Fully Dynamic and Memory-Adaptative Spatial Approximation Trees*

Diego Arroyuelo[1] Gonzalo Navarro[2] Nora Reyes[1]

[1] *Depto. de Informática*
Universidad Nacional de San Luis
Ejército de los Andes 950
San Luis, Argentina
{*darroy,nreyes*}*@unsl.edu.ar*

[2] *Center for Web Research*
Dept. of Computer Science
University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract

Hybrid dynamic spatial approximation trees are recently proposed data structures for searching in metric spaces, based on combining the concepts of spatial approximation and pivot based algorithms. These data structures are hybrid schemes, with the full features of dynamic spatial approximation trees and able of using the available memory to improve the query time. It has been shown that they compare favorably against alternative data structures in spaces of medium difficulty.

In this paper we complete and improve hybrid dynamic spatial approximation trees, by presenting a new search alternative, an algorithm to remove objects from the tree, and an improved way of managing the available memory. The result is a fully dynamic and optimized data structure for similarity searching in metric spaces.

Key Words: databases, data structures, algorithms, metric spaces.

*This work was partially supported by CYTED VII.19 RIBIDI Project (all authors) and Millenium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile (second author).

1 Introduction

"Proximity" or "similarity" searching is the problem of looking for objects in a set close enough to a query. This has applications in a vast number of fields. The problem can be formalized with the *metric space model* [2]: There is a universe $\mathcal{U}$ of objects, and a positive real-valued distance function $d : \mathcal{U} \times \mathcal{U} \longrightarrow \mathbb{R}^+$ defined among them, which satisfies the metric properties: *strict positiveness* ($d(x, y) = 0 \Leftrightarrow x = y$), *symmetry* ($d(x, y) = d(y, x)$), and *triangle inequality* ($d(x, z) \leqslant d(x, y) + d(y, z)$). The smaller the distance between two objects, the more "similar" they are. We have a finite database $S \subseteq \mathcal{U}$ that can be preprocessed to build an index. Later, given a *query* $q \in \mathcal{U}$, we must retrieve all similar elements in the database. We are mainly interested in the *range query*: Retrieve all elements in S within distance r to q, that is, $\{x \in S,\ \ d(x, q) \leqslant r\}$.

Generally, the distance is expensive to compute, so one usually defines the search complexity as the number of distance evaluations performed. Proximity search algorithms build an *index* of the database to speed up queries, avoiding the exhaustive search. Many of these indexes are based on pivots (Section 2).

In this paper we complete and improve a hybrid index for metric space searching built on the *dsa–tree* [3], an index supporting insertions and deletions that is competitive in spaces of medium difficulty, but unable of taking advantage of the available memory. This was enriched with a pivoting scheme in [1]. Pivots use the available memory to improve query time, and in this way they can beat any other structure, but too many pivots are needed in difficult spaces. Our new structure was still dynamic and made better use of memory, beating both *dsa-trees* and basic pivots. Now we present a new search alternative, a deletion algorithm, and a way of managing the available memory for *hybrid dynamic spatial approximation trees*. In this way we complete and improve our previous work [1].

2 Pivoting Algorithms

Essentially, pivoting algorithms choose some elements p_i from the database S, and precompute and store all distances $d(a, p_i)$ for all $a \in S$. At query time, they compute distances $d(q, p_i)$ against the pivots. Then the *distance by pivots* between $a \in S$ and q gets defined as $\mathcal{D}(a, q) = \max_{p_i} |d(a, p_i) - d(q, p_i)|$.

It can be seen that $\mathcal{D}(a, q) \leqslant d(a, q)$ for all $a \in S,\ q \in \mathcal{U}$. This is used to avoid distance evaluations. Each a such that $\mathcal{D}(a, q) > r$ can be discarded because we deduce $d(a, q) > r$ without actually computing $d(a, q)$. All the elements that cannot be discarded this way are directly compared against q.

Usually pivoting schemes perform better as more pivots are used, this way beating any other index. They are, however, better suited to "easy" metric spaces [2]. In hard spaces they need too many pivots to beat other algorithms.

3 Dynamic Spatial Approximation Trees

In this section we briefly describe dynamic *sa–trees* (*dsa-trees* for short), in particular the version called *timestamp with bounded arity* [3], on top of which we build.

3.1 Insertion Algorithm

The *dsa–tree* is built incrementally, via insertions. The tree has a maximum arity. Each tree node a stores a timestamp of its insertion time, $time(a)$, and its covering radius, $R(a)$, which is the maximum distance to any element in its subtree. Its set of children is called $N(a)$, the *neighbors* of a. To insert a new element x, its point of insertion is sought starting at the tree root and moving to the neighbor closest to x, updating $R(a)$ in the way. We finally insert x as a new (leaf) child of a if (1) x is closer to a than to any $b \in N(a)$, and (2) the arity of a, $|N(a)|$, is not already maximal. Neighbors are stored left to right in increasing timestamp order. Note that the parent is always older than its children.

3.2 Range Search Algorithm

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

Note that, at the time an element x was inserted, a node a may not have been chosen as its parent because its arity was already maximal. So, at query time, we must choose the minimum distance to x only among $N(a)$. Note also that, when x was inserted, elements with higher timestamp were not yet present in the tree, so x could choose its closest neighbor only among older elements. 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. That is, we enter into subtree b_i if $d(q, b_i) \leqslant \min\left(d(q, b_1), \ldots, d(q, b_{i-1})\right) + 2r$.

We use timestamps to reduce the work inside older neighbors. Say that $d(q, b_i) > d(q, b_{i+j}) + 2r$. We have to enter subtree b_i anyway because b_i is older. However, only the elements with timestamp smaller than $time(b_{i+j})$ 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 subtree b_i with timestamp larger than $time(b_{i+j})$ we can stop the search in that branch.

Algorithm 1 performs range searching. Note that, except in the first invocation, $d(a, q)$ is already known from the invoking process.

3.3 Deletion Algorithm

To delete an element x, the first step is to find it in the tree. In which follows, we do not consider the location of the object as part of the deletion problem, although in [3] we have shown how to proceed if necessary. 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.

The deletion of elements by *rebuilding subtrees* ensures that the resulting tree is exactly as if the deleted element had never been inserted. Thus, no degradation can occur due to repeated deletions. In such algorithm, when node $x \in N(a)$ is deleted, we disconnect x from the main tree. Hence all its descendants must be reinserted. Moreover, elements in the subtree of a that are younger than x have been compared against x to decide their insertion point. Therefore, these elements, in absence of x, could choose another path if we reinsert them into the tree. Then, we retrieve all the elements younger

RANGE SEARCH (Node a, Query q, Radius r, Timestamp t)

1. **if** $time(a) < t \ \wedge \ d(a,q) \leqslant R(a) + r$ **then**
2. **if** $d(a,q) \leqslant r$ **then** report a
3. $d_{min} \leftarrow \infty$
4. **for** $b_i \in N(a)$ in increasing timestamp order **do**
5. **if** $d(b_i,q) \leqslant d_{min} + 2r$ **then**
6. $k \leftarrow \min\{j > i, \ d(b_i,q) > d(b_j,q) + 2r\}$
7. RANGE SEARCH$(b_i, q, r, time(b_k))$
8. $d_{min} \leftarrow \min\{d_{min}, \ d(b_i,q)\}$

Algorithm 1: Range query algorithm on a *dsa–tree* with root a.

than x that descend from a (i.e. those whose timestamp is greater, which includes its descendants) and reinsert them into the tree, leaving the tree as if x had never been inserted.

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

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

Note that in this method the covering radii can become overestimated, because they are never reduced due to a deleted element. If we delete an element x, every $a \in A(x)$ such that x was the farthest element in its subtree will possibly have its $R(a)$ overestimated. In spite of it, this problem does not seem to affect much search performance since it does not significantly degrade over time (see [4] for more details).

4 Fully Dynamic Sa–trees with Pivots

Hybrid dynamic sa–trees were defined in [1], although without handling deletions. We review some of their main features and then present a deletion algorithm.

Pivoting techniques can trade memory space for query time, but they perform well on easy spaces only. A *dsa–tree*, on the other hand, is suitable for searching spaces of medium difficulty. However, it uses a fixed amount of memory, being unable of taking advantage of additional memory to improve query time. The idea is to obtain a hybrid data structure that gets the best of both worlds, by enriching *dsa–trees* with pivots. The result is better than both building blocks.

We choose different pivots for each tree node, such that *we do not need any extra distance evaluations against pivots*, either at insertion or search time. Recall that, after we find the insertion point of a new element x, say $x \in N(a)$, x has been compared against all its ancestors in the tree, all the

siblings of its ancestors, and its own siblings in $N(a)$. At query time, when we reach node x, some distances between q and the aforementioned elements have also been computed. So, we can use (some of) these elements as pivots to obtain better search performance, without introducing extra distance computations. Next we present different ways to choose the pivots of each node.

4.1 H–DSAT1: Using Ancestors as Pivots

A natural alternative is to regard the ancestors of each node as its pivots. Let $\mathcal{A}(x)$ be the set of ancestors of $x \in S$. We define $P(x) = \{(p_i,\ d(x,p_i)),\ \ p_i \in \mathcal{A}(x)\}$. This set is computed during insertion of x, by storing some of the distance evaluations computed in this process. We store $P(x)$ at each node x and use it to prune the search.

4.1.1 Insertion Algorithm

To insert an element x, we set $P(x) = \emptyset$ and begin searching for the insertion point of x. For each node a we choose in our path, we add $(a, d(x,a))$ to $P(x)$. When the insertion point of x is found, $P(x)$ contains the distances to the ancestors of x. Note that we do not perform any extra distance evaluations to build $P(x)$. Thus, the construction cost of a H–DSAT1 is *the same* of a *dsa–tree*.

4.1.2 Range Search Algorithm

For range searching, we modify the *dsa-tree* algorithm to use the set $P(x)$ stored at each tree node x. We recall that, given a set of pivots, the distance by pivots $\mathcal{D}(a,q)$ is a lower bound for $d(a,q)$.

Consider again Algorithm 1. If at step 1 it holds that $\mathcal{D}(a,q) > R(a) + r$, then surely $d(a,q) > R(a) + r$, and hence we can stop the search at node a without actually evaluating $d(a,q)$. An element a in S is said to be *feasible* for query q if $\mathcal{D}(a,q) \leqslant R(a) + r$. That is, is feasible that a or some element in its subtree lies within the search radius of q.

At search time, $\mathcal{D}(a,q)$ can be computed without additional evaluations of d. Suppose that we reach node p_k of the structure and want to decide if the search must follow into the subtree of $x \in N(p_k)$. At this point, we have computed all distances $d(q,p_i),\ p_i \in \mathcal{A}(x)$. If $\mathcal{A}(x) = \{p_1, \ldots, p_k\}$, then these distances are $d(q,p_1), \ldots, d(q,p_k)$. As the set $P(x) = \{(p_1, d(x,p_1)), \ldots, (p_k, d(x,p_k))\}$ is stored in the node of x, then the distances $d(x,p_i)$ and $d(q,p_i)$ needed to compute $\mathcal{D}(x,q)$ are present, at no extra costs.

The distances $d(q,p_i)$ are stored in a stack as the search goes up and down the tree. The sets $P(x)$ are also stored in root-to-x order, for example in a linear array, so that references to the pivots in $P(x)$ (first component of pairs) are unnecessary to correctly compute $\mathcal{D}$, and we save space.

The *covering radius feasible neighbors* of node a (feasible neighbors), denoted $F(a)$, are the neighbors $b \in N(a)$ such that $\mathcal{D}(b,q) \leqslant R(b) + r$. The other neighbors are said to be *infeasibles*.

At search time, if we reach node a, only the feasible neighbors of a could be taken into account, as other subtrees can be discarded completely. Observe that these subtrees have been discarded using $\mathcal{D}$ and not d and, as we have explained, $\mathcal{D}$ is computed for free. However, it does not immediately follow that we obtain for sure an improvement in search performance. The reason is that infeasible nodes still serve to reduce d_{min} in Algorithm 1, which in turn may save us entering into younger siblings. Hence, by saving computations against infeasible nodes, we may have to enter into new siblings later. This is an intrinsic price of our method.

RANGE SEARCH H–DSAT1 (Node a, Query q, Radius r, Timestamp t)

1. **if** $time(a) < t \ \wedge \ d(a,q) \leqslant R(a) + r$ **then**
2. **if** $d(a,q) \leqslant r$ **then** report a
3. $d_{min} \leftarrow \infty$
4. $F(a) \leftarrow \{b \in N(a), \ \mathcal{D}(b,q) \leqslant R(b) + r\}$
5. **for** $b_i \in N(a)$ in increasing timestamp order **do**
6. **if** $b_i \in F(a) \ \wedge \ \mathcal{D}(b_i,q) \leqslant d_{min} + 2r$ **then**
7. **if** $d(b_i,q) \leqslant d_{min} + 2r$ **then**
8. $k \leftarrow \min\{j > i, \ d(b_i,q) > d(b_j,q) + 2r\}$
9. RANGE SEARCH H–DSAT1$(b_i, q, r, time(b_k))$
10. **if** $d(b_i,q)$ has already been computed **then** $d_{min} \leftarrow \min\{d_{min}, \ d(b_i,q)\}$

Algorithm 2: Range searching for query q with radius r in a H–DSAT1 with root a.

Now we present a new search alternative not devised in [1]. The idea is to use $\mathcal{D}$ along with the *hyperplane criterion* to save distance computations at search time. For any feasible element b_i such that $\mathcal{D}(b_i,q) > d_{min} + 2r$, it holds that $d(b_i,q) > d_{min} + 2r$. Hence, we can stop the search in the feasible node b_i without evaluating $d(b_i,q)$.

In which follows we present different alternatives of the search algorithm. The Algorithm 2 shows the first alternative.

However, in step 8 we run into the risk of comparing infeasible elements against q. This is done in order to use timestamp information as much as possible, but it introduces the undesired effect of reducing the benefits of pivots in our data structure. We present some improvements to this weakness.

Optimizing using $\mathcal{D}$. We make use of $\mathcal{D}$ at search time not only to determine the feasibility of a node and prune the search space saving distance evaluations, but also to decrease the number of infeasible elements that are compared directly against q in step 8 of the algorithm. We search inside b_i using the timestamp t of a younger sibling b_k of b_i. Fortunately, some of the necessary comparisons can be saved by making use of $\mathcal{D}$. The key observation is that $d(b_i,q) \leqslant \mathcal{D}(b_j,q) + 2r$ implies $d(b_i,q) \leqslant d(b_j,q) + 2r$, so if $d(b_i,q) \leqslant \mathcal{D}(b_j,q) + 2r$ we can conclude that b_j is not of interest in step 8, hence saving the computation of $d(b_j,q)$. Although we save some distance computations and obtain the same result, still there will be infeasible elements compared against q. We call H–DSAT1D this search method.

Using Timestamps of Feasible Neighbors. The use of timestamps is not essential for the correctness of the algorithms. Any larger value would do, although the optimal choice is to use the smallest correct timestamp. Another alternative is to compute a safe approximation to the correct timestamp, but ensuring that no infeasible elements are ever compared against q. Note that every feasible neighbor of a node will be compared against q inevitably. If for $b_i \in F(a)$ it holds that $d(b_i,q) \leqslant d_{min} + 2r$, then we compute the oldest timestamp t among the reduced set $\{b_{i+j} \in F(a), \ d(b_i,q) > d(b_{i+j},q) + 2r\}$, and stop the search inside b_i at nodes whose timestamp is newer than t. This ensures that only feasible

elements are compared against q, and under that condition it uses as much timestamping information as possible. We call H–DSAT1F this alternative.

4.1.3 Deletion Algorithm

We adapt the algorithm of *rebuilding subtrees* [4] to delete an element $x \in N(a)$ from the tree, such that it takes into account the existence of pivots. The reinsertion process involve distance evaluations, some of which are already precomputed as pivot information. We show how to take advantage of this.

Note that a is a pivot of every element in its subtree. In other words, the distance $d(b, a)$ has been stored in $P(b)$, for every b in the subtree of a. As a result, we can save at least one distance evaluation for each element to be reinserted. Furthermore, if y is an older sibling of x, and y is a pivot (ancestor) of b then we can save the distance evaluation $d(b, y)$ when reinserting b.

As $P(b)$ is stored using a linear array, the position of $d(b, a)$ in $P(b)$ can be easily computed. If a lies at level i of the tree, then $d(b, a)$ is at position i of the array.

It is important to note that, after reinserting b, the node a and the ancestors of a will be ancestors of b. Then, we must keep in $P(b)$ the distances between b and the ancestors of a. The other distances in $P(b)$ are discarded before reinserting b. Finally, the new set $P(b)$ is completed using some of the distance evaluations produced at reinsertion time.

The deletion process of $x \in N(a)$ can be resumed as follows. We retrieve all the elements younger than x from the subtree rooted a, then disconnect them from the main tree, discard the distances to the pivots that are not of interest after reinserting the node, sort the nodes in increasing order of timestamp and reinsert them one by one (reusing some distances in P), searching their reinsertion point from a.

The result is very important: we do not introduce extra distance evaluations in the deletion process, and even more, the deletion cost can be reduced by using pivots.

4.2 H–DSAT2: Using Ancestors and their Older Siblings as Pivots

We aim at using even more pivots than H–DSAT1, to improve even more the search performance. At search time, when we reach a node a, q has been compared against all the ancestors and some of the older siblings of ancestors of a. Hence, we use this extended set of pivots for each node a.

4.2.1 Insertion Algorithm

The only difference in a H–DSAT2 is in the $P(x)$ sets we compute. Let $x \in S$ and $\mathcal{A}(x) = \{p_1, \ldots, p_k\}$ be the set of its ancestors, where p_i is the ancestor at tree level i. Note that $p_{i+1} \in N(p_i)$. Hence, $(b, d(x, b)) \in P(x)$ if and only if (1) $b \in \mathcal{A}(x)$, or (2) $p_i, p_{i+1} \in \mathcal{A}(x) \wedge b \in N(p_i) \wedge time(b) < time(p_{i+1})$.

4.2.2 Range Search Algorithm

For range searching, to compute $\mathcal{D}(x, q)$ we need the distances between q and the pivots of x stored in a stack. But it is possible that some of the pivots of x have not been compared against q because they were infeasible. In order to retain the same pivot order of $P(x)$, we push invalid elements into the stack when infeasible neighbors are found. $\mathcal{D}$ is then computed having this in mind. We define the same variants of the search algorithm for H–DSAT2, which only differ from H–DSAT1 in the way of computing $\mathcal{D}$.

4.2.3 Deletion Algorithm

Now suppose we want to delete an element $x \in N(a)$ in H–DSAT2. Again, a is a pivot of each element b in the subtree rooted a. Thus, we can avoid at least one distance evaluation for element to be reinserted. But we can do more. The comparison $d(b, y)$ can be avoided, for all $y \in N(a)$ older than x, since those distances have been stored in $P(b)$. The process of computing the position of the distances in $P(b)$ is not so direct as before: these must be computed as we retrieve the nodes to be reinserted. Because of the features of H–DSAT2, it is possible that the deletion cost can be reduced even more.

5 Limiting the Use of Storage

In practice, available memory is bounded. Our data structures, as have been defined, use memory in a non-controlled way (each node uses as much pivots as the definition requires). This fact rules out our solutions for many real-life situations. We show how to adapt the structures to fit the available memory. The idea is to restrict the number of pivots stored in each node to a value k, by holding a subset of the original set of pivots. As a result, the data structures lose some performance at search time. A way of minimizing such degradation is to choose a "good" set of pivots for each node.

5.1 Choosing Good Pivots

We study empirically the features of the pivots that discard elements at search time. In our experiment, each time a pivot discards an element, we mark that pivot (for more details see Section 6).

Because of the insertion process of H–DSAT1, the latest pivots of a node should be good since they are close, and hence good representatives, of the node. We verify experimentally that most discards using pivots were due to the latter ones. Figure 1 (left) shows that a small number of latter pivots per node suffices. In dimension 5, about 10 pivots per node discard all the elements that can be discarded using pivots. In higher dimensions, even less pivots are needed. We call H–DSAT1 k Latest to this alternative.

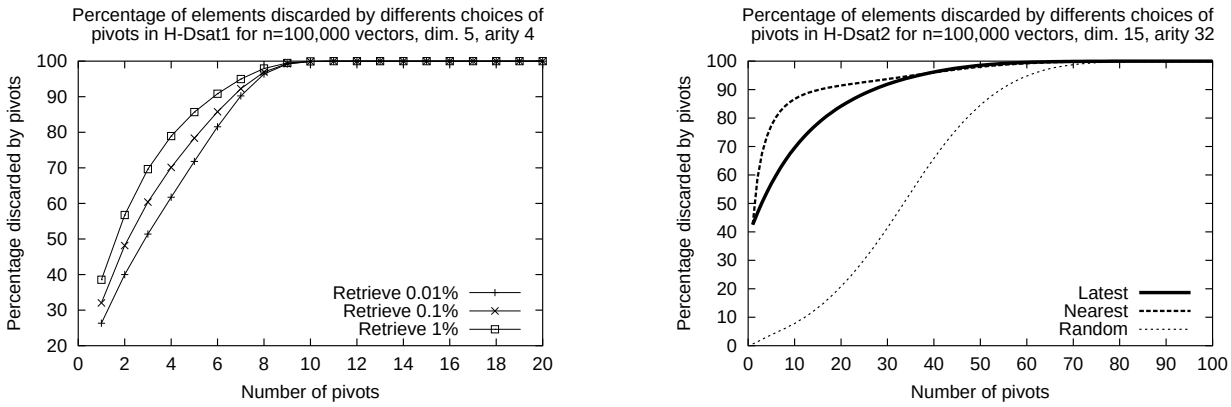

Figure 1: Percentage of elements discarded using the latest pivots in H–DSAT1 (left), and using the latest and nearest pivots in H–DSAT2 (right).

The ancestors of a node are close to it, but the siblings of the ancestors are not necessarily close. So we expect that using the k latest pivots in H–DSAT2 (H–DSAT2 k Latest) does not perform as well as before. An obvious alternative is H–DSAT2 k Nearest, which uses the k nearest pivots, not the k latest. Figure 1 (right) confirms that less nearest pivots are needed to discard the same number of nodes as latest pivots. However, note that for H–DSAT2 k Nearest we need to store the references to the pivots in order to compute $\mathcal{D}$. Hence, given a fixed amount of memory, this alternative must use less pivots per node than the others.

We have introduced a parameter k in our data structures, which can be easily tuned since it depends on the available memory. When $k = 0$ the data structure becomes the original *dsa–tree*, and when $k = \infty$ it becomes our unrestricted data structures of Section 4.

5.2 Choosing Good Nodes

The dual question is whether some tree nodes profit more from pivots than others. We experimentally study the features of the elements that are discarded using pivots. The result is that, for all the metric spaces used, the discarded elements are located near the leaves in the tree. In the vector space of dimension 5, the percentage varies from 40% to 60% (depending of the query radius), while in the space of dimension 15, almost the 100% of the elements discarded by pivots are leaves. In the dictionary this percentage varies from 80% to 90%.

The reason is that the covering radii of the nodes decrease as we go down in the tree, being zero in the leaves. As the covering radius infeasibility condition for a node a is $\mathcal{D}(a, q) > R(a) + r$, the probability of discarding a increases when $R(a)$ decreases.

Suppose that we restrict the number of pivots per node to a value k. As leaves are discarded more frequently than internal nodes, we consider an alternative that profits from this fact when using limited memory. The idea is to move the storage of pivots to the leaves smoothly and dynamically.

We have a parameter ρ, which is $0 \leqslant \rho \leqslant 1$. rho allows us to determine the number of pivots per node such that: *(1)* internal nodes have ρk pivots (unless they do not have so many to choose), and *(2)* external nodes have all the pivots that the scheme permits (unless there is not enough available space). The way to implement this is as follows: When an external node becomes internal it retains ρk of its pivots, and it yields the others to the public repository, and when a new external node appears it takes from the repository all the pivots that it needs (whenever the repository has that many, in other case it takes all the available ones).

In this way, each new element attempts to take a number of pivots as close as possible to its original number of pivots, and memory usage tends to move dynamically to the leaves. The parameter ρ allows us to control the degree of movement of storage to the leaves. Note that when $\rho = 0$, all the pivots move to the leaves, and when $\rho = 1$ the memory management has no effect.

Figure 2 shows the experimental query cost for H–DSAT2 k Nearest, in the vector space of dimension 15, using $k = 5$ and $k = 35$ pivots, and values 0, 0.1, 0.5, and 1 for ρ. In this metric space, we get the best performance with $\rho = 0$.

6 Experimental Results

In this section we present a series of experiments performed on our data structures. We have evaluated our structures in three metric spaces. First, a dictionary of 69,069 English words under edit distance

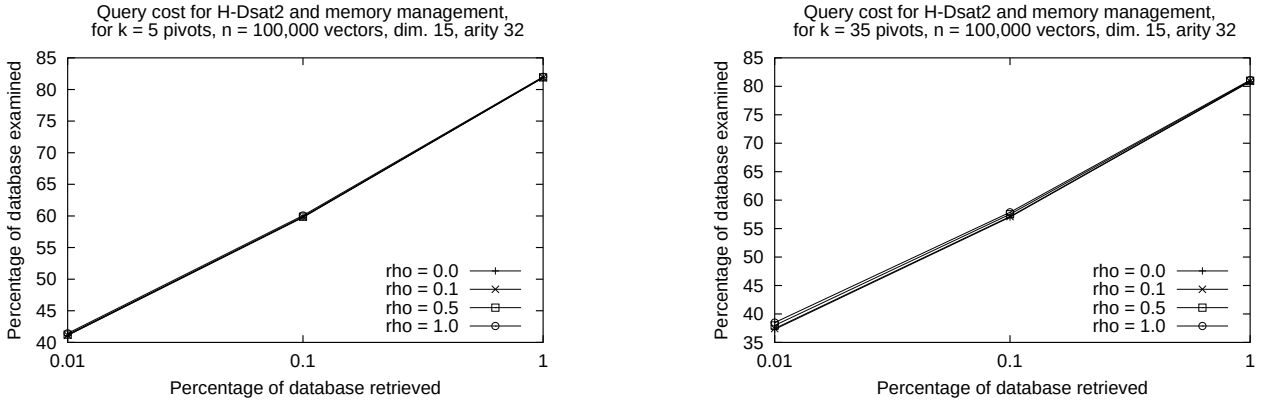

Figure 2: Performance of H–DSAT2 k Nearest with memory management, and various values of ρ. We selected values of $k = 5$ pivots (left) and $k = 35$ pivots (right).

(minimum number of character insertions, deletions and substitutions to make the strings equal), of interest in spelling applications. The other spaces are real unitary cubes in dimensions 5 and 15 under Euclidean distance, using 100,000 uniformly distributed random points. We treat these just as metric spaces, disregarding coordinate information.

In all cases, we left apart 100 random elements to act as queries. The data structures were built 20 times varying the order of insertions. We tested arities 4, 8, 16, and 32. Each tree built was queried 100 times, using radii 1 to 4 in the dictionary, and radii retrieving 0.01%, 0.1%, and 1% of the set in vector spaces.

In [1] we show that H–DSAT1F outperformed H–DSAT1D, clearly in the dictionary and slightly in vector spaces. The results are similar on H–DSAT2. Also we show experimentally that our structures are competitive, as our best versions of H–DSAT1 and H–DSAT2 largely improve upon *dsa–trees*. This shows that our structures make good use of extra memory. H–DSAT2 can use more memory than H–DSAT1, and hence its query cost is better.

However, there is a price in memory usage, e.g., H–DSAT1 needs 1.3 to 4.0 times the memory of *dsa–tree*, while H–DSAT2 requires 5.2 to 17.5 times. Hence the interest in comparing how well our structures use limited memory compared to others. Figure 3 and Figure 4 compare against a generic pivot data structure, using the same amount of memory in all cases. We also show a *dsa–tree* as a reference point, as it uses a fixed amount of memory. In easy spaces (dimension 5 or dictionary) we do better when there is little available memory, but in dimension 15 H–DSAT2 is always the best. More pivots are needed to beat H–DSAT in harder problems.

7 Conclusions

In this paper we have completed a hybrid scheme for similarity searching in metric spaces. Such scheme is basically a *dsa–tree*, except that a set of pivots is associated with each node. The set of pivots of each node is chosen in such a way that no extra distance evaluations are introduced at insertion time: we only save some of the distance computations that occur when inserting a new element. At search time, when we reach a node, we use pivots to prune the search space for free.

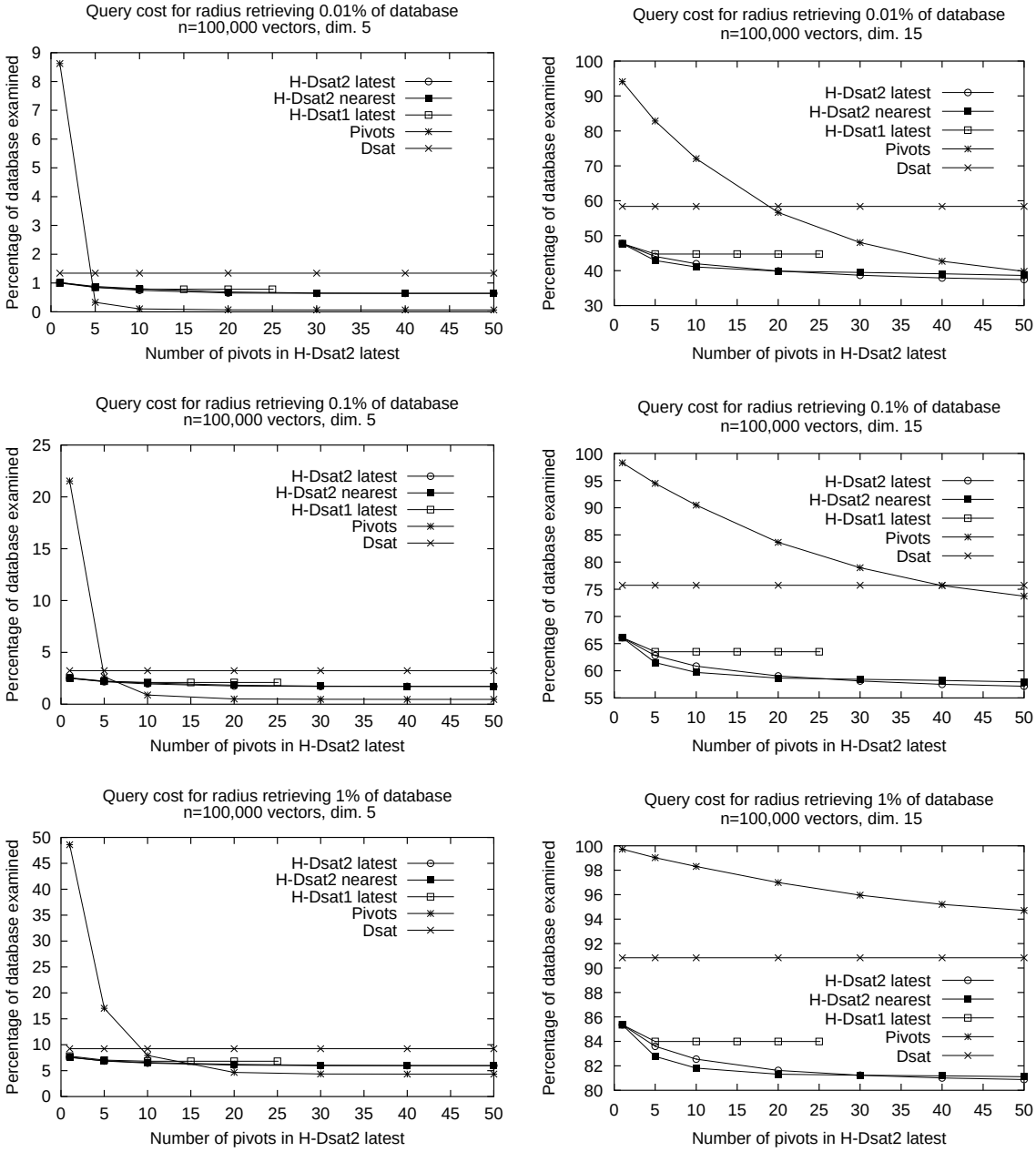

Figure 3: Query cost of H–DSAT1F and H–DSAT2F versus a pivoting algorithm, in vector spaces.

We study the way to choose good pivots when the amount of memory is limited. Several alternatives are explored and evaluated.

In this paper we have also presented a method to delete elements from a *hybrid dynamic spatial approximation tree*. This method has shown to be better than that of the original method over a *dsa–tree*.

The outcome is a fully dynamic data structure that can be managed through insertions and dele-

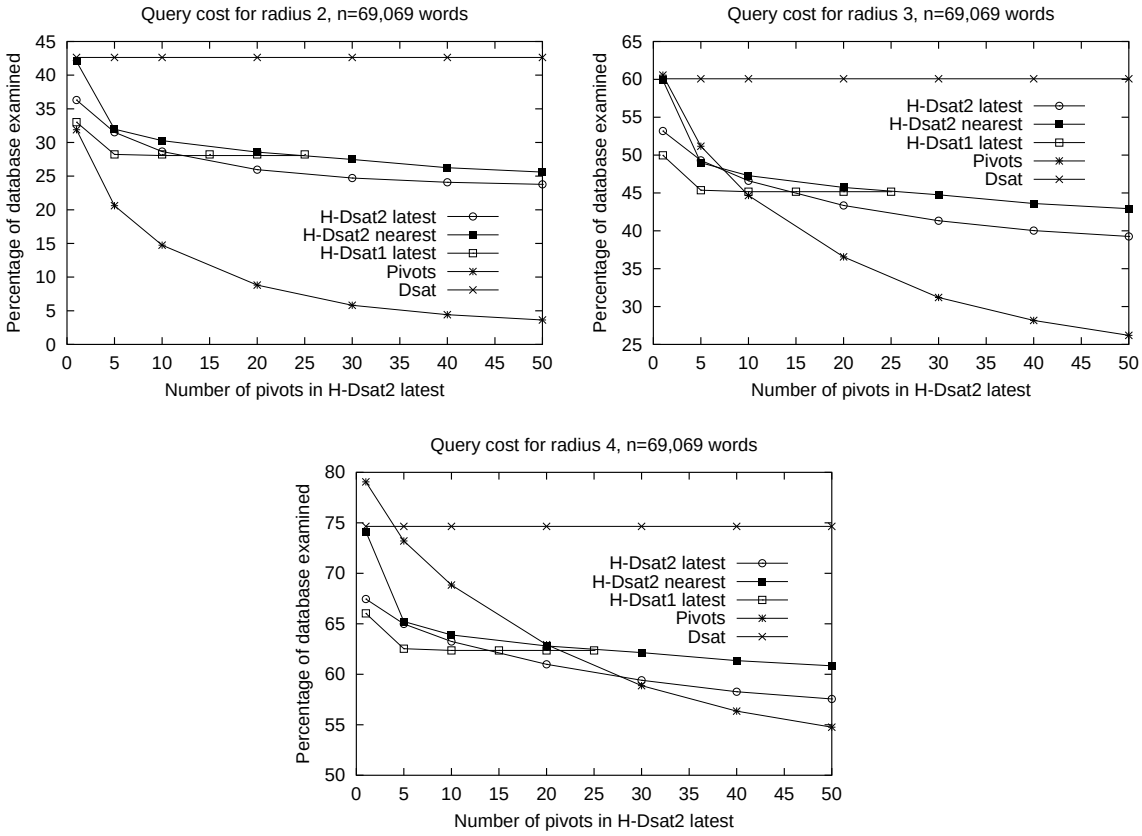

Figure 4: Query cost of H–DSAT1F and H–DSAT2F versus a pivoting algorithm, in the dictionary.

tions over arbitrarily long periods of time without any reorganization, and that can take advantage of available memory to improve search and deletion costs.

References

[1] D. Arroyuelo, F. Muñoz, G. Navarro, and N. Reyes. Memory–adptative dynamic spatial approximation trees. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS. Springer, 2003. To appear.

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

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

[4] G. Navarro and N. Reyes. Improved deletions in dynamic spatial approximation trees. In *Proc. of the XXIII International Conference of the Chilean Computer Science Society (SCCC'03)*. IEEE CS Press, 2003. To appear.