. (2006).** Similarity Search: The Metric Space Approach, volume 32 of Advances in Database Systems. Springer.

Article received on 25/07/2021; accepted on 16/09/2021.
Corresponding author is Antonio Camarena-Ibarrola.