A hybrid approach to boost the permutation index for similarity searching

Karina Figueroa[1], Antonio Camarena-Ibarrola[1], Nora Reyes[2], Rodrigo Paredes[3], and Braulio Ramses Hernández Martínez[1]

[1] Universidad Michoacana
{karina.figueroa,antonio.camarena}@umich.mx,1578615h@umich.mx
[2] Universidad de San Luis, Argentina nreyes@unsl.edu.ar
[3] Universidad de Talca, Chile raparede@utalca.cl

Abstract. We propose a hybrid strategy that combines three ideas, namely, a convenient way for reducing the length of the permutations, using a permutation similarity measure adjusted for these clipped permutations, and the use of the closest permutant of each object as a pivot for it. In this way, we increase the discriminability of the permutation index in order to reduce even more the number of distance computations without reducing the answer quality. The performance of our proposal is tested using two classical real-world databases: NASA and Colors which are part of the SISAP project's metric space benchmark. We reduced more than 30% of the number of distance evaluations needed to solve the queries on both databases.

Keywords: Similariy search · Permutant-based index · Permutation similarity measures

1 Introduction

Nowadays, multimedia databases are widely used, and of course, the information retrieval is a crucial task. Similarity searching is the only operation that makes sense with this kind of data because two elements are never exactly the same. The similarity is a concept that depends on the database's domain, it is modeled and defined by experts of each field, and it is frequently expensive to compute in terms of arithmetic operations, I/O events, etc. Naturally, when a query is given, the goal is to answer it as quickly as possible. One way to achieve efficiency is to reduce the number of distance computations for answering a query.

There are two kinds of similarity queries, namely, *K-Nearest Neighbor query* $NN_K(q)$ and *Range query* $R(q,r)$. The $NN_K(q)$ retrieves the K database elements that are the most similar to q. The $R(q,r)$ finds the elements of the database whose distance to q is lower than or equal to the radius r.

One way to represent the problem is by mapping it to a metric space [5]. A metric space is a pair $(\mathbb{U}, d)$, where $\mathbb{U}$ is the universe of valid objects and d is a distance function that allows us to compare any two objects from $\mathbb{U}$. Let $\mathbb{X} \subseteq \mathbb{U}$ be the database of interest and $n = |\mathbb{X}|$. As we assume that the function d

is expensive to compute, our goal is to minimize the use of d when answering queries. One issue in this kind of problems is the intrinsic dimension [10] because when it is high, the distance between any pair of different objects tends to be the same, so searching complexity rises as the intrinsic dimension increases.

Assuming we cannot establish a total order over a multimedia database, we have to resort to using a proximity index. An index is a data structure that allows us to obtain a candidate list without sequentially scanning the entire database (unthinkable for huge datasets). There are three well-known index families, namely, the ones based on *pivots*, the ones based on *compact partitions*, and recently, the ones based on *permutations*. Pivot-based and compact-partition-based indexes are *exact* proximity indexes, while permutations-based ones are *approximate*; in the sense that we may lose a few relevant objects from the query answers, but accepting this loss allows us to improve dramatically the searching time.

In this paper, we propose a hybrid method to improve the performance of the *permutations*-based indexes, combining three main ideas: the first one is to conveniently reduce the length of the permutations stored within the index, the second is adapting the permutation similarity measure for these clipped permutations, and the third one is to use the closest permutant of each object as a pivot for it. This novel strategy allows us to improve the already remarkable performance of the permutation-based index when solving similarity queries.

The performance of our proposal is tested using two classical real-world databases: NASA and Colors, which are part of the SISAP project's metric space benchmark available at [8]. We reduce more than 30% of the number of distance evaluations needed to solve the queries on both databases.

The rest of this article is organized as follows: in Section 2 we describe the related work on metric spaces and similarity search. Section 3 shows our novel hybrid index and Section 4 gives its experimental evaluation using two real world datasets from SISAP library [8]. Finally, we expose conclusions and some possible extensions for this work in Section 5.

2 Related Work

Similarity searching in metric spaces has been studied in three leading families of algorithms: pivot-based algorithms [12, 5] (for low intrinsic dimension), partition-based algorithms [4, 6] (for medium to high intrinsic dimension), and permutation-based algorithms [3, 1] (for high intrinsic dimension). As we aforementioned, the permutation-based approach is one of the best representative methods to solve approximate similarity searches. In the following we briefly describe the pivot-based and permutation-based algorithms as they are relevant for this work.

2.1 Pivot-based algorithm

Pivot-based algorithms use a subset of database objects $P = \{p_1, p_2, \ldots, p_k\} \subseteq \mathbb{X}$ to compute pseudo-coordinates. Each database object $x \in \mathbb{X}$ is represented

by a vector containing its k distances to every pivot $p_i \in P$. Let $D(u, P) = (d(x, p_1), \ldots, d(x, p_k))$ be this vector. Given a query $R(q, r)$, we first represent q in the same coordinate system as $D(q, P) = (d(q, p_1), \ldots, d(q, p_k))$. Thus, by virtue of the triangle inequality, any object $x \in \mathbb{X}$ can be discarded if $|d(p_i, x) - d(p_i, q)| > r$ for any pivot $p_i \in P$. Finally, to obtain the query answer, all the non-discarded database objects are directly compared with q and only the objects whose distance is within threshold r are reported.

2.2 Permutation-based algorithm

This kind of indexes use some distinguished elements from the database $\mathbb{X}$ as references points of view. These elements are called *permutants*. The main idea of this method was introduced in [2]. Let $\mathbb{P}$ be the permutant set, formally, $\mathbb{P} = \{p_1, \ldots, p_k\} \subseteq \mathbb{X}$. For the sake of producing the index, each $u \in \mathbb{X}$ computes $D(u, \mathbb{P}) = \{d(u, p_1), \ldots, d(u, p_k)\}$, that is, u computes its distance to every permutant. Then, each object u sorts the set $\mathbb{P}$ using the distances computed in $D(u, \mathbb{P})$ in increasing order. This ordering is called the *permutation* of u, which is denoted by Π_u. Therefore, the permutant in the first position of Π_u is the closest one, and so on. Inversely, let Π_u^{-1} be the inverse of the permutation Π_u, so we can use Π_u^{-1} to identify the position of any permutant in Π_u.

As an example, Figure 1 depicts a subset of points in $\mathbb{R}^2$, considering Euclidean distance. The set of permutants is $\mathbb{P} = \{p_1, p_2, p_3, p_4, p_5, p_6\}$; that is, $k = 6$. If we consider the object $u \in \mathbb{X}$, $D(u, \mathbb{P}) = (3, 4, 6, 3, 2, 5)$ where $d(u, p_1) = 3$ and so on; then $\Pi_u = (5, 1, 4, 2, 6, 3)$. It can be noticed that the closest permutant is p_5, because $d(u, p_5) = 2$. The inverse permutation Π_u^{-1} is $(2, 4, 6, 3, 1, 5)$. Then, $\Pi_u^{-1}(p_2) = 4$ means p_2 is in the 4th position in Π_u. We note that we need $O(nk)$ distance computations to obtain all the permutations.

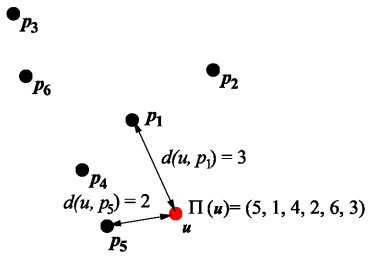


Fig. 1. Example of a permutation considering $\mathbb{P} \subset \mathbb{R}^2$ using Euclidean distance.

As two identical elements must have exactly the same permutation, we expect that two similar elements have similar permutations. Therefore, when we search for elements similar to a query q, the problem is to find objects whose permutations similar to Π_q. The advantage of this approach is that computing

the permutation similarity is cheaper than computing the *distance function d*. There are different measures to compute similarity between permutations [11]. One of the most used is the Spearman Footrule measure, defined as:

$$F(u,q)_k = F(\Pi_u, \Pi_q) = \sum_{i=1}^{k=|\mathbb{P}|} |\Pi_u^{-1}(p_i) - \Pi_q^{-1}(p_i)| \quad (1)$$

The basic method stores the whole permutation of each database object, hence it needs $O(nk)$ space.

An interesting member of this index family is the Metric Inverted File. In the next section we briefly describe it along with some of its improvements.

2.3 Metric Inverted File (MIFi)

Amato and Savino proposed using an inverted file of permutations [1], where each permutant in $\mathbb{P}$ has its respective entry in the inverted file. We call MIFi index this approach. To produce the index, they define parameter $m_i < |\mathbb{P}|$ which is used during the preprocessing time. For each permutant $p \in \mathbb{P}$ the MIFi index stores the list of the elements $u \in \mathbb{X}$ such that its permutation Π_u has the permutant p within the first m_i positions. The list for each permutant p stores pairs (u, pos), where u denotes an object in $\mathbb{X}$ and pos referes to the position of p within the permutation Π_u.

Given the query q, we need to determine Π_q. The MIFi index uses another parameter $m_s \ll m_i$ for searching. The MIFi search method only retrieves the posting lists of the first m_s permutants in Π_q and next, it unites all of them to obtain the candidate set. Finally, all the elements in the union of the lists are directly compared with q using the distance d to produce the query answer. Authors in [1] proposed a variant of the Spearman Footrule permutation similarity measure, because each permutation was clipped by the parameter m_i.

In the works [9, 7], the authors improved the performance of the MIFi index in two ways. On the one hand, each posting list stores only elements u but not the positions pos [9] and the short permutation of each element is maintained. They also proposed a new way to compute the Spearman Footrule measure. On the other hand, to reduce the candidate list size, even more, a new parameter m_{s_r} is selected according to the radius of the similarity query [7] (instead of the fixed-parameter m_s from the MIFi index).

3 Our proposal

We look to improve even more the performance of the alternative to MIFi index presented in [7]. Our proposal considers several aspects. The first one is to have smaller permutations in the index, the second one is to use one of the permutants as a pivot, and the third is to consider a modified permutation similarity measure. After explaining in detail these three main aspects of our contribution, we show how to combine them in order to produce our novel index and its respective searching algorithm.

3.1 Clipped permutations

Instead of having a maximum global length m_i, each permutation can be shortened with a different length, by considering for each $u \in \mathbb{X}$ its appropriate permutation prefix. To do so, we consider the distance to its closest permutant; that is, $d(u, p_{\Pi_u(1)}) = r_u$, and keep for u those permutants within a distance lower than or equal to $2r_u$. We ran preliminary experiments to determine that $2r_u$ performs well when solving queries, but this clearly deserves further study. We call Π'_u the clipped permutation of u.

Let m_u denote the length of the trimmed permutation for element u; that is, $m_u = |\Pi'_u|$ is the length of the prefix selected. In this case, since some objects could have only one permutant within distance $2r_u$ from u, we propose using a minimum global length $m_{\min}$ for all the permutations. Likewise, since some objects could have all the permutants within distance $2r_u$ from u, we also propose a maximum global length $m_{\max}$ for all the permutations.

In our example of Section 2.2, Figure 1, for element u we obtained $r_u = 2$. So, its clipped permutation is $\Pi'_u = (5, 1, 4, 2)$ because $d(u, p_2) = 4 \leq 2r_u$. It can be noticed that, by this way of shortening the permutations, each permutation would have a different length.

3.2 Including a single pivot

When we search for a query q, we have to compute Π_q by calculating all the distances between q and every permutant in $\mathbb{P}$. Besides, we know that if we keep the distances from the element $u \in \mathbb{X}$ to all the permutants in $\mathbb{P}$, we can use them to obtain lower bounds of the distance from u to q, as in a pivot-based algorithm. Hence, we can discard the elements whose lower bounds exceed the search threshold r. However, storing all the distances between the elements $u \in \mathbb{X}$ and the permutants $p \in \mathbb{P}$ is expensive.

We also know that a good pivot for estimating the distance from u to q is some element similar to u; so, we decided to use the permutant closest to u as its pivot. Then, we already have the pivot identifier and we only need to store the distance to it. Therefore, we only need one distance for each object $u \in \mathbb{X}$, which implies that we keep exactly n extra distances in the index, which is negligible for the index size.

Continuing with our example in Figure 1, the closest permutant of object u is p_5, so, we use it as pivot and store the distance 2.

3.3 Permutation similarity measure

Given a query q, we need to calculate all the distances between q and the permutants in $\mathbb{P}$ to obtain Π_q. At this point, we have the complete query permutation (with length k) and the distances $D(q, \mathbb{P})$. Thus, it is possible to compare any clipped permutation Π'_u with Π_q, using the same Equation 2 proposed in [7]:

$$F^*(u, q)_{m_u} = F^*(\Pi'_u, \Pi_q) = \sum_{i=1}^{m_u} |i - \Pi_q^{-1}(\Pi_u(i))| \tag{2}$$

If all the clipped permutations had the same size, we could directly use Equation 2, as it computes a similarity measure that is fair when all the permutation prefixes have the same size. However, this is not the case, thus we have to readapt the similarity measure considering different sizes of prefixes.

This adaptation corresponds to define mechanisms to apply penalties when we find a permutant that does not belong to Π'_u and, analogously, when we miss a permutant from the prefix of Π_q. Fortunately, they also improve the discriminability of our proposal, as can be seen in Section 4.

Penalty when a permutant does not belong to Π'_u Each permutant clipped from Π'_u adds a penalty that considers how big is the displacement of the remaining permutants in Π'_u with respect to their positions in Π_q. We call the maximum of all these displacements *maxi*. So, we add $maxi \cdot (k - m_u)$ to F^*.

Note that if the permutants in Π'_u are placed in the prefix of Π_q, this penalty is very mild. Also, the penalty increases as long as displacements are bigger.

Penalty when missing a permutant from the prefix of Π_q Two permutations starting with the same permutants give a strong suggestion that the respective objects could be similar. Likewise, if some of the permutants in the prefix of Π_q does not belong to Π'_u, we have a strong indicator that object u could be irrelevant to the query q.

So, we need to establish a criterion about what is this prefix. Analogously to Π'_u, considering the query radius r and using $D(q, \mathbb{P})$, we compute how many permutants have their distances from q within threshold $2r$. This value is called m_q, so the prefix of m_q permutants is called Π'_q. Notice that, if $m_q < m_{\min}$ then m_q is set to $m_{\min}$. Otherwise, if $m_q > m_{\max}$ then m_q is set to $m_{\max}$.

Therefore, we determine how many permutants in Π'_q are missing in Π'_u. We call this value c. Finally, as this is a strong indicator that u is not relevant to q, we strongly penalize the measure with $c \cdot F^*$. Of course, if all the permutants in Π'_q occur in Π'_u, this term is zero. But, the more the number of missing permutants in Π'_u, the greater the penalty (and each increment is also very strong).

Resulting permutation similarity measure We use Equation 2 and these two penalties in order to compute the permutation similarity measure. The obtained measure is depicted in Algorithm 1.

The variable t accumulates the similarity measure, c is initialized as m_q so we start by assuming that we miss all the permutant in Π'_q, and *maxi* is initialized as zero. Then, we compute a **for** cycle to review all the permutants in Π'_u (Lines 3 throu 9). Line 4 computes the displacement Δ_i for each permutant in Π'_u and accumulate it in t. Line 5 updates the value of *maxi*, when the displacement increases. In Line 6, we verify whether the permutant $\Pi_u(i)$ belongs to the prefix Π'_q, in whose case, we decrease c by 1 (Line 7), as we found another permutant within Π'_q. Finally, in Line 10 we apply the penalties and return the permutation similarity measure.

Algorithm 1 distanceBetweenPermutations($\Pi_q^{-1}, m_q, \Pi_u, m_u, k$)

```
1: OUTPUT: Reports modified Spearman Footrule.
2: t ← 0, c ← m_q, maxi ← 0
3: for i ← 1 to m_u do
4:    Δ_i ← |i − Π_q^{-1}(Π_u(i))|,   t ← t + Δ_i
5:    maxi ← max(Δ_i, maxi)
6:    if Π_q^{-1}(Π_u(i)) < m_q then
7:       c ← c − 1
8:    end if
9: end for
10: return  t ← t + maxi · (k − m_u) + c · t
```

3.4 Solving similarity queries

Given the dataset $\mathbb{X}$, we chose a subset of k objects at random to compute the permutations. Then, for each object $u \in \mathbb{X}$ we compute its permutation and its clipped version Π'_u, and we store both Π'_u and the distance to the closest permutant in the index.

Given a query q, we compute its permutation Π_q and its prefix Π'_q. Since we have the distance between u and its closest permutant, and we already compute $D(q, \mathbb{P})$ at querying time, then, we can calculate a lower bound of $d(u, q)$ with any permutant. Thus, for each u, $d(u, q)$ is lower bounded by $|d(u, p) - d(p, q)|$, using the closest permutant to u as a pivot. Then, if $|d(u, p) - d(p, q)| > r$ then u can be discarded from the candidate list. Only the non-discarded objects are included in the candidate list. We sort increasingly the candidate list according to our adapted permutation similarity measure. Next, a small portion of this list is traversed and compared with q using the distance function d.

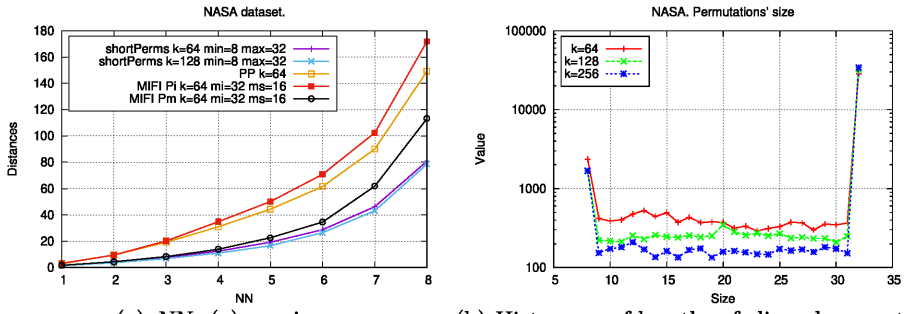
4 Experimental Results

The performance of our proposal has been tested using two classical real-world databases: NASA and Colors. These datasets are available from SISAP project's metric space benchmark set [8]. Any quadratic form can be used as a distance on these spaces, so we chose Euclidean distance as the simplest meaningful alternative for both databases.

4.1 NASA

NASA is a dataset with 20-dimensional vectors. They were generated from images downloaded from NASA[4] and there is not duplicate vectors. The total number of vectors is 40,150. The first 39,650 are indexed and the remaining 500 vectors are used as queries.

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



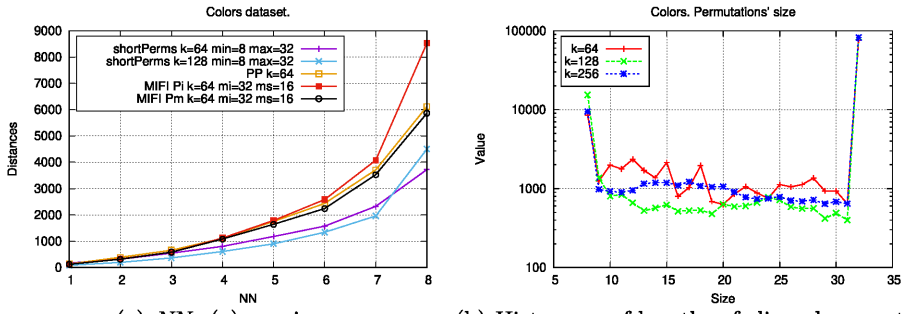
(a) $NN_K(q)$ queries. (b) Histogram of lengths of clipped permut.

Fig. 2. Performance with NASA dataset.

In Figure 2(a), we show the performance of our proposal (shortPerms) along with those of the basic permutation idea (PP, with 64 permutants), the MIFi using m_i for the missing values during the Spearman Footrule computation (MIFI Pi, using similar space of our index), and the MIFi modified as in [7] (MIFI Pm, using similar space of our index). These experiments account for how many distances are needed to obtain the true $NN_K(q)$ answer, varing $K \in [1, 8]$. Notice that our proposal makes 30% fewer distance computations than the best technique proposed in [7] for NN_8. In Figure 2(b), we show the histogram for different lengths of clipped permutations, considering that $m_{\min} = 8$ and $m_{\max} = 32$.

It is remarkable that using 64 permutants we can get clipped permutations with different m_u lengths and smaller than the original size (64), as shown in Figure 2(b). The average length of the clipped permutation is 27. We note that our index having almost half the space of PP with 64 permutants, behaves better. Moreover, when using a more extensive set of permutants (128 concerning 64), the searching costs are almost the same, since the clipped permutations have almost the same size. If we use more permutants, we increase the construction cost of the index. Therefore, it is not worth using more permutants because our proposal always leave only a few permutants that are good ones. In fact, in Figure 2(a) we omit the plot for $k = 256$ as the results are similar to those of $k = 64$ and 128.

During searches, our proposal outperforms the other variants that build the index with the same number of permutants and construction costs. This behavior may be not only due to a good clipped permutation but also to the pruning ability that gives the stored distance to the nearest permutant of each element.



(a) $NN_K(q)$ queries. (b) Histogram of lengths of clipped permut.

Fig. 3. Performance with Colors dataset.

4.2 Colors

This dataset consists of 112,682 color histograms represented as 112-dimensional feature vectors, from an image dataset [5]. Similarly with the NASA dataset, the first elements are indexed and the last 500 color histograms are used as queries.

In Figure 3(a), we show the performance of our proposal. Again, it is compared with the MIFi algorithms and the permutation-based algorithm (PP). In this dataset, our proposal needs 37% fewer distances than the best technique used in [7]. As it occurs in the NASA space, the number of permutants used does not significantly affect the search performance. On Figure 3(b), we show the histogram for each length of clipped permutations. Notice that all permutations have almost the same length for short permutations independent of the original permutation size, again with an average length of 27 permutants.

Newly, if we fix the number of permutants used to build the different alternatives of the indexes whose construction costs are the same, our proposal outperforms the others during searches.

5 Conclusions and Future Work

In this paper, we propose a new strategy for reducing the length of the permutations, which we call *clipped permutations*. We also propose a permutation similarity measure adapted for this clipping. Our approach also takes advantage of storing only one distance per database element, that is well selected, to obtain a lower bound of the distance between the element and the query. This stored distance allows for discarding many elements. This way we can use a smaller hybrid index and at the same time improving the search performance.

We have tested the performance of our proposal with two classical real-world databases: NASA and Colors, obtained from SISAP project's metric space

[5] Available at `http://www.dbs.informatik.uni-muenchen.de/~seidl/DATA/-histo112.112682.gz`.

benchmark set [8]. Our experimental results proved that our approach surpasses the other known permutation-based techniques. Therefore, the combination of these three contributions significantly improves searching performance of a permutation-based index for approximate proximity searching. As it can be noticed, we reduced more than the 30% the distance evaluations needed to solve the queries on both databases.

As future work, we will assess how scalable this method is, considering very large datasets. We will analyze how the values of $m_{\min}$ and $m_{\max}$ affect the lengths of the permutations and in consequence the impact on storage and search performance. Besides, we plan to study the actual prunning ability of the stored distances to the nearest permutant of each database element. Also, we consider investigating how the number of permutants used during the index construction affects the search performance considering other metric spaces.

References

1. Amato, G., Savino, P.: Approximate similarity search in metric spaces using inverted files. In: Proc. 3rd Intl. ICST Conf. on Scalable Information Systems, INFOSCALE 2008, Vico Equense, Italy, June 4-6, 2008. p. 28 (2008)
2. Chávez, E., Figueroa, K., Navarro, G.: Proximity searching in high dimensional spaces with a proximity preserving order. In: Proc. 4th Mexican Intl. Conf. in Artificial Intelligence (MICAI'05). pp. 405–414. LNAI 3789 (2005)
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.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters **26**(9), 1363–1376 (2005)
5. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Proximity searching in metric spaces. ACM Computing Surveys **33(3)**, 273–321 (2001)
6. Ciaccia, P., Patella, M., Zezula, P.: M-tree: an efficient access method for similarity search in metric spaces. In: Proc. 23rd Conf. on Very Large Databases (VLDB'97). pp. 426–435 (1997)
7. Figueroa, K., Camarena-Ibarrola, A., Reyes, N.: Shortening the candidate list for similarity searching using inverted index. In: Mexican Conf. on Pattern Recognition. vol. 12725, pp. 89–97. LNCS Springer (2021). https://doi.org/10.1007/978-3-030-77004-4_9
8. Figueroa, K., Navarro, G., Chávez, E.: Metric spaces library (2007), available at `http://www.sisap.org/Metric_Space_Library.html`
9. Figueroa, K., Reyes, N., Camarena-Ibarrola, A.: Candidate list obtained from metric inverted index for similarity searching. In: Martínez-Villaseñor, L., Herrera-Alcántara, O., Ponce, H., Castro-Espinoza, F.A. (eds.) Advances in Computational Intelligence. pp. 29–38. Springer International Publishing, Cham (2020)
10. Navarro, G., Paredes, R., Reyes, N., Bustos, C.: An empirical evaluation of intrinsic dimension estimators. Inf. Syst. **64**, 206–218 (Mar 2017). https://doi.org/10.1016/j.is.2016.06.004
11. Skala, M.: Counting distance permutations. J. of Discrete Algorithms **7**(1), 49–61 (Mar 2009). https://doi.org/10.1016/j.jda.2008.09.011
12. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach, Advances in Database Systems, vol. 32. Springer (2006)