Enlarging Nodes to Improve Dynamic Spatial Approximation Trees [*]

Marcelo Barroso
Departamento de Informática
Universidad Nacional de San Luis
Argentina
mabarros@unsl.edu.ar

Nora Reyes
Departamento de Informática
Universidad Nacional de San Luis
Argentina
nreyes@unsl.edu.ar

Rodrigo Paredes
Departamento de Ciencias de la Computación
Universidad de Talca
Curicó, Chile
raparede@utalca.cl

ABSTRACT

The *metric space model* allows abstracting many similarity search problems. Similarity search has multiple applications especially in the multimedia databases area. The idea is to index the database so as to accelerate similarity queries. Although there are several promising indices, few of them are dynamic, i.e., once created very few allow to perform insertions and deletions of elements at a reasonable cost.

The *Dynamic Spatial Approximation Trees* (*DSA–trees*) have shown to be a suitable data structure for searching high dimensional metric spaces or queries with low selectivity (i.e., large radius), and are also completely dynamic. The performance of *DSA–trees* is directly related to the amount of backtracking in search time. To boost the performance in this data structure a sufficient condition is to maintain in the nodes elements close-to-each-other. In this work we propose a new data structure for searching in metric spaces, based on the *DSA–tree*, which holds its virtues and takes advantage of element clusters, which are present in many metric spaces, and can also make better use of available memory to improve searches. In fact, we use these element clusters to improve the spatial approximation.

Categories and Subject Descriptors

H.3.1 [**Information storage and retrieval**]: Content analysis and indexing—*indexing methods*; H.3.3 [**Information storage and retrieval**]: Information Search and Retrieval —*search process*

General Terms

Algorithms, Experimentation

[*]Partially Supported by PPUA Project 01-10-269 and Fondo de Desarrollo de Investigación, Universidad de Talca.

Keywords

Metric space searching, DSA–trees

1. INTRODUCTION

Similarity search is a suitable way to find in a database any kind of unstructured data and has applications in a vast number of fields. Some examples are next generation databases (e.g. storing images, fingerprints, or audio clips), text searching, information retrieval, machine learning and classification, image quantization and compression, computational biology, and function prediction. These applications have some common characteristics, where one of the main ones is that the concept of exact search is of no use and we search instead for similar objects.

The similarity search problem can be formally defined through the concept of *metric space*, which provides a formal framework that is independent of the application domain. There is a universe U of objects and a nonnegative *distance function* $d : U \times U \longrightarrow \mathbb{R}^+$ defined among them. This distance satisfies the three axioms that make the set a *metric space*: *strict positiveness* ($d(x, y) = 0 \Leftrightarrow x = y$), *symmetry* ($d(x, y) = d(y, x)$) and *triangle inequality* ($d(x, z) \leq d(x, y) + d(y, z)$). The smaller the distance between two objects, the more "similar" they are. We have a finite database $S \subseteq U$, which is a subset of the universe of objects and can be preprocessed (to build an index, for example). Later, given an object from the universe (a query q), we must retrieve all similar elements in the database. There are two typical queries of this kind:

Range query: Retrieve all elements within distance r to q in S. This is, the set $\{x \in S, d(x, q) \leq r\}$.

Nearest neighbor query (k-NN): Retrieve the k closest elements to $q \in S$. That is, a set $A \subseteq S$ such that $|A| = k$ and $\forall\ x \in A, y \in S - A, d(x, q) \leq d(y, q)$.

In this paper we are devoted to range queries, as nearest neighbor queries can be rewritten as range queries in an optimal way [8]. The distance is considered expensive to compute (think, for instance, in comparing two fingerprints). Hence, 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.

Since, the computational cost of determining the similarity among objects is known to be a significant part of the total running time, a number of data structures and algorithms have been devised to deal efficiently with large collections of data [13, 12, 5]. Given a database of $n = |S|$ objects, queries can be trivially answered by performing n distance evaluations. The goal is to build an index of the database to speed up queries, avoiding the exhaustive search and computing a minimal amount of distances. All those structures work on the basis of discarding elements using the triangle inequality, and most of them use the classical divide-and-conquer approach (which is a general algorithmic approach).

There are effective methods to search on D–dimensional spaces considering that the distance function belongs to the Minkowski's distance function family $L_p = (\sum_{1 \le i \le d} |x_i - y_i|^p)^{1/p}$, such as *kd*–trees [2, 3] but for roughly 20 dimensions or more those structures cease to work well. For a survey on these methods see [7]. We focus in this paper in general metric spaces, although the solutions are well suited also for D–dimensional spaces. It is interesting to notice that the concept of "dimensionality" is related to "easiness" or "hardness" of searching a D–dimensional space: higher dimensional spaces have a probability distribution of distances among elements whose histogram is more concentrated and with larger mean. This makes the work of any similarity search algorithm more difficult. We extend this idea, following [5], by saying that a general metric space is high dimensional when its histogram of distances is concentrated.

Among all the techniques for metric space indexing we are interested in the *dynamic* data structures, where the database is unknown beforehand and the objects arrive to the index at random times as well as the queries. Static data structures may benefit from the *full knowledge* of the database to select the best reference points for a particular data structure. A dynamic data structure cannot make such strong assumptions about the database and will not have statistics about all the database.

The *Dynamic Spatial Approximation Tree* (*DSA–tree*) is a recently proposed data structure for searching in metric spaces [11], based on a novel concept: rather than dividing the search space, approach the query spatially, that is, start at some point in the space and get closer and closer to the query [10]. The *DSA–tree* behaves better than the other existing data structures on metric spaces of high dimension or queries with low selectivity, which is the case in many applications. This index is fully dynamic and is incrementally built via insertions. As such, the tree root will be the first object arriving, and this is repeated recursively at every level in the tree. The *DSA–tree* supports insertion and deletion of elements and has proven to be competitive in all dimensionalities but unable of taking advantage of the available memory.

Unlike some other metric data structures [5, 4], the (*DSA–tree*) does not take advantage if the metric space has clusters, or can improve the search at the expense of using more memory. Our proposal, based on *DSA–tree* is still dynamic, competitive, and makes better use of the available memory (however, it still cannot improve by using more memory). If the *DSA–tree* grouped objects that are very close to each other it could achieve better search performance by having to do less backtracking. We propose a new data structure that performs the spatial approximation on groups of objects or *clusters*, rather than individual objects, and thus reduces search costs.

2. PREVIOUS WORK

Algorithms to search in general metric spaces can be divided into two large areas: *pivot-based* algorithms and *compact partition-based* ones. Pivot-based algorithms are better suited for low dimensional metric spaces, while compact partitions ones deal better with high dimensional metric spaces. Although the former can improve by using more memory, they need more and more memory to beat the latter as dimension grows. On the other hand, indices based on compact partitions use a fixed amount of memory and cannot be improved by giving them more space. However, there are algorithms that combine ideas from both areas. See [12, 13, 5, 9] for more complete surveys.

Pivot-Based Algorithms.

The idea is to use a set of k distinguished elements ("pivots") $p_1 \ldots p_k \in S$ and storing, for each dataset element x, its distance to the k pivots $(d(x, p_1) \ldots d(x, p_k))$. Later, given the query q, its distance to the k pivots is computed $(d(q, p_1) \ldots d(q, p_k))$. Now, if for some pivot p_i it holds that $|d(q, p_i) - d(x, p_i)| > r$, then we know by the triangle inequality that $d(q, x) > r$ and therefore do not need to explicitly evaluate $d(x, p)$. All the other elements that cannot be discarded using this rule are directly compared with the query.

Clustering Algorithms.

This second trend consists of dividing the space into zones as compact as possible, usually in a recursive fashion, and storing a representative point ("center") for each zone plus a few extra data that permit quickly discarding the zone at query time. Two criteria can be used to delimit a zone. The first one is the *Voronoi region*, where we select a set of centers and put every other point inside the zone of its closest center. The regions are bounded by hyperplanes and the zones are analogous to Voronoi regions in vector spaces. Let $\{c_1 \ldots c_m\}$ be the set of centers. At query time we evaluate $(d(q, c_1), \ldots, d(q, c_m))$, choose the closest center c and discard every zone whose center c_i satisfies $d(q, c_i) > d(q, c) + 2r$, as its Voronoi area cannot intersect the query ball. The second criterion is the *covering radius* $cr(c_i)$, which is the maximum distance between c_i and an element in its zone. If $d(q, c_i) - r > cr(c_i)$, then there is no need to consider zone i. The two criteria can be combined.

Combining Clustering with Pivots.

There are some indices that combine both ideas by dividing the space into compact zones and, at the same time, storing distances to some distinguished elements (pivots) [1].

3. DYNAMIC SPATIAL APPROXIMATION TREES

In this section we briefly describe *DSA–trees*, in particular the version called *timestamp with bounded arity* (reported in [11] as one of the better options for this dynamic tree), on top of which we build our approach.

3.1 Insertion Algorithm

The *DSA–tree* is built incrementally via insertions. The tree has a maximum arity denoted by $MaxArity$. Each tree node a stores a *timestamp* of its insertion time, $time(a)$,

Algorithm 1 Insertion of a new element x into a *DSA–tree* with root a using timestamping and bounded arity.

```
Insert(Node $a$, Element $x$)

1. $R(a) \leftarrow \max(R(a), d(a,x))$
2. $c \leftarrow \text{argmin}_{b \in N(a)} d(b,x)$
3. If $d(a,x) < d(c,x) \wedge |N(a)| < MaxArity$ Then
4.    $N(a) \leftarrow N(a) : x$
5.    $N(x) \leftarrow \langle\rangle$, $R(x) \leftarrow 0$
6.    $T(x) \leftarrow CurrentTime$
7.    $CurrentTime \leftarrow CurrentTime + 1$
8. Else Insert($c$,$x$)
```

its *covering radius*, $R(a)$, and its set of children $N(a)$ (the *neighbors* of a). To insert a new element x, its point of insertion is sought starting at the tree root and moving to the neighbor closest to x, updating $R(a)$ in the way. We finally insert x as a new (leaf) child of a if **(1)** x is closer to a than to any $b \in N(a)$, and **(2)** the arity of a, $|N(a)|$, is not already maximal. In other case, we insert x in the subtree of the closest element $b \in N(a)$. Neighbors are stored left to right in increasing timestamp order. Note that each element is older than its children and than its next sibling. See Algorithm 1.

3.2 Search Algorithm

The idea for range searching is to replicate the insertion process of relevant elements. That is, we act as if we wanted to insert q but keep in mind that relevant elements may be at distance up to r from q. So in each decision for simulating the insertion of q we permit a tolerance of $\pm r$, so that it may be that relevant elements were inserted in different children of the current node, and backtracking is necessary.

We have to consider two facts. The first is that, at the time an element x was inserted, a node a in its path may not have been chosen as its parent because its arity was already maximal. So, at query time, instead of choosing the closest to x among $\{a\} \cup N(a)$, we may have chosen only among $N(a)$. Hence, we perform the minimization only among elements in $N(a)$. The second fact is that, at the time x was inserted, elements with higher timestamp were not yet present in the tree, so x could choose its closest neighbor only among elements older than itself.

Hence, we consider the neighbors $\{b_1, \ldots, b_k\}$ of a from oldest to newest, disregarding a, and perform the minimization as we traverse the list. This means that we enter into the subtree of b_i if $d(q, b_i) \leqslant \min\{d(q, b_1), \ldots, d(q, b_{i-1})\} + 2r$. That is, we always enter into b_1; we enter into b_2 if $d(q, b_2) \leqslant d(q, b_1) + 2r$; and so on. Let us stress again the reason: between the insertion of b_i and b_{i+j} there may have appeared new elements that have chosen b_i just because b_{i+j} was not yet present, so we may miss an element if we do not enter into b_i because of the existence of b_{i+j}.

Up to now we do not really need the exact timestamps but just to keep the neighbors sorted by timestamp. We make better use of the timestamp information in order to reduce the work done inside older neighbors. Say that $d(q, b_i) > d(q, b_{i+j}) + 2r$. We have to enter into the subtree of b_i anyway because b_i is older. However, only the elements with timestamp smaller than that of b_{i+j} should be considered when searching inside b_i; younger elements have seen b_{i+j} and they cannot be interesting for the search if they are inside b_i. As parent nodes are older than their descendants, as soon as we find a node inside the subtree of b_i with timestamp larger than that of b_{i+j} we can stop the search in that branch, because all its subtree is even younger.

Algorithm 2 Searching for q with radius r in a *DSA–tree* rooted at a, built with timestamping and bounded arity.

```
RangeSearch(Node $a$, Query $q$, Radius $r$,
            Timestamp $t$)

1. If $T(a) < t \ \wedge \ d(a,q) \leq R(a) + r$ Then
2.   If $d(a,q) \leq r$ Then Report $a$
3.   $d_{min} \leftarrow \infty$
4.   For $b_i \in N(a)$ Do // ascend. timestamps
5.     If $d(b_i,q) \leq d_{min} + 2r$ Then
6.       $t' \leftarrow \min\{t\} \cup \{T(b_j), j > i \ \wedge$
                          $d(b_i,q) > d(b_j,q) + 2r\}$
7.       RangeSearch($b_i$,$q$,$r$,$t'$)
8.       $d_{min} \leftarrow \min\{d_{min}, d(b_i,q)\}$
```

Algorithm 2 depicts the search process. Note that $d(a, q)$ is always known except in the first invocation, and the initial t is $+\infty$.

4. OUR PROPOSAL

As we mention previously, we build our proposal on the *DSA–tree*. The new data structure, called *DSACL–tree* for short, performs the spatial approximation on groups or *clusters* of objects that are very close to each other, rather than individual objects. By this way it can reduce search costs, because it has to do less backtracking. The new data structure is still dynamic, competitive, and makes better use of the available memory.

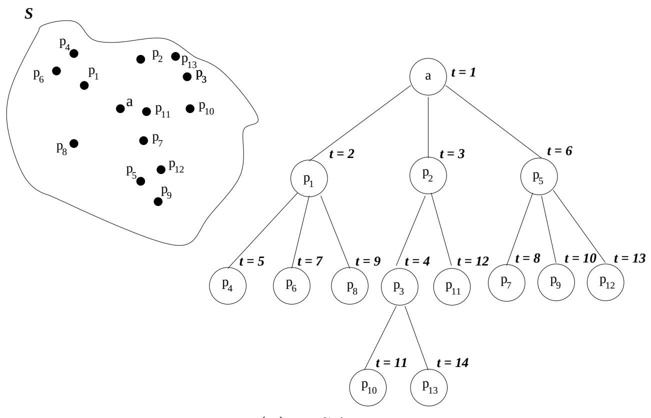
Therefore, in the *DSACL–tree* each node represents a cluster of very similar objects, for short we refer to it simply as *cluster*. Thus, we relate the clusters by their proximity in the metric space. So, each node of the tree would be able to store multiple database objects, reducing the number of nodes with respect to the original *DSA–tree*. This reduce the number of pointers used by the tree structure, thus makes a better use of the available memory.

As in the *DSA–tree* we build the *DSACL–tree* incrementally, considering a *maximum arity* and maintaining information of the *timestamp* (time of insertion of each element). We also register the *timestamp time*(c) of each node c in the tree, that is, the time when this node (and its cluster) was created. Each node c keeps an object *center*(c) as the *center* of the cluster and the k *nearest objects* (*cluster*(c)) seen in its subtree, and is connected with their clusters-neighbors $N(c)$. The cluster also has a *cluster radius* $rc(c)$, that is, considering the objects in increasing order to the *center*(c) the distance of the k-th object in the *cluster*(c). Any object further away from the center than $rc(c)$ would become part of another tree node, which could be a new neighbor in some cases, since the arity is bounded in the same way as *DSA–tree*. Each node c also stores the maximum distance between the *center*(c) and the farthest object in its subtree $R(c)$ (as *DSA–tree* does), called *covering radius* of the subtree of c.

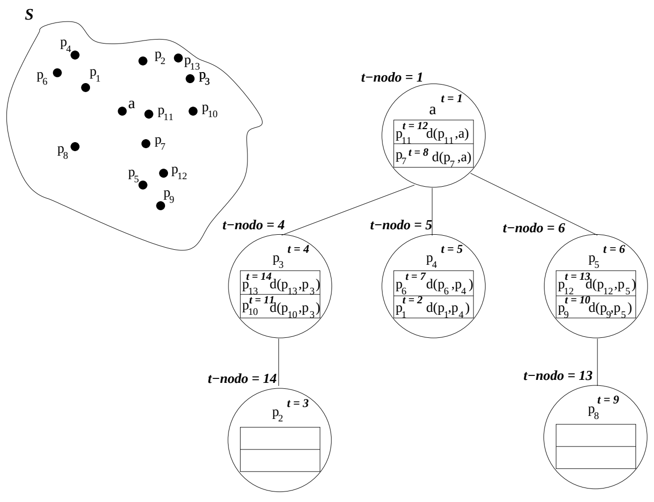
Since each node c represents a cluster centered in *center*(c) with at most k objects within *cluster*(c), we maintain the distances between *center*(c) and all the objects in *cluster*(c) ordered by increasing distance to the center. At search time, we can use these stored distances in order to minimize the number of distance computations using the triangle inequality. Besides, if $x_1, \ldots, x_k$ are the objects in *cluster*(c)

sorted by distances, the covering radius of the cluster will be $rc(c) = d(center(c), x_k)$. Therefore, for each object x_i inside the cluster, we stored its insertion moment $time(x_i)$ and the distance $d(center(c), x_i)$. It is clear it is not necessary to really register $rc(c)$ because it can be obtained from the stored distances inside the node.

Figure 1 allows us to see graphically the differences between *DSA–tree* and *DSACL–tree* for the same metric space in $\mathbb{R}^2$, using Euclidean distance. The insertion sequence in both cases is $a, p_1, p_2, \ldots, p_{13}$ and the maximum arity is 3. For the *DSACL–tree* the maximum size of the cluster is 2.



(a) *DSA–tree*



(b) *DSACL–tree*

Figure 1: Comparison between a *DSA–tree* (a) and *DSACL–tree* (b) for the same metric space $S \subset \mathbb{R}^2$, using Euclidean distance.

Figure 2 depicts, for the same metric space used as example, how are grouped the elements in clusters within S and how they are related into the tree.

Now, we describe the insertion process.

4.1 Insertion

When we insert a new element x in a *DSACL-tree* we have to consider two cases. The first is when the new element becomes the center of a new node. The second is when

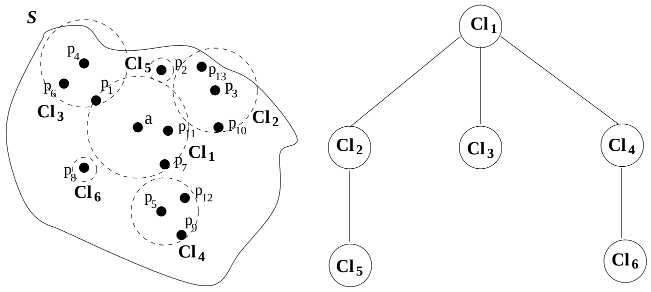


Figure 2: Clusters and their relations from the *DSACL–tree*, for the same example of Figure 1.

x should be inserted into a cluster of a node c, that is, x must belong to the $cluster(c)$, which means x is one of the k-nearest elements for the $center(c)$ at that current time.

As we have already mentioned, to incrementally build the *DSACL–tree* we maintain the same considerations as *DSA–tree*, i.e. we set the maximum arity of the tree and store the insertion time of each object, denoted by $timestamp(\cdot)$. We also maintain the creation time, $time(c)$, for each node c in the tree and set the maximum size k of each cluster node. Moreover, given a node c, we store in its $cluster(c)$ the k nearest elements seen in its subtree sorted by increasing distance to the $center(c)$, and for each of them we register its timestamp and the distance to the center.

When we insert a new element x in the cluster of a node c (that can occurs when $d(center(c), x) < rc(c)$) and we have achieved the maximum cluster size ($|cluster(c)| = k$), we need to remove an element from $cluster(c)$ to create a free slot for x. As objects within the cluster are sorted in increasing distance ($cluster(c) = x_1, \ldots, x_k$) we simple exclude x_k and include x. This also reduce the volume of $cluster(c)$. Of course we have to reinsert x_k. For this sake, we start x_k's reinsertion process starting from node c (and not from the root), because the fact that x_k was previously inserted in node c implies that either (i) x_k is closer to c than to each other ancestor in the path from the root, or (ii) each of those ancestors has reached its maximum arity. Finally, the timestamp of the reinserted object is updated to the new insertion time.

When we insert a new element x into a *DSACL–tree* the whole insertion process is the following. It begins in the node which is the tree root, and in each center node c considered, we first evaluate if x must belong to $cluster(c)$, this is to achieve the smallest possible volume for the current cluster. Thus, we have the following cases:

1. If x should not belong to $cluster(c)$, we need to determine if x is closer to $center(c)$ than the centers of the neighboring nodes in $N(c)$:
   (a) If x is closer to $center(c)$ and the arity of c is not already the maximum, we insert x as the center of a new neighboring node of c. In this case we create a new node b with $center(b) = x$, and we add b in $N(c)$.
   (b) If x is not closer to the $center(c)$, or the arity of c has achieved the maximum value, we continue the insertion process from the c's neighbor node b ($b \in N(c)$) whose $center(b)$ is the closest to x. Recursively with b we have to check whether x

must belong to the $cluster(b)$, x has to be inserted as a center of a new neighboring node, or we have to continue down the tree until we find the correct insertion point.

2. On the other hand, if there is a free slot in c or x is closer of the center than its current covering radius, then x must belong to $cluster(c)$. In the first case ($|cluster(c)| < k$), we simple insert x with its distance $d(center(c), x)$ into the $cluster(c)$ preserving the increasing order of distances. In the second case ($d(center(c), x) < rc(c)$), we can reduce $rc(c)$ by excluding the farthest element within the cluster and inserting x (and its distance) into $cluster(c)$. Next, we need to reinsert the excluded element by searching its correct insertion point from the node c as we mention.

In the last case it is important to notice what happens when the cluster is full, but x can reduce $rc(c)$. As x must belong to $cluster(c)$, we remove x_k, the farthest current element in $cluster(c)$ from it former cluster and then we reinsert x_k in the tree. Therefore, x_k has to choose the nearest element between $center(c)$ and the centers of the neighboring nodes in $N(c)$. If the former $center(c)$ is the nearest one, we insert a new node b whose $center(b) = x_k$ in $N(c)$, if the arity is not maximal. Otherwise, we continue the search of the insertion point from the nearest center of a neighboring node in $N(c)$, following the same procedure previously detailed. This can recursively reduce the volume of other clusters.

Algorithm 3 illustrates the whole insertion process. The function is invoked as `InsertCl(a, x)`, where a is the root node of the tree and x is the element to be inserted. As can be seen, as in the *DSA–tree* we only have to follow a path from the tree root to the cluster, or to the node which will be its father node when this element has to be a new center of a neighboring node. $MaxArity$ is the upper bound of the arity, k is the maximum size of a cluster, and $CurrentTime$ is the current timestamp, which increases before each insertion.

The tree can be built incrementally. The first element inserted x will create a single node a, which become the tree root, whose center will be $center(a) = x$ with $rc(a) = 0$, $cluster(a) = \emptyset$, $N(a) = \emptyset$, and $R(a) = 0$. Then, the following k insertions will be responsible to fill $cluster(a)$ completely. In this situation, the next insertion will create a new node in the neighborhood of a, whose center will be the farthest from $center(a)$.

As can be noticed, an insertion cannot change the center of a node, but it could create (at most) a new node with one corresponding center. Nevertheless, the insertion of a new object can affect many nodes, by changing their clusters with successive replacements of objects inside the clusters in a path from the tree root to a leaf.

Finally, we can observe that in *DSACL-tree* occurs the same as in *DSA–tree*: We cannot guarantee that a new element x will become a neighbor of the first node a satisfying that x does not belong to $cluster(a)$ and it is nearest to $center(a)$ than any other center of neighboring nodes $b \in N(a)$. The reason is that the node a had already reached the maximum arity, so x will be forced to choose the node in $N(a)$ whose center is the nearest one to continue the insertion process. This fact will have implications in the search procedure.

Algorithm 3 Insertion of a new element x into a *DSACL–tree* with a as tree root, using timestamping and bounded arity.

```
InsertCl (Node $a$, Element $x$)
 1. $R(a) \leftarrow \max(R(a), d(center(a), x))$
  /* Let $rc(c)$ be the distance from $center(a)$
     to the farthest element in $cluster(a)$ */
 2. If $((|cluster(a)| < k) \vee (d(center(a), x) < rc(a)))$ Then
 3.    $cluster(a) \leftarrow cluster(a) \cup \{x\}$
 4.    $d'(x) \leftarrow d(center(a), x)$
 5.    $timestamp(x) \leftarrow CurrentTime$
 6.    If $(|cluster(a)| = k + 1)$ Then
 7.       $y \leftarrow argmax_{z \in cluster(a)} d'(z)$
 8.       $cluster(a) \leftarrow cluster(a) - \{y\}$
 9.       InsertCl($a$,$y$)
10. Else
11.    $c \leftarrow \text{argmin}_{b \in N(a)} d(center(b), x)$
12.    If $d(center(a), x) < d(center(c), x) \wedge |N(a)| < MaxArity$
          Then /* $b$ becomes a new node, neighbor of $a$,
                  with $center(b) = x$ */
13.       $N(a) \leftarrow N(a) \cup \{b\}$
14.       $center(b) \leftarrow x$
15.       $N(b) \leftarrow \emptyset, R(b) \leftarrow 0$
16.       $cluster(b) \leftarrow \emptyset$
17.       $timestamp(x) \leftarrow CurrentTime$
18.       $time(b) \leftarrow CurrentTime$
19.    Else
20.       InsertCl ($c$,$x$)
```

4.2 Range Search

When performing a range query, we proceed in a similar way as *DSA–tree*, that is we perform the spatial approximation to the query via the centers of nodes. As we mentioned previously, the idea for range searching is to replicate the insertion process of the relevant elements to the query. That is, we act as if we wanted to insert q but keeping in mind that relevant elements may be at distance up to r from q, so in each decision we simulate the insertion of q permitting a tolerance of $\pm r$. So that it may be that relevant elements were inserted in a cluster, in different children of the current node, and backtracking is necessary.

We have two important facts to consider during searches. The first one is that at the time an element x was inserted, a node a in its path may not have been chosen as its parent because its arity was already maximal. So, at query time, instead of choosing the closest center to x among $\{a\} \cup N(a)$, we may have chosen only among $N(a)$. Hence, we perform the minimization only among elements in $N(a)$. The second fact is that, at the time x was inserted, elements with higher timestamp were not yet present in the tree, so x could choose its closest center of a neighbor only among elements older than itself.

Moreover, because we have clusters within the nodes, in each node a reached during the search we have to check whether $cluster(a)$, whose radius is $rc(a)$, intersects with the query (q, r). If there is no intersection we can discard all the elements in the cluster without any comparison with q. However, if there is intersection, we still can use $center(a)$ as a pivot. That is, we can avoid some distance computation considering the stored distances to the $center(a)$ and the triangle inequality to discard any element $x_i \in cluster(a)$ that satisfies $|d(center(a), x_i) - d(center(a), q)| > r$. In this case, non discarded elements must be compared directly with q.

It is very important to notice that, if the query is strictly contained inside the cluster of a node a reached during the search, we can stop the process on the subtree of a, because in no other place of this subtree we could find any relevant elements to the query.

We also can use the timestamps and covering radii information to prune searches, as *DSA–tree* does.

Algortithm 4 depicts the *DSACL–tree* search algorithm. It is initially invoked as `RangeSearchCl(`a, q, r`,`$CurrentTime$`)`, where a is the tree root.

Algorithm 4 Range Search of q with radius r in a *DSACL-tree* with tree root a.

```
RangeSearchCl (Node $a$, Query $q$, Radius $r$, Timestamp $t$)
 1. If $time(a) < t \wedge d(center(a), q) \leq R(a) + r$ Then
 2.   If $d(center(a), q) \leq r$ Then Report $a$
 3.   If $(d(center(a), q) - r \leq rc(a)) \vee$
         $(d(center(a), q) + r \leq rc(a))$ Then
 4.     For $c_i \in cluster(a)$ Do
 5.       If $|(d(center(a), q) - d'(c_i)| \leq r$ Then
 6.         If $d(c_i, q) \leq r$ Then Report $c_i$
 7.     If $d(center(a), q) + r < rc(a)$ Then Return
 8.   $d_{min} \leftarrow \infty$
 9.   For $b_i \in N(a)$ in increasing order of timestamp Do
10.     If $d(center(b_i), q) \leq d_{min} + 2r$ Then
11.       $k \leftarrow \min\{j > i, d(center(b_i), q) > d(center(b_j), q) + 2r\}$
12.       RangeSearchCl ($b_i$,$q$,$r$,$time(b_k)$)
13.     $d_{min} \leftarrow \min\{d_{min}, d(center(b_i), q)\}$
```

Figure 3 shows graphically the different situations that can occur during the search process in a *DSACL–tree*. In Figure (a), we find that $d(center(a), q) \leq R(a) + r$, so we need to evaluate the subtree of the node a, but we do not enter into its cluster because $d(center(a), q) - r > rc(a)$, thus there are not relevant elements for the query. In Figure (b), $d(center(a), q) - r \leq rc(a)$, then we need to consider the elements in $cluster(a)$, but in this case we can use $center(a)$ as pivot. Finally, in Figure (c), after evaluate the elements in $cluster(a)$, since $d(center(a), q) + r < rc(a)$ we can conclude the search process in this subtree as the query is strictly contained into $cluster(a)$.

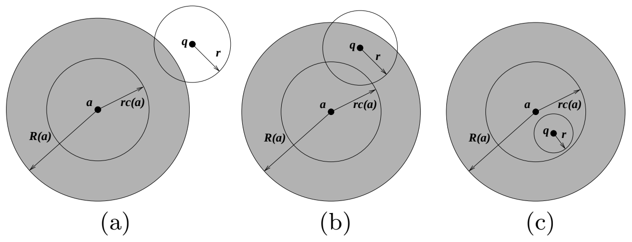


Figure 3: All the possible cases where it is satisfied that $d(center(a), q) \leq R(a) + r$**.**

5. EXPERIMENTAL RESULTS

For the experiments we have selected four widely different metric spaces. They are available from the SISAP Metric Library (`www.sisap.org`). Using these databases we can give a broad picture of the performance of our index.

***English*:** is a dictionary of 69,069 English words. We use the *edit distance* or *Levenshtein distance*, that is, the minimum number of character insertions, deletions, and substitutions needed to make two strings equal.

***Documents*:** 1,265 documents under the Cosine similarity, from TREC-3 collection. In this model the space has one coordinate per term and documents are seen as vectors in this space. The distance we use is the angle between the vectors.

***NASA*:** 40,700 20-dimensional feature vectors, generated from NASA images, using Euclidean distance.

***Histograms*:** 112,682 8-D color histograms (112-dimensional vectors) from an image database. Euclidean distance is used for this metric space, because it is the simplest meaningful alternative, but any quadratic form can be used as a distance.

For the search experiments, we build the indices with 90% of the points and use the other 10% (randomly chosen) as queries. All our results are averaged over 10 index constructions using different permutations of the datasets. We have considered range queries retrieving on average 0.01%, 0.1% and 1% of the dataset. This corresponds to radii 0.140, 0.150 and 0.195 for the documents; 0.12, 0.285 and 0.53 for the images; and 0.051768, 0.082514 and 0.131163 for the histograms. Words have a discrete distance, so we used radii 1 to 4, which retrieved on average 0.003%, 0.037%, 0.326% and 1.757% of the dataset, respectively. The same queries are used for all the experiments on the same datasets. For economy of space, we place all the experimental results together in Figure 4.

For all the considered metric spaces we start by determining experimentally which is the best cluster size for searching, so we tested with sizes: 10, 50, 100, 150, 200, and 250. The best size is 50 elements in English space and 10 elements in the others. In the following results, we set the cluster size in 50 elements for all the experiments on *DSACL–trees* for English space and in 10 elements for Documents, NASA, and Histograms spaces.

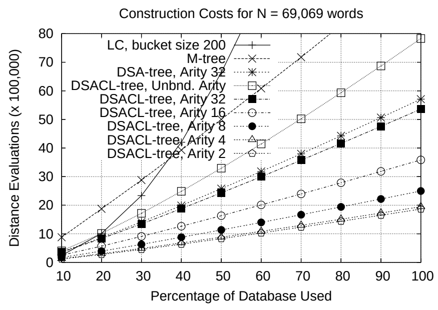
Then, we have to select the best maximum arity for each space. We try several values of maximum arities, namely 2, 4, 8, 16, and 32, but we also evaluate how it is affected the search on our tree when we do not bound the arity. Figure 4, left column, shows the construction costs of the *DSACL–tree* with all the arities tested. As can be seen, construction costs increases as arity grows.

Additionally, the left column also shows the construction costs for the other indices which *DSACL-tree* is compared with (that performance comparison is given in Section 5.1).

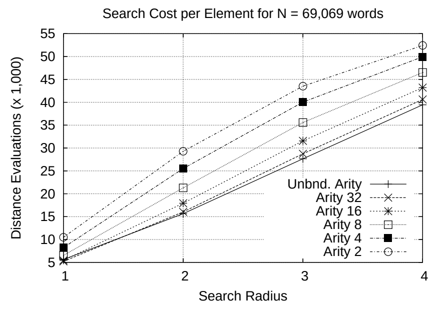
Figure 4, middle column, depicts the results obtained in searches. As can be seen for the English space (Figure (b)) we select the option without bounded arity, for the Documents (Figure (e)) maximum arity of 4 is the best one, and for NASA space and for Histograms (Figures (h) and (k), respectively) the best arity is 8. The rule of thumb is that low arities are good for low dimensions or small search radii and viceversa (higher arities are better for high dimensions or low selectivity).

5.1 Comparison with other indices

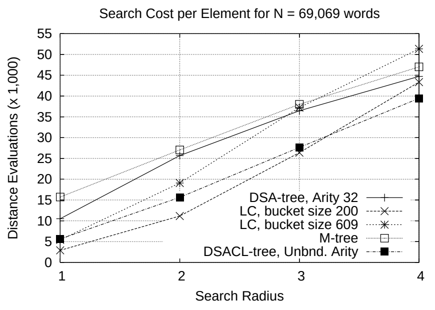
In order to evaluate the competitiveness of the *DSACL–tree*, we compare its search performance with three data structures. One of them is clearly the *DSA–tree*, the second one is the *M–tree* [6], that is the best-known dynamic secondary-memory data structure and its code is freely avail-



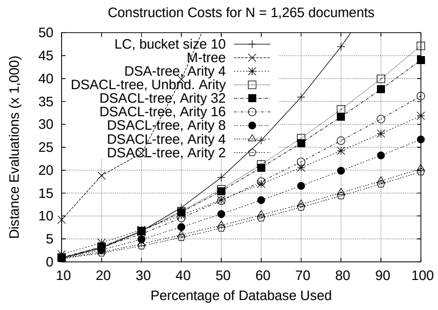
(a) English space



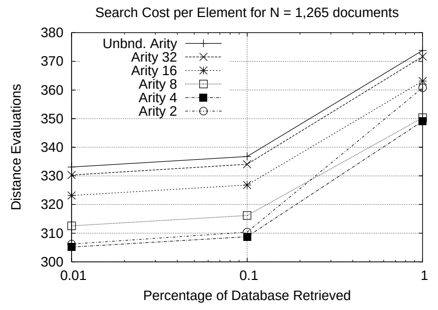
(b) English space



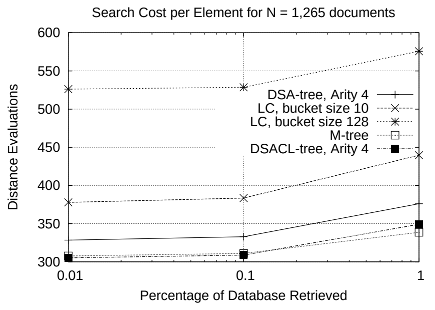
(c) English space



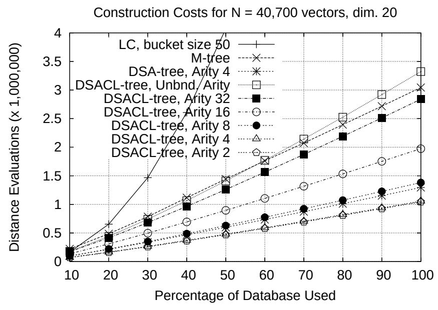
(d) Documents space



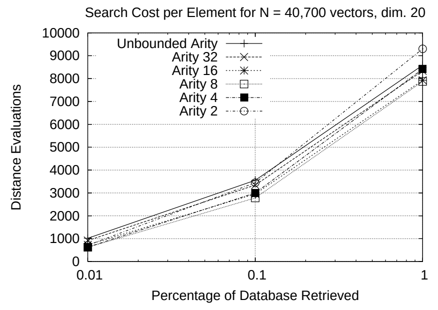
(e) Documents space



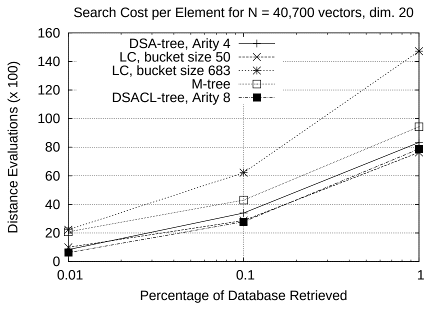
(f) Documents space



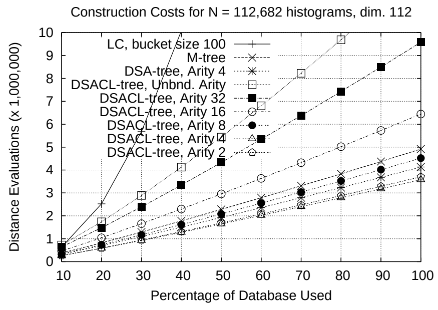
(g) NASA space



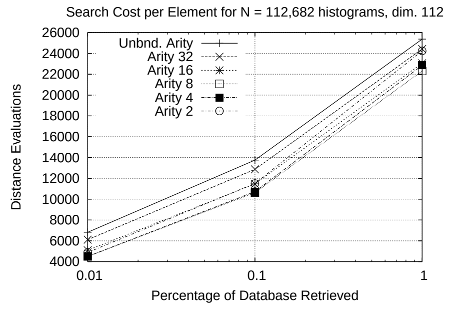
(h) NASA space



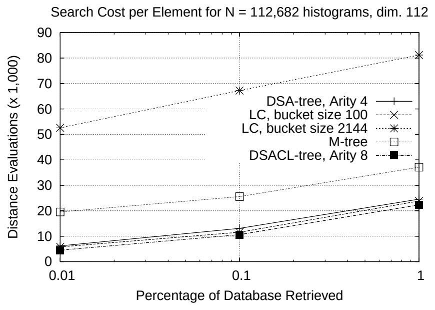
(i) NASA space



(j) Histograms space



(k) Histograms space



(l) Histograms space

Figure 4: In the first row we show the results for the English dictionary, using cluster size of 50 objects. In the following rows, we show results for the other spaces, using cluster size of 10 objects. In Plot (a), (d), (g), and (j), LC reaches $26.9 \cdot 10^6$, $73.3 \cdot 10^3$, $16.8 \cdot 10^6$, and $64.0 \cdot 10^6$ distance evaluations, respectively. In Plot (a) and (d), $M-tree$ reaches $10.6 \cdot 10^6$ and $145 \cdot 10^3$, distance evaluations, respectively. In Plot(j), *DSACL-tree* with unbounded arity reaches $12.7 \cdot 10^6$ distance evaluations. On the left column, we show construction costs both for the *DSACL–tree* with different arities and for the alternative indices. On the middle column, we show *DSACL–tree* search costs. On the right column, we compare the *DSACL–tree* search costs with the costs of all the alternative indices *DSACL–tree* is compared with.

able[1], and the last one is the *List of Clusters* [4] because, despite it is not dynamic, it works well in high dimensional spaces. We choose these indices because they have demonstrated to be competitive and its codes are freely available[2].

For the *M–tree* we have used the parameter setting suggested by the authors[3]. For the *DSA–tree* we used the best parameters reported in [11] for each considered metric space. For the *List of Clusters* we show the best cluster size experimentally determined for each space, but in order to make a fair comparison we also show for each metric space the option with the cluster size that obtains similar construction costs than *DSACL–tree*.

Figures 4(a), (d), (g), and (j) show that, in general, the *DSATCL–tree* is a very economical alternative in terms of distance computations during index construction. In fact, the plots show that it is similar, and sometimes cheaper, than the basic *DSA–tree*.

In the those plots, we show the construction cost of a *LC* with best cluster size. As can be notice, constructing this particular *LC* is very costlier. In terms of space requirement, the *M-tree* is the index using more space, and the *DSACL–tree* is the second alternative. The *DSA–tree* and the *LC* use little space in order to index the objects.

Finally, in terms of object retrieval, Figures 4(c), (f), (i), and (l) show that the best suited *DSACL–tree* for each metric space is consistently the best or second best in searching performance for all the spaces considered.

6. CONCLUSIONS AND FUTURE WORKS

In this work we present the *DSACL–tree* which is a new index for searching metric spaces. This new index enhances the good features of the *DSA–tree* (spatial approximation and dynamism) by taking into account the element clusters present in the metric space. In fact, each node of the *DSACL–tree* not only stores a single object as the root of a tree, but also it maintains a bucket to store the elements which are the closest ones with respect to the current root. This allows us to reduce the backtracking in the tree improving the cost of retrieval relevant elements when performing a proximity query. We have shown some empirical evidence that our new index is competitive with other dynamic indices such as *DSA–tree* and *M–tree*.

Future work considers a secondary memory version of the *DSACL–tree*. We also pretend to perform an exhaustive experimental study of our data structure, both to understand its behavior in other metric spaces and also to evaluate the quality of the clusters produced in the *DSACL–tree*.

[1]At `http://www-db.deis.unibo.it/research/Mtree/`

[2]Codes of *DSA–tree* and *List of Clusters* are available at SISAP Metric Space Library (`www.sisap.org`)

[3]`SPLIT_FUNCTION = G_HYPERPL, PROMOTE_PART_FUNCTION = MIN_RAD, SECONDARY_PART_FUNCTION = MIN_RAD, RADIUS_FUNCTION = LB, MIN_UTIL = 0.2. We use PAGE SIZE = 4KB`

7. ACKNOWLEDGMENTS

We wish to thank Mauricio Marin and Gonzalo Navarro for their valuable comments.

8. REFERENCES

[1] D. Arroyuelo, F. Muñoz, G. Navarro, and N. Reyes. Memory-adaptative dynamic spatial approximation trees. In *Proc. 10th Intl. Symp. on String Processing and Information Retrieval (SPIRE'03)*, LNCS 2857, pages 360–368. Springer, 2003.

[2] J. Bentley. Multidimensional binary search trees used for associative searching. *Comm. of the ACM*, 18(9):509–517, 1975.

[3] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. on Software Engineering*, 5(4):333–340, 1979.

[4] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[5] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001.

[6] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. 23rd Conf. on Very Large Databases (VLDB'97)*, pages 426–435, 1997.

[7] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[8] G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report CS-TR-4199, University of Maryland, Computer Science Department, 2000.

[9] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. on Database Systems*, 28(4):517–580, 2003.

[10] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal*, 11(1):28–46, 2002.

[11] G. Navarro and N. Reyes. Dynamic spatial approximation trees. *ACM Journal of Experimental Algorithmics*, 12:article 1.5, 2008. 68 pages.

[12] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[13] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.