Compact and Efficient Permutations for Proximity Searching

Karina Figueroa Mora[1] and Rodrigo Paredes[2]

[1] Universidad Michoacana de San Nicolás de Hidalgo, México
[2] Universidad de Talca, Chile
karina@fismat.umich.mx, raparede@utalca.cl

Abstract. Proximity searching consists in retrieving the most similar objects to a given query. This kind of searching is a basic tool in many fields of artificial intelligence, because it can be used as a search engine to solve problems like *kNN* searching. A common technique to solve proximity queries is to use an index. In this paper, we show a variant of the permutation based index, which, in his original version, has a great predicting power about which are the objects worth to compare with the query (avoiding the exhaustive comparison). We have noted that when two permutants are close, they can produce small differences in the order in which objects are revised, which could be responsible of finding the true answer or missing it. In this paper we pretend to mitigate this effect. As a matter of fact, our technique allows us both to reduce the index size and to improve the query cost up to 30%.

1 Introduction

Proximity or similarity searching has become a fundamental task in different areas, for instance artificial intelligence and pattern recognition. The common elements in these areas are an object set and a similarity measure among its objects. The similarity is modeled by a distance function defined by experts in each application domain (for instance, Euclidean distance), which tells how similar two objects are. Objects are manipulated as black boxes and the only operation permitted is to measure the distance towards another object.

Formally, a metric space can be seen as a pair $(\mathbb{X}, d)$, where $\mathbb{X}$ is a universe of objects and d is a function $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+ \cup \{0\}$ that measures the distance between them. The distance function must satisfy the following properties for all $u, v, p \in \mathbb{U}$: positiveness, $d(u, v) \geq 0$, symmetry $d(u, v) = d(v, u)$, reflexivity $d(u, v) = 0$ iff $u = v$, and the triangle inequality $d(u, p) + d(p, v) \geq d(u, v)$.

Let $\mathbb{U}$ be a subset of objects from $\mathbb{X}$ of size n. In metric space searching, we basically consider two kinds of queries: range queries and k-Nearest-Neighbor queries. The range query retrieves all the objects within a given query radius, that is, $(q, r)_d = \{u \in \mathbb{U}, d(u, q) \leq r\}$. The k-Nearest-Neighbor query retrieves the (first) k elements of $\mathbb{U}$ closest to the query q; formally, $kNN(q)_d = \{u \in \mathbb{U}, v \in \mathbb{U} \setminus kNN(q)_d, d(u, q) \leq d(v, q)\}$ and $|kNN(q)_d| = k$.

When solving a problem with metric space searching techniques, we split it in two parts, preprocessing time and query time. During the preprocessing time, we build offline a data structure, the index, in order to solve future online queries. During the query time, we use the index in order to retrieve the objects from $\mathbb{U}$ that are relevant to the query. The cost of the searching can be evaluated in different ways, for instance, time to process the index, time consumed by side computations, or space; but most of them can be neglected because the distance function usually is expensive to compute (think, for instance, in comparing two documents or two fingerprints). Hence, our results will be presented in terms of the number of distance computations performed.

There are many algorithms for proximity searching in metric spaces, many of them are surveyed in [3]. Basically, they are divided in two classes, exact searching and approximated searching, even when similarity searching is already an approximated search. There are three families of proximity searching algorithms: pivots based, partition based and, nowadays, the permutation based algorithm.

Pivots are quite studied and it is known that this technique does not resist the curse of dimensionality[3], but in low dimensional spaces (2–8), it has good performance. By the other hand, partition based algorithms have good performance in medium and high dimension (8–20). However, in very high dimension (20–), exact algorithms fail and we have to use approximated or probabilistic approaches, the permutation based approach being one of the most successful ones. This last approach still has some deficiencies. Our proposal is a permutation variant, where we try to mitigate the effect when two permutants are so close that they can confuse the query procedure. Our experimental results show improvements up to 30% both in space and in distance computations performed.

2 Previous and related work: permutations

In [1], the authors present a novel technique called the permutation based algorithm. During the preprocessing, a subset $\mathbb{P} = \{p_1, p_2, \ldots, p_{|\mathbb{P}|}\} \subset \mathbb{U}$ is selected out of the database, which are called the permutants. From each $u \in \mathbb{U}$, we compute its distance to all the permutants (that is, compute $d(u, p)$ for all $p \in \mathbb{P}$) and sort them increasingly. Then, for each object $u \in \mathbb{U}$, we store in the index just the order of the permutants (not the distances), first the closest and so on. We define Π_u as a permutation of $(1 \ldots |\mathbb{P}|)$ so that, 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 index is composed by all the permutations for every object in $\mathbb{U}$, so it needs $O(n|\mathbb{P}|)$ memory cells.

The crux of this index is that two equal objects must have the same permutation, while similar objects will hopefully have similar permutations. So, if

[3] Empirically, the curse shows that the bigger the space dimensionality is, the harder the problem becomes. A detailed explanation can be found in [3].

Π_u is similar to Π_q we expect that object u is close to query q. Thus, we have changed the problem from searching $\mathbb{U}$ to searching the permutation set.

At query time, we compute Π_q and compare it with all the permutations stored in the index. Then, we traverse $\mathbb{U}$ by increasing permutation dissimilarity. If we limit the number of distance computations we obtain a probabilistic search algorithm. Fortunately, the order induced by Π_q is extremely promissory.

Now we explain how to compare two permutations Π_u and Π_q of $(1 \dots |\mathbb{P}|)$ using Spearman Rho S_ρ metric. In [5], S_ρ is defined as:

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

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

Let us give an example of $S_\rho(\Pi_q, \Pi_u)$. Let $\Pi_q = (6, 2, 3, 1, 4, 5)$ be the permutation of the query and $\Pi_u = (3, 6, 2, 1, 5, 4)$ that of an element u. Permutant p_3 in permutation Π_u is found two positions off with respect to its position in Π_q. The differences of position for each permutant within the permutations are: $1-3, 2-1, 3-2, 4-4, 5-6, 6-5$. Their squares make $S_\rho(\Pi_q, \Pi_u) = 8$. In Algorithm 1, we show how we compute this similarity in time $\Theta(|\mathbb{P}|)$.

Algorithm 1 SpearmanRho($\Pi_u^{-1}, \Pi_q, |\mathbb{P}|$)

1: INPUT: Π_u^{-1} is the inverse permutation for u, Π_q is the permutation of q, and $|\mathbb{P}|$ is the size of permutation.
2: OUPUT: Reports the similarity between Π_u and Π_q.
3: $t \leftarrow 0$
4: **for** $i \leftarrow 1$ to $|\mathbb{P}|$ **do**
5: $\quad t \leftarrow t + |i - \Pi_u^{-1}(\Pi_q(i))|^2$
6: **end for**
7: **return** t

There are other similarity measures between permutations [5]. However, they have a similar performance, and Spearman Rho shows the best balance between accuracy and time to compute (see [2] for more details).

In [1], the authors show that this technique is unbeatable in high dimensional spaces. In [4], the authors kept just the closest permutants and made an inverted index to reduce the index space and get a good performance during the query time. They gained a fast permutation similarity computation but lost precision.

There are other attempts for reducing the space used by the permutation index. For example, in [9] every partition is split and just used its first and last part. They improved the space used by the index but lost precision. In general, all the attempts to reduce the space of the permutation based algorithm sacrifice precision. In [7], the authors show an analysis of which part of the permutation is the most important, concluding that its beginning is the most important section and the middle portion is the least important.

In [8], the authors proposed to use groups of permutants instead of managing them independently. They reduce the number of distances used and the new permutations are almost as large as the original ones.

3 Our proposal

The main feature of the permutation index is its predicting power about which are the objects worth to compare with the query, and what is the order to do so. If we are allowed to do few distance computations, we would like that all the relevant objects were at the beginning of the revision order. However, small differences in the permutations can change this ordering. Moreover, since the permutation distances are discrete, the order changes are especially important at the beginning of the revision order.

We note that when two permutants are close, they can produce small differences in the permutations. When this occurs, we say those permutants collide. With limited amount of work, these differences could be responsible of finding the true answer or missing it. In this paper, we mitigate the collision effect.

3.1 Our contribution

Our contribution tries to reduce both the error rate of the permutation based algorithm and the space used for every object in the index. An $\alpha-$collision of two permutants is defined as follow: let $p_1, p_2 \in \mathbb{P}$, permutant p_1 and p_2 $\alpha-$collide with respect to object $u \in \mathbb{U}$ if $|d(u, p_1) - d(u, p_2)| \leq \alpha$. Our proposal is to keep only the permutants free of $\alpha-$collisions for every permutation. The intuition of the method is that if we prevent permutants collisions then we only consider the most important permutants for every permutation. Our results show that we have reduced the size of permutations without losing precision.

Figure 1 illustrates the $\alpha-$collision idea. Blank points are the permutants p_1, p_2, p_3, and p_4; white ones are objects u_1 and u_2; and the query q is the filled point. According to Spearman Rho, $S_\rho(q, u_1) = 6$ and $S_\rho(q, u_2) = 0$. This implies that q is possibly closer to u_2 than u_1, but it is clearly false. With our technique we detect that both p_1 $\alpha-$collides with p_2 and p_3 with p_4 with respect to u_1. So, the new permutation for u_1 is $\Pi_{u_1} = (1, 3)$, and for u_2, $\Pi_{u_2} = (2, 3, 4)$. We only need 2 permutants to learn that u_1 is the closest to q.

Formally, every permutation is computed as follow. We have a set of permutants $\mathbb{P} = \{p_1, \dots p_{|\mathbb{P}|}\} \subseteq \mathbb{U}$. Let $u \in \mathbb{U}$, u computes its distances to the permutants and sorts them. u keeps those permutants free of $\alpha-$collisions. Algorithm 2 shows how to compute the partial permutation for an element u.

Algorithm 3 shows how to compute partial Spearman Rho in time $\Theta(|\mathbb{P}|)$. Note that the size of the reduced permutation (m) could become much too smaller when compared to $|\mathbb{P}|$. With $\alpha = 0$ we return to the original permutation based algorithm.

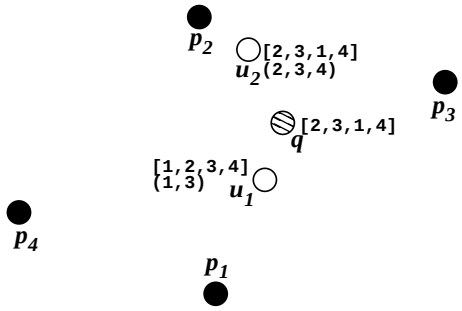

Fig. 1. Example of collision between permutants. p_1, p_2, p_3, p_4 are permutants, q is the query and u_1, u_2 are objects in the database. Note that the original permutations are in brackets and the compact ones in parenthesis.

Algorithm 2 Algorithm: build-PartialPermutation$(u, \mathbb{P}, \alpha)$

```
INPUT: u is an object, ℙ is a set of permutants and α is a parameter.
OUTPUT: u's permutation, without α−collisions.
D[] is an array of tuples (d, p)
for i ← 1 to |ℙ| do
    D[i] ← (d(u, p_i), i)
end for
Sort(D) // by component d
(Π_u(1), last, cont) ← (D[1].p, 1, 2)
for i ← 2 to |ℙ| do
    if D[i].d > (D[last].d + α) then
        Π_u(cont) ← D[i].p
        (last, cont) ← (i, cont + 1)
    end if
end for
return Π_u
```

Algorithm 3 SearmanRhoPartial(Π_u, Π_q)

```
INPUT: Π_u, Π_q are the permutations of u and q, respectively. Π_u may be reduced.
OUPUT: Reports the similarity between Π_u and Π_q.
(t, c) ← (0, 0)
for i ← 1 to |Π_q| do
    T[i] ← 0 // initializing
end for
for i ← 1 to |Π_u| do
    T[Π_u(i)] ← i
end for
for i ← 1 to |Π_q| do
    if T[Π_q(i)] ≠ 0 then
        (t, c) ← (t + |c − T[Π_q(i)]|^2, c + 1)
    end if
end for
return t
```

3.2 Solving queries

In order to solve range and *kNN* queries, first we need to compute the permutation similarity between the query and every point in the database. After that, we need to sort the objects in order to find the closest one, this takes $O(n \log n)$. This algorithm is probabilistic and this technique only show a good way to review the elements. Nevertheless, this technique preserves (and sometime improves) the predicting power of the permutation based index. In the next section, we show our experimental results.

4 Experimental evaluation

In the experimental evaluation, we used two kind of databases, namely, synthetic and real-world datasets. All the plots have a comparison with the basic technique, that is, when $\alpha = 0$. Despite that our variant is probabilistic, the retrieval in all the cases is 100%.

4.1 Synthetic Databases

The synthetic sets are random vectors uniformly distributed in the unitary cube. Then, we can control the space dimensionality and evaluate how it influences in our technique. We use datasets of dimension 16 to 32, with 10,000 objects.

In Figure 2, we use permutant sets of increasing size (ranging from 8 through 256 permutants) for a dataset in dimension 16. In the plots, $|\mathbb{P}|$ shows the initial number of permutants used per each object in the dataset, and the line in red is a separation-line just to read easily the results. Figure 2 (left) shows that as α grows, the number of non-colliding permutants decreases (the separation line is placed at 16 permutants). In the right side, we show the number of distances used to solve a query. Every point under the separation-line makes less distances than the original algorithm with 16 randomly chosen permutant, and also most of them use less permutants (this can be checked reading the left plot).

In order to illustrate our point, we compare the performance of 16 and 64 permutants. Starting with 64 permutants with the original technique (line in cyan with $\alpha = 0$), we can reduce both the number of distance computations to solve a query and also the number of non-collision permutants by changing the parameter α. In fact, with $\alpha \in [0.05, 0.1]$, Figure 2 (left) shows that the technique reduces from 64 to 9 permutants (about 44% less memory compared with 16 permutants under the original technique). For this number of permutants, Figure 2 (right) shows that the algorithm requires from 200 to 350 distance computations (in the best case, it uses up to 43% less distance computations).

The gain in terms of distance computations or the number of permutants is calculated as follow. Let $\mathbb{C}(|\mathbb{P}|, \alpha)$ and $\mathbb{T}(|\mathbb{P}|, \alpha)$ be the computed distances and the number of permutants with parameters $|\mathbb{P}|$ and α, respectively.

$$gain_{\mathbb{C}} = \frac{\mathbb{C}(|\mathbb{P}|, \alpha)}{\mathbb{C}(|\mathbb{P}|, \alpha = 0)} * 100 \tag{2}$$

$$gain_{\mathbb{T}} = \frac{\mathbb{T}(|\mathbb{P}|, \alpha)}{\mathbb{T}(|\mathbb{P}|, \alpha = 0)} * 100 \tag{3}$$

The best results for dimension 16 are showed in Figure 3. In left side, we use as reference $|\mathbb{P}| = 8$ permutants with $\alpha = 0$. The best tradeoff is where distances and permutants are intersected. For example, using in the left the best result with $|\mathbb{P}| = 64$, we improve the original technique up to 30% both in distances and permutants. In Figure 3 using as reference $|\mathbb{P}| = 16$ (right side), we improve the previous technique up to 40% using $|\mathbb{P}| = 128$. In this plot we have just the lines with gain in both distances and permutants. In other words, this is the zoom of Figure 2 under the red line.

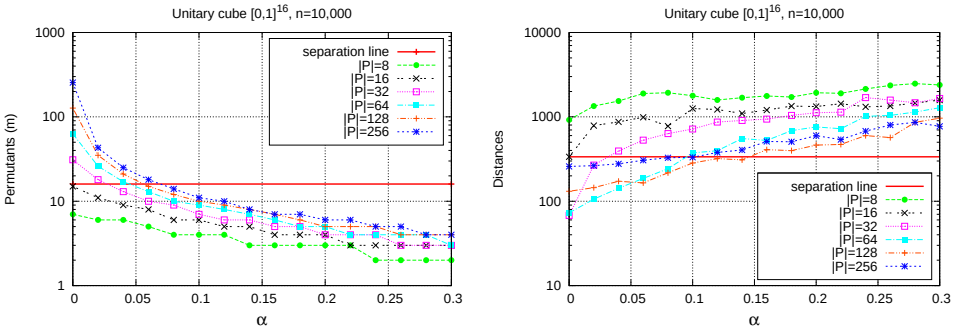

Fig. 2. Unitary cube in dimension 16. (Left) Size of permutation (m) and distances (right) when α grows.

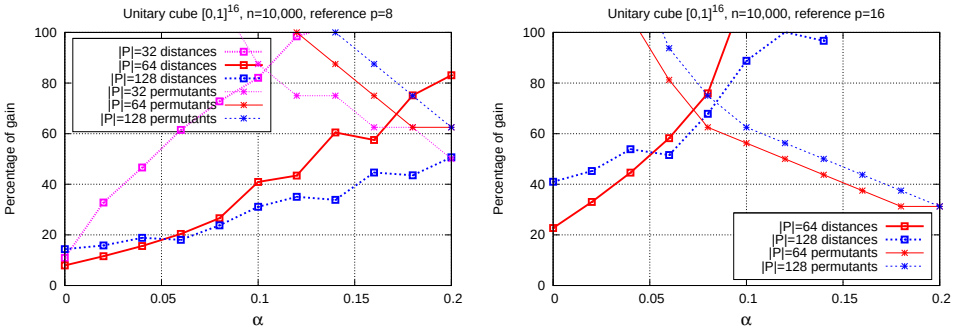

Fig. 3. The plots show the percentage of gain respect to the original permutation based algorithm. This is the zoom of the figure 2 in region under red line.

For dimension 32, we obtain similar results. For example, in Figure 4 using $|\mathbb{P}| = 8$, the original idea made 1,080 distances, while starting with $|\mathbb{P}| = 256$,

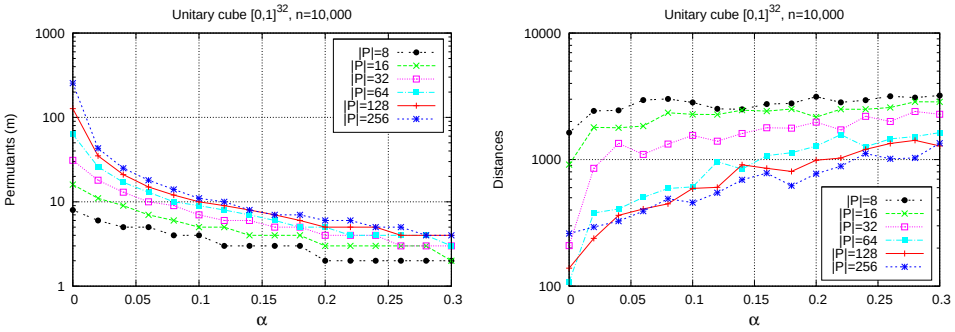

Fig. 4. Unitary cube in dimension 32. (Left) Size of permutation (m) and distances (right) when α grows.

when $\alpha = 0.15$ we have reduced the permutation set to $m = 8$ permutants in average per object. We also reduce the number of distances computed up to 750, that is almost 200 distances less, about 30% less distances computations.

4.2 Real Databases

In this section, we show the performance of our heuristic in a real-world space of images. The dataset used was obtained from the web site called **Flickr**, using the URL provided by the SAPIR collection [6]. The content-based descriptors extracted from the images were: Color Histogram $3\times3\times3$ using RGB color space (a 27 dim vector), Gabor Wavelet (a 48 dim vector), Efficient Color Descriptor 8×1 using both RGB and HSV color space (two 32 dim vectors), and Edge Local 4×4 (a 80 dim vector). The distance function used was Euclidean distance. The dataset size was 1 million of images.

Figure 5 shows the performance of our technique and also it compares our proposal with permutations by groups (introduced in [8]). Permutations by groups use 8, 16 and 32 groups and 2 permutants per group ($(|\mathbb{P}| = 8, 2), (|\mathbb{P}| = 16, 2), (|\mathbb{P}| = 32, 2)$. Since this technique is not affected by the parameter α, we plot it as a straight line. Notice that using 16 permutants and $\alpha = 5$, we can improve the number of distances computed using less permutants in the index.

5 Contributions and Future Work

Similarity searching is a very important operation in multimedia databases and other database applications containing complex objects. It involves finding objects in a dataset similar to a query object q, based on some distance measure d. In this paper we have introduce a novel way to reduce the size of permutation based algorithm without losing precision. To do so, we introduce the concept of α-collision: two permutants p_1 and p_2 $\alpha-$collide with respect to $u \in \mathbb{U}$, if

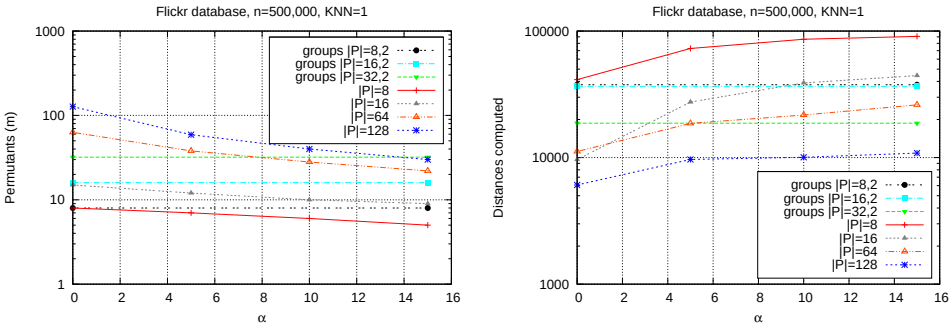

Fig. 5. Flickr database. (Left) Size of permutation (m) and distances (right) when α grows.

$|d(u,p_1) - d(u,p_2)| \leq \alpha$. We propose to keep just the non-α-colliding permutants for every object. Our algorithm improves the original technique up to 30% in both computed distances and the size of the permutant set. We note that α varies from a dataset to another. As future work, we are interested in turning out this probabilistic algorithm into an exact one.

References

1. 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 2005). pp. 405–414. LNCS 3789 (2005)
2. Chávez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI) 30(9), 1647–1658 (2009)
3. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (Sep 2001)
4. 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)
5. Fagin, R., Kumar, R., Sivakumar, D.: Comparing top k lists. SIAM J. Discrete Math. 17(1), 134–160 (2003)
6. Falchi, F., Kacimi, M., Mass, Y., Rabitti, F., Zezula, P.: Sapir: Scalable and distributed image searching. In: SAMT (Posters and Demos). vol. 300 of CEUR Workshop Proceedings, pp. 11–12 (2007)
7. Figueroa, K., Paredes, R.: Finding good permutants for proximity searching in metric spaces. In: Proc. 2010 Intl. Conf. on Information Security and Artificial Intelligenge (ISAI 2010). vol. 1, pp. 320–323. IEEE Computer Society Press (2010)
8. 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)
9. Sadit, E., Chávez, E.: On locality sensitive hashing in metric spaces. In: Proc. 3rd Intl. Workshop on Similarity Search and Applications (SISAP 2010). pp. 67–74. ACM press (2010)