The List of Clusters Revisited

Eric Sadit Tellez and Edgar Chávez

Universidad Michoacana de San Nicolás de Hidalgo, México
sadit@lsc.fie.umich.mx, elchavez@fismat.umich.mx

Abstract. One of the most efficient index for similarity search, to fix ideas think in speeding up k-nn searches in a very large database, is the so called *list of clusters*. This data structure is a counterintuitive construction which can be seen as extremely unbalanced, as opposed to balanced data structures for exact searching. In practical terms there is no better alternative for exact indexing, when every search return all the incumbent results; as opposed to approximate similarity search. The major drawback of the list of clusters is its quadratic time construction.

In this paper we revisit the list of clusters aiming at speeding up the construction time without sacrificing its efficiency. We obtain similar search times while gaining a significant amount of time in the construction phase.

1 Introduction

Many pattern recognition and data mining tasks can be stated in terms of proximity searching. In its more general abstraction, the similarity searching problem consist on find a set of items following a given constraint over an involved query, using a distance function as the only information source available, that is, the internal object structure cannot be used. Under this statement of the problem, it is possible to design and created fast proximity searching structures, called proximity indexes. Those indexes are general enough to be used careless of the specific description of a database.

Unfortunately, most of these indexes are not capable to scale on the growth of the databases, such that only a few hundred of thousand items can be handled even on modern hardware. This issue is particularly noticeable when objects are defined on high intrinsic dimensional spaces, as described by Chavez and Navarro [1].

Unlike exact searching, the performance of the indexes depends heavily on the data, and the complexity bounds cannot be established without considering the intrinsic dimensionality of the data collection.

The above dependency have been documented thoroughly in the literature, including several books and surveys [2,1,3]. This condition is called the *curse of dimensionality* (CoD), that in practice imply that a clever index cannot outperform a sequential scan if the data is intrinsically high dimensional.

From a practical point of view, we can classify the indexes as effective on a specific range over the intrinsic dimensionality line. A long standing index, with asymptotic optimal performance on the number of computed distances, is AESA [4,5] which can be seen as a pivot-based index using all the database objects as pivots. Nevertheless, in practice AESA is restricted to very small datasets, since it has a quadratic dependance on the size of the database, for both preprocessing time and memory requirements. Also, a hidden complexity is present even on query time, i.e. $O(n^2)$ basic operations (arithmetic and logical) are required (additional to the computed distances). In general,

J.A. Carrasco-Ochoa et al. (Eds.): MCPR 2012, LNCS 7329, pp. 187–196, 2012.

pivot based indexes with linear number of pivots such as LAESA [5] can trade speed at query time for the size of the index. One way to obtain a good speed/space tradeoff is by using a compact index such as the Fixed Queries Array (FQA) [6], or the Fixed Queries Trie [7].

Another way to cope with the space usage is the technique of the List of Clusters [8], which is faster than LAESA or the FQA for any practical space bounds, specially when the data is high dimensional. Table 1 compares complexities among search-efficient indexes, here ρ is the number of bits used to represent a distance, m the number of centers in the LC, ℓ the number of pivots used by LAESA, and α is a fixed value between 0 and 1 that depends on the intrinsic dimension of the database. Summarizing, indexes that allow fast searches are highly expensive at the preprocessing step and/or in memory requirements.

Table 1. Complexities of the most faster proximity searching algorithms for a fixed dimensionality dataset

method	preprocessing distances	searching distances	memory
List of clusters (LC)	$O(n^2)$	$O(n^\alpha)$	$O(n \log n + m\rho)$ bits
AESA	$O(n^2)$	$O(1)$	$O(n^2\rho)$ bits
Linear AESA (LAESA)	$O(n\ell)$	$O(n^\alpha)$	$O(n\ell\rho)$ bits

In this paper we propose a new index for metric searching allowing fast searches, using $O(n^\alpha)$ distances per query, and $O(n \log n + m\rho)$ bits of space, and a preprocessing time of $O(nm^\alpha)$, with $\alpha \leq 1$. Our index is inspired on the LC data structure, and use a similar searching procedure hence it can be plug into applications already using the LC without modification.

The key difference of or approach is in the construction phase. The LC have a quadratic construction time which prevents its usage in large databases, probably in the same way AESA cannot be used in practice. With our proposal it is possible to index large databases. Furthermore our index can be built in parallel, making efficient use of modern hardware. So, our approach is faster to build than the original LC with a very small penalty in the searching complexity, that makes the index capable to deal with complex search pattern analysis and data mining procedures even on very large datasets.

1.1 Preliminaries

A metric space is a pair (U, d), with U a set and $d(\cdot, \cdot)$ a distance function $d : U \times U \to \mathbb{R}^+$ with the usual metric properties, $\forall u, v, w \in U$, $d(u, v) \geq 0$ with $d(u, v) = 0 \iff u = v$, $d(u, v) = d(v, u)$, and $d(u, w) + d(w, v) \geq d(u, v)$. These properties are known as strict positiveness, symmetry, and the triangle inequality, respectively. A database S is a finite subset of U, $S \subseteq U$ of size $n = |S|$. Proximity search can be formulated in terms of two operations, namely

- *k nearest neighbor query*. Retrieve the k closer elements of a query q in S. $\mathsf{k\text{-}nn}(q) = \{u \mid d(q, u) \leq d(q, v)\ \forall\ v \in S\}$ subject to $|\mathsf{k\text{-}nn}(q)| = k$. Ties are arbitrarily broken.
- *range r query*. Obtain all the objects in S which distance to q is within the range r, i.e. $(q, r)_d = \{u \in S \mid d(q, u) \leq r\}$.

1.2 Pivot Based Indexes

The purpose of an index is to avoid a sequential scan. The pivot trick consist in filtering the database S by using repeatedly the triangle inequality to bound the distance from an object to the query. A set of distinguished points $P = \{p_1, p_2, \cdots, p_m\} \subseteq U$ (the pivots) are used to define a filtering distance, always bounded from above by the original distance d. Let $D(u, v) = \max_{1 \leq i \leq m} |d(u, p_i) - d(v, p_i)|$. Using the triangle inequality, it is inmediate $D(u, v) \leq d(u, v)$ and hence it implies $(q, r)_d \subseteq (q, r)_D$. Notice that D is a well known metric, i.e. the Minkowski's ∞ norm. Nevertheless, it is well known that the pivot space cannot be easily indexed with a vector space index, since P has a very large dimensionality. Chavez and Navarro [1] nicely survey pivot indexes and its core properties.

The index retrieve $(q, r)_D$ using only m distance computations, just distances to the pivots. When the intrinsic dimension of the data set is high, the CoD implies that even a significant increase in the number of pivots (say to the limits of the available memory to store the distance matrix), barely decreases $(q, r)_D \setminus (q, r)_d$. For easier instances of metric spaces the decrease may be significant, implying that a pivot based index will have a single parameter for the end user, i.e., if the time to get an answer is not satisfactory, then she simple increases the number of pivots.

The above simple rule can be used as long as the cost of obtaining $(q, r)_D$ and the amount of memory used to maintain the distances is bounded. A plain table of ℓ pivots is neither efficient for processing $(q, r)_D$ nor efficient in space usage. In a tree data structure as the Fixed Height Fixed Queries tree [9] the time is sublinear but this comes with the overhead of maintaining pointers in addition to the ℓn distances. Other pivot based tree data structures, like the Vantage Point tree (VPT) [10] or the Burkhard-Keller tree (BKT) [11], cannot use more pivots because they can only represent as much distances as the path length (the sum of the paths from every node to the root). One interesting alternative is the Fixed Queries Array (FQA) [6] which uses a few bits per pivot, and a logarithmic penalty over a tree data structure.

The limit in the number of pivots usable for indexing is the size of the database (unless pivots outside the database are used for indexing). Taking all the objects in the database is precisely what the AESA algorithm does [4,5]. This algorithm is optimal because the number of distance computations to obtain the nearest neighbor search is $O(1)$ for a fixed dataset. The algorithm is useful only for small databases because storing $O(n^2\rho)$ bits for distances is fairly unpractical.

1.3 Compact Partition Indexes

Another approach to proximity searching consist in partitioning the database in compact regions. Most of the compact partitioning indexes in the literature are hierarchical, with a recursive rule as follows: A set of centers $c_1, c_2, \cdots, c_m \in S$ is selected per node, such that every c_i is the center of a T_i subtree. The set of centers are used to partition the database such that each T_i is spatially compact. For example, $u \in T_i$ if $i = \arg\min_{1 \leq j \leq m} d(u, c_j)$. The covering radius $cov(c_i) = \max_{u \in T_i} d(c_i, v)$ is stored for each node. This construction is applied recursively. A query $(q, r)_d$ is solved recursively starting from the root node. If $d(q, c_i) \leq r$ then $c_i \in (q, r)_d$, and T_i must be explored if $|d(q, c_i) - cov(c_i)| \leq r$.

In general the recursion can be stopped at any level, and objects below that level in each node are stored together in a bucket. In this type of indexes there is not a simple

recipe as in the case of pivot indexing. Chavez and Navarro [1], surveys a great variety of compact partition indexes, and its common behavior.

The List of Clusters. A surprising data structure is the List of Clusters (LC). Using a linear amount of identifiers, is able to outperform all other indexes, specially when the data is high dimensional. The LC unbalances a tree data structure until it resembles a linked list. This strategy contrasts with that followed on exact searching; where achieving balance is a paramount and a myriad of balancing algorithms exist. In the case of approximate searching, an unbalanced structure has be proven to be useful. The drawback of the approach is a quadratic construction time and have the same origin of its unmatched performance.

Let explore with more detail the construction and the searching algorithm for the LC, as introduced by Chavez and Navarro [8]. Define $I_{S,c,cov(c)} = \{u \in S \setminus \{c\} \mid d(c,u) \leq cov(c)\}$ as the bucket of *internal* elements, which lie inside center ball of c, and $E_{S,c,cov(c)} = \{u \in S \mid d(c,u) > cov(c)\}$ as the rest of the elements (the *external* ones). Now the process is repeated recursively inside E. The construction procedure returns a list of triplets (c_i, r_i, I_i) (center,radius,bucket) and it is shown in algorithm 1.

ALGORITHM 1. The construction algorithm of the LC. The operator :: is the list constructor. It is not hard to remove the tail recursion to make it iterative.

Input: The set of objects to be indexed, m.
Output: The list of clusters.
Build(S)

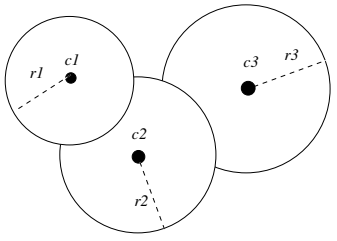
```
1: if S = ∅ then
2:     return empty list
3: end if
4: Select c ∈ S
5: Select a radius cov(c)
6: I ← {u ∈ S \ {c}, d(c,u) ≤ cov(c)}
7: E ← S \ I
8: return (c, cov(c), I)::Build(E)
```

Please note that the number of centers in the algorithm 1 is unknown beforehand. There are two possible parameters, the number of objects inside a ball, or the radius of the ball. This defines in a intrinsic way the number of centers. As proposed in the original paper, we select the number of centers, i.e. n/m, as a simple way to select the required parameters.

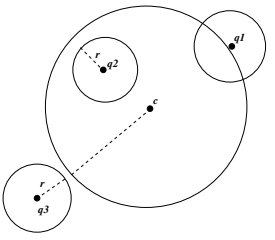
The searching procedure is described in algorithm 2. When the intrinsic dimensionality of the data is high, then most of the balls need examination. To bound the searching complexity in [8] the authors used probabilistic arguments to show the complexity of the searching, showing it must be $O(n^{\alpha})$ distance computations, for some $\alpha \leq 1$ which depend on the distribution of the data.

1.4 Our Contribution

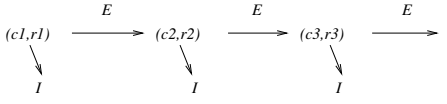
This paper introduces a simple and effective metric index, the *Reverse Nearest Neighbor* List of Clusters (**Rev-LC**). The preprocessing time of Rev-LC is way smaller than the



(a) The influence zones of three centers taken in this order: c_1, c_2, c_3.



(b) The three cases of query ball versus center ball.



(c) The list arrangement for the data structure.

Fig. 1. Illustrations of the LC's data structure and the querying procedure. In all figures $cov(c_i) = r_i$. The cases of querying are showin in figure 1(b), the case depicted by q_1 requires to consider the current bucket and the rest of centers. For q_2 we can prune the search inside the rest of the partitions. For q_3 we can avoid considering the current bucket.

ALGORITHM 2. The searching algorithm. The main loop (line 2) visits triples in the order found at L.

Input: The list of clusters L, the query $(q, r)_d$.
Output: The result set R.

```
1: Let R ← ∅
2: for all (c, cov(c), I) ∈ L do
3:    Let d_cq = d(c, q)
4:    R ← R ∪ {c} if d_cq ≤ r
5:    if d_cq ≤ cov(c) + r then
6:       for all u ∈ I do
7:          R ← R ∪ {u} if d(u, q) ≤ r
8:       end for
9:    end if
10:   stop loop if d_cq < cov(c) − r
11: end for
```

LC in most cases. The central idea is to select the centers beforehand (say m of them) instead of obtaining them in the recursive construction of the list.

Our construction time is in worst case $O(nm)$, but it is likely to achieve $O(nm^\alpha)$ for some $\alpha \leq 1$.

Since all the centers are selected beforehand, once certain order is established, populating the balls around each center can be done in parallel. This particular feature make the index suitable for taking advantage of modern hardware. We conducted a thorough experimentation to demonstrate the efficiency of the new index.

2 Revisiting the List of Clusters

The searching algorithm depicted in figure 1(b) can work with *any* partition of the database. It should be clear that if we have an arbitrary partition in the mathematical sense, i.e. $S = \cup I_i$ and $I_i \cap I_j = \emptyset$, then querying each one of the partition elements I_i is equivalent to searching the entire database S. The key is to avoid buckets not being relevant to the query. As pointed out in [1] in this schema we can fit most of the indexes, the difference consists on the rule to discard partition elements.

If we want to reduce the $O(n^2)$ construction complexity of the LC then we need to examine the origin of the problem. Please notice that in the LC, illustrated in figure 1, the next center is selected from the unassigned objects, and that the tail recursion is responsible for the quadratic behavior.

A solution is to choose m centers beforehand and define a Dirichlet domain, just as if it where a GNAT [12] of one level with very large arity. For convenience, we unzip LC's triplets into its three components, i.e. the centers $C = c_1, c_2, \cdots, c_m$, the buckets $I = I_1, I_2, \cdots, I_m$, and the covering radii $COV = cov(c_1), cov(c_2), \cdots, cov(c_m)$. In order to simplify algorithms, both I and COV are indexed with its entry number i, and with the corresponding c_i center.

ALGORITHM 3. Construction of the Rev-LC

Input: The number of centers, m.
Output: The Rev-LC index, i.e. C, I's, and COV.

```
1: C is initialized selecting m random centers from S
2: for all i = 1 to m do
3:     I_i ← ∅
4:     COV_i ← 0
5: end for
6: for all u ∈ S \ C do
7:     Let c_i to be the nearest neighbor of u in C (ties are arbitrarily broken).
8:     I_i ← I_i ∪ {u}
9:     COV_i ← max {COV_i, d(c_i, u)}
10: end for
```

The construction of Rev-LC is depicted by algorithm 3. If we assume no ties for the nearest neighbors (NN), an alternative succinct definition is $I_c = \{u \in \{S \setminus C\} \mid NN(u, C) = c\}$, and $COV_c = \max \{d(c_i, u) \mid u \in I_c\}$, i.e., buckets are populated with the reverse neighbors of each $c \in C$.

The searching procedure is very similar to the original LC since the discarding rule is the same. We can apply the searching algorithm 2, only avoiding the last condition (line 10). This is because the Rev-LC divides S into m regions, contrary to the recursive binary division of the LC.

Assuming C is large enough to resemble the distribution of S, and that the value of $cov(c_i)$ on average corresponds to an small percentile of the cumulative distribution function of the distances, we would have the same performance conditions of the LC, and hence theorem 1 also holds.

Theorem 1 (Chavez and Navarro [8]). *The number of distance computations performed by the (R-)LC to solve some query is $O(n^\alpha)$ for some $\alpha \leq 1$.*

Notice that to follow this theorem we require $m = O(n^\beta)$, for some $\beta < 1$, and in fact, close to 1, such that $n/m = O(n^{1-\beta})$ is quite small, and enough to produce compact buckets (in the radii sense). Nevertheless, the resulting α is basically larger than that exposed by the LC. Based on this theorem, we can produce the following conclusion about the construction of the Rev-LC.

Theorem 2. *The preprocessing cost of the Rev-LC is* $O(nm^\alpha + m^3/n)$ *for some* $\alpha \leq 1$.

Proof. From algorithm 3, the preprocessing cost for *very* high intrinsic dimensional datasets requires $O(nm - m^2)$ distance computations, which is a very pessimistic assumption. For the construction we need $n - m$ nearest neighbor searches over C. We can use the Rev-LC index for this smaller set. By theorem 1, we need n nearest neighbor searches in the smaller set C with cost $O(m^\alpha)$. The additional $O(m^3/n)$ term comes from the construction of of the Rev-LC index for C, but this time using a sequential search to retrieve the NN and using the same proportion of centers. □

Please notice that the term m^3/n should be small enough, i.e. $m^3/n < nm/2$, follows that $n/m > \sqrt{2}$. If $\gamma = n/m$ this can be expressed as $n^3/(n\gamma^3) \ll n^2/(2\gamma)$, yielding to a simplified formulae $n^2/\gamma^3 \ll n^2/(2\gamma)$.

Even if $\gamma = O(1)$ the cost would be small enough. For example, if $\gamma = 12$ (i.e. a suggested value of the bucket size for high intrinsic dimensional data sets according to [8]), we obtain that $n^2/1728 \ll n^2/24$. Thus, the $O(m^3/n)$ overhead is negligible, we can take only the significant term $O(nm^\alpha)$ as the processing time for building the Rev-LC index for S.

The parallelization of the construction is straightforward, unlike the LC algorithm. A simple modification of the algorithm 3 is required at line 6, here we must search the nearest neighbor in C in parallel, line 7. Finally, the rest of the lines inside the loop must be serialized. We call this version of the algorithm as Parallel Rev-LC (PRev-LC).

3 Description of Datasets

Due to space restrictions we show only experiments for synthetic datasets, showing the performance as a function of the dimension of the data and the size of the database.

It is worth noticing that even if our datasets are vector spaces, we are not using the coordinates to discard elements. We use the distance as a black box. This allows to work with the data disregarding its representation, all we need is a distance function to index the data.

We use six databases of randomly generated vectors in the unitary hypercube are used. We generate $n = 10^6$ random vectors of 4, 8, 12, 16, 20, and 24 dimensions. We averaged two hundred nearest neighbor queries in the plots. Each query object is a randomly generated vector of the dimension of the dataset.

4 Experiments

All the algorithms were written in C#, with the Mono framework[1]. Algorithms and indexes are available as open source software in the *natix* project[2]. The experimentation

[1] `http://www.mono-project.org`
[2] `http://www.natix.org`

was executed in a four quad-core Intel Xeon 2.40 GHz workstation with 32 GiB of RAM, running CentOS Linux. The entire databases and indexes are maintained in main memory and without exploiting any parallel capabilities of the workstation, excepting for the PRev-LC. On the parallel version, the setup was left to the default configuration of the *parallel tasks* of the mono framework.

We present an experimental comparison of our Rev-LC index against the LC, in both preprocessing and querying time. As it is customary we count the number of distances computed in each one of them to compare. We also measured the total time elapsed in the construction.

4.1 Construction Time

We selected a cheap distance for testing the construction. Since we will count the number of distance computations this is a fair benchmark. Our choice is a four dimensional dataset of one million vectors. The results are reported in table 2. The time to build the LC is more than twice larger than the Rev-LC for n/m of 1024 and 128, this grows to 17.5 times for $n/m = 16$, it is evident the advantage of our method. The parallel algorithm (PRev-LC) have the faster preprocessing times, it runs close to 10 times faster than the LC for $n/m = 1024$, and more than 46 times faster for $n/m = 16$.

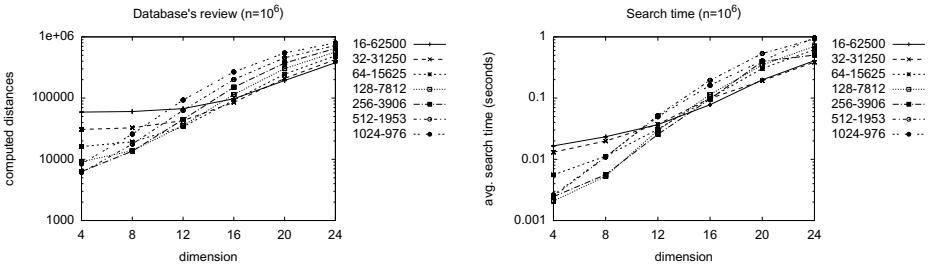
Table 2. Construction time for random vectors of dimension 4 and $n = 10^6$

method	n/m	m	preprocessing time	
			seconds	human readable
LC	1024	976	331.13	5 min 31.13 sec.
LC	128	7812	2056.5	34 min 16.52 sec.
LC	16	62500	16163.16	4 hours 29 min.
Rev-LC	1024	976	168.65	2 min 48.65 sec.
Rev-LC	128	7812	895.41	14 min 55.41 sec.
Rev-LC	16	62500	920.61	15 min 20.61 sec.
PRev-LC	1024	976	35.50	35.50 sec.
PRev-LC	128	7812	327.57	5 min 27.57 sec.
PRev-LC	16	62500	348.11	5 min 48.11 sec.

4.2 Search Performance

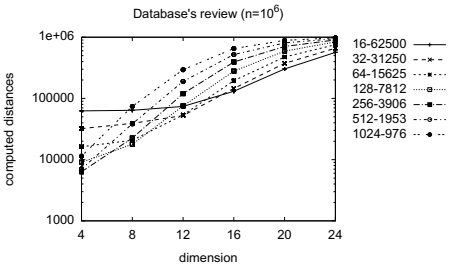
In the figure 2, the number of distance computations to retrieve the nearest neighbor in different intrinsic dimensional datasets is shown. The LC have fixed bucket size while the Rev-LC has a fixed number of centers, m. Each plot title is the bucket size (or the expected bucket size in the case of Rev-LC), and its number of centers (in this order).

From figure 2 we learn that there is a small penalty in the Rev-LC when compared to the LC. The number of distances is shown in figures 2(b) and 2(d), for LC and Rev-LC respectively. We must notice that the behavior of the Rev-LC is more faithful to the LC for larger m (smaller n/m values), this is an effect of Theorem 1. Furthermore, the real time behavior is more tight than the cost driven on counting distance computations. Nevertheless, we must encourage that the Rev-LC never reaches the performance of the LC, but this can be improved by increasing the number of centers, and hence we can make the difference as small as desired. In addition, the faster preprocessing step makes the Rev-LC and PRev-LC a competitive real option for large datasets.

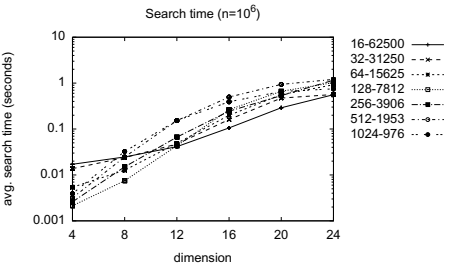


(a) LC's review of the database

(b) LC's search time



(c) Rev-LC's review of the database



(d) Rev-LC's search time

Fig. 2. Behavior of the Rev-LC and the LC for the nearest neighbor search for increasing intrinsic dimension

5 Conclusions and Future Work

We presented a new index for general metric spaces. This index uses a similar searching algorithm of the well known List of Clusters. The key difference is a better construction time and the trivial parallelization of the index. This last feature makes the index suitable to exploit better the available power of modern hardware.

There is also immediate use for metric join operations as the one described in [13]. In addition, a dynamic index, i.e that supporting insertions and deletions, seems to be simpler to implement for the Rev-LC than for the LC, since the invariant is easier to maintain. Also, the dynamic Rev-LC promises to be useful even on highly transactional environments. Finally, we are working on the parallelization of the searching process too, trying to maximize the exploitation of the parallel capabilities available in modern multicore architectures.

References

1. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. 33(3), 273–321 (2001)
2. Samet, H.: Foundations of Multidimensional and Metric Data Structures, 1st edn. The Morgan Kaufman Series in Computer Graphics and Geometic Modeling. Morgan Kaufmann Publishers, University of Maryland at College Park (2006)

3. Hjaltason, G.R., Samet, H.: Index-driven similarity search in metric spaces (survey article). ACM Trans. Database Syst. 28(4), 517–580 (2003)
4. Vidal Ruiz, E.: An algorithm for finding nearest neighbours in (approximately) constant average time. Pattern Recognition Letters 4, 145–157 (2005)
5. Micó, M.L., Oncina, J., Vidal, E.: A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. Pattern Recogn. Lett. 15, 9–17 (1994)
6. Chávez, E., Marroquin, J., Navarro, G.: Fixed queries array: A fast and economical data structure for proximity searching. Multimedia Tools and Applications (MTAP) 14(2), 113–135 (2001)
7. Chávez, E., Figueroa, K.: Faster Proximity Searching in Metric Data. In: Monroy, R., Arroyo-Figueroa, G., Sucar, L.E., Sossa, H. (eds.) MICAI 2004. LNCS (LNAI), vol. 2972, pp. 222–231. Springer, Heidelberg (2004)
8. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recogn. Lett. 26, 1363–1376 (2005)
9. Baeza-Yates, R., Navarro, G.: Fast approximate string matching in a dictionary. In: Proc. 5th International Symposium on String Processing and Information Retrieval (SPIRE), pp. 14–22. IEEE CS Press (1998)
10. Uhlmann, J.: Satisfying general proximity/similarity queries with metric trees. Information Processing Letters 40(4), 175–179 (1991)
11. Burkhard, W., Keller, R.: Some approaches to best-match file searching. Communications of the ACM 16(4), 230–236 (1973)
12. Brin, S.: Near neighbor search in large metric spaces. In: Proceedings of the 21th International Conference on Very Large Data Bases, VLDB 1995, pp. 574–584. Morgan Kaufmann Publishers Inc., San Francisco (1995)
13. Paredes, R., Reyes, N.: Solving similarity joins and range queries in metric spaces with the list of twin clusters. J. of Discrete Algorithms 7, 18–35 (2009)