Improved Deletions in Dynamic Spatial Approximation Trees *

Gonzalo Navarro
Dept. of Computer Science
University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Nora Reyes
Depto. de Informática
Universidad Nacional de San Luis
Ejército de los Andes 950, San Luis, Argentina
nreyes@unsl.edu.ar

Abstract

*The Dynamic Spatial Approximation Tree (*dsa–tree*) is a recently proposed data structure for searching in metric spaces. It has been shown that it compares favorably against alternative data structures in spaces of high dimension or queries with low selectivity. The* dsa–tree *supports insertion and deletions of elements. However, it has been noted that deletions degrade the structure over time, so the structure cannot be regarded as fully dynamic in the sense that deletions are not sustainable for long periods of time.*

In this paper we propose and study a new method to handle deletions over the dsa–tree*, which is shown to be superior to the former in the sense that it does not affect search time at all. Indeed, we show that the resulting tree is exactly as if the deleted element had never been inserted. The outcome is a fully dynamic data structure that can be managed through insertions and deletions over arbitrarily long periods of time without any reorganization.*

1. Introduction

"Proximity" or "similarity" searching is the problem of looking for objects in a set close enough to a query under a certain (expensive to compute) distance. This has applications in a vast number of fields. All those applications can be formalized with the *metric space model* [3]. That is, there is an universe $\mathcal{U}$ of objects, and a positive real valued distance function $d : \mathcal{U} \times \mathcal{U} - \rightarrow \mathbb{R}^+$ defined among them. This distance may (and ideally does) satisfy the three axioms that make the set a metric space: *strict positiveness*, *symmetry*, and *triangle inequality*. The smaller the distance between two objects, the more "similar" they are. We have a finite database $S \subseteq \mathcal{U}$, which is a subset of the universe and can be preprocessed. Later, given a new object from the universe (a *query* q), we must retrieve all similar elements found in the database. There are two typical queries of this kind:

Range query: retrieve all elements within distance r to q in S.

Nearest neighbor query (k-NN): retrieve the k closest elements to q in S.

The distance is considered expensive to compute. Hence, it is customary to define the complexity of the search as the number of distance evaluations performed. We consider the number of distance evaluations instead of the CPU time because the CPU overhead over the number of distance evaluations is negligible in the *dsa–tree*, unlike other structures.

In this paper we are devoted to range queries. In [5] is shown how to build an nearest neighbors algorithm range-optimal using a range algorithm, so we can restrict our attention to range queries.

A particular case of this problem arises when the space is a set of D-dimensional points and the distance belongs to the Minkowski L_p family. There are effective methods to search in D-dimensional spaces [4, 1]. However, for roughly 20 dimensions or more those structures cease to work well. We focus in this paper on general metric spaces, although the solutions are well suited also for D-dimensional spaces. It is interesting to notice that the concept of "dimensionality" can be translated to metric spaces as well [2, 3]. We say that a general metric space is high dimensional when its histogram of distances is concentrated.

Proximity search algorithms build an *index* of the database and perform queries using this index, avoiding the exhaustive search. For general metric spaces, there exist a number of methods to preprocess the database in order to reduce the number of distance evaluations [3]. All those structures work on the basis of discarding elements using the triangle inequality, and most use the classical divide-and-conquer approach. (which is not specific of metric space searching).

*This work has been partially supported CYTED VII.19 RIBIDI Project (both authors) and Millenium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile (first author).

The Spatial Approximation Tree (*sa–tree*) is a recently proposed data structure of this kind [6, 7], based on a novel concept: approach the query spatially, that is, start at some point in the space and get closer and closer to the query. It has been shown that the *sa–tree* gives better space-time tradeoffs than the other existing structures on metric spaces of high dimension or queries with low selectivity [7], which is the case in many applications. The *sa–tree*, however, has some important weaknesses. The first is that, compared to other indexes, it is relatively costly to build in low dimensions The second is that, in low dimensions or for queries with high selectivity (small r or k), its search performance is poor when compared to simple alternatives. The third is that it is a static data structure: once built, it is hard to add/delete elements to/from it. These weaknesses make the *sa–tree* unsuitable for important applications such as multimedia databases.

The *dsa–tree* is a dynamic version of the *sa–tree* and overcomes its drawbacks. The dynamic *sa–tree* can be built incrementally (i.e., by successive insertions) at the same cost of its static version, and the search performance is unaffected. It has been shown that it compares favorably against alternative data structures in spaces of high dimension or queries with low selectivity [9]. The *dsa–tree* supports insertion and deletions of elements. However, it has been noted that deletions degrade the structure over time, so the structure cannot be regarded as fully dynamic in the sense that deletions are not sustainable for long periods of time.

In this paper we present a new deletion algorithm that does not degrade the search performance over time. Only with such a deletion algorithm can we consider that the *dsa–tree* is a fully dynamic data structure. Although the new deletion method is more costly than the previous, it can be invoked sparsely, so as to have an amortized deletion cost comparable to the insertion cost of the *dsa–tree*. The previous deletion algorithm was cheaper but degraded search costs. We show that this new algorithm yields better tradeoffs between search performance and deletion cost.

Full dynamism is not so common in metric data structures [3]. While permitting efficient insertions is quite usual, deletions are rarely handled. In several indexes one can delete some elements, but there are selected elements that cannot be deleted at all. This is particularly problematic in the metric space scenario, where objects could be very large (e.g., images) and deleting them physically may be mandatory. Our algorithms permit deleting any element from a *dsa–tree*. This is remarkable on a data structure whose original conception was markedly static [6].

The outcome is a much more practical data structure that can be useful in a wide range of applications. We expect the *dsa–tree*, with the new deletion algorithm, to replace the static version in the developments to come.

For the experiments of this paper we have selected two metric spaces, which are the real unitary cube in dimension 15 and 5, using Euclidean distance, where we generated 100,000 random points with uniform distribution. We have tested our *dsa–tree* on these synthetic sets of random points in a D-dimensional space: every coordinate has been chosen uniformly and independently in $[0, 1)$. However, we have not used the fact that the space has coordinates, treating the points as abstract objects in an unknown metric space. This choice allows us to control the exact dimensionality we are working with, which is not so easy if the space is a general metric space or the points come from a real situation. The results on these two spaces are representative of those on several other metric spaces we tested, for lack of space we omit those results.

This paper is organized as follows: In Section 2 we give a description of the *dsa–trees*. Section 3 presents our new improved deletion method, and Section 4 contains results obtained from experimentations. Finally, in Section 5 we conclude and discuss about possible extensions for our work.

2. Dynamic Spatial Approximation Trees

In this section we briefly describe dynamic *sa–trees* (*dsa–trees* for short) [8, 9, 10, 11].

2.1. Insertion Algorithm

To construct the *dsa–tree* incrementally we fix a maximum tree arity, and also keep a timestamp of the insertion time of each element. Each node a in the tree is connected to its children, which form a set of elements called $N(a)$, the *neighbors* of a. When inserting a new element x, its point of insertion is found by beginning from the tree root a and performing the following procedure. We add x to $N(a)$ (as a new leaf node) if (1) x is closer to a that to any element $b \in N(a)$, and (2) the arity of node a, $|N(a)|$, is not already maximal. Otherwise we force x to choose the closest neighbor in $N(a)$ and keep walking down the tree in a recursive manner, until we reach a node a such that x is closer to a that any $b \in N(a)$ and the arity of node a is not maximal (this eventually occurs at a tree leaf). At this point we add x at the end of the list $N(a)$, put the current timestamp to x and increment the current timestamp. The following information is kept in each node a of the tree: the set of neighbors $N(a)$, the timestamp $time(a)$ of the insertion time of the node, and the covering radius $R(a)$ with the distance between a and the farthest element in the subtree of a.

Note that by reading neighbors from left to right we have increasing timestamps. It also holds that the parent is always older than its children. The *dsa–tree* can be built by

starting with a first single node a where $N(a) = \emptyset$ and $R(a) = 0$, and then performing successive insertions.

2.2. Range 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, when 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 $d(q, b_i) \leqslant \min(d(q, b_1), \ldots, d(q, b_{i-1})) + 2r$. Let us stress again the reason: between the insertion of b_i and b_{i+j} there may have appeared new elements that chose 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.

Figure 1 shows the algorithm to perform range searching. Note that, except in the first invocation, $d(a, q)$ is already known from the invoking process.

```
RangeSearch (Node a, Query q, Radius r,
             Timestamp t)
1. If time(a) < t  ∧  d(a,q) ≤ R(a) + r Then
2.    If d(a,q) ≤ r Then Report a
3.    d_min ← ∞
4.    For b_i ∈ N(a)
     // in increasing timestamp order
5.       If d(b_i,q) ≤ d_min + 2r Then
6.          k ← min {j > i,  d(b_i,q) > d(b_j,q) + 2r}
7.          RangeSearch (b_i, q, r, time(b_k))
8.       d_min ← min{d_min, d(b_i,q)}
```

Figure 1. Searching q with radius r in a *dsa–tree*.

2.3. Deletions

To delete an element x, the first step is to find it in the tree. Unlike most classical data structures, doing this is not equivalent to simulating the insertion of x and seeing where it leads us to in the tree. The reason is that the tree was different at the time x was inserted. If x were inserted again, it could choose to enter a different path in the tree, which did not exist at the time of its first insertion.

An elegant solution to this problem is to perform a range search with radius zero, that is, a query of the form $(x, 0)$. This is reasonably cheap and will lead us to all the places in the tree where x could have been inserted.

On the other hand, whether this search is necessary is application dependent. The application could return a handle when an object was inserted into the database. This handle can contain a pointer to the corresponding tree node. Adding pointers to the parent in the tree would permit to locate the path for free (in terms of distance computations). Hence, in which follows, we do not consider the location of the object as part of the deletion problem, although we have shown how to proceed if necessary.

We had studied several alternatives to delete elements from a *dsa–tree* in [9, 11]. From the beginning we have discarded the trivial option of marking the element as deleted without actually deleting it. As explained, this is likely to be unacceptable in most applications. We assume that the element has to be physically deleted. We may, if desired, keep its node in the tree, but not the object itself.

It should be clear that a tree leaf can always be removed without any complication, so we focus on how to remove internal tree nodes.

2.3.1. Reinserting Subtrees

A widespread idea in the Euclidean range search community is that reinserting the elements of a disk page may be beneficial because, with more elements in the tree, the space can be clustered better. We follow this principle now to obtain a method with costly deletions but good search performance.

When node x is deleted, we disconnect the subtree rooted at x from the main tree. This operation does not affect the correctness of the remaining tree, but we have now to reinsert the subtrees rooted at the nodes of $N(x)$. To do

this efficiently we try to reinsert complete subtrees whenever possible.

In order to reinsert a subtree rooted at y, we follow the same steps as for inserting a fresh object y, so as to find the insertion point a. The difference is that we have to assume that y is a "fat" object with radius $R(y)$. That is, we can choose to put the whole subtree rooted at y as a new neighbor of a only if $d(y,a)+R(y)$ is smaller than $d(y,b)$ for any $b \in N(a)$. Similarly, we can choose to go down by neighbor $c \in N(a)$ only if $d(y,c)+R(y)$ is smaller than $d(y,b)$ for any $b \in N(a)$. When none of these conditions hold, we are forced to split the subtree rooted at y into its elements: one is a single element y, and the others are the subtrees rooted at $N(y)$. Once we split the subtree, we continue the insertion process with each constituent separately.

Every time we insert a node or a subtree, we pick a fresh timestamp for the node or the root of the subtree. The elements inside the subtree should get fresh timestamps while keeping the relative ordering among the subtree elements. The easiest way to do this is to assume that timestamps are stored relative to those of their parent. In this way, nothing has to be done. We need, however, to store at each node the maximum differential time stored in the subtree, so as to update $CurrentTime$ appropriately when a whole subtree is reinserted. This is easily done at insertion time and omitted in the pseudocode for simplicity.

During reinsertion, we also modify the covering radii of the tree nodes a traversed. When inserting a whole subtree we have to add $d(y,a)+R(y)$, which may be larger than necessary. This involves at search time a price for having reinserted a whole subtree in one shot.

Note that it may seem that, when searching the place to reinsert subtrees of a removed node x, one could save some time by starting the search at the parent of x. However, the tree has changed since the subtree of x was created, and new choices may exist now.

Figure 2 shows the algorithm to reinsert a tree with root y into a *dsa–tree* rooted at a. The deletion of a node x is done by first locating it in the tree (say, $x \in N(b)$), then removing it from $N(b)$, and finally reinserting every subtree $y \in N(x)$ using `Reinsert(a,y)`.

Optimization. A further optimization to the subtree reinsertion process makes a more clever use of timestamps. Say that x will be deleted, and let $A(x)$ be the set of ancestors of x, that is, all the nodes in the path from the root to x. For each node c belonging to the subtree rooted at x we have $A(x) \subset A(c)$. So, when node c was inserted, it was compared against all the neighbors of every node in $A(x)$ whose timestamp was lower than that of c. Using this information we can avoid evaluating distances to these nodes when revisiting them at the time of reinserting c. That is, when looking for the neighbor closest to c, we know that the one in $A(x)$ is closer to c than any older neighbor, so we have to consider only newer neighbors. Note that this is valid as long as we reenter the same path where c was inserted previously.

```
Reinsert (Node a, Node y)
1. If |N(a)| < MaxArity Then M ← {a} ∪ N(a)
2. Else M ← N(a)
3. c1 ← argmin_{b∈M} d(b,y)
4. c2 ← argmin_{b∈M−{c1}} d(b,y)
5. If d(c1,y) + R(y) ≤ d(c2,y)
6. Then // keep subtree together
7.     R(a) ← max(R(a), d(a,y) + R(y))
8.     If c1 = a Then // insert it here
9.         N(a) ← N(a) ∪ {y}
10.        time(y) ← CurrentTime
11.    Else Reinsert (c1, y) // go down
12. Else // split subtree
13.    For z ∈ N(y) Do Reinsert (a, z)
14.    N(y) ← ∅, R(y) ← 0
15.    Reinsert (a, y)
```

Figure 2. Simple algorithm to reinsert a subtree with root y into a *dsa–tree* with root a.

The average cost of subtree reinsertion is as follows. Assume that we just reinsert the elements one by one. Assuming that the tree has always arity A and that it is perfectly balanced, the average size of a randomly chosen subtree turns out to be $\log_A n(1+o(1))$. As every (re)insertions costs $A \log_A n(1+o(1))$, the average deletion cost is $(A \log_A^2 n)(1+o(1))$. This is much more costly than an insertion.

3. A New Deletion Technique

Reinsertion of subtrees and reinsertion by element have shown that it is possible to delete elements from a *dsa–tree* at a reasonable cost, but it has been noted that deletions degrade the structure over time, so that deletions are not sustainable for long periods of time.

This degradation is partially caused by inevitable overestimation of covering radii. Figure 3 shows query costs on the space of vectors in dimension 5, without correcting the covering radii after each deletion (above), and correcting them (below). We can observe that the overestimation of covering radii is not the only reason for degradation.

Now we propose and study a new method to handle deletions over the *dsa–tree*. Our idea is to ensure that the resulting tree is exactly as if the deleted element had never been inserted. This ensures that no degradation can occur due to repeated deletions. The new method is called *rebuilding subtrees*.

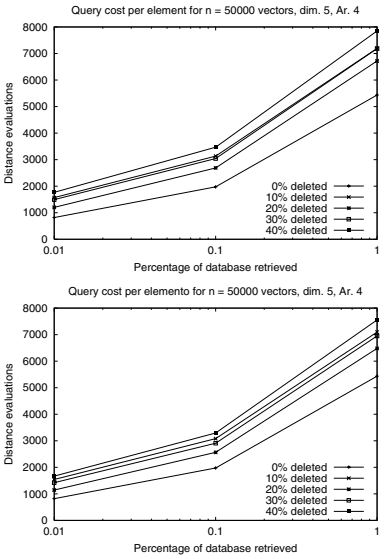

Figure 3. Query costs for different percentages of database deleted, without correcting the covering radii (above) and correcting them (below).

When node $x \in N(a)$ is deleted, we disconnect x from the main tree. Hence all its descendants must be reinserted. Moreover, elements in the subtree of a that are younger than x have been compared against x to decide their insertion point. Therefore, these elements, in absence of x, could choose another path if we reinsert them into the tree. Then, we retrieve all the elements younger than x that descend from a (i.e. those whose timestamp is greater, which includes its descendants) and reinsert them into the tree, leaving the tree as if x had never been inserted.

If we reinsert the elements younger than x like completely new elements, that is if they get fresh timestamps, we must search the appropriate point of reinsertion beginning at tree root. On the other hand, if we maintain their timestamp we can begin reinsertion process from a, so we can save many comparisons. In order to leave the resulting tree exactly as if x never had been inserted, we must reinsert the elements in the original order, that is the elements must be reinserted in increasing order of timestamp.

Hence, when node $x \in N(a)$ is deleted we retrieve all the elements younger than x from the subtree rooted a, then disconnect them from the main tree, sort them in increasing order of timestamp and reinsert them one by one, searching their reinsertion point from a.

Figure 4 shows the algorithm to retrieve from the subtree of a all the elements younger than x. We denote $T(b)$ the set of elements in the subtree rooted at b for simplicity. Figure 5 illustrates the algorithm to rebuild subtrees, which invokes to `RetrieveTS (a,x)`, according to this technique.

```
RetrieveTS (Node a, Node x)
 1. Q ← {a} , T ← ∅
 2. While Q not empty
 3.     b ← first element of Q
 4.     Q ← Q − {b}
 5.     For v ∈ N(b)
 6.         If timestamp(v) > timestamp(x) Then
 7.             N(b) ← N(b) − {v}
 8.             T ← T ∪ T(b)
 9.         Else
10.             Q ← Q ∪ {v}
11.  Return T
```

Figure 4. Algorithm to retrieve from the subtree rooted a all the elements younger than $x \in N(a)$.

```
RebuildTS (Node a, Node x)
1. T ← RetrieveTS (a,x)
2. Sort T by timestamp (older first)
3. N(a) ← N(a) − {x}
4. For v ∈ T
5.     Insert (a, v)
        // without changing its timestamp
```

Figure 5. Algorithm to rebuild the subtree with root a in a *dsa–tree*, after the deletion of $x \in N(a)$.

Note that in this method the covering radii can also become overestimated, because they are never reduced due to a deleted element. That is, if we delete an element x, every $a \in A(x)$ such that x was the farthest element in its subtree will possibly have its $R(a)$ overestimated. In spite of it, this problem does not seem to affect much search performance since, as can be seen in Figure 6, it does not significantly degrade over time (we have considered the same space used in Figure 3).

The average cost of rebuilding subtrees in a *dsa–tree* with arity A is $(A^2/4)log_A^2 n(1 + o(1))$ (we omit the proof for lack of space), that is more costly than reinsertion element by element (or of subtrees), and this difference grows as the arity tree grows. As we will see, however, this is compensated by a better search time.

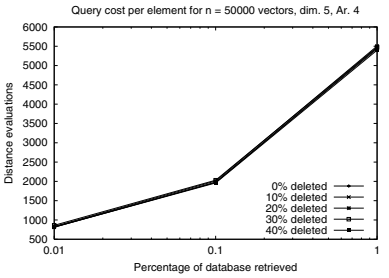

Figure 6. Query costs for different percentages of database deleted, using rebuilding of subtrees.

3.1. Optimization

We analyze two posible optimizations to rebuilding subtrees. Say that x will be deleted from the subtree rooted at node a (that is $x \in N(a)$). The first one makes a more clever use of timestamps. We can observe that there can be elements younger than x which not will change their insertion point when we reinsert them into the subtree rooted a. These elements are younger than the first child of x and also than the next sibling of x. For these elements, the available options at reinsertion time will be the same of insertion time, so they will choose the same. So we can avoid computing their new insertion place.

A further optimization to the subtree rebuilding process uses the previous work done during the insertion to save distance evaluations. That is, when node y was inserted, it was compared against all the neighbors of every node in $A(x)$ whose timestamp was lower than that of y. Using this information we can avoid evaluating distances to these nodes when revisiting them at the time of reinserting y. That is, when looking for the neighbor closest to y, we know that the one in $A(x)$ is closer to y than any older neighbor, so we have to consider only newer neighbors. Note that this is valid as long as we reenter the same path where y was inserted previously.

3.2. Fake Nodes

Another alternative to delete element x is to leave its node in the tree (without content) and mark it as deleted. We call these nodes *fake*. Although cheap and simple at deletion time, we must now figure out how to carry out a consistent search when some nodes do not contain an object. This alternative was also considered previously in [9], because it is a general form of amortizing the cost of one deletion over many.

Basically, if node $b \in N(a)$ is fake, we do not have enough information to avoid entering into the subtree of b once we have reached a. So we cannot include b in the minimization and have to enter always its subtree (except if we can use the timestamp information of b to prune the search).

The search performed at insertion time, on the other hand, has to follow just one path in the tree. In this case, one is free to choose inserting the new element into any fake neighbor of the current node, or into the closest non-fake neighbor. A good policy is, however, trying not to increase the size of subtrees rooted at fake nodes, as eventually they will have to be rebuilt (see later). Hence, although deletion is simple, the search process degrades its performance.

3.3. Combining both Methods

We have two methods. Fake nodes delete elements for free but degrade the search performance of the tree. Subtree rebuilding makes a costly subtree rebuilding but maintains the search quality of the tree. Note that the cost of rebuilding a subtree would not be much different if it contained fake nodes, so we could remove all the fake nodes with a single subtree rebuilding, therefore amortizing the high cost of the rebuilding over many deletions.

Our idea is to ensure that every subtree has at most a fraction α of fake nodes. We say that such subtrees are "balanced". When we mark a new node $x \in N(a)$ as fake, we check if we have not unbalanced it. In this case, x is discarded and all the younger non-fake elements reinserted in increasing order of timestamp. The only difference is that we never insert a fake node, but we discard it. A complication is that removing x may unbalance several ancestors of x, even if x is just a leaf that can be directly removed, and even if the ancestor is not rooted at a fake node. As an example, consider a unary tree of height $3n$ where all the nodes at distance $3i$ from the root, $i \geq 0$, are fake. The tree is balanced for $\alpha = 1/3$, but removing the leaf or marking it as fake its parent unbalances every node. We opt for a simple solution. We look for the lowest ancestor of x that gets unbalanced and rebuild all the subtree from the parent of x. As an example in ours experiment on vectors in dimension 15, using $\alpha = 30\%$, deleting 10% of the elements the "real" α is 2.2%, deleting 20% is 3.9%, deleting 30% is 5.5% and deleting 40% produces 6.6%, that is clearly lower than the α chosen as parameter.

This technique has a nice performance property. Since we reinsert the non-fake elements, we have the guarantee that the fraction α of its elements are fake. This means that if the size of the subtree to rebuild is m, we pay on average $A(1-\alpha)m$ reinsertions for each αm deletions made in the subtree. Hence the amortized cost of a deletion is at most $((1-\alpha)/\alpha)A^2 log_A n$. This is almost true, since because of the problem mentioned in the above paragraph we sometimes cannot guarantee the given fraction of fake

nodes. In practice, however, all the subtrees easily satisfy the criterion.

Asymptotically, the tree works as if we permanently had a fraction α of fake nodes. Hence, we can control the tradeoff between deletion and search cost. Note that pure fake nodes corresponds to $\alpha = 100\%$ and pure rebuilding of subtree to $\alpha = 0\%$.

3.3.1. Optimization

A further optimization to those considered in Section 3.1, allows us saving more distance evaluations when we rebuild a subtree. We rebuild a subtree when we find the lowest ancestor y of x, whose fraction of fake nodes exceeds α. Node y can be in one of two possible situations: (1) y is a fake node, or (2) y is not a fake node. In both cases we can save distance evaluations. In (1) since y is a fake node, then y must be discarded, and we proceed as pure rebuilding of subtrees. This means discarding y in the subtree of its parent and reinserting the non-fake elements younger than y (sorted by timestamp). In (2), element y cannot be discarded, so we retrieve the timestamp t of the oldest fake node in the subtree rooted at y and reinsert all the non-fake elements not older than t (sorted by timestamp).

4. Experimental Comparison

Let us now compare the reinsertion of subtrees versus rebuilding of subtrees on the space of vectors in dimension 15 using arity 16. Figure 7 shows the cost to delete 10% of database with arity 16 (above), and how the arity affects the search cost using pure rebuilding of subtrees (below). As it can be seen, our new method is much more expensive (but it preserves the quality of the database over time, as seen).

We compare now the three methods to handle deletions on vectors in dimension 15 using different arities, that is pure fake nodes, pure rebuilding of subtrees and the combined method. Figure 8 shows the deletion cost for the first 10% of the database using arity 16 (above) and 32 (below) and different α. We can note that pure rebuilding of subtrees is very costly, but as soon as we allow $\alpha = 1\%$ the deletion costs decrease considerably.

On the other hand, let us consider how the search costs are affected by the fraction α of fake nodes and by deletions. We search on an index built on half of the elements of the database. This half is built by inserting more elements and then removing enough elements to leave 50% of the set in the index. So, we compare the search on sets of the same size where a percentage of the elements has been deleted in order to leave the set in that size. For example, 30% deletions means that we inserted 80000 elements and then removed 30000, so as to leave 50000 elements (half of the set).

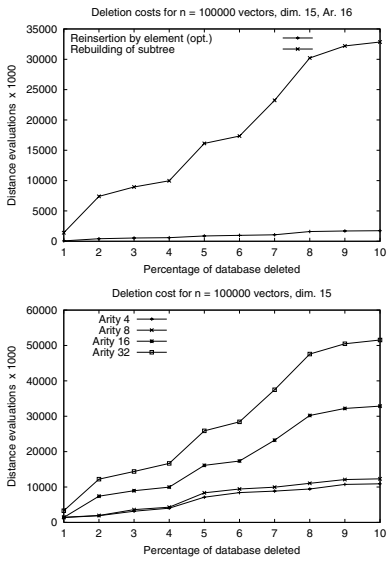

Figure 7. Deletion costs of 10% of database with arity 16 (above), and how arity affects search costs (below), using rebuilding of subtrees in dim. 15.

Figure 9 shows the results for $\alpha = 0\%$ (pure rebuilding of subtrees), 1%, 3% and 10%. As can be seen, even considering $\alpha = 10\%$ the search quality does not degrade considerably as the number of deletions grows. Figure 10 shows the same data in a way that permits comparing the change of search costs as α grows, considering 10% and 40% of elements deleted. As α grows, the search cost increases because of the need to enter every neighbor of fake nodes. The difference in search cost ceases to be reasonable as early as $\alpha = 30\%$, but it is not significant to lower α. So one has to choose the right tradeoff between deletion and search cost depending on the application. A good tradeoff for vectors in dimension 15 is $\alpha = 10\%$.

Figure 11 shows the comparison between the reinsertion of subtrees and the new rebuilding of subtrees. In each case we show search costs versus deletion costs, when deleting 10% or 40% of database, and considering range searches where 0.01%, 0.1% and 1% of database is retrieved. For example, we can note that deleting 10% of database, a given search cost is more costly to achieve (in terms of deletion cost) under rebuilding of subtrees than under the reinsertion of subtrees. However, the rebuilding of subtrees allows us to achieve cheaper search costs if we pay more expensive deletion costs (which is not possible at all with the former method). On the other hand, considering 40% of database

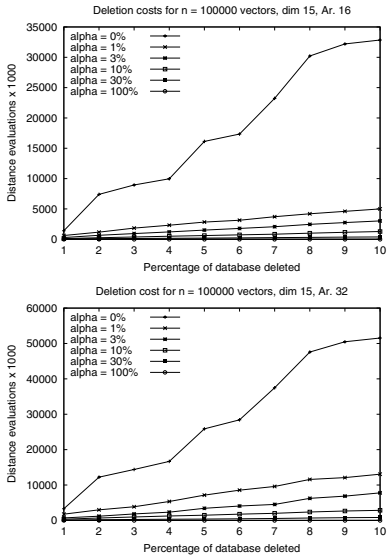

Figure 8. Deletion costs combining rebuilding of subtrees with fake nodes, for arities 16 (above) and 32 (below) in dim. 15.

deleted, rebuilding of subtrees offers a better tradeoff than subtree reinsertion: Rebuilding of subtrees obtains the same search costs at lower deletion costs. In a real scenario, where the database evolves over time and suffers many insertions and deletions, the difference between both methods will favor more and more clearly the new technique.

Figure 12 shows how the deletion costs of the combined method improve with the optimizations proposed in Section 3.1 and Section 3.3.1, for different values of α. Figure 13 shows the deletion costs with optimized rebuilding of subtrees for different α.

We compare the methods by deleting different percentages of the database to make appreciable not only the deletion cost per element but also to show the cumulative effect of deletions over the structure.

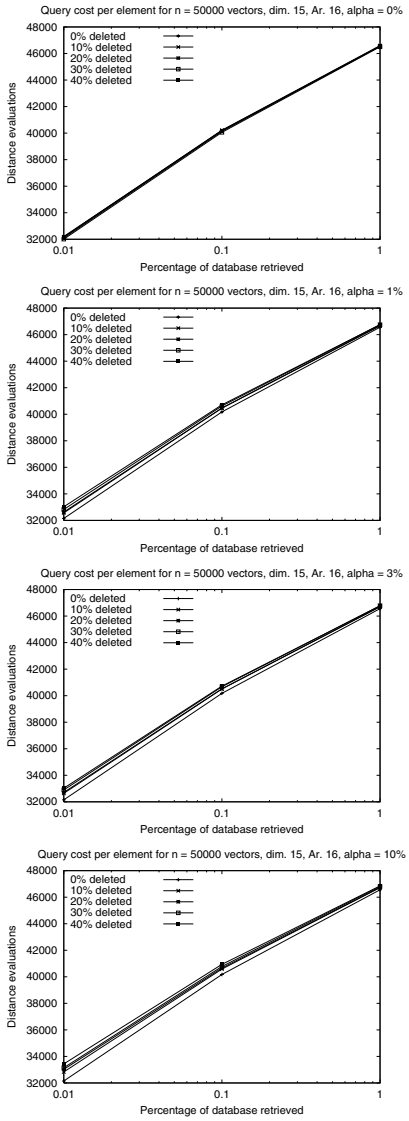

Figure 9. Search costs combining the rebuilding of subtrees and fake nodes for different α.

5. Conclusions

In this paper we have presented a new method to delete elements from a *dsa–tree*. This method has shown to be better than the former because the tradeoff between deletion and search cost improves.

The outcome is a fully dynamic data structure that can be managed through insertions and deletions over arbitrarily long periods of time without any reorganization. We also obtain optimizations for deletion costs in both methods: pure rebuilding of subtrees and the combined method with fake nodes.

Our new dynamic *dsa–tree* stands out as a practical and efficient data structure that can be used in a wide range of applications, while retaining the good features of the original data structure.

We are currently pursuing in the direction of making the *dsa–tree* work efficiently in secondary memory. In that case both the number of distance computations and disk accesses are relevant.

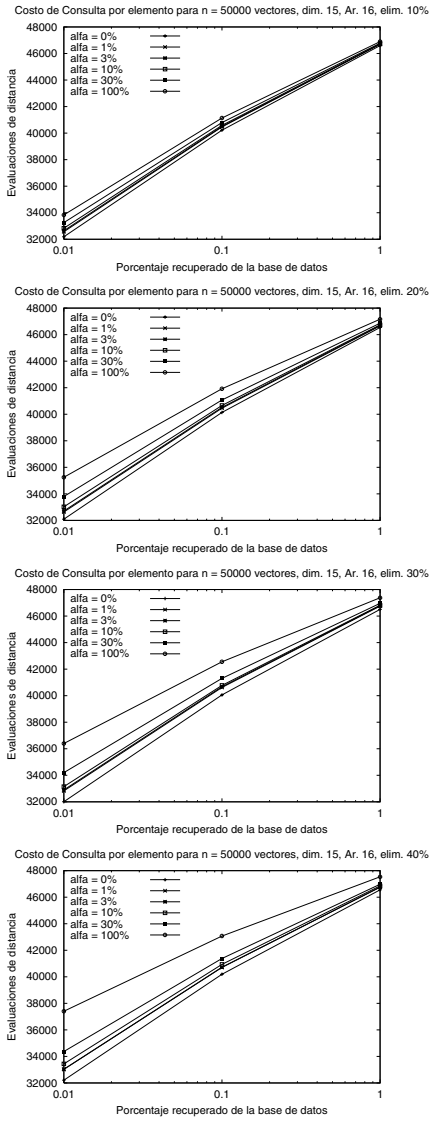

Figure 10. Search costs combining rebuilding of subtrees and fake nodes, comparing α and deleting 10%, 20%, 30% and 40% (above to below) of database.

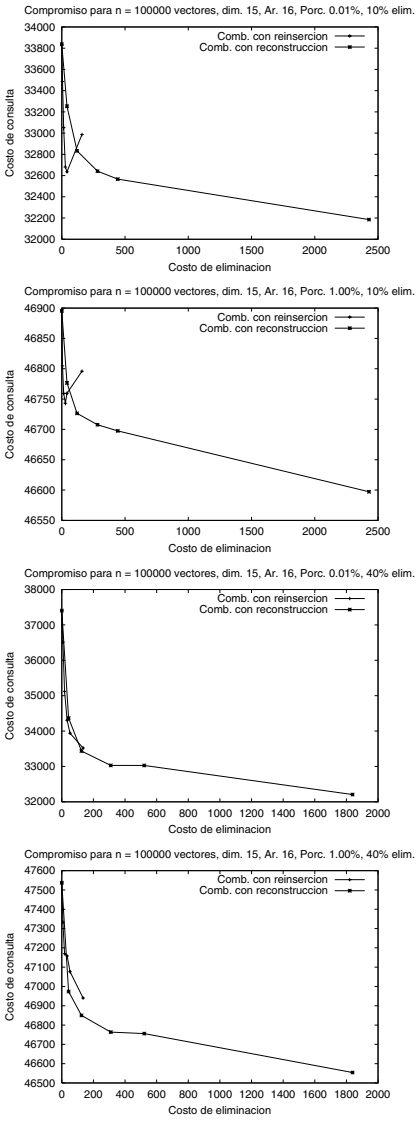

Figure 11. Tradeoff between reinsertion of subtrees and rebuilding of subtrees.

References

[1] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[2] S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

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

[4] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231,

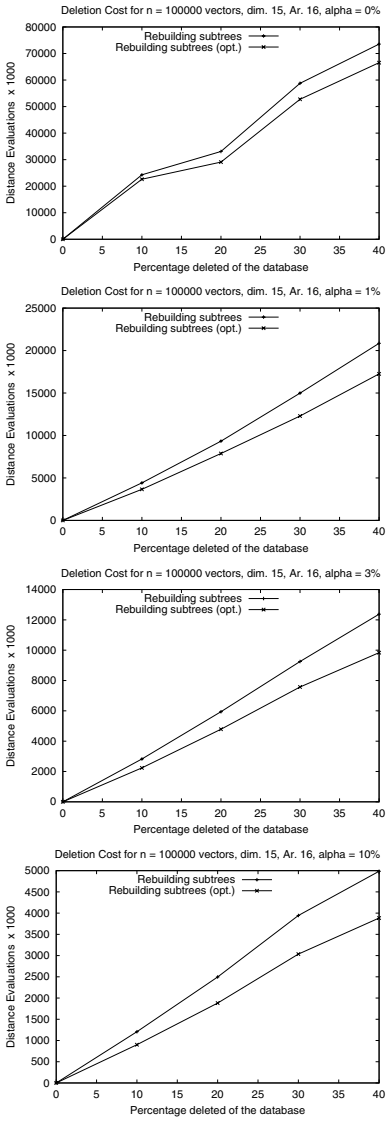

Figure 12. Deletion costs for the optimized combined method.

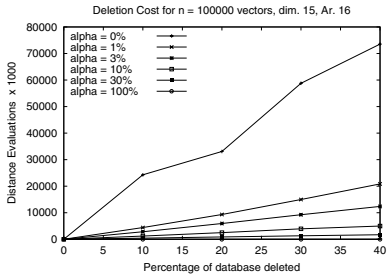

Figure 13. Deletion costs for the optimized combined method, comparing different α.

1998.

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

[6] G. Navarro. Searching in metric spaces by spatial approximation. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, pages 141–148. IEEE CS Press, 1999.

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

[8] G. Navarro and N. Reyes. Dynamic spatial approximation trees. In *Proc. XXI Conference of the Chilean Computer Science Society (SCCC'01)*, pages 213–222. IEEE CS Press, 2001.

[9] G. Navarro and N. Reyes. Fully dynamic spatial approximation trees. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, LNCS 2476, pages 254–270. Springer, 2002.

[10] G. Navarro and N. Reyes. Improved dynamic spatial approximation trees. In *Proceedings of the XXVIII Latin American Conference on Informatics (CLEI'02)*, page 74, Montevideo, Uruguay, 2002. Abstract in print, complete papers in CD-ROM.

[11] N. Reyes and G. Navarro. Eliminación en árboles de aproximación espacial dinámicos. In *Actas del VIII Congreso Argentino de Ciencias de la Computación (CACIC'02)*, pages 821–833, Buenos Aires, Argentina, 2002. Complete papers in CD-rom. In Spanish.