An Effective Permutant Selection Heuristic for Proximity Searching in Metric Spaces *

Karina Figueroa[1] and Rodrigo Paredes[2]

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

Abstract. The permutation based index has shown to be very effective in medium and high dimensional metric spaces, even in difficult problems such as solving reverse k-nearest neighbor queries. Nevertheless, currently there is no study about which are the desirable features one can ask to a permutant set, or how to select good permutants. Similar to the case of pivots, our experimental results show that, compared with a randomly chosen set, a good permutant set yields to fast query response or to reduce the amount of space used by the index. In this paper, we start by characterizing permutants and studying their predictive power; then we propose an effective heuristic to select a good set of permutant candidates. We also show empirical evidence that supports our technique.

1 Introduction

Proximity or *similarity* searching is the problem of, given a data set and a similarity criterion, finding elements within the set that are close or similar to a given query. This is a natural extension of the classical problem of exact searching. It is motivated by data types that cannot be queried by exact matching, such as multimedia databases containing images, audio, video, documents, and so on. In this framework, the exact comparison is just a type of query, while close or similar objects can be queried as well.

There exist several computer applications where the concept of similarity retrieval is of interest (see [4] for a comprehensive survey on those applications). Some examples are *machine learning and classification*, where a new element must be classified according to its closest existing element; *image quantization and compression*, where only some samples can be represented and those that cannot must be coded as their closest representable one; *text retrieval*, where we look for words in a text database allowing a small number of errors, or we look for documents similar to a given query or document; *computational biology*, where we want to find a DNA or protein sequence in a database allowing some errors due to typical variations; and *function prediction*, where past behavior is extrapolated to predict future behavior, based on function similarity.

* 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.

Proximity/similarity queries can be formalized using the metric space model [4, 7, 10, 12]. The model assumes that there is a universe $\mathbb{X}$ of objects and a non-negative distance function $d : \mathbb{X} \times \mathbb{X} \to R^+ \cup \{0\}$ defined among them. Objects in $\mathbb{X}$ do not necessarily have coordinates (think, for instance, in strings, images, audio, or video). On the other hand, the distance function gives us a dissimilarity criterion to compare objects from the universe. Therefore, the smaller the distance between two objects, the more "similar" they are.

The distance satisfies the following properties that make the tuple $(\mathbb{X}, d)$ a metric space: $d(x, y) \geq 0$ (*positiveness*), $d(x, y) = d(y, x)$ (*symmetry*), $d(x, x) = 0$ (*reflexivity*), and $d(x, z) \leq d(x, y)+d(y, z)$ (*triangle inequality*). These properties hold for many reasonable similarity functions.

In the typical scenario of the metric space search problem, there is a finite database or dataset of interest $\mathbb{U} \subset \mathbb{X}$. Later, when a new query object $q \in \mathbb{X} \setminus \mathbb{U}$ arrives, its proximity query consists in retrieving objects from $\mathbb{U}$ that are relevant (that is, similar) to q. There are two basic kinds of queries. The first is the *range query* $d(q, r)$, which retrieves all the elements in $\mathbb{U}$ which are within distance r to q. Formally, $(q, r) = \{u \in \mathbb{U}, d(q, u) \leq r\}$. The second is the *k-nearest neighbor query* $NN_k(q)$, which retrieves the k closest-to-q elements in $\mathbb{U}$. This is, $|NN_k(q)| = k$, and $\forall\ u \in NN_k(q)$ and $v \in \mathbb{U} \setminus NN_k(q)$, $d(u, q) \leq d(v, q)$.

Given the database, these similarity queries can be trivially answered by performing $n = |\mathbb{U}|$ distance evaluations. Yet, as the distance function is assumed to be expensive to compute (think, for instance, when comparing two fingerprints), it is customary to define the complexity of the search as the number of distance evaluations performed, disregarding other components such as CPU time for side computations and even I/O time. Thus, the ultimate goal is to build offline an index in order to speed up online queries.

The Permutation Index [3] is one of the most effective similarity search indices. It is particularly well suited for medium and high dimensional spaces. Despite that its practical version is focused in the case of retrieving an approximate answer for a given query, its effectiveness allows us to consider it as an (almost) exact answer. As a matter of fact, experimental results shown in [3] reveal that the Permutation Index is extremely promissory. Nevertheless, there is no study about which are the desirable features one can ask to a permutant set, or how to select good permutants. This is the main focus of this paper.

Similar to the case of traditional pivots [1, 2], our experimental results show that a good permutant set yields to fast query response or to reduce the amount of space used by the index compared to a randomly chosen set. In fact, in the metric space of Flickr images, we only need to review a 1.15% of the dataset to obtain the nearest neighbor of a given query.

The rest of this paper is organized as follows. In Section 2, we cover related work on metric space indices. Then, in Section 3, we start by characterizing permutants and studying their prediction power, and then we propose an effective heuristic to select a good permutant candidate set. Next, Section 4 shows the empirical evidence that supports our technique. We finally draw our conclusions and future work directions in Section 5.

2 Related Work

2.1 Previous Work on Choosing Pivots

The pivot based indices are one of the most popular in the field of metric space searching. These algorithms select a set of pivots $\{p_1 \dots p_k\} \subseteq \mathbb{U}$ and store a table of kn distances $d(p_i, u),\ i \in \{1 \dots k\}, \forall\ u \in \mathbb{U}$. To solve a range query (q, r), pivot-based algorithms measure $d(q, p_i)$ for all the pivots, and use the fact that, because of the triangle inequality, $d(q, u) \geq |d(q, p_i) - d(u, p_i)|$, so they can discard every $u \in \mathbb{U}$ such that $|d(q, p_i) - d(u, p_i)| > r$, since this implies $d(q, u) > r$ (which any of the pivots). The elements u that still cannot be discarded at this point are directly compared against q.

In general, these indices choose the pivots randomly among the database objects. However, there are some preliminary attempts to choose pivots [11, 8] which are based in two ideas: good pivots are far away from the rest of the database, and also, far away of each other pivots. These techniques have good performance in some metric spaces but they fail in others.

There are two works [1, 2] on how to systematically select a good pivot set.

In [1], the authors propose an efficiency measure to compare two pivot sets, maximizing the mean of the distance distribution among pivots. Based on that criterion, they also provide three optimization techniques to select good sets of pivots: (*i*) produce several pivot sets at random and choose the set with the highest average distance; (*ii*) choose pivots incrementally, the first pivot p_1 is the object that have the maximum average distance to other objects, the second one p_2 is selected so that the subset $\{p_1, p_2\}$ has the maximum average distance, and so on until selecting all the pivots; and (*iii*) starting with a random set of pivot, in each iteration the pivot having the smallest contribution to the average distance is removed and replaced by a better pivot.

In [2], the authors propose a dynamic pivot selection technique, which combines the third alternative in [1] with the Sparse Spatial Selection (SSS) technique [9]. The idea is to use the same SSS's pivot insertion criterion but checking if any of the already selected pivots becomes redundant (in the sense that its contribution to some efficiency criterion is low) or the new pivot candidate is redundant with respect to the current pivot set.

2.2 The Permutation based Index

Let $\mathbb{P} \subset \mathbb{U}$ be a subset of permutants (in the literature, they are also called anchors). Each element $u \in \mathbb{U}$ induces a preorder $\leq_u$ given by the distance from u towards each permutant, defined as $y \leq_u z \Leftrightarrow d(u, y) \leq d(u, z)$, for any pair $y, z \in \mathbb{P}$. The relation $\leq_u$ is a preorder and not an order because some permutants can be at the same distance of u. So, it could be possible to find $y, z \in \mathbb{P}$, such that $y \leq_u z \wedge z \leq_u y$.

Let $\Pi_u = i_1, i_2, \dots, i_{|\mathbb{P}|}$ be the permutation of u, where permutant $p_{i_j} \leq_u p_{i_{j+1}}$. Permutants at the same distance take an arbitrary but consistent order. Every object in $\mathbb{U}$ computes its preorder of $\mathbb{P}$ and associates it to a permutation, which

is stored in the index (this index does not store distances). Thus, a simple implementation needs $n|\mathbb{P}|$ space.

The crux of this index is that two equal objects must have the same permutation, while similar objects will hopefully have similar ones. So if Π_u is similar to Π_q we expect that u is close to q. Thus, we have changed the problem from searching the dataset $\mathbb{U}$ to searching the permutation set.

At query time, we compute Π_q and compare it with all the permutations stored in the index. So, we traverse $\mathbb{U}$ in the order $\leq_{\Pi_q}$ induced by Π_q (by increasing permutation dissimilarity). If we limit the number of distance computations, we obtain a probabilistic search algorithm (in the sense that with some probability, we are able to find the right answer to the query). Fortunately, the order $\leq_{\Pi_q}$ induced is extremely promissory, as reported in [3] for range queries.

Similarity between the permutations of q and u can be measured by Kendall Tau K_τ, Spearman Footrule S_F, or Spearman Rho S_ρ metric [5], among others. K_τ can be seen as the number of swaps that a bubble-sort-like algorithm has to do in order to make two permutations equal. On the other hand, using $\Pi_u^{-1}(i_j)$ to denote the position of permutant p_{i_j} in the permutation Π_u, S_F and S_ρ are defined as follows:

$$S_F(\Pi_u, \Pi_q) = \sum_{j=[1,|\mathbb{P}|]} |\Pi_u^{-1}(i_j) - \Pi_q^{-1}(i_j)| \tag{1}$$

$$S_\rho(\Pi_u, \Pi_q) = \sqrt{\sum_{j=[1,|\mathbb{P}|]} |\Pi_u^{-1}(i_j) - \Pi_q^{-1}(i_j)|^2} \tag{2}$$

As S_ρ is monotonous, we can simple use S_ρ^2. For example, let $\Pi_u = (42153)$ and $\Pi_q(32154)$ be the object $u \in \mathbb{U}$ and query $q \in \mathbb{X} \setminus \mathbb{U}$ permutations, respectively. So, $K_\tau(\Pi_u, \Pi_q) = 7$, $S_F(\Pi_u, \Pi_q) = 8$ and $S_\rho^2 = (\Pi_u, \Pi_q) = 32$.

3 Finding Good Permutants

In the following, we will describe our approach to select good permutants. It is based on experimental observations that lead us to the final methodology to obtain a good permutant set. We start by characterizing the permutants, and then we propose a heuristic to select a permutant set.

3.1 Characterizing the Permutants

Let us consider a single permutant p from the set $\mathbb{P}$. For this permutant, the maximum difference of positions between the query permutation Π_q and the permutation of any object in the database Π_u is $|\mathbb{P}| - 1$. This is obvious, however it suggests a simple way to determine how much a permutant can contribute in the searching process. Permutants at the beginning or the end of a permutation can produce an important increase in the permutation distance, while the ones in the middle of the permutation produce a mild increase. Note that, such increase is the one that changes the order $\leq_{\Pi_q}$ (induced by the permutation of the query).

Based in this observation, we start our study by considering how much contributes a portion of the permutation to the order $\leq_{\Pi_q}$. To illustrate our point we perform the following experiment. We divide the permutation Π_q in three equal portions, and show the results when searching in a metric space using the permutants that fall in two thirds of the permutation Π_q (that is, we exclude a portion). To do this, we compute the permutation distance using Equation 3.

$$S_F(\Pi_u, \Pi_q) = \sum_{j=[1,|\mathbb{P}| \wedge \Pi_q^{-1}(i_j) \notin \texttt{portion}]} |\Pi_q^{-1}(i_j) - \Pi_u^{-1}(i_j)| \qquad (3)$$

For this experiment, we use $(\mathbb{R}^D, L_2)$ considering a uniform dataset composed by 80,000 vectors with dimension $D \in [8, 128]$, and show the average over 100 randomly chosen queries. According to the results of [3] we use permutant sets of size $D \pm 2$, that is, a direct relation with the dimension of data.

Fig. 1 shows the results when we discard either the left, middle, or right portions of the permutation. The figure shows that the best results are obtained when we discard the middle portion; this is in agreement with the intuition that the permutants that fall in this section do not heavily increase the permutation distance. On the other hand, the figure also says that the permutants in the left portion are the most important ones in terms of inducing the order $\leq_{\Pi_q}$, since in high dimension ($D = 128$), removing the left partition produces the biggest degradation in the search efficiency.

This observation leads us to the core of our method to find good permutants. The main idea is trying to pick permutants that do not fall in the middle portion. To do this, we can look for permutants that concentrate their occurrences in some portions of the object permutations (hopefully in the left one).

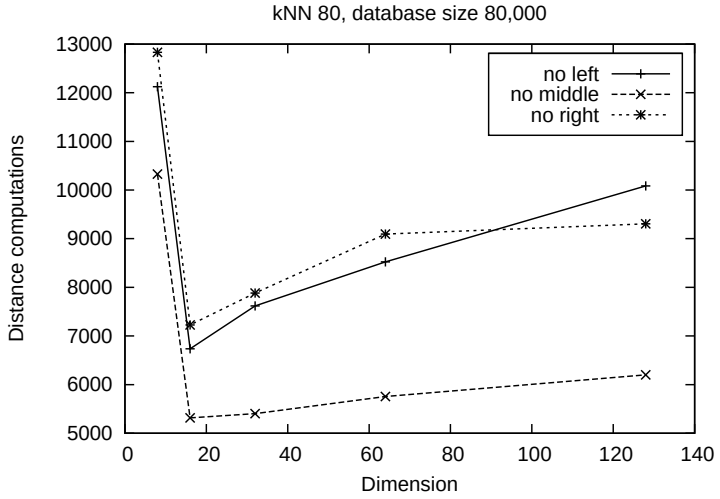

Fig. 1. Retrieval performance when excluding one portion of the permutation.

3.2 A Simple Heuristic to Find Good Permutants

Given a permutant, one can compute the histogram of its positions in the permutations of all the elements in $\mathbb{U}$. If that histogram is not concentrated, its variance is large when compared with a random permutant.

Therefore, we now perform another preliminary experiment in order to verify this assumption. In this experiment we compare the retrieval performance of three sets. The first is a randomly chosen permutant set of size D, the second is composed by D permutants having low variance in their histogram of positions, and the third are the D permutants with high variance. This experiment was done using the same setup of the previous section.

To choose permutant sets with low and high variance, we take a random sample of size $c = \left\lfloor \frac{1+\sqrt{1+8n}}{2} \right\rfloor$ (where $n = |\mathbb{U}|$). We select this value since the full comparison among all the elements in the sample requires $\frac{c(c-1)}{2} < n$ distance computations. So, we maintain controlled the number distance computations. If $D > c$, we repeat the process until we have enough permutants to make the selection. But, in practical cases $D = O(\sqrt{n})$, so with one iteration is enough.

Fig. 2 shows that the best similarity query performance is verified with the set whose permutants present the largest variance. The difference is clearly noticeable, in particular in high dimensional spaces. In fact, in dimension $D = 128$, the high variance set needs 79% of the distance computations required by the random set; and for $D = 64$, the high variance set requires just 76%. The second performance is achieved by the random chosen set. This behavior is counter intuitive as we would expect that having a criterion is better that choosing permutants at random.

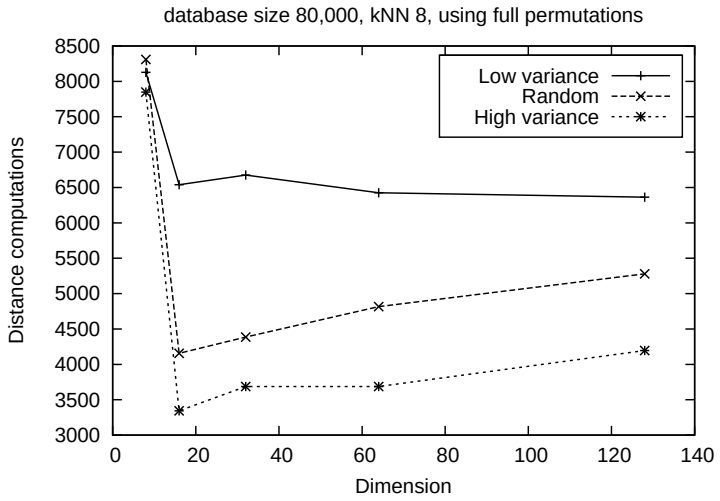

Fig. 2. Search performance of a random permutant set versus low and high variance permutant sets.

4 Experimental Evaluation

We show the performance of our heuristic in two real-world spaces of images.

4.1 Flickr

The set of image objects were taken from Flickr, using the URL provided by the SAPIR collection [6]. The content-based descriptors extracted from the images were: Color Histogram $3 \times 3 \times 3$ using RGB color space (27 dimension vector), Gabor Wavelet (48 dimension vector), Efficient Color Descriptor (ECD) 8×1 using RGB color space (32 dimension vector), ECD 8×1 using HSV color space (32 dimension vector), and Edge Local 4×4 (80 dimension vector). For each image, all the components are concatenated in a single vector of 219 components. We compare the vectors using Euclidean distance. The dataset contains 1 million of images.

Fig. 3 shows the performance of our technique when solving k-nearest neighbor queries. We test three values of k, namely 1, 10, and 20, and the three criteria used in Section 3.2, that is, excluding the each one of the three portions (left, middle and right). We repeat the experiment using permutant set of size 128 and 256. Notice that we also compare our criteria with the complete permutation of 128 or 256 permutants, respectively. These experiments show that we make almost the same computational effort using two thirds of the permutations than using the complete one. In practice, the center or the right portion does not really give much information.

Usually, users retrieve a few numbers of images per query. So for illustrate our results, we remark that with $k = 1$, we need to compare only 1.15% to 0.89% of the database (for 128 or 256 permutants). If we use higher values of k, our index needs to review a larger fraction of the database, as expected; however, the query performance is sublinear with respect to k.

4.2 Nasa

A set of 40,150 20-dimensional feature vectors, generated from images downloaded from NASA[3] and with duplicate vectors eliminated. We also use Euclidean distance.

At Fig. 4, we show the performance of our technique when solving k-nearest neighbor queries ($k \in \{1, 10, 20\}$), and two permutant set sizes $\{128, 256\}$. Notice that we compare our criteria with the complete permutation of 128 or 256 permutants (that is, considering the three parts, namely, left, center and right). Once again, the complete permutation uses more permutants than the others (exactly $\mathbb{P}/3$ more), and that can explain why it is better than the others. However, permutations without center (or right) part have the same power of prediction.

[3] at `http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html`

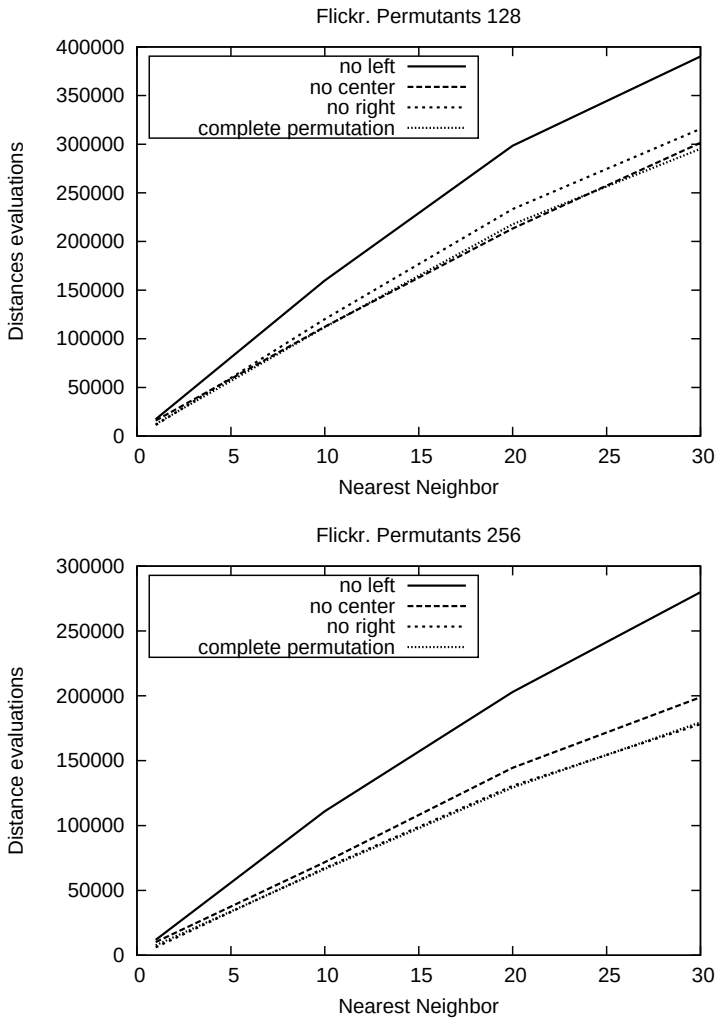

Fig. 3. Performance of our technique using real database (FLICKR images).

5 Conclusions

In this paper we started by characterizing permutants and studying their prediction power. Next, we proposed an effective heuristic to select a good permutant candidate set. We also showed empirical evidence that supports our technique. The experimental results shows that our approach is very promising when retrieving objects from high dimensional spaces and also from real-world datasets.

In fact, in the Flickr real-world metric space, which contains 1 million of images represented with a vector of 219 dimensions, using 128 permutants, we

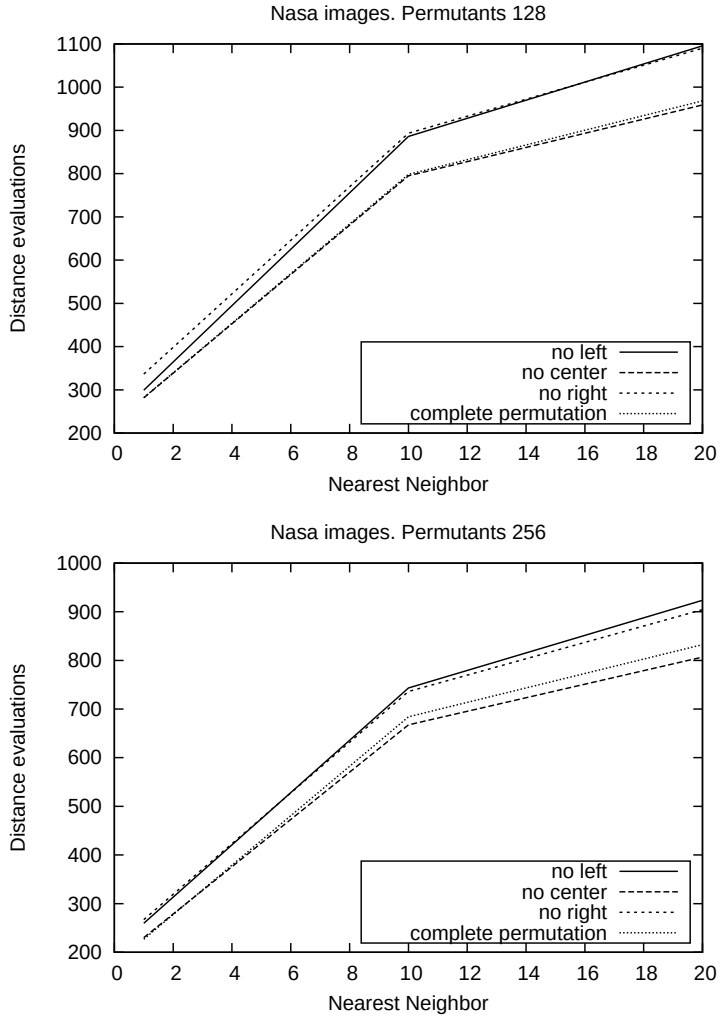

Fig. 4. Performance of our technique using real database (NASA images).

only need to review 1.15% of the dataset to obtain the nearest neighbor of a given query. We obtain a similar performance in the space of Nasa images (represented by 20-dimensional feature vectors), where, using a permutant set of size 128 and our criteria, we need to review 2.24% of the dataset in order to retrieve the 10 nearest neighbors of a given query.

In the future work, we are interesting on compressing the permutants through removing the middle of the permutations, as we state here. We are also very interested in how to characterize better the set of permutants. Finally, the high variance permutant set of Fig. 2 is composed by permutants that concentrate

their occurrences in some specific positions of the permutation. In future work we also study how to we refine this permutant set.

References

1. Bustos, B., Navarro, G., Chávez, E.: Pivot selection techniques for proximity searching in metric spaces. Pattern Recognition Letters 24(14), 2357–2366 (2003)
2. Bustos, B., Pedreira, O., Brisaboa, N.R.: A dynamic pivot selection technique for similarity search. In: Proc. 1st Workshop on Similarity Search and Applications (SISAP'08). pp. 105–112 (2008)
3. Chávez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI) 30(9), 1647–1658 (2009)
4. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquin, J.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (Sep 2001)
5. Fagin, R., Kumar, R., Sivakumar, D.: Comparing top k lists. SIAM J. Discrete Math. 17(1), 134–160 (2003)
6. Falchi, F., M.Kacimi, Mass, Y., Rabitti, F., Zezula, P.: SAPIR: Scalable and distributed image searching. In: SAMT (Posters and Demos). vol. 300, pp. 11–12. CEUR Workshop Proceedings (2007)
7. Hjaltason, G., Samet, H.: Index-driven similarity search in metric spaces. ACM Transactions Database Systems 28(4), 517–580 (2003)
8. Micó, L., Oncina, J., Vidal, E.: A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. Pattern Recognition Letters 15, 9–17 (1994)
9. Pedreira, O., Brisaboa., N.R.: Spatial selection of sparse pivots for similarity search in metric spaces. In: Proc. SOFSEM 2007: Theory and Practice of Computer Science. pp. 434–445. LNCS 4362, Springer (2007)
10. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann (2006)
11. Yianilos, P.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93). pp. 311–321 (1993)
12. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search – The Metric Space Approach, Advances in Database System, vol. 32. Springer (2006)