Fully Dynamic Spatial Approximation Trees *

Gonzalo Navarro[1] and Nora Reyes[2]

[1] Center for Web Research, Dept. of Computer Science, University of Chile,
Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl
[2] Depto. de Informática, Universidad Nacional de San Luis,
Ejército de los Andes 950, San Luis, Argentina. nreyes@unsl.edu.ar

Abstract. The Spatial Approximation Tree (*sa-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. Its main drawbacks are: costly construction time, poor performance in low dimensional spaces or queries with high selectivity, and the fact of being a static data structure, that is, once built, one cannot add or delete elements. These facts rule it out for many interesting applications.
In this paper we overcome these weaknesses. We present a dynamic version of the *sa-tree* that handles insertions and deletions, showing experimentally that the price of adding dynamism is rather low. This is remarkable by itself since very few data structures for metric spaces are fully dynamic. In addition, we show how to obtain large improvements in construction and search time for low dimensional spaces or highly selective queries. The outcome is a much more practical data structure that can be useful in a wide range of applications.

1 Introduction

The concept of "approximate" searching has applications in a vast number of fields. Some examples are non-traditional databases (e.g. storing images, fingerprints or audio clips, where the concept of exact search is of no use and we search instead for similar objects); text searching (to find words and phrases in a text database allowing a small number of typographical or spelling errors); information retrieval (to look for documents that are similar to a given query or document); machine learning and classification (to classify a new element according to its closest representative); image quantization and compression (where only some vectors can be represented and we code the others as their closest representable point); computational biology (to find a DNA or protein sequence in a database allowing some errors due to mutations); and function prediction (to search for the most similar behavior of a function in the past so as to predict its probable future behavior).

All those applications have some common characteristics. There is a universe $\mathbb{U}$ of *objects*, and a nonnegative *distance function* $d : \mathbb{U} \times \mathbb{U} \longrightarrow \mathbb{R}^+$ defined among them. This distance may (and ideally does) satisfy the three axioms that make the set a *metric space*: strict positiveness ($d(x, y) = 0 \Leftrightarrow x = y$), symmetry ($d(x, y) = d(y, x)$)

* 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).

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* $S \subseteq \mathbb{U}$, which is a subset of the universe of objects and can be preprocessed (to build an index, for example). 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. This is, $\{x \in S\ ,\ d(x,q) \leq r\}$.

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 $\forall x \in A, y \in S - A, d(x,q) \leq d(y,q)$.

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. Given a database of $|S| = n$ objects, queries can be trivially answered by performing n distance evaluations. The goal is to structure the database such that we perform less distance evaluations.

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: $L_p = (\sum_{1 \leq i \leq D} |x_i - y_i|^p)^{1/p}$. For example $p = 2$ yields Euclidean distance. 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: the typical feature in high dimensional spaces with L_p distances is that the probability distribution of distances among elements has a very concentrated histogram (with larger mean as the dimension grows), making the work of any similarity search algorithm more difficult [2, 3]. We say that a general metric space is high dimensional when its histogram of distances is concentrated.

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).

The Spatial Approximation Tree (*sa-tree*) is a recently proposed data structure of this kind [5, 6], 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. Apart from being algorithmically interesting by itself, 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, 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 (it is harder to build in high dimensions, but in this case the competing indexes are even more costly). 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.

Overcoming these drawbacks is the aim of this paper. We present a dynamic version of the *sa-tree* that handles insertions and deletions. We show that the dynamic *sa-tree* can be built incrementally (i.e., by successive insertions) at the same cost of its static version, and that the search performance is unaffected. We also show that one can remove elements from the structure at about the same cost of an insertion, with a very small penalty in the search performance.

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 *sa-tree*. This is remarkable on a data structure whose original conception was markedly static [5].

In addition to the above achievement, we find out how to obtain large improvements in construction and search time for low dimensional spaces or highly selective queries. The method consists of limiting the tree arity and involves new algorithmic insights on this data structure. The lower the arity, the cheaper to build the tree. However, at search time, the best arity depends on the dimension and the query selectivity. In particular, for low dimensions, we obtain improved construction and search time simultaneously.

The outcome is a much more practical data structure that can be useful in a wide range of applications. We expect the dynamic *sa-tree* to replace the static version in the developments to come.

This work builds over [7], where it was shown that insertions on the *sa-tree* could be reasonably handled. We improve their insertion algorithm and also permit deletions, thus obtaining a fully dynamic data structure. In addition, we capture in the tree arity the parameter that permits adapting it better to different dimensions. The original *sa-tree* adapts itself to the dimension, but not optimally.

For the experiments of this paper we have selected two metric spaces. The first is a dictionary of 69,069 English words. The distance is the edit distance, that is, the minimum number of character insertions, deletions and replacements to make the strings equal. The second space is the real unitary cube in dimension 15 using Euclidean distance, where we generated 100,000 random points with uniform distribution.

In both cases, we built the indexes with 90% of the points and used the other 10% (randomly chosen) as queries. For the experiments with deletions in an index of n elements, we select at random a fraction of those n elements and delete them from the index. The results on these two spaces are representative of those on several other metric spaces we tested: NASA images, dictionaries in other languages, Gaussian distributions, other dimensions, etc.

2 The Spatial Approximation Tree

We describe briefly in this section the static *sa-tree* data structure. It needs $O(n)$ space, $O(n \log^2 n / \log\log n)$ construction time, and sublinear search time: $O(n^{1-\Theta(1/\log\log n)})$

in high dimensions and $O(n^\alpha)$ ($0 < \alpha < 1$) in low dimensions. It is experimentally shown to offer better space-time tradeoffs than other data structures when the dimension is high or the query radius is large. For more details see the original work [5, 6].

2.1 Construction

We select a random element $a \in S$ to be the root of the tree. We then select a suitable set of neighbors $N(a)$ satisfying

Condition 1: (given a, S) $\quad \forall x \in S,\ x \in N(a) \Leftrightarrow \forall y \in N(a) - \{x\},\ d(x,y) > d(x,a)$.

That is, the neighbors of a form a set such that any neighbor is closer to a than to any other neighbor. The "only if" ($\Leftarrow$) part of the definition guarantees that if we can get closer to any $b \in S$ then an element in $N(a)$ is closer to b than a, because we put as direct neighbors all those elements that are not closer to another neighbor. The "if" part ($\Rightarrow$) aims at putting as few neighbors as possible.

Notice that the set $N(a)$ is defined in terms of itself in a non-trivial way and that multiple solutions fit the definition. For example, if a is far from b and c and these are close to each other, then both $N(a) = \{b\}$ and $N(a) = \{c\}$ satisfy the definition.

Finding the smallest possible set $N(a)$ seems to be a nontrivial combinatorial optimization problem, since by including an element we need to take out others (this happens between b and c in the example of the previous paragraph). However, simple heuristics which add more neighbors than necessary work well. We begin with the initial node a and its "bag" holding all the rest of S. We first sort the bag by distance to a. Then, we start adding nodes to $N(a)$ (which is initially empty). Each time we consider a new node b, we check whether it is closer to some element of $N(a)$ than to a itself. If that is not the case, we add b to $N(a)$.

At this point we have a suitable set of neighbors. Note that Condition 1 is satisfied thanks to the fact that we have considered the elements in order of increasing distance to a. The "only if" part of Condition 1 is clearly satisfied because any element not satisfying it is inserted in $N(a)$. The "if" part is more delicate. Let $x \neq y \in N(a)$. If y is closer to a than x then y was considered first. Our construction algorithm guarantees that if we inserted x in $N(a)$ then $d(x,a) < d(x,y)$. If, on the other hand, x is closer to a than y, then $d(y,x) > d(y,a) \geq d(x,a)$ (that is, a neighbor cannot be removed by a new neighbor inserted later).

We now must decide in which neighbor's bag we put the rest of the nodes. We put each node not in $\{a\} \cup N(a)$ in the bag of its closest element of $N(a)$ (*best-fit* strategy). Observe that this requires a second pass once $N(a)$ is fully determined.

We are done now with a, and process recursively all its neighbors, each one with the elements of its bag. Note that the resulting structure is a tree that can be searched for any $q \in S$ by spatial approximation for nearest neighbor queries. The reason why this works is that, at search time, we repeat exactly what happened with q during the construction process (i.e. we enter into the subtree of the neighbor closest to q), until we reach q. This is is because q is present in the tree, i.e., we are doing an exact search after all.

Finally, we save some comparisons at search time by storing at each node a its covering radius, i.e., the maximum distance $R(a)$ between a and any element in the subtree rooted by a. The way to use this information is made clear in Section 2.2.

Figure 1 depicts the construction process. It is first invoked as `BuildTree(`a`,`$S-\{a\}$`)` where a is a random element of S. Note that, except for the first level of the recursion, we already know all the distances $d(v,a)$ for every $v \in S$ and hence do not need to recompute them. Similarly, some of the $d(v,c)$ distances at line 9 is already known from line 6. The information stored by the data structure is the root a and the $N()$ and $R()$ values of all the nodes.

```
BuildTree (Node a, Set of nodes S)
 1. N(a) ← ∅        // neighbors of a
 2. R(a) ← 0        // covering radius
 4. For v ∈ S in increasing distance to a Do
 5.     R(a) ← max(R(a), d(v,a))
 6.     If ∀b ∈ N(a),  d(v,a) < d(v,b) Then N(a) ← N(a) ∪ {v}
 7. For b ∈ N(a) Do S(b) ← ∅
 8. For v ∈ S − N(a) Do
 9.     c ← argmin_{b∈N(a)} d(v,b)
10.     S(c) ← S(c) ∪ {v}
11. For b ∈ N(a) Do BuildTree (b, S(b))
```

Fig. 1. Algorithm to build the *sa-tree*.

2.2 Searching

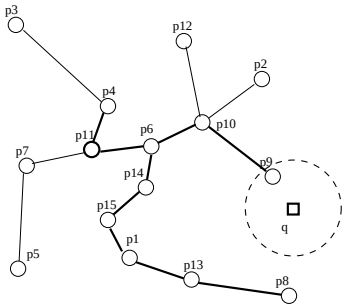
Of course it is of little interest to search only for elements $q \in S$. The tree we have described can, however, be used as a device to solve queries of any type for any $q \in \mathbb{U}$. We consider first range queries with radius r.

The key observation is that, even if $q \notin S$, the answers to the query are elements $q' \in S$. So we use the tree to pretend that we are searching for an element $q' \in S$. We do not know q', but since $d(q,q') \leq r$, we can obtain from q some distance information regarding q': by the triangle inequality it holds that for any $x \in \mathbb{U}$, $d(x,q) - r \leq d(x,q') \leq d(x,q) + r$.

Hence, instead of simply going to the closest neighbor, we first determine the closest neighbor c of q among $\{a\} \cup N(a)$. We then enter into *all* neighbors $b \in N(a)$ such that $d(q,b) \leq d(q,c) + 2r$. This is because the virtual element q' sought can differ from q by at most r at any distance evaluation, so it could have been inserted inside any of those b nodes. In the process, we report all the nodes q' we found close enough to q. (A more sophisticated search scheme is given in [6], but it cannot be applied to our dynamic version, so we prefer to omit it.)

Finally, the covering radius $R(a)$ is used to further prune the search, by not entering into subtrees such that $d(q,a) > R(a) + r$, since they cannot contain useful elements.

Figure 2 illustrates the search process on the left, starting from the tree root p_{11}. Only p_9 is in the result, but all the bold edges are traversed. On the right, we give the search algorithm, initially invoked as `RangeSearch(`a`,`q`,`r`)`, where a is the tree root. Note that in the recursive invocations $d(a, q)$ is already computed.




```
RangeSearch (Node a, Query q,
             Radius r)
1. If d(a,q) ≤ R(a) + r Then
2.    If d(a,q) ≤ r Then Report a
3.    d_min ← min {d(c,q), c ∈ {a} ∪ N(a)}
4.    For b ∈ N(a) Do
5.       If d(b,q) ≤ d_min + 2r Then
6.          RangeSearch (b,q,r)
```

Fig. 2. On the left, an example of the search process. On the right, the algorithm to search for q with radius r in a *sa-tree*.

We can also perform nearest neighbor searching by simulating a range search where the search radius is reduced as we proceed. We have a priority queue of subtrees sorted by the known lower bound distance between the subtree and q. Initially, we insert the *sa-tree* root in the data structure. Iteratively, we extract the (as far as it is known) closest subtree, process its root, and insert all its subtrees in the queue. This is repeated until the queue gets empty or the lower bound distance is larger than r. For lack of space we omit further details.

3 Incremental Construction

The construction of the *sa-tree* needs to know all the elements of S in advance. In particular, it is difficult to add new elements once the tree is already built. To insert a new element x, we should go down the tree by the closest neighbor until x must become a neighbor of the current node a, that is, until x is closer to a than to any $b \in N(a)$ (Condition 1). All the subtree rooted at a must be rebuilt from scratch, since some nodes that went into another neighbor could prefer now to get into the new neighbor x.

Several insertion alternatives have been previously considered [7, 6]. The best methods turned out to be the so-called "timestamping" and "insertion at the fringe". We propose here a novel technique based on ideas from these two methods.

Timestamping permits inserting an element with a technique very similar to that of the static construction, by recording the time every element was inserted. Remarkably, this technique obtained a performance very similar to that of the static version, by avoiding any reconstruction. Insertion at the fringe, on the other hand, limits the maximum tree size where a new element can be inserted, with the aim of reconstructing only small subtrees. The technique permits us avoiding insertion at the point where Condition 1 would require it, delaying it to a point downwards the tree. Surprisingly,

this technique even *improved* the performance in low dimensions, so there was a factor largely compensating the cost of the reconstructions. Where this factor came from was not clear at that time [7].

We have pursued this line and determined that the key fact is that these trees have a reduced arity. Moreover, the main reason of the poor performance of the *sa-tree* in low dimensional spaces is its excessively high arity (the tree automatically adapts its arity to the dimension, but not optimally). Hence we decided to focus directly on the maximum permitted arity and made it a tuning parameter. The same delaying technique used to limit the tree size to rebuild is now used to limit the tree arity. Moreover, by merging this technique with timestamping, we have no reconstruction cost to compensate, so we get the best of both worlds.

Observe that one of the nice features of the original *sa-tree* was that it had no parameter to set, so any non-expert could just use it. Our new parameter does not harm in this sense, because it can be set to ∞ to obtain the same performance of the original *sa-tree*. On the other hand, very large improvements can be obtained in low dimensions by appropriately setting the maximum tree arity. We get into the details now.

3.1 Insertion

To construct the *sa-tree* incrementally we fix a maximum tree arity, and also keep a timestamp of the insertion time of each element. When inserting a new element x, we add it as a neighbor at the appropriate point a (Condition 1) only if the arity of node a is not already maximal. Otherwise, even when x is closer to a than to any $b \in N(a)$, we force x to choose the closest neighbor in $N(a)$ and keep walking down the tree, until we reach a node a where Condition 1 is satisfied (x is closer to a than to 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.

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. Note also that now it is not sure anymore that a new inserted element x is a neighbor of the first node a that satisfies Condition 1 in its path. It may be that the arity of a was maximal and x was forced to choose a neighbor of a. This has implications in the search process that will be considered soon.

Figure 3 illustrates the insertion process. We follow only one path from the tree root to the parent of the inserted element. The function is invoked as `Insert(`a`,`x`)`, where a is the tree root and x is the element to be inserted. The *sa-tree* can now be built by starting with a first single node a where $N(a) = \emptyset$ and $R(a) = 0$, and then performing successive insertions.

Figure 4 compares the cost of incremental construction using our technique against static construction for increasing subsets of the database. We show arities 4, 8, 16 and 32. In both cases, the construction performance improves as we reduce the tree arity, being by far better than the static construction (twice as fast on strings and four times faster on vectors). Note that if we permit a sufficiently large arity (e.g., 32 on strings) the incremental version becomes somewhat worse than the static version (whose arity is unlimited). This shows that the reduced arity is a key factor in lowering construction costs.

```
Insert (Node a, Element x)
1.  R(a) ← max(R(a), d(a, x))
2.  c ← argmin_{b∈N(a)} d(b, x)
3.  If d(a, x) < d(c, x) ∧ |N(a)| < MaxArity Then
4.      N(a) ← N(a) ∪ {x}
5.      N(x) ← ∅,  R(x) ← 0
6.      time(x) ← CurrentTime
7.  Else Insert (c, x)
```

Fig. 3. Insertion of a new element x into a dynamic *sa-tree* with root a. $MaxArity$ is the maximum tree arity and $CurrentTime$ is the current time, incremented in each insertion.

This is clear, as the insertion cost with arity A is $A \log_A n$. On unlimited arity the average arity is $A = O(\log n)$, so the construction cost per element is $O(\log^2 n / \log\log n)$ [6]. We consider next how a reduced arity affects search time.

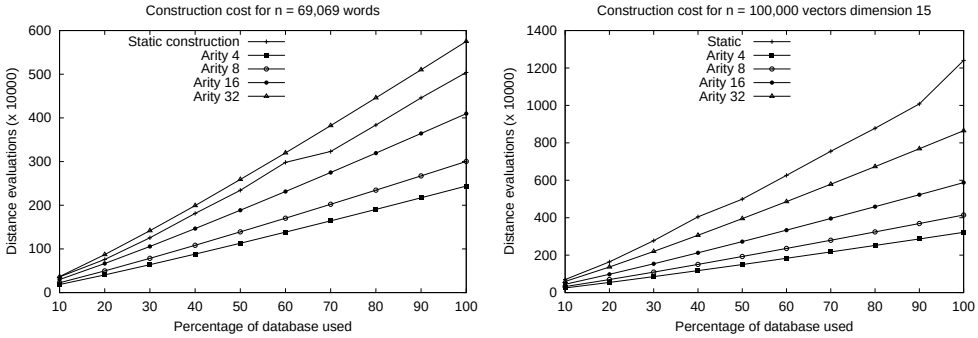


Fig. 4. Static versus dynamic construction costs.

3.2 Searching

At search time 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 instead of choosing the closest to x among $\{a\} \cup N(a)$, we may have chosen only among $N(a)$. This means that we have to remove $\{a\}$ from the minimization of line 3 in Figure 2. The second fact to consider is that, at the time x was inserted, elements with higher timestamp were not present in the tree, so x could choose its best 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 $d(q, b_i) \leq \min(d(q, b_1), \ldots, d(q, b_{i-1})) + 2r$. That is, we always enter into b_1; we enter into b_2 if $d(q, b_2) \leq d(q, b_1) + 2r$; 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 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 can 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 5 shows the search algorithm, initially invoked as `RangeSearch(`a`,`q`,`r`,`∞`)`, where a is the tree root. Note that $d(a, q)$ is always known except in the first invocation. Despite of the quadratic nature of the loop implicit in lines 4 and 6, the query is of course compared only once against each neighbor.

```
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 Do
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)}
```

Fig. 5. Searching q with radius r in a dynamic *sa-tree*.

Figure 6 compares this technique against the static one. In the case of strings, the static method provides slightly better search time compared to the dynamic technique. In vector spaces of dimension 15, arities 16 and 32 improve (by a small margin) the static performance. We have also included an example in dimension 5, showing that in low dimensions small arities largely improve the search time of the static method. The best arity for searching depends on the metric space, but the rule of thumb is that low arities are good for low dimensions or small search radii.

The percentage retrieved in the space of strings for search radius 1 is 0.003%, for 2 is 0.037%, for 3 is 0.326% and for 4 is 1.757% aproximately.

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 *sa-tree*, unlike other structures.

It is important to notice that we have obtained dynamism and also have improved the construction performance. In some cases we have also (largely) improved the search performance, while in other cases we have paid a small price for the dynamism. Overall,

this turns out to be a very convenient choice. This technique can be easily adapted to nearest neighbor searching with the same results.

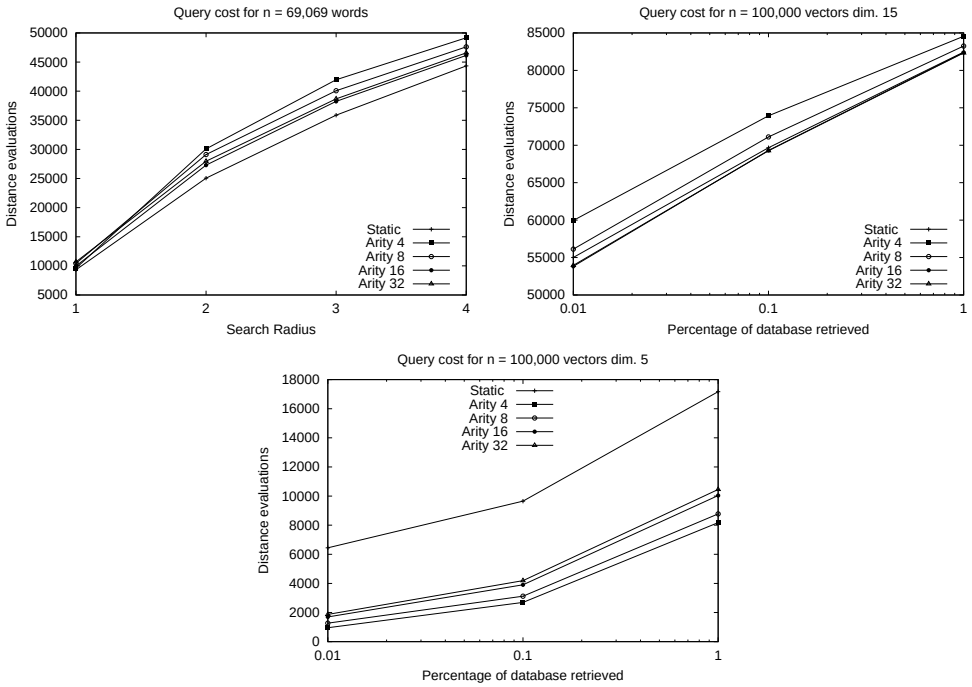


Fig. 6. Static versus dynamic search costs.

4 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 compu-

tations). 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 have studied several alternatives to delete elements from a dynamic *sa-tree*. 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.

4.1 Fake Nodes

Our first 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.

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.

4.2 Reinserting Subtrees

A widespread idea in the Euclidean range search community is that reinserting the elements of a disk page may be benefical 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 the 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 time the subtree of x was created, and new choices may exist now.

Figure 7 shows the algorithm to reinsert a tree with root y into a dynamic *sa-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`)`.

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

Fig. 7. Simple algorithm to reinsert a subtree with root y into a dynamic *sa-tree* with root a.

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.

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 $\log_A n$. As every (re)insertions costs $A \log_A n$, the average deletion cost is $A \log_A^2 n$. This is much more costly than an insertion.

4.3 Combining both Methods

We have two methods. Fake nodes delete elements for free but degrade the search performance of the tree. Subtree reinsertion make a costly subtree reinsertion but try to maintain the search quality of the tree. Note that the cost of reinserting a subtree would not be much different if it contained fake nodes, so we could remove all the fake nodes with a single subtree reinsertion, therefore amortizing the high cost of the reinsertion 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 as fake, we check if we have not unbalanced it. In this case, x is discarded and its subtrees reinserted. The only difference is that we never insert a subtree whose root is fake, rather, we split the subtree and discard the fake root.

A complication is that removing the subtree rooted at 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 three is balanced for $\alpha = 1/3$, but removing the leaf or marking 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 reinsert all the subtree rooted at x. Because of this complication, we reinsert whole subtrees only when they have no fake nodes.

This technique has a nice performance property. Even if we reinserted the elements one by one (instead of whole subtrees), we would have the guarantee that we would reinsert a subtree only when a fraction α of its elements were fake. This would mean that if the size of the subtree to rebuild were m, we would pay $m(1-\alpha)$ reinsertions for each αm deletions made in the subtree. Hence the amortized cost of a deletion would be at most $(1-\alpha)/\alpha$ times the cost of an insertion, that is, $(1-\alpha)/\alpha\ A \log_A n$. Asymptotically, the tree would work 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 $\alpha = 1$ and pure subtree reinsertion to $\alpha = 0$.

4.4 Experimental Comparison

Let us now compare the three methods to handle deletions on the space of words using arity 16. Figure 8 shows the deletion cost for the first 10% (left) or 40% (right) of the

database. On the left we have shown the case of full subtree reinsertion (that is, reinserting the subtrees after each deletion), with and without the final optimization proposed. As it can be seen, we save about 50% of the deletion cost with the optimization. We also show that one can only rarely insert whole subtrees, as reinserting the elements one by one has almost the same cost. Hence the algorithms could be simplified without sacrificing much. We also show the combined method with $\alpha = 1\%$, 3% and 5%. On the right we have shown much larger values of α, from 0% (full reinsertion) until 100% (pure fake nodes), as well as larger percentages of deletions (only the optimized version of reinsertions is used from now on).

We compare the methods 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.

It can be seen that, even with full reinsertion, the individual deletion cost is not so high. For example, the average insertion cost in this space is about 58 distance computations per element. With the optimized method, each deletion costs about 173 distance computations, i.e., 3 times the cost of an insertion. The combined method largely improves over this: using α as low as 1% we have a deletion cost of 65 distance computations, which is close to the cost of insertions, and with α =3% this reduces to 35.

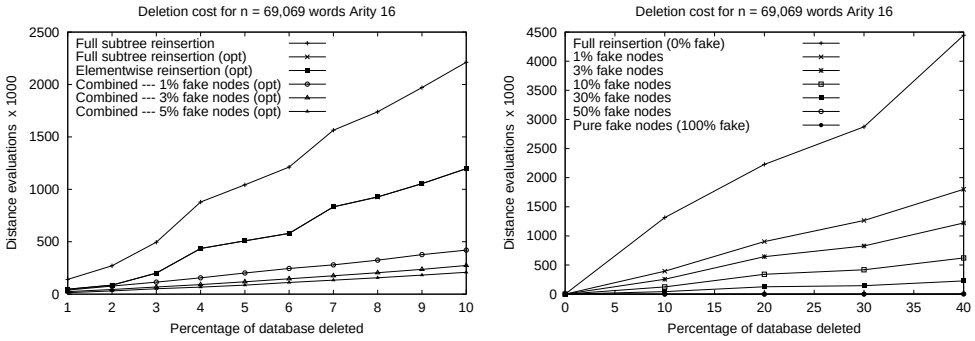


Fig. 8. Deletion costs using different methods.

Let us now consider how the search costs are affected 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 49,335 elements and then removed 14,800, so as to leave 34,534 elements (half of the set).

Figure 9 shows the results. As it can be seen, even with full reinsertions ($\alpha = 0\%$) the search quality degrades, albeit hardly noticeably and non-monotonically with the number of deletions made. As α grows, the search costs increase because of the need to enter every children of fake nodes. The difference in search cost ceases to be reasonable

as early as $\alpha = 10\%$, and in fact it is significant even for $\alpha = 1\%$. So one has to choose the right tradeoff between deletion and search cost depending on the application. A good tradeoff for strings is $\alpha = 1\%$.

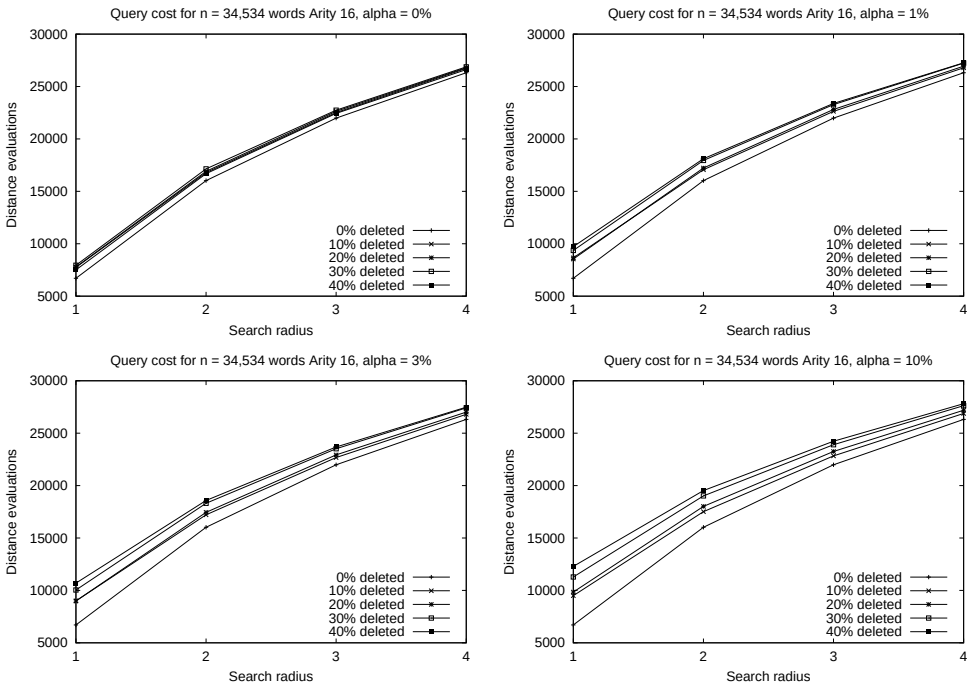


Fig. 9. Search costs using different deletion methods. In reading order we show the cases of $\alpha = 0\%$, 1%, 3% and 10%.

Figure 10 shows the same data in a way that permits comparing the change in search cost as α grows.

5 Conclusions

We have presented a dynamic version of the *sa-tree* data structure, which is able of handling insertions and deletions efficiently without affecting its search quality. Very few data structures for searching metric spaces are fully dynamic. Furthermore, we have shown how to improve the behavior of the *sa-tree* in low dimensional spaces, both for construction and search costs.

The *sa-tree* was a promising data structure for metric space searching, with several drawbacks that prevented it from being practical: high construction cost in low dimensional spaces, poor search performance in low dimensional spaces or queries with high selectivity, and unability to accommodate insertions and deletions.

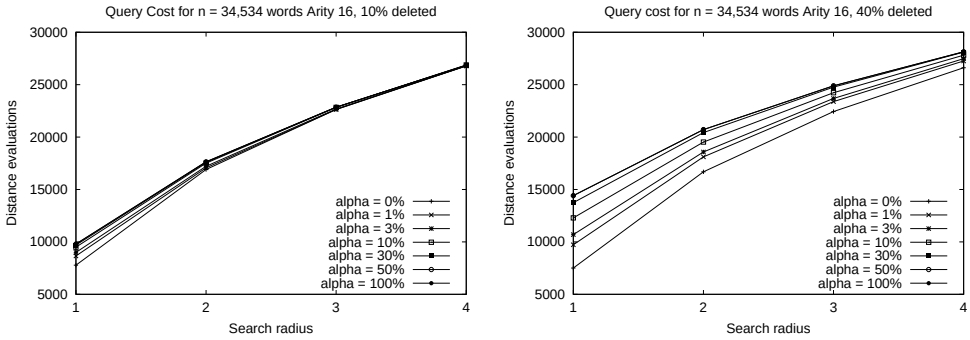


Fig. 10. Search costs using different deletion methods, comparing α. On the left we have deleted 10% of the database, on the right, 40%.

We have addressed all these weaknesses. Our new dynamic *sa-tree* stand 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.

As an example to give an idea of the behavior of our dynamic *sa-tree*, let us consider the space of vectors in dimension 15 using arity 16. We save 52.63% of the static construction cost, and improve the search time by 0.91% on average. A deletion with full element reinsertion costs on average 143 distance evaluations, which is 2.43 times the cost of an insertion. If we allow 10% of fake nodes in the structure, then the cost of a deletion drops to 17 and the search time becomes 3.04% worse than the static version.

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

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, 1998.
5. 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.
6. G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 2002. To appear.
7. 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.