Improved Dynamic Spatial Approximation Trees *

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

and

Nora Reyes
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. The main drawback of the *sa-tree* was that it was a static data structure, that is, once built, it was difficult to add new elements to it. This ruled it out for many interesting applications.

We have already proposed several methods to handle insertions in the *sa-tree*. In this paper we propose and study a new method that is an evolution over previous ones. We show that the new method is superior to the former and that it permits fast insertions while keeping a good search efficiency.

Keywords: databases, data structures, algorithms.

Resumen

El Arbol de Aproximación Espacial (*sa-tree*) es una estructura de datos para búsqueda en espacios métricos propuesta recientemente. Se ha mostrado que tiene buen desempeño comparada contra estructuras de datos alternativas en espacios de alta dimensión o consultas de baja selectividad. La principal desventaja que presentó *sa-tree* fue la de ser una estructura de datos estática, es decir, era dificultoso agregarle nuevos elementos una vez construida. Esto la descartaba para muchas aplicaciones interesantes.

Ya hemos propuesto varios métodos para manejar inserciones en el *sa-tree*. En este artículo proponemos un método nuevo que es una evolución sobre algunos previos. Mostramos que el nuevo método es superior a los primeros y que permite inserciones rápidas mientras que mantiene una buena eficiencia de búsqueda.

Palabras claves: bases de datos, estructuras de datos, algoritmos.

*This work was supported in part by CYTED VII.19 RIBIDI Project.

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 U of *objects*, and a non-negative *distance function* $d : U \times U \longrightarrow R^+$ 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 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: $L_p = (\sum_{1 \leq i \leq D} |x_i - y_i|^p)^{1/p}$. The best known special cases are $p = 1$ (Manhattan distance), $p = 2$ (Euclidean distance) and $p = \infty$ (maximum distance), that is, $L_\infty = \max_{1 \leq i \leq D} |x_i - y_i|$.

There are effective methods to search on D-dimensional spaces, such as kd-trees [1] or R-trees [4]. However, for roughly 20 dimensions or more those structures cease to work well. 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" 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]. In the extreme case we have a space where $d(x, x) = 0$ and $\forall y \neq x,\ d(x, y) = 1$, where it is impossible to avoid a single distance evaluation at search time. We say that a general metric space is high dimensional when its histogram of distances is concentrated.

There are a number of methods to preprocess the set in order to reduce the number of distance evaluations. 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, 7], 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 elements to it. These weaknesses make the *sa-tree* unsuitable for important applications such as multimedia databases.

Permitting incremental construction is the aim of this paper. We present a dynamic version of the *sa-tree* that handles insertions efficiently. 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.

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 [6], where it was shown that insertions on the *sa-tree* could be reasonably handled. 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.

2 The Spatial Approximation Tree

We describe briefly in this section the static *sa-tree* data structure. For lack of space we do not cover alternative structures for metric space searching; the reader is referred to [3] for details.

The *sa-tree* 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 references [5, 7].

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))
```

Figure 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.

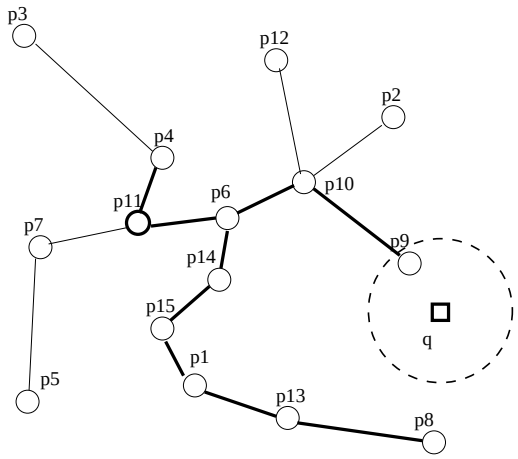
The key observation is that, even if $q \notin S$, the answers to the query are elements $q' \in S$. So we use the tree to pretend that we are searching for an element $q' \in S$. We do not know q', but since $d(q, q') \leq r$, we can obtain from q some distance information regarding q': by the triangle 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 [7], 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.

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




```
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)
```

Figure 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*.

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 *sa-tree* is a structure whose construction algorithm needs to know all the elements of S in advance. In particular, it is difficult to add new elements under the *best-fit* strategy once the tree is already built. Each time a new element is inserted, we must go down the tree by the closest neighbor until the new element must become a neighbor of the current node a. 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.

Several insertion alternatives have been previously considered [6, 7]. The best methods turned out to be the so-called "timestamping" and "insertion at the fringe". In this section we discuss and empirically evaluate these alternatives to permit insertion of new elements into an already built *sa-tree*. Then we propose a novel technique based on ideas from these two methods and we show that it is better than the others.

For the experiments we have selected two metric spaces. The first is a dictionary of 69,069 English words, from where we randomly chose queries. The distance in this case is the edit distance, that is, 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. We generated 100,000 random points with uniform distribution. For the queries, we build the indexes with 90% of the points and use the other 10% (randomly chosen) as queries. The results on these two spaces are representative of those on many other metric spaces we tested: NASA images, dictionaries in other languages, Gaussian distributions, other dimensions, etc.

As a comparison point for which follows, a static construction costs about 5 million comparisons for the dictionary and 12.5 million for the vector space.

Now we explain the two best methods previously proposed to permit insertion of new elements into an already built *sa-tree*. These methods show that it is possible to build the *sa-tree* incrementally.

3.1 Timestamping

This alternative has resemblances with other two also considered in [6], but it is more sophisticated consists in keeping a timestamp of the insertion time of each element. When inserting a new element, we add it as a neighbor at the appropriate point using *best-fit* and do not rebuild the tree. Let us consider that neighbors

are added at the end, so by reading them left to right we have increasing insertion times. It also holds that the parent is always older than its children.

At search time, we consider the neighbors $\{v_1, \ldots, v_k\}$ of a from oldest to newest. We perform the minimization while we traverse the neighbors, that is, we enter into the subtree of v_1 if $d(q, v_1) \leq d(q, a) + 2r$; into the subtree of v_2 if $d(q, v_2) \leq \min(d(q, a), d(q, v_1)) + 2r$; and in general into the subtree of v_i if $d(q, v_i) \leq \min(d(q, a), d(q, v_1), \ldots, d(q, v_{i-1})) + 2r$. This is because v_{i+j} can never take out an element from v_i.. This is because between the insertion of v_i and v_{i+j} there may have appeared new elements that preferred v_i just because v_{i+j} was not yet a neighbor, so we may miss an element if we do not enter into v_i because of the existence of v_{i+j}.

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

An alternative view, equivalent as before but focusing on maximum allowed radius instead of maximum allowed timestamp, is as follows. Each time we enter into a subtree y of v_i, we search for the siblings v_{i+j} of v_i that are older than y. Over this set, we compute the maximum radius that permits to avoid processing y, namely $r_y = \max(d(q, v_i) - d(q, v_{i+j}))/2$. If it holds $r < r_y$, we do not need to enter into the subtree y.

Let us now consider nearest neighbor searching. Assume that we are currently processing node v_i and insert its children y in the priority queue. We compute r_y as before and insert it together with y in the priority queue. Later, when the time to process y comes, we consider our current search radius r^* and discard y if $r^* < r_y$. If we insert a children z of y, we put it the value $\min(r_y, r_z)$.

This was an excellent alternative to the static construction in the case of our vector space example, providing basically the same construction and search cost with the added value of dynamism. In the case of the dictionary, the timestamping technique is significantly worse than the static one.

3.2 Inserting at the Fringe

Yet another good alternative also considered in [6] is as follows. We can relax Condition 1 (Section 2.1), whose main goal is to guarantee that if q is closer to a than to any neighbor in $N(a)$ then we can stop the search at that point. The idea is that, at search time, instead of finding the closest c among $\{a\} \cup N(a)$ and entering into any $b \in N(a)$ such that $d(q, b) \leq d(q, c) + 2r$, we exclude the subtree root $\{a\}$ from the minimization. Hence, we *always* continue to the leaves by the closest neighbor and others close enough. This seems to make the search time slightly worse, but the cost is marginal.

The benefit is that we are not forced anymore to put a new inserted element x as a neighbor of a, even when Condition 1 would require it. That is, at insertion time, even if x is closer to a than to any element in $N(a)$, we have the choice of not putting it as a neighbor of a but inserting it into its closest neighbor of $N(a)$. At search time we will reach x because the search and insertion processes are similar.

This freedom opens a number of new possibilities that deserve a much deeper study, but an immediate consequence is that we can insert always at the leaves of the tree. Hence, the tree is read-only in its top part and it changes only in the fringe. However, we have to permit the reconstruction of small subtrees so as to avoid that the tree becomes almost a linked list. So we permit inserting x as a neighbor when the size of the subtree to rebuild is small enough, which leads to a tradeoff between insertion cost and quality of the tree at search time.

As can be seen in [6], some reconstruction sizes yield the same and even better search time compared to the static version, which shows that it may be benefical to move elements downward in the tree. This fact makes this alternative a very interesting choice deserving more study.

4 A New Incremental Construction Technique

Timestamping permitted 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 [6].

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.

4.1 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.

```
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)
```

Figure 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, which is incremented after each new insertion.

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. This is clear, as the insertion cost with arity A is $A \log_A n$, while on unlimited arity it was shown in [7] that the average arity is $A = O(\log n)$, so the construction cost per element is $O(\log^2 n / \log\log n)$.

The question is how a reduced arity affects search time. We consider this next.

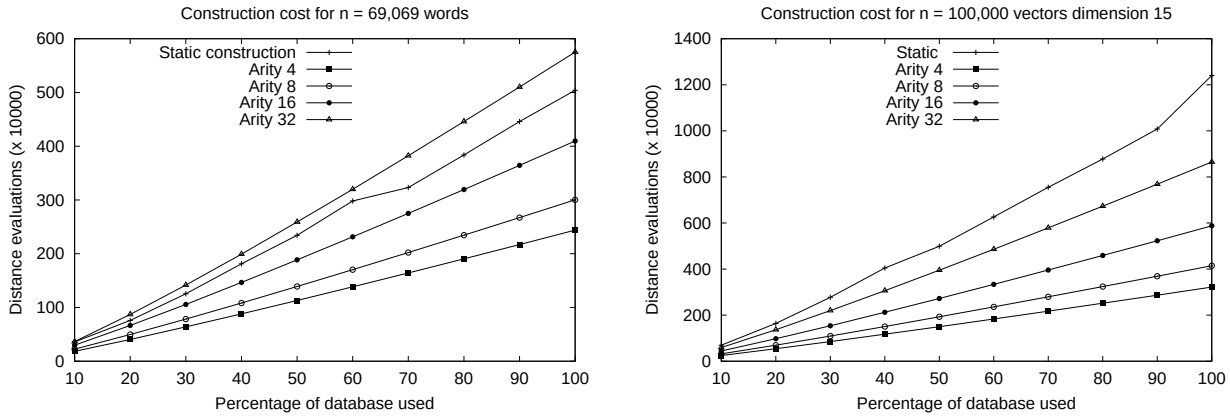


Figure 4: Static versus dynamic construction costs.

4.2 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.

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

```
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)}
```

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

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.

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.

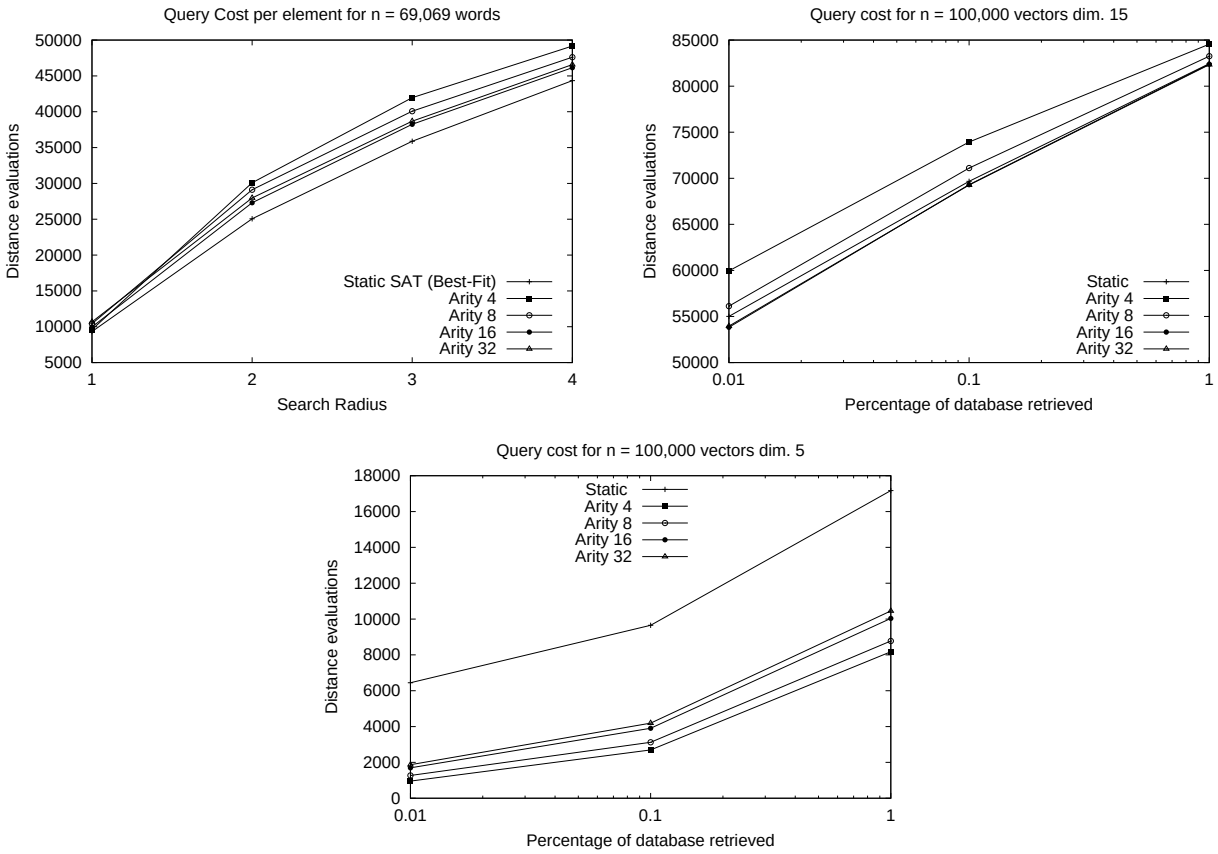


Figure 6: Static versus dynamic search costs.

This technique can be easily adapted to nearest neighbor searching with the same results, but we omit the description for lack of space.

We show now a comparison of construction and search costs between the previous incremental methods and the new method proposed. Figure 7 shows the construction and search costs in the space of strings, of the static *sa-tree*, timestamp, insertion at the fringe, and our new timestamp with bounded arity. We have used the best parameters for all the methods. Figure 8 shows the same comparison for the space of vectors in dimension 15.

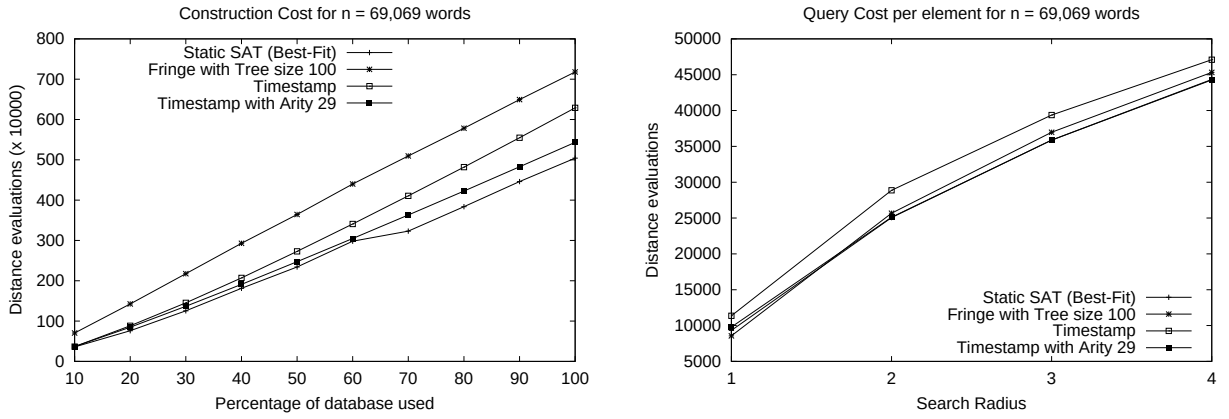


Figure 7: Construction and search costs for different methods in the space of strings.

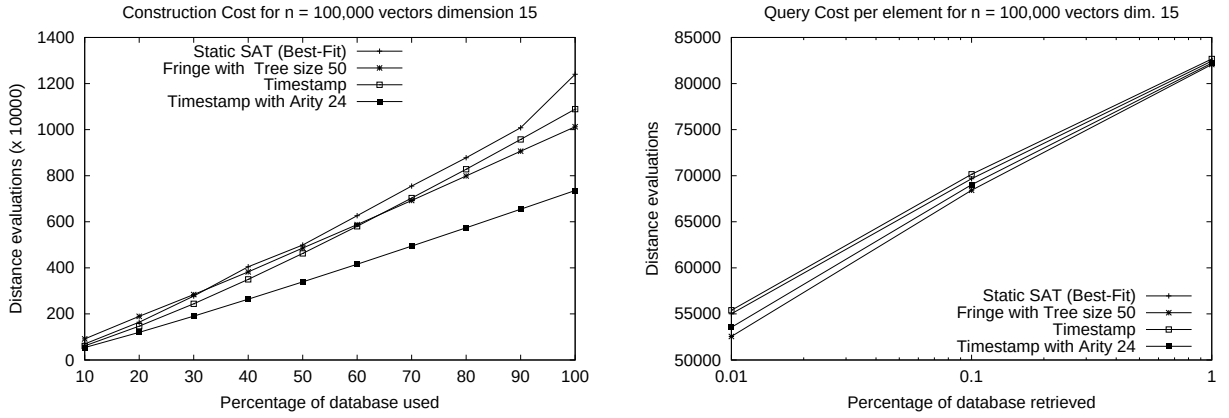


Figure 8: Construction and search costs for different methods in the space of vectors in dimension 15.

As it can be seen, our new method largely outperforms previous incremental methods at construction time, and at the same time it remains competitive (when not the best) at search time. In the case of coordinates it outperforms by far the static version, while in the case of strings it is almost equal, with the added value of dynamism.

We have introduced the tree arity as a new tuning parameter. A remaining issue is how difficult is it to find the best arity and how costly is a small error in this tuning.

We show some results about optimal arities for the two spaces considered. Figure 9 shows arities around the optimal value. The best arity found is 24 for the vector space in dimension 15 and arity 29 for the space of strings. It is important to note that, around these optimal, the costs differ by very few distance evaluations. Table 1 gives the numbers for the space of strings.

This shows that, if we fail to choose the best arity to construct the *sa-tree* by a reasonable margin, the price paid at search time is not very significative.

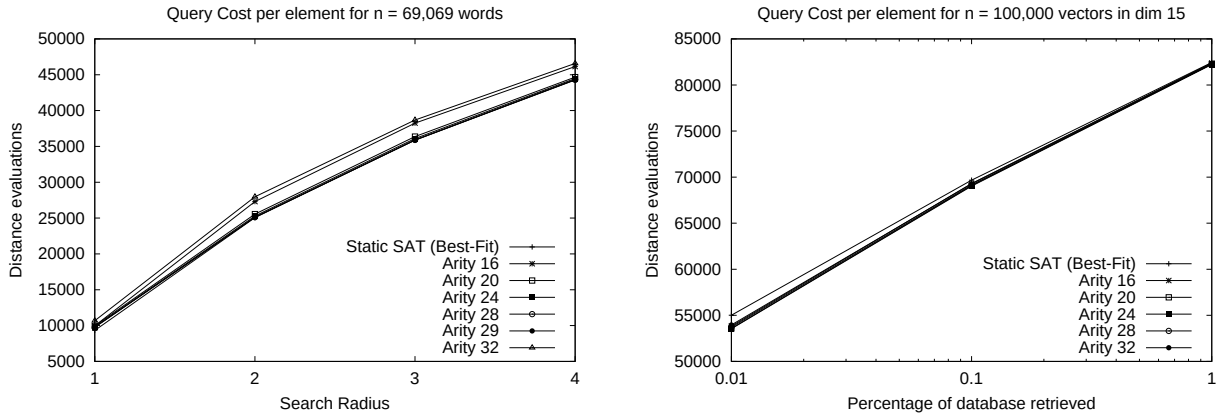


Figure 9: Static vs. dynamic search costs for different arities.

Method	Search Radius			
	1	2	3	4
Static	9324.89	25065.42	35883.64	44335.43
Arity 16	9972.58	27298.47	38249.70	46146.64
Arity 20	9958.61	25554.92	36367.81	44665.95
Arity 24	9816.64	25286.59	36088.06	44471.21
Arity 28	9762.68	25123.40	35898.67	44296.86
Arity 32	10679.11	27971.59	38685.50	46575.76
Arity 29	9795.26	25110.16	35862.23	44268.24

Table 1: Numeric values of Figure 9 (left).

5 Conclusions

We have presented a new technique to modify the *sa-tree* in order to make it a dynamic data structure supporting insertions, without degrading its current performance. We have shown that this new alternative is better than the others previously considered. In fact, only now we have understood the key factors that made our previous work [6] reasonably successful. In this paper we used that knowledge to combine the two best previous methods into a new one that has the best from each and outperforms both of them. Furthermore, we have shown how to improve the behavior of the *sa-tree* in low dimensional spaces, both for construction and search costs.

This work is a first step of a broader project [8] which aims at a fully dynamic data structure for searching in metric spaces, which can also work on secondary memory. We have not touched this last aspect in this paper but it is our current focus.

References

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

[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] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[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 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.

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

[8] N. Reyes. Dynamic data structures for searching metric spaces. MSc. Thesis, Univ. Nac. de San Luis, Argentina, 2002. In progress. G. Navarro, advisor.