Memory-Adaptative Dynamic Spatial Approximation Trees*

Diego Arroyuelo[1], Francisca Muñoz[2], Gonzalo Navarro[2], and Nora Reyes[1]

[1] Depto. de Informática, Univ. Nac. de San Luis, Argentina.
[2] Center for Web Research, Dept. of Computer Science, Univ. of Chile.
{darroy,nreyes}@unsl.edu.ar, {franmuno,gnavarro}@dcc.uchile.cl

Abstract. Dynamic spatial approximation trees (*dsa–trees*) are efficient data structures for searching metric spaces. However, using enough storage, pivoting schemes beat *dsa–trees* in any metric space. In this paper we combine both concepts in a data structure that enjoys the features of *dsa–trees* and that improves query time by making the best use of the available memory. We show experimentally that our data structure is competitive for searching metric spaces.

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* [1]: 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, $\{x \in S, \quad d(x,q) \leqslant r\}$.

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 (Sec. 2).

In this paper we present a hybrid index for metric space searching built on the *dsa–tree*, an index supporting insertions and deletions that is competitive in spaces of medium difficulty, but unable of taking advantage of the available memory. This is enriched with a pivoting scheme. 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 is still dynamic and makes better use of memory, beating both *dsa-trees* and basic pivots.

Unlike previous work [3], (1) we use local rather than global pivots, and provide empirical evidence in favor of this decision, (2) we use pivots for free.

* Supported in part by CYTED VII.19 RIBIDI Project and, the third author, Millenium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

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 [1]. 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* [2], on top of which we build. Deletion algorithms are omitted for lack of space.

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.

```
RANGE SEARCH (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.              RANGE SEARCH(b_i, q, r, time(b_k))
8.          d_min ← min {d_min,  d(b_i,q)}
```

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

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 $d(q, b_i) \leqslant \min(d(q, b_1), \ldots, d(q, b_{i-1})) + 2r$.

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.

4 A Dsa–tree with Pivots

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. Our 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 $P(x) = \{(p_i,\ d(x,p_i)),\quad p_i \in \mathcal{A}(x)\}$. We store $P(x)$ at each node x and use it to prune the search.

Insertion Algorithm. 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*.

Range Search Algorithm. 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 Alg. 1. If at step 1 it holds $\mathcal{D}(a,q) > R(a)+r$, then $d(a,q) > R(a)+r$, and we can stop the search at node a without 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, it is feasible that a or some element in its subtree lie within the search radius of q.

We compute $\mathcal{D}(a,q)$ at search time without additional distance evaluations. Assume we reach node p_k and want to decide whether the search must enter subtree $x \in N(p_k)$. 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 $d(q,p_1),\ldots,d(q,p_k)$. In a H–DSAT1, we store $P(x) = \{(p_1,d(x,p_1)),\ldots,(p_k,d(x,p_k))\}$ at node x. Hence, all the elements needed to compute $\mathcal{D}(x,q)$ are present, at no extra cost.

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, so that references to the pivots in $P(x)$ (first component of pairs) are unnecessary and we save space.

The *feasible neighbors* of node a, 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 *infeasible*.

At search time, if we reach node a, we may consider only its feasible neighbors, as other subtrees can be wholly discarded. Although they are discarded using $\mathcal{D}$, which is computed for free, it does not immediately follow that we obtain for sure an improvement in search time. The reason is that infeasible nodes still serve to reduce d_{min} in Alg. 1, which in turn may save us entering younger siblings. Hence, by saving computations against infeasible nodes, we may have to enter new siblings later. This is an intrinsic tradeoff of our method.

Alg. 2 shows the basic search approach. Note that 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 also reduces the benefits of using pivots. The following alternatives are improvements to this weakness.

H–DSAT1D*: Optimizing using* $\mathcal{D}$. We use $\mathcal{D}$ not only to determine feasibility and hence prune subtrees, but also to decrease the number of infeasible elements directly compared against q in step 8. Some of those comparisons can be saved by using $\mathcal{D}$. The key observation is that $d(b_i,q) \leqslant \mathcal{D}(b_j,q)+2r$ implies $d(b_i,q) \leqslant$

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** $b_i \in F(a)$ **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** $d_{min} \leftarrow \min\{d_{min}, \ d(b_i,q)\}$

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

$d(b_j,q) + 2r$, so we can conclude that b_j is not of interest in step 8 without computing $d(b_j,q)$. Although we save some distance computations and obtain the same result, still there will be infeasible elements compared against q.

H–DSAT1F*: Using Timestamps of Feasible Neighbors.* Timestamps are not essential for the correctness of the algorithm. Although the optimal choice is to use the smallest correct timestamp, any larger value would do. So we compute a safe approximation to the correct timestamp, while ensuring that no infeasible elements are ever compared against q. Note that every feasible neighbor of a node will be compared against q inevitably. So, if for $b_i \in F(a)$ it holds $d(b_i,q) \leqslant d_{min} + 2r$, then in step 8 we compute the oldest timestamp t among the reduced set $\{b_{i+j} \in F(a), \ \ d(b_i,q) > d(b_{i+j},q) + 2r\}$. This uses as much timestamping information as possible without considering infeasible elements.

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.

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) $p_i, p_{i+1} \in \mathcal{A}(x) \wedge b \in N(p_i) \wedge time(b) < time(p_{i+1})$.

Range Search Algorithm. As before, 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

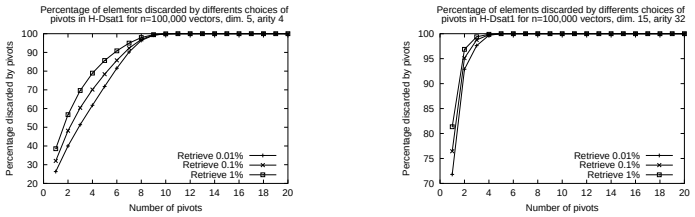

Fig. 1. Percentage of elements discarded using the latest pivots in H–DSAT1.

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 $\mathcal{D}$.

5 Limiting the Use of Storage

In practice, available memory is bounded. Our data structures use memory in a non-controlled way (each node uses as much pivots as the definition requires). This rules them out for many real-life situations. In order to adapt our structures to fit the available memory, we restrict the number of pivots stored in each node to a value k, holding a subset of the original set of pivots. As a result, the performance of our data structures may degrade at search time. A way of minimizing this effect is to choose a "good" set of pivots for each node.

We study empirically which pivots discard more elements at search time. See Sec. 6 for details on the experiments.

Good Pivots in H–Dsat1. Because of the insertion process, 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. Fig. 1 shows that a small number of latter pivots per node suffice. 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. This alternative will be called H–DSAT1 k Latest.

Good Pivots in H–Dsat2. 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 (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. Fig. 2 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.

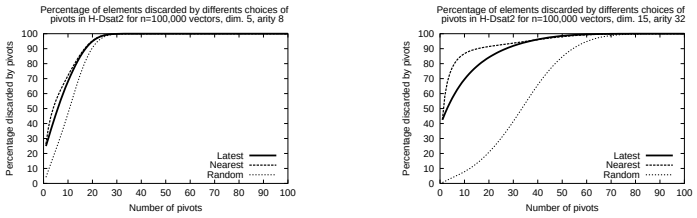

Fig. 2. Comparison between the quality of the pivots in a H–DSAT2, retrieving 0.01% of database. Unlike the others, random pivots degrade quickly as the dimension grows.

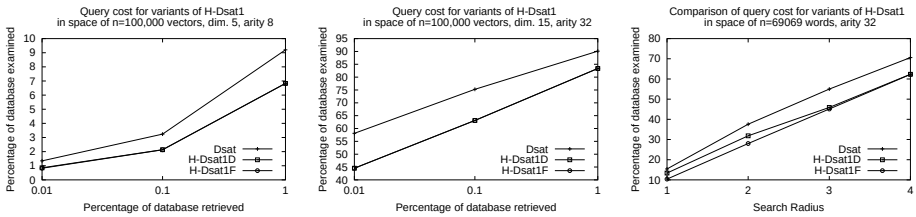

Fig. 3. Comparison of query cost for variants of H–DSAT1.

6 Experimental Results

We have evaluated our structures over three metric spaces. First, a dictionary of 69,069 English words under edit distance (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.

Fig. 3 shows that H–DSAT1F outperformed H–DSAT1D, clearly in the dictionary and slightly in vector spaces. The results are similar on H–DSAT2.

Fig. 4 shows 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. Fig. 5 compares 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

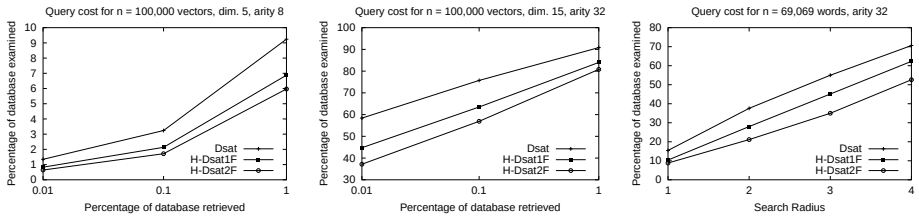

Fig. 4. Comparison of query cost among our structures.

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

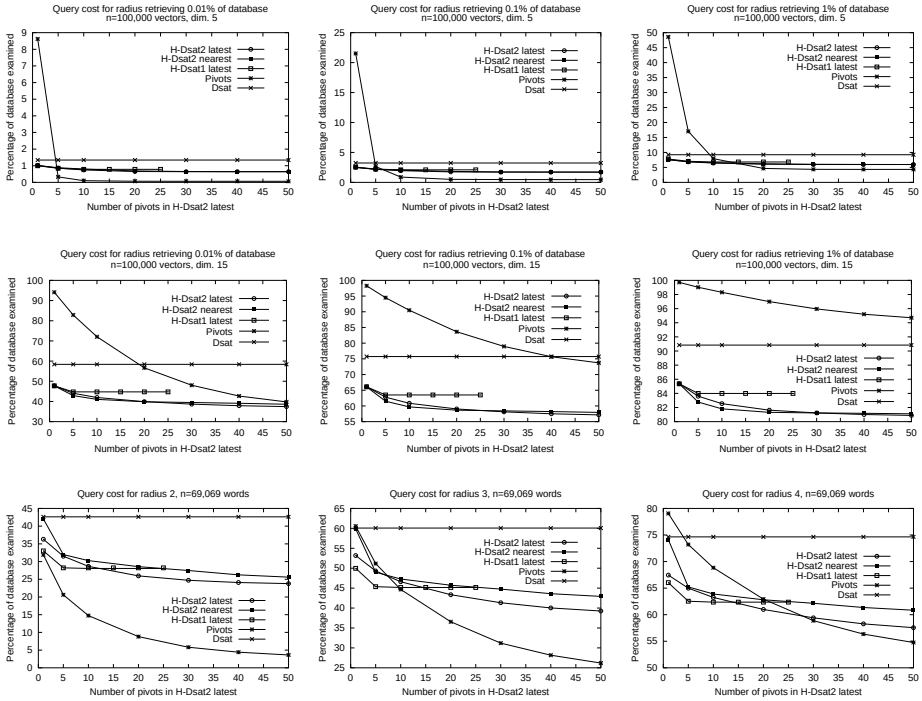

Fig. 5. Query cost of H–DSAT1F and H–DSAT2F versus a pivoting algorithm.

References

1. E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
2. G. Navarro and N. Reyes. Fully dynamic spatial approximation trees. In *Proc. SPIRE'02*, LNCS 2476, pp. 254–270, 2002.
3. C. Traina, A. Traina, R. Santos Filho and C. Faloutsos. How to improve the pruning ability of dynamic metric access methods. In *Proc. CIKM'02*, pp. 219–226, 2002.