Efficient Group of Permutants for Proximity Searching

Karina Figueroa Mora[1], Rodrigo Paredes[2], and Roberto Rangel[1]

[1] Universidad Michoacana de San Nicolás de Hidalgo, México
[2] Universidad de Talca, Chile
karina@fismat.umich.mx, raparede@utalca.cl, a0529275g@correo.fie.umich.mx

Abstract. Modeling proximity searching problems in a metric space allows one to approach many problems in different areas, e.g. pattern recognition, multimedia search, or clustering. Recently there was proposed the permutation based approach, a novel technique that is unbeatable in practice but difficult to compress. In this article we introduce an improvement on that metric space search data structure. Our technique shows that we can compress the permutation based algorithm without loosing precision. We show experimentally that our technique is competitive with the original idea and improves it up to 46% in real databases.

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 a database (e.g. a set of objects) and a similarity measure among its objects (e.g. Euclidean distance). The similarity is modeled by a distance function defined by experts in each application domain, which tells how similar two objects are. The objects are manipulated as black boxes and the only operation permitted is to measure their distance towards another object. Usually, the distance function is quite expensive to compute, therefore our goal is avoiding to make comparisons between objects.

Another problem in these days is that there are huge databases and usually these data use considerably less structure and much less precise queries than traditional database system. Example are multimedia data like images or videos where is common query-by-example search. In view of these challenges a way to face up is building an index that allows us to search quickly. In particular, we are proposing to use a metric space index. A metric space consists in a dataset and a function distance (formally it is described at Section 2).

All metric space search algorithms rely on an index, that is, a data structure that maintains some information on the database in order to save some distance evaluations at search time. Chávez et al. [2] give a complete survey in this area (the two main types of indices), but recently in metric space indices was a third proposal, the permutation-based algorithm [1] which is unbeatable in practice but is difficult to compress. All the proposals to compress this kind of index are prepared to loose precision at retrieval but reducing the size of the index [3,7].

J.-F. Martínez-Trinidad et al. (Eds.): MCPR 2011, LNCS 6718, pp. 42–49, 2011.
© Springer-Verlag Berlin Heidelberg 2011

In this paper we face the compression of the index, by a novel index using cluster techniques. At Section 2 we introduce the basic concepts and discuss the previous work in permutation-based algorithms. At Section 3 we explain our proposal and finally at Section 4 we present the experimental part that support our technique. We finish with our conclusions and future work at Section 5.

2 Basic Concepts and Related Work

Formally, the proximity search problem in a metric space may be stated as follows: there is a universe $\mathbb{X}$ of objects and a nonnegative *distance function* $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$ defined among them. The distance satisfies the axioms that make the set a metric space: reflexivity ($d(x, x) = 0$), strict positiveness ($x \neq y \Rightarrow d(x, y) > 0$), symmetry ($d(x, y) = d(y, x)$), and triangle inequality ($d(x, z) \leq d(x, y) + d(y, z)$). Usually, this distance is expensive to compute. We have a finite database $U \subseteq \mathbb{X}$, of size n, which is a subset of the universe of objects.

Basically, there are two kind of queries: *range query* and *K-Nearest Neighbor query* (K-NN). The first one consists in retrieving those objects within a radius to a given query, that is $R(q, r) = \{d(u, q) \leq r \mid \forall\ u \in \mathbb{U}\}$, the second one is to retrieve the K elements of $\mathbb{U}$ that are closest to q.

Most of the existing approaches to solve the search problem are *exact algorithms* which retrieve exactly the elements of $\mathbb{U}$ as specified above. In [2,6,8,9], most of those approaches are surveyed and explained in detail. These kind of indices usually have a good performance with a few dimensions (two to eight). However, in high dimensions they compare the whole database. An alternative in high dimension is the permutation-based algorithms (PBA).

In [1], the authors show the technique PBA as follows. Let $\mathbb{P} \subseteq \mathbb{U}$ be a set of distinguished objects from the database, called *permutants*. Each element of the space, $u \in \mathbb{U}$, defines a *permutation* Π_u, where the elements of $\mathbb{P}$ are written in increasing order of distance to u. Ties are broken using any consistent order, for example the order of the elements in $\mathbb{P}$.

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

Given permutations Π_u and Π_q of $(1 \ldots k)$, we can compare them using Spearman Rho. It is defined as[1]

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

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. A particular element p_3 in permutation Π_u is found two positions off with respect to its

[1] The actual definition in [4] corresponds to $\sqrt{S_\rho(\Pi_q, \Pi_u)}$ in our terminology. We omit the square root because it is monotonous and hence does not affect the ordering.

position in Π_q. The differences of position of each permutant within its permutation are: $1-2, 2-3, 3-1, 4-4, 5-6, 6-5$, and the sum of their squares is $S_\rho(\Pi_q, \Pi_u) = 8$.

There are other similarity measures between permutations [4], however in [1] the authors show that all of them have a similar performance but specially Spearman Rho has a balance between accuracy and time to compute it.

An attempt to compress the index is shown in [3]. They proposed to chose just a few of permutants (the closest ones) in each permutation, and use an inverted index to keep them. In this way they compress the index. They improve in searching time and the space used by the index but retrieval is sacrificed. They can lost up to 20% of the retrieval using less permutants.

There is another attempt to compress the index sacrificing the retrieval which is described in [7]. Basically, they represent the permutation using just two bits, and using another similarity between permutations. They show a good performace but after all retrieval is sacrificed.

3 Our Proposal

The basic idea of a PBA consists in selecting a set of permutants, and produce all the permutations (by comparing every object in the database against the permutants, and sorting in increasing order).

Our proposal consists in using a *set of permutants* instead of a single permutant per each component in the permutation. In this way, we use the same amount of space that the original one but we have more precision.

Formally, we select sets of permutants $\mathbb{G} = \{G_1, G_2, \ldots, G_k\}$ where $\mathbb{G} \subseteq \mathbb{U}$ and $G_i \cap G_j = \emptyset$, $\forall\ i, j,\ 1 \leq i, j \leq k$. Each group has m permutants. Then, $\forall\ u \in \mathbb{U}$, we compared u against the groups, that is, $D_i(u, G_i)$, $1 \leq i \leq k$ and we sorted D_i by proximity to u. In the next section we discuss the criteria to compute D_i.

3.1 Proximity to a Group

An important factor of the performance in our technique is how to decide the proximity to the groups (i.e., how to compute D). We propose treating each group as a cluster and we can use the methods to classify a cluster.

- Single Linkage: That is, we consider the lowest distance towards all the objects in the group, $D_{i_{min}}(u, G_i) = \min_{\forall p \in G_i}\ d(p, u)$.
- Complete Linkage: In this case we consider the greatest distance towards all the objects in the group $D_{i_{max}}(u, G_i) = \max_{\forall p \in G_i}\ d(p, u)$.
- Average: The third propose is the average of distances, that is, $D_{i_{av}}(u, G_i) = \frac{\sum_{\forall p \in G_i}\ d(p,u)}{|G_i|}$

For simplicity, in the follow we will use only $Dmin$, $Dmax$ o Dav.

Figure 1 describes two (of the three) criteria of proximity towards the groups. Permutants are the black points. $G_1 = \{p_1, p_2, p_3\}$, $G_2 = \{p_4, p_5, p_6\}$, and

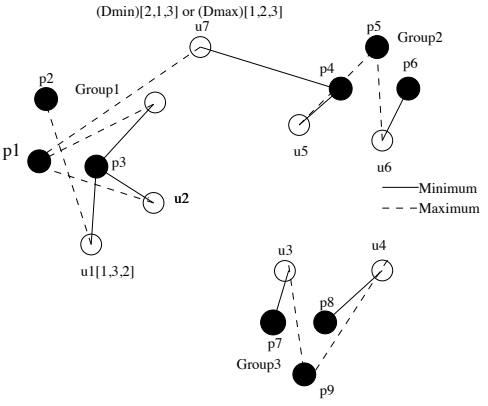

Fig. 1. Criteria for proximity to a group. Continuous line is for the minimum distance (single linkage) to the group, and discontinued line is for the maximum distance (complete linkage).

$G_3 = \{p_7, p_8, p_9\}$. Every point has a line to the closest group according to the criterion used. Notice that each permutation depends on these criteria. For example, u_1 does not change its permutation when considering single or complete criterion, but u_7 does. In fact, using the *single linkage* $\Pi_{u_7} = 2, 1, 3$ ($Dmin(u_7, G_2) \leq Dmin(u_7, G_1) \leq Dmin(u_7, G_3)$); and using the *complete linkage* $\Pi_{u_7} = 1, 2, 3$ ($Dmax(u_7, G_1) \leq Dmax(u_7, G_2) \leq Dmax(u_7, G_3)$). In this figure, we have not drawn the average criterion because it depends on each point and it can confuse the idea. At the experimental section we can see the performance of these criteria.

3.2 Selecting Good Permutants per Group

Figure 1 shows the permutant groups closer each other, but that is not mandatory. In this section we consider other options:

1. Random (RTG). That is choosing elements at random to form a group.
2. Closer to its group (CTG). For this heuristic we propose choosing one permutant p and to pick up closest ones to p for the rest of the group.
3. Farther to its group (FTG). Unlike to the previous one we use the opposite, that is the farther ones to p.

Algorithm 1 shows our proposal. Notice that before to use this algorithm we need to compute every Π_u, $\forall\ u \in \mathbb{U}$. The fraction f of the database to traverse depends on dimension and number of permutants, to name a few. For more reference see [1].

Analysis. Our technique uses the same amount of space as the original one, that is $\Theta(kn)$, where k is the number of groups. We notice that, we can pack several group identifiers in a single machine word. Also, we keep a small vector

Algorithm 1. gPermutation(q,r,f)

```
1: INPUT: q is a query and r its radius, f is the fraction of the database to traverse.
   We have Π_u, ∀u ∈ 𝕌.
2: OUTPUT: Reports a subset of those u ∈ 𝕌 that are at distance at most r to q.
3: Let A[1, n] be an array of tuples and 𝕌 = {u_1, ..., u_n}
4: Compute Π_q with the same criterion used (Dmin, Dmax, Dav) to build every Π_u
5: Every group was formed using RTG, CTG or FTG
6: for i ← 1 to n do
7:    A[i] ← ⟨u_i, S_ρ(Π_{u_i}, Π_q)⟩
8: end for
9: SortIncreasing(A) // by second component of tuples
10: for i ← 1 to f · n do
11:    Let A[i] = ⟨u, s⟩
12:    if d(q, u) ≤ r then
13:       Report u
14:    end if
15: end for
```

with the identifiers of the permutants within the groups. That is $\Theta(km)$, where m is the group size (we can also try to save some space by packing the identifiers somehow). Another important issue to emphasize is that our technique spends the same time as the original to solve a query because we use the same procedure that the original idea at algorithm 1 (lines 6-15).

4 Experimental Section

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 real life databases. 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 10,000 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.

Figure 2 shows the performance of our technique in different dimensions. We use all the parameters proposed (FTG, CTG, RTG), and Dav, $Dmin$ and $Dmax$. The line with label $m = 1$ is the original permutant idea (one permutant per group). Our technique can improve the original one in dimensions up to 64. As can be seen Dav has better performance than $Dmax$ and $Dmin$, and CTG and RTG has better performance over FTG most of the time.

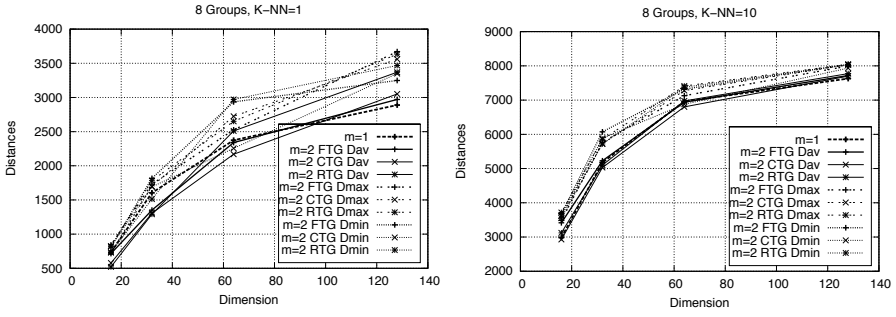

Fig. 2. Vectors uniformly distributed on the unitary cube n=10,000, K-NN=1,10

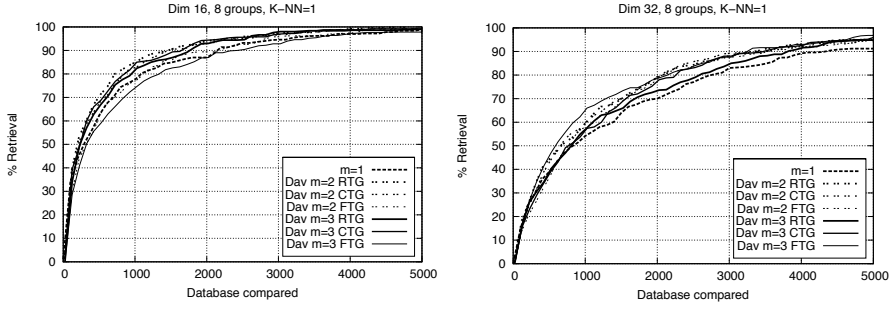

Fig. 3. Dimension 16 and 32, n=10,000, K-NN=1. We use the three criteria to sort objects by proximity to the groups, and the three criteria to choose permutants per group.

Figure 3 shows that experiments with $m = 2$ and $m = 3$ retrieve faster than $m = 1$. In this case, for simplicity we only kept $8n$ bytes for the index. We also use 8×2 or 8×3 machine words to identify each of the permutants per group.

However, we can pack the permutant index in just $3n$ bytes plus the space of the permutant identifiers (as we are considering just eight groups, so we need 24 bits for each permutation).

4.2 Real Databases

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

Flickr. The set of image objects were taken from Flickr, using the URL provided by the SAPIR collection [5]. The content-based descriptors extracted from the images were: Color Histogram $3 \times 3 \times 3$ using RGB color space (a 27 dim vector), Gabor Wavelet (a 48 dim vector), Efficient Color Descriptor (ECD) 8×1 using RGB color space (a 32 dim vector), ECD 8×1 using HSV color space (a 32 dim

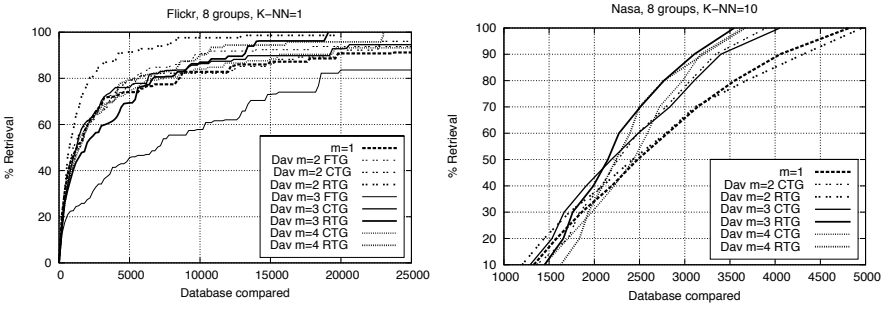

Fig. 4. Real-life database (left)Flickr database, (right) Nasa database. K-NN=1,10.

vector), and Edge Local 4×4 (a 80 dim vector). The distance function used was Euclidean distance. The dataset size was 1 million of images.

Nasa. A set of 40,150 20-dimensional feature vectors, generated from images downloaded from NASA[2] and with duplicate vectors eliminated. We also use Euclidean distance.

At figure 4 the original idea is labeled (black line) with $m = 1$, that is 1 permutant per group (randomly selected). With our technique we can retrieve the 100% of the query using up to 46% less comparison using two permutants per group ($m = 2$ and RTG). With three permutants per group we can also get faster the 100% of retrieval but it grows slower than $m = 2$. Notice that comparing just 5000 distances our technique has retrieved 90% of the data while the original idea has retrieved only 75% of the data. The plots also show that the FTG strategy has the worst performance when choosing the permutant groups. Our technique has a better performance on a real database.

5 Conclusions and Future Work

The permutation based algorithm for proximity searching is unbeatable in high dimension, but this kind of index has not been improved using less space. All the attempts to compress the index lose precision at retrieval.

In this article we present an improvement on the permutation-based algorithm. The main idea is to have a set of permutants instead of a single permutant (as the original idea). With our heuristic we can improve the permutation based algorithm using the same amount of space as the original idea. The experimental part shows that our technique can improve the performance of the original idea up to 46% in real databases.

In future work we will select different amount of permutants per group. That is, some groups will have more permutants than others.

[2] At `http://www.dimacs.rutgers.edu/ Challenges/Sixth/software.html`

Acknowledgements

This paper was partially supported by National Council of Science and Technology (CONACyT) of México and by Universidad Michoacana de San Nicolás de Hidalgo, México.

References

1. 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)
2. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (2001)
3. Esuli, A.: Mipai: using the pp-index to build an efficient and scalable similarity search system. In: Similary Searching and Applications, pp. 146–148 (2009)
4. Fagin, R., Kumar, R., Sivakumar, D.: Comparing top k lists. SIAM J. Discrete Math. 17(1), 134–160 (2003)
5. Falchi, F., Kacimi, M., Mass, Y., Rabitti, F., Zezula, P.: Sapir: Scalable and distributed image searching. In: CEUR Workshop Proceedings SAMT (Posters and Demos), vol. 300, pp. 11–12 (2007)
6. Hjaltason, G., Samet, H.: Index-driven similarity search in metric spaces. ACM Transactions Database Systems 28(4), 517–580 (2003)
7. Sadit, E., Chávez, E.: On locality sensitive hashing in metric spaces. In: Similarity Search and Applications, pp. 67–74. ACM Press, New York (2010), ISBN: 978-1-4503-0420-7
8. Samet, H.: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling). Morgan Kaufmann Publishers Inc., San Francisco (2005)
9. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach. In: Advances in Database Systems, vol. 32, Springer, Heidelberg (2006)