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 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 *is that it is a static data structure, that is, once built, it is difficult to add new elements to it. This rules it out for many interesting applications.*

In this paper we overcome this weakness. We propose and study several methods to handle insertions in the sa-tree*. Some are classical folklore solutions well known in the data structures community, while the most promising ones have been specifically developed considering the particular properties of the* sa-tree*, and involve new algorithmic insights in the behavior of this data structure. As a result, we show that it is viable to modify the* sa-tree *so as to permit fast insertions while keeping its good search efficiency.*

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.

*Partially supported by Fondecyt grant 1-000929.

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: $L_p = (\sum_{1 \le i \le d} |x_i - y_i|^p)^{1/p}$. The best known special cases are $p = 1$ (Manhattan distance), $p = 2$ (Euclidean distance) and $p = \infty$ (maximum distance), that is, $L_\infty = \max_{1 \le i \le d} |x_i - y_i|$.

There are effective methods to search on d-dimensional spaces, such as kd-trees [2] or R-trees [13]. 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 [5, 10]. 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 [16], which is 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. It has been shown that the *sa-tree* behaves better 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*, unlike other data structures, does not have parameters to be tuned by the user of each application. This makes it very appealing as a general purpose data structure for metric searching, since any non-expert seeking for a tool to solve his/her particular problem can use it as a black box tool, without the need of understanding the complications of an area he/she is not interested in. Other data structures have many tuning parameters, hence requiring a big effort from the user in order to obtain an acceptable performance.

On the other hand, the main weakness of the *sa-tree* is that it is not dynamic. That is, once it is built, it is difficult to add new elements to it. This makes the *sa-tree* unsuitable for dynamic applications such as multimedia databases.

Overcoming this weakness is the aim of this paper. We propose and study several methods to handle insertions in the *sa-tree*. Some are classical folklore solutions well known in the data structures community, while the most promising ones have been specifically developed considering the particular properties of the *sa-tree*. As a result, we show that it is viable to modify the *sa-tree* so as to permit fast insertions while keeping its good search efficiency. As a related byproduct of this study, we give new algorithmic insights in the behavior of this data structure.

2. Previous Work

Algorithms to search in general metric spaces can be divided in two large areas: pivot-based and clustering algorithms. (See [10] for a more complete review.)

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

Several algorithms [23, 15, 7, 18, 6, 8] are almost direct implementations of this idea, and differ basically in their extra structure used to reduce the CPU cost of finding the candidate points, but not in their number of distance evaluations.

There are a number of tree-like data structures that use this idea in a more indirect way: they select a pivot as the root of the tree and divide the space according to the distances to the root. One slice corresponds to each subtree (the number and width of the slices differs across the strategies). At each subtree, a new pivot is selected and so on. The search backtracks on the tree using the triangle inequality to prune subtrees, that is, if a is the tree root and b is a children corresponding to $d(a, b) \in [x_1, x_2]$, then we can avoid entering in the subtree of b whenever $[d(q, a) - r, d(q, a) + r]$ has no intersection with $[x_1, x_2]$.

Several data structures use this idea [3, 22, 14, 24, 4, 25].

Clustering algorithms. The second trend consists in dividing the space in zones as compact as possible, normally recursively, and storing a representative point ("center") for each zone plus a few extra data that permits quickly discarding the zone at query time. Two criteria can be used to delimit a zone.

The first one is the *Voronoi area*, where we select a set of centers and put each other point inside the zone of its closest center. The areas are limited by hyperplanes and the zones are analogous to Voronoi regions in vector spaces. Let $\{c_1 \dots c_m\}$ be the set of centers. At query time we evaluate $(d(q,c_1),\dots,d(q,c_m))$, choose the closest center c and discard every zone whose center c_i satisfies $d(q,c_i) > d(q,c) + 2r$, as its Voronoi area cannot intersect with the query ball.

The second criterion is the *covering radius* $cr(c_i)$, which is the maximum distance between c_i and an element in its zone. If $d(q,c_i) - r > cr(c_i)$, then there is no need to consider zone i.

The techniques can be combined. Some techniques using only hyperplanes are [22, 19, 12]. Some techniques using only covering radii are [11, 9]. One using both criteria [5].

Nearest neighbor queries. To answer 1-NN queries, we simulate a range query with a radius that is initially $r^* = \infty$, and reduce r^* as we find closer and closer elements to q. At the end, we have in r^* the distance to the closest elements and have seen them all. Unlike a range query, we are now interested in quickly finding close elements in order to reduce r^* as early as possible, so there are a number of heuristics to achieve this. One of the most interesting is proposed in [21], where the subtrees yet to be processed are stored in a priority queue in a heuristically promising ordering. The traversal is more general than a backtracking. Each time we process the root of the most promising subtree, we may add its children to the priority queue. At some point we can preempt the search using a cutoff criterion given by the triangle inequality.

k-NN queries are handled as a generalization of 1-NN queries. Instead of one closest element, the k closest elements known are maintained, and r^* is the distance to the farthest to q among those k. Each time a new candidate appears we insert it into the queue, which may displace another element and hence reduce r^*. At the end, the queue contains the k closest elements to q.

3. The Spatial Approximation Tree

We describe briefly in this section the *sa-tree* data structure. It needs linear space $O(n)$, reasonable construction time $O(n \log^2 n / \log\log n)$ 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 improve over other data structures when the dimension is high or the query radius is large. For more details see the original references [16, 17].

3.1. 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 the following property:

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 3.2.

Figure 1 depicts the construction process. It is firstly 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, $d(v, c)$ at line 10 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)

N(a) ← ∅        /* neighbors of a */
R(a) ← 0        /* covering radius */
Sort S by distance to a (closer first)
for v ∈ S do
   R(a) ← max(R(a), d(v,a))
   if ∀b ∈ N(a), d(v,a) < d(v,b)
      then N(a) ← N(a) ∪ {v}
for b ∈ N(a) do S(b) ← ∅
for v ∈ S − N(a) do
   Let c ∈ N(a) be the one minimizing d(v,c)
   S(c) ← S(c) ∪ {v}
for b ∈ N(a) do BuildTree (b, S(b))
```

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

3.2. 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 $q \in U$. We start with 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 $x \in U$, $d(x, q) - r \leq d(x, q') \leq d(x, q) + r$.

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.

Moreover, notice that, in an exact search for a $q' \in S$, the distances between q' and the nodes we traverse get reduced as we step down the tree. That is,

Observation 1: Let $a, b, c \in S$ such that b descends from a and c from b in the tree. Then $d(c, b) \leq d(c, a)$.

The same happens, allowing a tolerance of $2r$, in a range search with radius r. That is, for any b in the path from a to q' it holds $d(q', b) \leq d(q', a)$, so $d(q, b) \leq d(q, a) + 2r$. Hence, while at first we need to enter into all the neighbors $b \in N(a)$ such that $d(q, b) - d(q, c) \leq 2r$, when we enter into those b the tolerance is not $2r$ anymore but it gets reduced to $2r - (d(q, b) - d(q, c))$.

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, starting from the tree root p_{11}. Only p_9 is in the result, but all the bold edges are traversed. Figure 3 gives the search algorithm, initially invoked as `RangeSearch(`a`,`q`,`r`,`$2r$`)`, where a is the tree root. Note that in the recursive invocations $d(a, q)$ is already computed.

Nearest neighbor searching. We can also perform nearest neighbor searching by simulating a range search where the search radius is reduced, just as explained at the end of Section 2. We have a priority queue of subtrees, most promising first. Initially, we insert the *sa-tree* root in the data structure. Iteratively, we extract the most promising subtree, process its root, and insert all its subtrees in the queue. This is repeated until the queue gets empty or its most promising subtree can be discarded (i.e., its promise value is bad enough). For lack of space we omit further details.

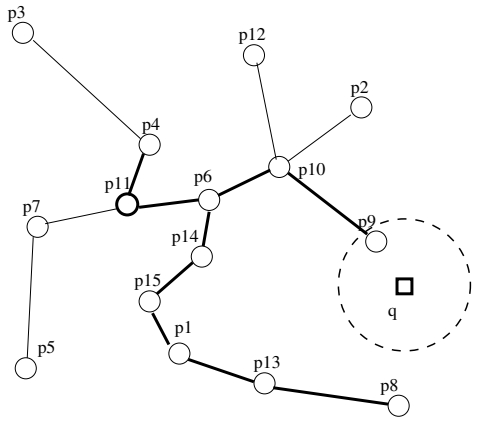


Figure 2. An example of the search process.

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

if d(a,q) ≤ r then Report a
if d(a,q) ≤ R(a) + r then
   d_min ← min{d(c,q), c ∈ {a} ∪ N(a)}
   for b ∈ N(a) do
      if d(b,q) − d_min ≤ t then
         RangeSearch (b,q,r,t − (d(b,q) − d_min))
```

Figure 3. Searching q with radius r in a *sa-tree*.

4. 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.

In this section we discuss and empirically evaluate different alternatives to permit insertion of new elements into an already built *sa-tree*. 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.

4.1. Rebuilding the Subtree

The naive approach rebuilds the whole subtree rooted at a once a new element x being inserted has to become a new neighbor of a. This has the advantage of preserving the same tree that is built statically, but, as Figure 4 shows for the case of the dictionary, the dynamic construction becomes too costly in comparison to a static one (140 times more costly in this example, almost 230 times more in our vector space).

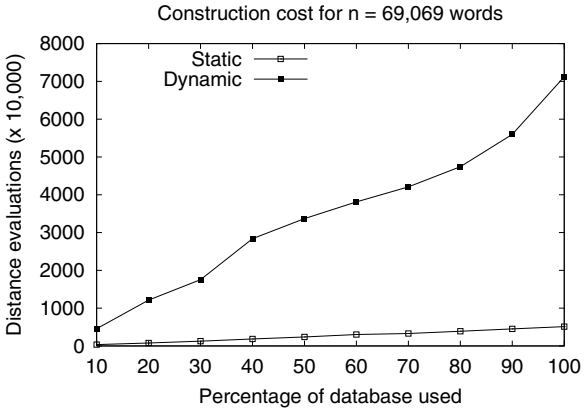
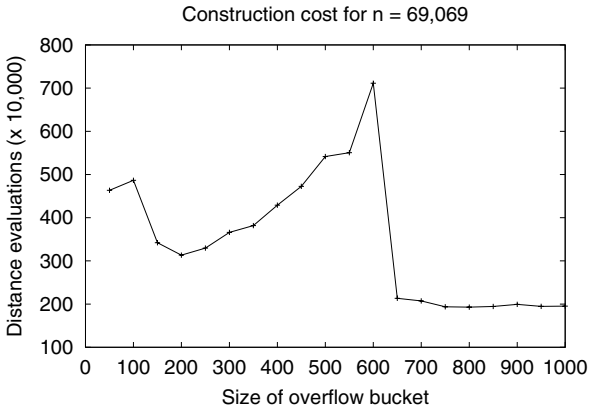


Figure 4. Construction cost by rebuilding subtrees.

4.2. Overflow Buckets

We can have an overflow bucket per node with "extra" neighbors that should go in the subtree but have not been classified yet. When the new element x must become a neighbor of a, we put it in the overflow bucket of a. Each time we reach a at query time, we also compare q against its overflow bucket and report any element near enough.

We must limit the size of the overflow buckets in order to maintain a reasonable search efficiency. We rebuild a subtree when its overflow bucket exceeds a given size. The main question is which is the tradeoff in practice between reconstruction cost and query cost. As smaller overflow buckets are permitted, we rebuild the tree more often and improve the query time, but the construction time raises.



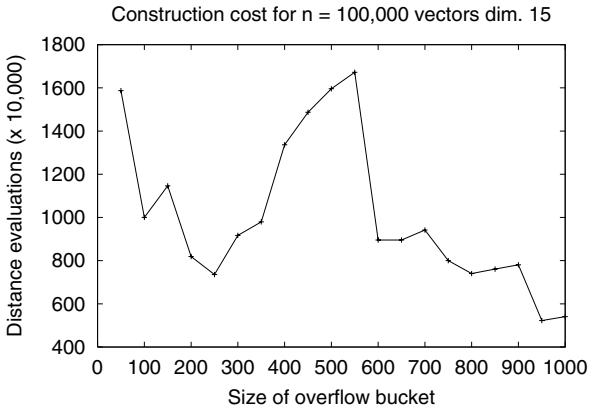
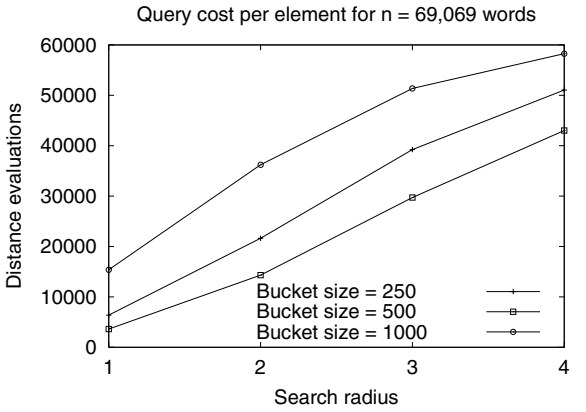


Figure 5. Construction costs using overflow buckets.



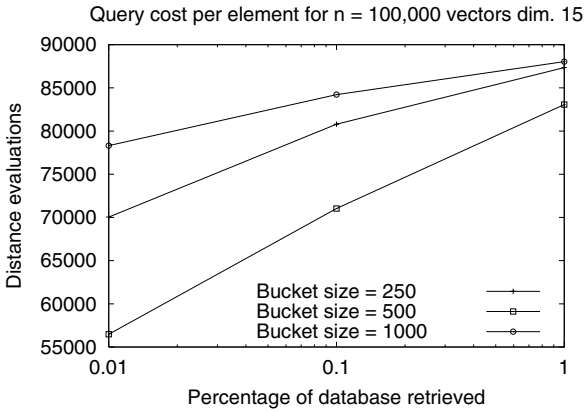


Figure 6. Search costs using overflow buckets.

Figure 5 shows the cost of the construction using different bucket sizes, which exhibits interesting fluctuations and in some cases costs even less than a static construction. This is possible because many unclassified elements are left in the buckets. For example, for bucket size 1,000, almost all the elements are in overflow buckets in the dictionary case and almost 60% in the vector case. These fluctuations appear because a larger bucket size may produce more rebuilds than a smaller one for a given set size n. The effect is well known, for example it appears when studying the number of splits as a function of the B-tree page size [1].

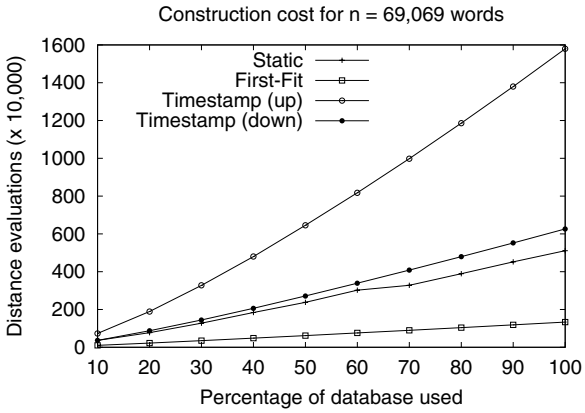
Figure 6 shows the search costs using overflow buckets. We searched with fixed radius 1 to 4 in the dictionary example and with radii retrieving 0.01%, 0.1% and 1% of the set in the vector example. We also performed nearest neighbor search experiments, which yielded similar results and are omitted for lack of space.

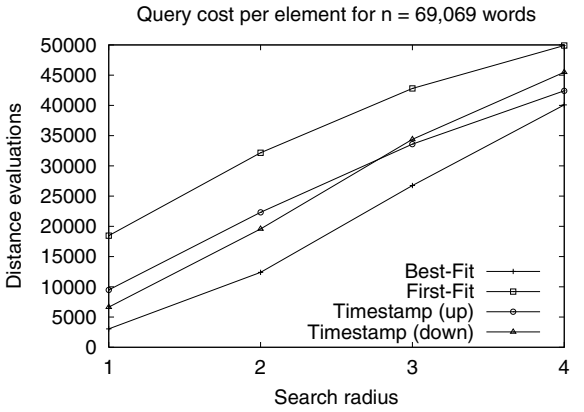
As can be seen by comparing the results to those of Figure 8, this technique is competitive against the static construction provided the correct bucket size is chosen. For example, with bucket size 500 we obtain almost the same search costs as for the static version, at the modest price of 10% extra construction cost for the dictionary and 30% for the vectors. The main problem in this method is its high sensitivity to the fluctuations, which makes it difficult to select a good bucket size. The intermediate bucket size 500 works well because at this point the elements in overflow buckets are 30% in the dictionary and 15% in the vectors.

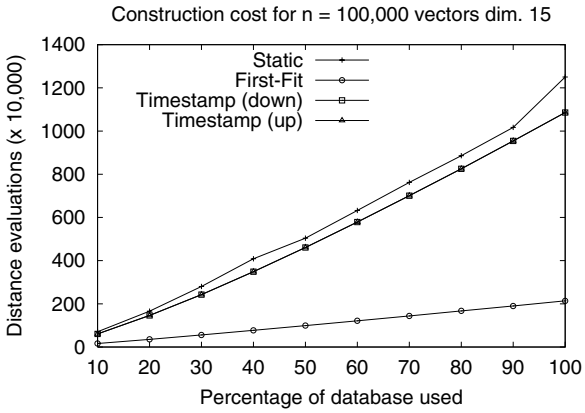
4.3. A First-Fit Strategy

Yet another solution is to change our *best-fit* strategy to put elements inside the bags of the neighbors of a at construction time. An alternative, *first-fit*, is to put each node in the bag of the first neighbor closer than a to q. Determining $N(a)$ and the bag of each other element can now be done all in one pass.

With the first-fit strategy, however, we can easily add more elements by pretending that the new incom-







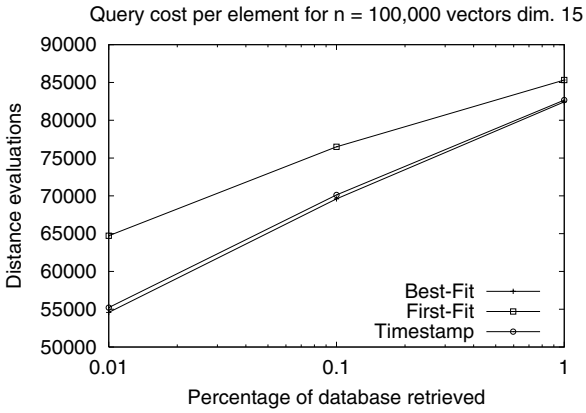


Figure 7. Construction costs using first-fit and using timestamps.

Figure 8. Search costs using first-fit and the two versions of the timestamping technique.

ing element x was the last one in the bag, which means that when it becomes a neighbor of a it can be simply added as the last neighbor of a, and there were no later elements that had the chance of getting into x. This allows building the structure by successive insertions.

Figure 7 shows that the construction (static or dynamic) using *first-fit* is much cheaper than using *best-fit*. Moreover, *first-fit* costs exactly the same and produces the same tree in the static or the dynamic case.

Range searching under the *first-fit* strategy is a little different. We consider the neighbors $\{v_1, \ldots, v_k\}$ of a in order. 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.

Figure 8 shows search times. As can be seen, the search overhead of the *first-fit* strategy is too high, at a point that makes the structure not competitive against other existing ones.

4.4. Timestamping

An alternative that has resemblances with the two previous but 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.

As seen in Figure 7, this alternative can cost a bit more or a bit less than static *best-fit* depending on the

case. Two versions of this methods, labeled "up" and "down" in the plot, correspond to how to handle the case of equal distances to the root and to the closest neighbor when inserting a new element. The former inserts the element as a new neighbor and the latter sends it to the subtree of the closest neighbor. This makes a difference only in discrete distances.

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, exactly as in Section 4.3. 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}.

Note that, although the search process is the same as under *first-fit*, the insertion puts the elements into their closest neighbor, so the structure is more balanced.

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 $\min(r_y, r_z)$.

Figure 8 compares this technique against the static one. As it can be seen, this is 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 (although the "up" behaves slightly better for nearest neighbor searching). The problem is that the "up" version is much more costly to build, needing more than 3 times the static construction cost.

4.5. Inserting at the Fringe

Yet another alternative is as follows. We can relax Condition 1 (Section 3.1), 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.

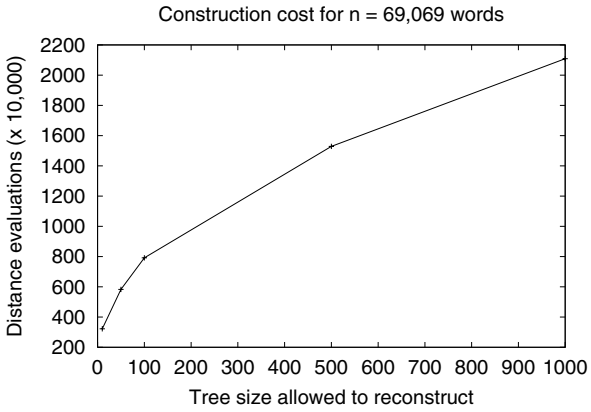
The benefit is that we are not forced anymore to put a new inserted element x as a neighbor of a, even when Condition 1 would require it. That is, at insertion time, even if x is closer to a than to any element in $N(a)$, we have the choice of not putting it as a neighbor of a but inserting it into its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

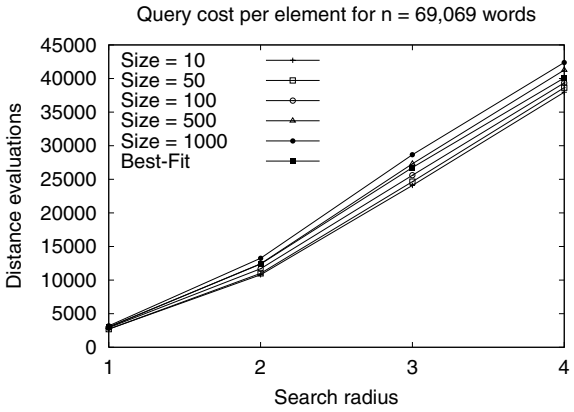
This freedom opens a number of new possibilities that deserve a much deeper study, but an immediate consequence is that we can insert always at the leaves of the tree. Hence, the tree is read-only in its top part and it changes only in the fringe.

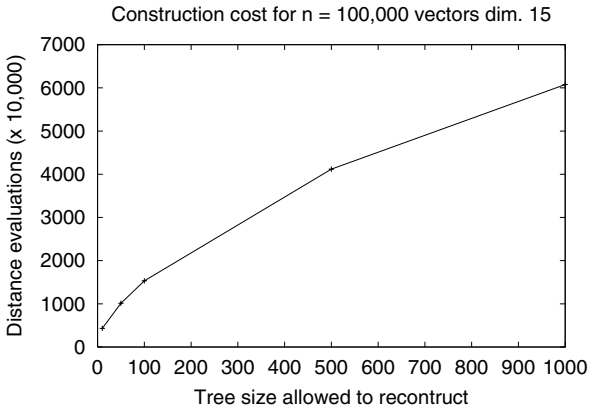
However, we have to permit the reconstruction of small subtrees so as to avoid that the tree becomes almost a linked list. So we permit inserting x as a neighbor when the size of the subtree to rebuild is small enough, which leads to a tradeoff between insertion cost and quality of the tree at search time.

Figure 9 shows the construction cost for different maximum tree sizes that can be rebuilt. As can be seen, permitting a tree size of 50 yields the same construction cost of the static version.

Finally, Figure 10 shows the search times using this technique. As can be seen, using a tree size of 50 permits 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.







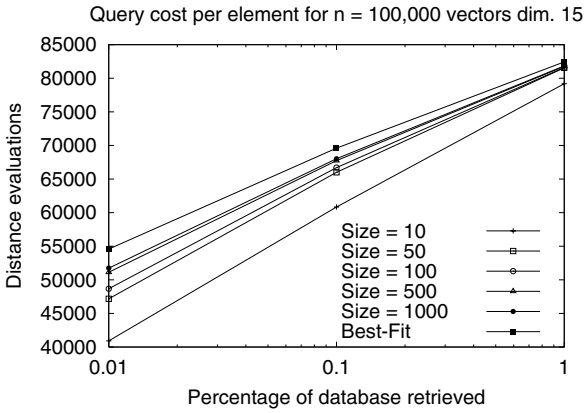


Figure 9. Construction costs inserting at the fringe.

Figure 10. Search costs using insertion in the fringe.

5. Conclusions

We have presented several techniques 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 there are many more alternatives than what appears at a first glance, and that the invariants of the *sa-tree* can be relaxed in ways unforeseen before this study (e.g. the fact that we can decide whether or not to add neighbors).

From the choices we have considered, the use of *overflow buckets* shows that it is possible to obtain construction and search times similar to those of the static version, although the choice of the bucket size deserves more study. *Timestamping* has also shown competitive in some metric spaces and not so attractive in others, a fact deserving more study. Finally, *inserting at the fringe* has shown the potential of even improving the performance of the static version, although studying the effect of the size of the fringe is required.

Other alternatives, such as *rebuilding* and *first-fit*, proved to be not competitive, although the latter offers very low construction costs, which could be interesting despite its much higher search cost.

It is clear now that making the *sa-tree* dynamic is affordable and that the structure can even be improved in a dynamic setup, contrary to our previous assumption that there would be a cost for the dynamism. On the other hand, we need to pursue more in the most promising alternatives in order to understand them better. Moreover, we have not considered deletions yet. These seem more difficult but always can be treated by marking the nodes as deleted and making periodic rebuilds.

This work is a first step of a broader project [20] 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. A simple solution to store the *sa-tree* in secondary storage is to try to store whole subtrees in disk pages so as to minimize the number of pages read at

search time. This has an interesting relationship with inserting at the fringe (Section 4.5), not only because the top part of the tree is read-only, but also because we can control the maximum arity of the tree so as to make the neighbors fit in a disk page.

References

[1] R. Baeza-Yates and P. Larson. Performance of B^+-trees with Partial Expansions. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):248–257, 1989.

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

[3] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, 1973.

[4] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. ACM Conference on Management of Data (SIGMOD'97)*, pages 357–368, 1997. Sigmod Record 26(2).

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

[6] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Conference on Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[7] E. Chávez, J. Marroquín, and R. Baeza-Yates. Spaghettis: an array based algorithm for similarity queries in metric spaces. In *Proc. 6th International Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 38–46. IEEE CS Press, 1999.

[8] E. Chávez, J. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, 2001. Kluwer.

[9] E. Chávez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *Proc. 7th International Symposium on String Processing and Information Retrieval (SPIRE'00)*, pages 75–86. IEEE CS Press, 2000.

[10] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 2001. To appear.

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

[12] F. Dehne and H. Nolteimer. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987. Pergamon Journals.

[13] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM Conference on Management of Data (SIGMOD'84)*, pages 47–57, 1984.

[14] L. Micó, J. Oncina, and R. Carrasco. A fast branch and bound nearest neighbor classifier in metric spaces. *Pattern Recognition Letters*, 17:731–739, 1996. Elsevier.

[15] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994. Elsevier.

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

[17] G. Navarro. *Searching in metric spaces by spatial approximation.* Technical Report TR/DCC-2001-4, Dept. of Computer Science, Univ. of Chile, 2001. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/jsat.ps.gz`.

[18] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, 1997.

[19] H. Nolteimer, K. Verbarg, and C. Zirkelbach. Monotonous Bisector* Trees – a tool for efficient partitioning of complex schenes of geometric objects. In *Data Structures and Efficient Algorithms*, LNCS 594, pages 186–203, 1992.

[20] N. Reyes. *Dynamic data structures for searching metric spaces.* MSc. Thesis, Univ. Nac. de San Luis, Argentina, 2001. In progress. G. Navarro, advisor.

[21] J. Uhlmann. Implementing metric trees to satisfy general proximity/similarity queries. Manuscript, 1991.

[22] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991. Elsevier.

[23] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[24] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*, pages 311–321, 1993.

[25] P. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, 2000.