List of Clustered Permutations for Proximity Searching ⋆

Karina Figueroa[1] and Rodrigo Paredes[2]

[1] Facultad de Ciencias Físico-Matemáticas,Universidad Michoacana, Mexico.
[2] Departamento de Ciencias de la Computación, Universidad de Talca, Chile.
karina@fismat.umich.mx, raparede@utalca.cl

Abstract. The *permutation based algorithm* has been proved unbeatable in high dimensional spaces, requiring $O(|\mathbb{P}|)$ distance evaluations when solving similarity queries (where $\mathbb{P}$ is the set of permutants); but needs n evaluations of the permutant distance to compute the order to review the metric dataset, requires $O(n|\mathbb{P}|)$ space, and does not take much benefit from low dimensionality. There have been several proposals to avoid the n computations of the permutant distance, however all of them lost precision. Inspired in the *list of cluster*, in this paper we group the permutations and establish a criterion to discard whole clusters according the permutation of their centers. As a consequence of our proposal, we now reduce not only the space of the index and the number of distance evaluations but also the cpu time required when comparing the permutations themselves. Also, we can use the permutations in low dimensions.

1 Introduction

Several modern applications —for instance, pattern recognition or multimedia retrieval— require similarity retrieval systems to find relevant objects when solving a query. In these applications the pattern is the same, the search problem is often stated in terms of expensive comparison between two objects in a huge database.

The problem can be mapped into a metric space $(\mathbb{X}, d)$, where a metric d compares objects out of a universe $\mathbb{X}$ and reveals how close is an object with respect to other. This metric must satisfy the follow properties: positiveness $d(x, y) \geq 0$, symmetry $d(x, y) = d(y, x)$ and triangle inequality $d(x, y) \leq d(x, z) + d(z, y)$. Given a dataset $\mathbb{U} \subset \mathbb{X}$, this kind of queries can be classify basically in two: range and k-nearest neighbor queries. The first one consists in retrieving those objects out of $\mathbb{U}$ within a radius to a given query, that is, $R(q, r) = \{d(u, q) \leq r, \forall\, u \in \mathbb{U}\}$; the second one is to retrieve the k elements of $\mathbb{U}$ that are closest to q.

In general metric spaces, the (black-box) distance function is the only way to distinguish between objects, and usually, the distance function is expensive

⋆ This work is partially funded by National Council of Science and Technology (CONACyT) of México, Universidad Michoacana de San Nicolás de Hidalgo, México, and Fondecyt grant 1131044, Chile.

to compute (e.g., consider comparing two multimedia objects). Hence the complexity is absorbed by the distances evaluated.

Since this kind of datasets lack of total order, to avoid a full linear scan, a dataset preprocessing, consisting in building an index structure that allows to get the answer with less effort, is common. In this respect, the *list of cluster* [5] is one of the most efficient algorithms in high dimensional spaces, however it takes $O(n^2)$ distances computation to make the index.

In other hand, the *permutation based algorithm* [3] has been proved unbeatable in practice but only works well in high dimensions, as the authors claim. To use this index, we have to compute the permutation of the query and compare it with all the dataset permutations so as to compute the order to review the permutations. This takes at least $O(|\mathbb{P}|)$ distances computations ($|\mathbb{P}|$ is the size of the permutations) and $O(n)$ evaluations of the permutation distance. There have been several proposal to avoid the sequential scanning in the permutation based algorithm, however all of them lost precision with respect to the original technique [7, 10].

In this article we combine the ideas of the list of cluster and the permutation based index and present a new metric index that improves on both of them. The rest of this paper is organized as follows. In Section 2 we introduce some basic concepts. Next, in Section 3 we describe the *List of Clustered Permutations* and in Section 4 we show the experimental evaluation of our technique. Finally, we draw our conclusions and future work directions in Section 5.

2 Previous work

One can approach the similarity search problem in either an exact or approximated way. In the first case, we want to retrieve all the objects satisfying the similarity query. The main families of this kind of algorithms are the pivot based indices and the ones based on compact partitions [6, 1]. In the second, the idea is to retrieve most of the relevant elements that fulfill the similarity query. In this case, we accept to miss some relevant elements for the sake of speed up the query solving. There are already some non-exact approaches [4, 3, 12, 9].

In this paper, we combine the list of clusters with the permutation based algorithm. Hence, in the next sections we describe both indices.

2.1 List of Clusters

There are many indices in metric spaces [6, 1]. One of the most economical and rather efficient is the *list of clusters* (LC) [5], because it uses $O(n)$ space and has an excellent performance in high dimension. Regretably, its construction requires $O(n^2)$ distance evaluations, which is very expensive. The LC is built as follows:

Firstly, a center c is selected from the database and a bucket size b is given. c chooses its b-closest elements out of the database and build the set I, which is the answer of a b-nearest neighbour query. Let cr_c be the distance from c to its farthest neighbor in I. The tuple (c,I,cr_c) is called a cluster. (Notice that the

parameter b could be replaced by specifying the global covering radius, but this alternative has worse performance [5].) This process is repeated recursively with the rest of the non-clustered objects.

To solve queries, the query object is compared with all the cluster centers. So, for each cluster, if the distance from its center to the query is larger than the its covering radius plus the query radius we can discard its whole bucket, otherwise we review it exhaustively.

2.2 Permutation based algorithm

In [3], the authors introduce the permutation based algorithm (PBA), a novel technique that shows a different way to sort the space. During the preprocessing time, a subset of objects $\mathbb{P} = \{p_1, p_2, \ldots, p_{|\mathbb{P}|}\} \subset \mathbb{U}$ is selected out of the database, which are called the permutants. Each $u \in \mathbb{U}$, computes its distance to all the permutants (that is, compute $d(u, p)$ for all $p \in \mathbb{P}$) and sort them increasingly by proximity. Then, for each object $u \in \mathbb{U}$, we store just the order of the permutants (not the distances) in the index.

If we define Π_u as the permutation of $(1 \ldots |\mathbb{P}|)$ for object u, so $\Pi_u(i)$ is the i-th cell in the u's permutation and $p_{\Pi_u(i)}$ denotes the i-th permutant. For instance, if $\Pi_u = (5, 1, 2, 4, 3)$ then $p_{\Pi_u(3)} = p_2$. Within the permutation, for all $1 \leq i < |\mathbb{P}|$ it holds either $d(p_{\Pi_u(i)}, u) < d(p_{\Pi_u(i+1)}, u)$ or, if there is a tie $(d(p_{\Pi_u(i)}, u) = d(p_{\Pi_u(i+1)}, u))$, then the permutant with the lowest index appears first in Π_u. We call the i-th permutant $\Pi_u(i)$, the *inverse permutation* Π_u^{-1}, and the position of i-th permutant $\Pi_u^{-1}(p_i)$. The set of all the permutations stored in the index needs $O(n|\mathbb{P}|)$ memory cells.

At query time, we compute the distance from the query q to all the permutants in $\mathbb{P}$ and calculate the query permutation Π_q. Next, Π_q is compared with all the permutations stored in the index, that is $O(n)$ permutation distances. In [11], authors introduce how to index the *permutations' space* as a new metric space, however they do not mix both kind of distances and they use a bigger index. Authors in [3] claim that the order induced by Π_q is extremely promising and a reviewing a small fraction of the dataset is enough to get a good answer.

The permutation distance is calculated as follows: let Π_u and Π_q permutations of $(1 \ldots |\mathbb{P}|)$. We compute how different is a permutation from the other using Spearman Rho S_ρ metric. In [8], S_ρ is defined as:

$$S_\rho(\Pi_u, \Pi_q) \;=\; \sqrt{\sum_{1 \leq i \leq |\mathbb{P}|} \left(\Pi_u^{-1}(i) - \Pi_q^{-1}(i)\right)^2} \qquad (1)$$

Since S_ρ is monotonic we omit the square root as it preserves the ordering.

The main disadvantage of the PBA is that its memory requirement could be prohibitive in some scenarios, especially where n is huge. Also, like other indices, the dimension of the space has an impact on the index performance; in particular, it has an effect on how long is the fraction to consider when solving the approximated query.

3 List of Clustered Permutations

The simplest way to reduce the time consumed when building a list of clusters is to avoid distance computations. For this sake, we have two possibilities: a bigger bucket size, or using another, cheaper, way to construct the structure. Follow the second possibility, we propose to combine the PBA with the LC. We choose a set of permutants, where each one within this set has a double role, as permutant and as a cluster center; and only the cluster centers store their permutation. We call this structure the *List of Clustered Permutations (LCP)*.

When solving a proximity query q with the standard PBA, we need to spend $|\mathbb{P}|$ distance evaluations to compute the query permutation Π_q, plus n evaluations of the permutation distance to compute the order induced by Π_q, and $O(fn)$ distance evaluations to compare q with the fraction f of the dataset objects that are the most promising to be relevant for the query. With the LCP, we spend only $|\mathbb{P}|$ ($< n$) evaluations of the permutation distance to compare Π_q with the permutation of each cluster center, and distances evaluations needed to review non-discarded clusters. In our experiments, we verify that this is an improvement over the traditional LC.

3.1 Building

Firstly, we randomly select a set $\mathbb{P}$ of centers and we compare every object within the database with this set. This way, we compute permutations for all the objects in the dataset. Then, we choose the first center and group its $b = \frac{n}{|\mathbb{P}|} - 1$ most similar objects according to the *permutation distance* (excluding all the cluster centers, so that no center can be inside the bucket of another one). We continue the process iteratively with the rest of elements in the dataset until every element is clustered. Every center keeps its covering radius cr_c (that is, the distance to the farthest object in the bucket), its bucket and its permutation (hence, we discard the permutations of all the objects within a bucket).

The space used is $n + |\mathbb{P}|^2$ cells, and the construction time is $O(n|\mathbb{P}|)$ evaluations of both the space distance and the permutation distance. Note that we can pack the whole LCP index using just $(n + |\mathbb{P}|^2) \log_2 |\mathbb{P}|$ bits.

3.2 Querying

The standard LC discards clusters with the covering radius rule. Let $d(q, c)$ be the distance between the query and the center, r the query radius, and cr_c the covering radius of center c. So, if $d(q, c) > r + cr_c$ the cluster is discarded.

Since we have permutations, we introduce a heuristic method to discard a cluster, modifying the criteria explained in [13]. Our preliminary experimental results shown that if we have an object (for instance, a cluster center), and its permutation has (just) one permutant that moved far away with respect to its position inside query permutation, then this object is not relevant, so we can discard it (and also its bucket). For instance, if the permutation of the query

is (1,2,3,4) and the permutation of the center is (4,1,2,3), even though most of both permutations are similar, the position shifting of permutant 4 suggests that the object can be discarded.

Of course, we need to establish a criterion to measure our finding. Basically, we need to know how much could a permutant move away inside the permutation of an object. So, using the query permutation and the range query radius, we estimate how far a permutant can shift. To do that, for a pair of permutants p_i, p_j, where p_i is closer to the query than p_j, and $d(p_j, q) - d(p_i, q) \leq r$, our method does not discard an object whose permutation has an inversion of these permutants; this is, it does not discard an object that is closer to p_j than to p_i. But, if the distance difference is larger, even though permutant inversion is possible there as a big chance that the object were irrelevant so the object can be discarded. Therefore, we take note of how many slots the permutant moves; this is computed in Algorithm 1.

Algorithm 1 ComputingShift(Q, r)

```
1: Let Q the set of pairs (permutant, distance) to q, sorted by distance
2: permShift ← 0
3: for i ← 0 to |ℙ| − 2 do
4:    cont ← 0, j ← i + 1
5:    while j < |ℙ| AND Q[j].dist − Q[i].dist ≤ r  do
6:       cont ← cont + 1, j ← j + 1
7:    end while
8:    permShift ← max {permShift, cont}
9: end for
10: return  permShift
```

In the query procedure, we discard a cluster center (and its bucket) when a permutant shifts more than tolerated.

4 Experiments

In this section we evaluate and compare the performance of our technique in different metric spaces, such as synthetic vectors on the unitary cube and a real life database. The experiments were run on an Intel Xeon workstation with 2.4 GHz CPU and 32 GB of RAM with Ubuntu server, running kernel 2.6.32-22.

4.1 Synthetic Databases

In these experiments we used a synthetic database with vectors uniformly distributed on the unitary cube. We use 100,000 points in different dimensions under Euclidean distance. As we can precisely control the dimensionality of the space, we use these experiments to show how much the predictive power of our technique varies with the dimensionality.

Since ours is an approximated method, we relax the discarding criteria by accepting bigger shifts and tabulate the results. They are shown in Figures 1, 2, and 3. In this plots, the labels bx000 indicates the size b of the LCP buckets. Since $b = \frac{n}{|\mathbb{P}|} - 1$, bx000 also fixes a value for $|\mathbb{P}|$.

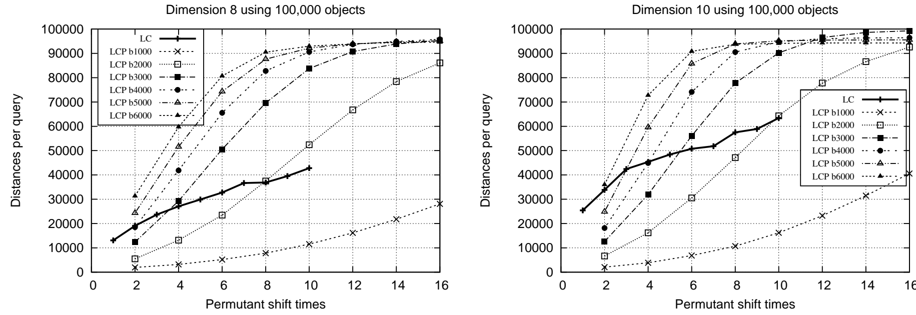

Fig. 1. Unitary cube using 100,000 vectors. (Left) Dimension 8, (right) dimension 10.

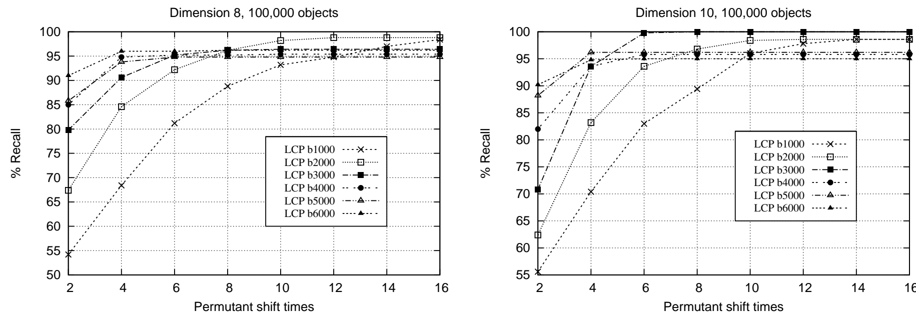

Fig. 2. The plots show the percentage of recall using the distances show in Figure 1.

In Figures 1 and 3, the solid line is the original LC. For this line, the axis x represent the bucket size per thousand (from 1,000 to 10,000). As expected, Figure 1 shows that the smaller the bucket size the better the query results, since it is easier to discard a cluster with any of both criteria (this applies both for LC and LCP). On the other hand, Figure 2 illustrates that as long as the shifting criterion is relaxed, the recall of the method improves; but, it also increases the number of distance evaluations needed to solve the query. In several cases, accepting eight times in the permutant shift is enough to obtain an acceptable recall, saving distance computations and cpu time. Finally, the time computed for our method is lower than the standard LC, as evidenced in Figure 3.

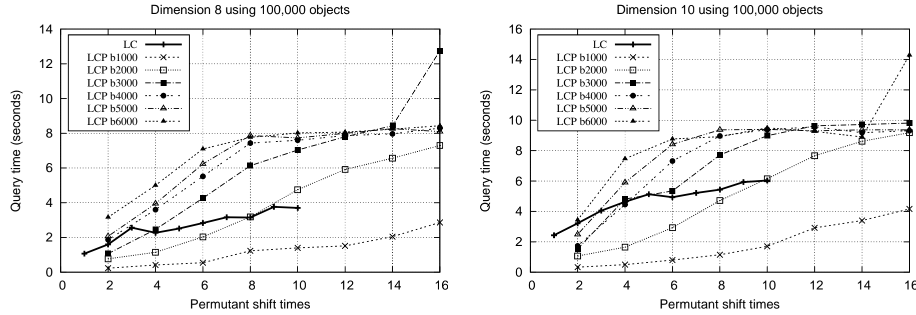

Fig. 3. Time consumed for experiments showed in Figures 1 and 2.

In order to illustrate the performance of our method, in dimension 10, using buckets of 1,000 objects and accepting eight times in the permutation shifting, our method requires 44% of distance evaluations of LC, obtains a 88% of recall and uses 48% of LC cpu time.

Note that the LCP index uses very little space: one identifier for each non-center object and only $|\mathbb{P}|^2$ cells for the permutations of centers. In this case, when using buckets of 1,000 objects (so $|\mathbb{P}| = 100$), this translates approximately to 7.7 bits per object.

Figure 4 compares the LTC with standar PBA. In order to perform a fair comparison, we allow 8 bits for each permutation, that is, four permutants coded in two bits. As can be seen, LTC with buckets of 1,000 objects outperforms by far the recall of standard PBA.

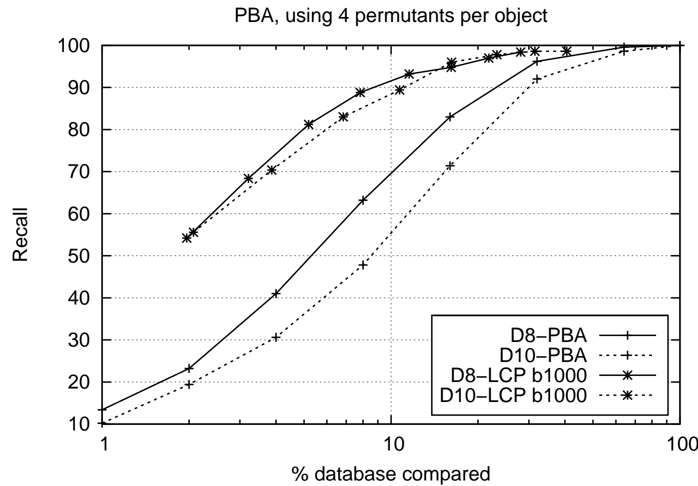

Fig. 4. Comparison between LCP and standard PBA.

4.2 Real Databases

In this section we show the performance of our heuristic in a real-world space of images.

Cophir Database In this section we show the experiments made on a large database. The CoPhIR is Content-based Photo Image Retrieval, with 1,000,000 of images [2] and buckets de 2,000 objects per cluster. For each image, the standard MPEG-7 image feature have been extracted. So, each image is a vector of 208 components.

In Figure 5, the label *List of Cluster* is the original technique retrieving the exact nearest neighbors. It shows that the LC requires to review almost 30% of the images. The label *Recall* is our proposal (it reviews from 1 to 7 % of the database) and the label *Distances* is the distance evaluation used to retrieval that recall. In this space, LCP performance is rather good. For instance, accepting forty times of shifting, we get the best retrieval (94%), reviewing just 7% of the database, in compare with LC that requires almost the 30% of the database.

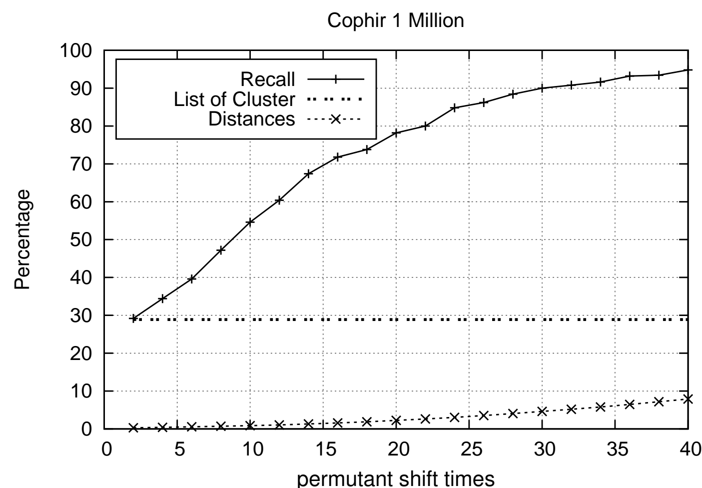

Fig. 5. 1 million of images. The solid line is the recall and the dashed line is the percentage of distance evaluations. The dotted line is the original LC.

5 Contributions and Future Work

Similarity searching is a very important operation in multimedia databases nowadays. It involves finding objects in a dataset similar to a query object q, based on some distance measure d. To do so, it is common to compute an index structure

in order to solve similarity queries efficiently. One of the most successful indices is the permutation based algorithm. In this paper, we present a novel way to index permutations so that we can save space when computing the distance between permutations. The advantage of our proposal is that it is now possible to use the permutations in low dimensions and also we propose a parameter to avoid to sequential scanning in the permutation based algorithm.

As a future work we consider two lines, namely:

1. For the sake of maintain small clusters, we can divide the LCP construction in three phases. In the first, we choose the permutants and compute the permutations for all the objects. In the second, we compute the clusters for the permutants, and finally, we compute the other clusters. This way, we expect to compute the LCP using very few distance computations, but the amount of work computing the permutation distances should increase.
2. Since our method uses very little memory, we want to explore the possibility of using short permutations for objects inside the clusters. This is supported by the facts that the beginning of the permutation is the most important data portion to process and that we can trade space in order to improve the recall results.

References

1. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Computing Surveys 33(3), 322–373 (2001)
2. Bolettieri, P., Esuli, A., Falchi, F., Lucchese, C., Perego, R., Piccioli, T., Rabitti, F.: CoPhIR: a test collection for content-based image retrieval. CoRR abs/0905.4627v2 (2009), http://cophir.isti.cnr.it
3. Chávez, E., Figueroa, K., Navarro, G.: Proximity searching in high dimensional spaces with a proximity preserving order. In: Proc. 4th Mexican Intl. Conf. on Artificial Intelligence (MICAI'05). pp. 405–414. LNAI 3789 (2005)
4. Chávez, E., Navarro, G.: Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. Information Processing Letters 85(1), 39–46 (2003)
5. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters 26(9), 1363–1376 (2005)
6. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Proximity searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (2001)
7. Esuli, A.: Mipai: using the pp-index to build an efficient and scalable similarity search system. In: Proc. 2nd Intl. Workshop on Similary Searching and Applications (SISAP 2009). pp. 146–148. IEEE Computer Society (2009)
8. Fagin, R., Kumar, R., Sivakumar, D.: Comparing top k lists. SIAM J. Discrete Math. 17(1), 134–160 (2003)
9. Figueroa, K., Chávez, E., Navarro, G., Paredes, R.: Speeding up spatial approximation search in metric spaces. ACM Journal of Experimental Algorithmics (JEA) 14, article 3.6 (2009), 21 pages, doi: http://doi.acm.org/10.1145/1498698.1564506
10. Figueroa, K., Paredes, R., Rangel, R.: Efficient group of permutants for proximity searching. In: Proc. 3rd Mexican Conference on Pattern Recognition (MCPR 2011). pp. 42–49. LNCS 6718, Springer (2011)

11. Figueroa, K., Frediksson, K.: Speeding up permutation based indexing with indexing. In: Proceedings of the 2009 Second International Workshop on Similarity Search and Applications. pp. 107–114. SISAP '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/SISAP.2009.12
12. Patella, M., Ciaccia, P.: Approximate similarity search: A multi-faceted problem. Journal of Discrete Algorithms 7(1), 36–48 (2009)
13. Skala, M.: Counting distance permutations. J. of Discrete Algorithms 7(1), 49–61 (Mar 2009), http://dx.doi.org/10.1016/j.jda.2008.09.011