New Deletion Method for Dynamic Spatial Approximation Trees

Fernando Kasián, Verónica Ludueña, Nora Reyes, and Patricia Roggero

Departamento de Informática, Universidad Nacional de San Luis,
San Luis, Argentina
{fkasian,vlud,nreyes,proggero}@unsl.edu.ar

Abstract. The Dynamic Spatial Approximation Tree (*DSAT*) is a data structure specially designed 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 *DSAT* supports insertion and deletions of elements. However, it has been noted that eliminations degrade the structure over time. In [8] is proposed a method to handle deletions over the *DSAT*, which shown to be superior to the former in the sense that it permits controlling the expected deletion cost as a proportion of the insertion cost.

In this paper we propose and study a new deletion method, based on the deletions strategies presented in [8], which has demonstrated to be better. The outcome is a fully dynamic data structure that can be managed through insertions and deletions over arbitrarily long periods of time without any reorganization.

Keywords: multimedia databases, metric spaces, similarity search

1 Introduction

“Proximity” or “similarity” searching is the problem of looking for objects in a set close enough to a query under a certain (expensive to compute) distance. Similarity search has become a very important operation in applications that deal with unstructured data sources. For example, multimedia databases manage objects without any kind of structure, such as images, fingerprints or audio clips. This has applications in a vast number of fields. Some examples are non–traditional databases, text searching, information retrieval, machine learning and classification, image quantization and compression, computational biology, and function prediction. All those applications can be formalized with the *metric space model* [3]. That is, there is an universe $\mathcal{U}$ of objects, and a positive real valued distance function $d : \mathcal{U} \times \mathcal{U} \longrightarrow \mathbb{R}^+$ defined among them. This distance may (and ideally does) satisfy the three axioms that make the set a metric space: *strict positiveness*, *symmetry*, and *triangle inequality*. The smaller the distance between two objects, the more “similar” they are. We have a finite database $S \subseteq \mathcal{U}$, which is a subset of the universe and can be preprocessed. 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.
Nearest neighbor query (k-NN): retrieve the k closest elements to q in S.

The distance is considered expensive to compute. Hence, it is customary to define the complexity of search as the number of distance evaluations performed. We consider the number of distance evaluations instead of the CPU time because the CPU overhead over the number of distance evaluations is negligible in the *DSAT*. In this paper we are devoted to range queries. In [5] is shown how to build an nearest neighbors algorithm range-optimal using a range algorithm, so we can restrict our attention to range queries.

Proximity search algorithms build an *index* of the database and perform queries using this index, avoiding the exhaustive search. For general metric spaces, there exist a number of methods to preprocess the database in order to reduce the number of distance evaluations [3]. All those structures work on the basis of discarding elements using the triangle inequality, and most use the classical divide-and-conquer approach. (which is not specific of metric space searching).

The Spatial Approximation Tree (*SAT*) is a proposed data structure of this kind [6, 7], based on a concept: approach the query spatially. It has been shown that the *SAT* gives better space-time tradeoffs than the other existing structures on metric spaces of high dimension or queries with low selectivity [7], which is the case in many applications. The *SAT*, however, has some important weaknesses: it is relatively costly to build in low dimensions; in low dimensions or for queries with high selectivity (small r or k), its search performance is poor when compared to simpler alternatives; and it is a static data structure: once built, it is hard to add/delete elements to/from it. These weaknesses make the *SAT* unsuitable for important applications such as multimedia databases.

The *DSAT* is a dynamic version of the *SAT* and overcomes its drawbacks. The dynamic *SAT* can be built incrementally (i.e., by successive insertions) at the same cost of its static version, and the search performance is unaffected. At first, the *DSAT* supports insertion and deletions of elements. However, that deletions degrade the structure over time, so in [8] was presented a deletion algorithm that does not degrade the search performance over time. This algorithm yielded better tradeoffs between search performance and deletion cost. In this paper we present another alternative method to delete an element of the *DSAT* which obtains better costs than methods described in [8].

Full dynamism is not so common in metric data structures [3]. While permitting efficient insertions is quite usual, deletions are rarely handled. In several indexes one can delete some elements, but there are selected elements that cannot be deleted at all. This is particularly problematic in the metric space scenario, where objects could be very large (e.g., images) and deleting them physically may be mandatory. Our algorithms permit deleting any element from a *DSAT*.

This paper is organized as follows: In Section 2 we give a description of the *DSAT*. Section 3 presents our new improved deletion method, and Section 4 contains the empirical evaluation of our proposal. Finally, in Section 5 we conclude and discuss about possible extensions for our work.

2 Dynamic Spatial Approximation Trees

In this section we briefly describe dynamic *SAT* (*DSAT* for short), in particular the version called *timestamp with bounded arity* presented in [8] as the better option to build incrementally the index, without any reconstruction after each insertion. To construct *DSAT* [8] incrementally a maximum tree arity is fixed, and also a timestamp of the

Algorithm 1 Insertion of a new element x into a *DSAT* with root a.

```
Insert(Node $a$, Element $x$)
 $R(a) \leftarrow \max(R(a), d(a,x))$
 $c \leftarrow \text{argmin}_{b \in N(a)} d(b,x)$
 If $d(a,x) < d(c,x) \wedge |N(a)| < MaxArity$ Then
     $N(a) \leftarrow N(a) \cup \{x\}$
     $N(x) \leftarrow \emptyset$, $R(x) \leftarrow 0$
     $time(x) \leftarrow CurrentTime$
     $CurrentTime \leftarrow CurrentTime + 1$
 Else Insert($c, x$)
```

insertion time of each element is kept. Each node a in the tree is connected to its children, which form a set of elements called $N(a)$, the *neighbors* of a. When inserting a new element x, its point of insertion is found by beginning from the tree root a and pe-rforming the following procedure. The element x is added to $N(a)$ (as a new leaf node) if (1) x is closer to a than to any element $b \in N(a)$, and (2) the arity of node a, $|N(a)|$, is not already maximal. Otherwise x is forced to choose the closest neighbor in $N(a)$ and keep walking down the tree in a recursive manner, until we reach a node a such that x is closer to a than any $b \in N(a)$ and the arity of node a is not maximal (this eventually occurs at a tree leaf). At this point x is added at the end of the list $N(a)$, the current timestamp is put to x and the current timestamp is incremented. The following information is kept in each node a of the tree: the set of neighbors $N(a)$, the timestamp $time(a)$ of the insertion time of the node, and the covering radius $R(a)$ with the distance between a and the farthest element in the subtree of a.

Note that by reading neighbors from left to right timestamps increase. It also holds that the parent is always older than its children. The *DSAT* can be built by starting with a first single node a where $N(a) = \emptyset$ and $R(a) = 0$, and then performing successive insertions. Algorithm 1 gives the insertion process.

2.1 Searching

The idea for range searching is to replicate the insertion process of relevant elements. That is, the process act as if it 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 a tolerance of $\pm r$ is permitted, so that it may be that relevant elements were inserted in different children of the current node, and backtracking is necessary.

Two facts have to be considered. The first is that, when an element x was inserted, a node a in its path may not have been chosen as its parent because its arity was already maximal. So, at query time, instead of choosing the closest to x among $\{a\} \cup N(a)$, it may have chosen only among $N(a)$. Hence, the minimization is performed only among elements in $N(a)$. The second fact is that, at the time x was inserted, elements with higher timestamp were not yet present in the tree, so x could choose its closest neighbor only among elements older than itself.

A better use of the timestamp information is made in order to reduce the work done inside older neighbors. Say that $d(q, b_i) > d(q, b_{i+j}) + 2r$. The process enters into the subtree of b_i anyway because b_i is older. However, only the elements with

Algorithm 2 Searching for q with radius r in a *DSAT* rooted at a.

```
RangeSearch(Node $a$, Query $q$, Radius $r$, Timestamp $t$)
If $time(a) < t \ \wedge \ d(a,q) \leq R(a) + r$ Then
    If $d(a,q) \leq r$ Then Report $a$
    $d_{min} \leftarrow \infty$
    For $b_i \in N(a)$ Do   /* in ascending timestamp order */
        If $d(b_i,q) \leq d_{min} + 2r$ Then
            $t' \leftarrow \min\{t\} \cup \{time(b_j), j > i \wedge d(b_i,q) > d(b_j,q) + 2r\}$
            RangeSearch($b_i, q, r, t'$)
            $d_{min} \leftarrow \min\{d_{min}, d(b_i,q)\}$
```

timestamp smaller than that of b_{i+j} should be considered when searching inside b_i; younger elements have seen b_{i+j} and they cannot be interesting for the search if they are inside b_i. As parent nodes are older than their descendants, as soon as a node inside the subtree of b_i with timestamp larger than that of b_{i+j} is found the search in that branch can stop, because all its subtree is even younger.

Algorithm 2 shows the process to perform range searching. Note that, except in the first invocation, $d(a,q)$ is already known from the invoking process.

2.2 Deletions

To delete an element x, the first step is to find it in the tree. Unlike most classical data structures, doing this is not equivalent to simulating the insertion of x and seeing where it leads us to in the tree. The reason is that the tree was different at the time x was inserted. If x were inserted again, it could choose to enter a different path in the tree, which did not exist at the time of its first insertion.

An elegant solution to this problem is to perform a range search with radius zero, that is, a query of the form $(x, 0)$. This is reasonably cheap and will lead us to all the places in the tree where x could have been inserted.

On the other hand, whether this search is necessary is application dependent. The application could return a handle when an object was inserted into the database, and therefore this search would not be necessary. This handle can contain a pointer to the corresponding tree node. Adding pointers to the parent in the tree would permit to locate the path for free (in terms of distance computations). Hence, in which follows, the location of the object is not considered as part of the deletion problem, although it has shown how to proceed if necessary.

Several alternatives to delete elements from *DSAT* were studied in [8]. From the beginning they discarded the trivial option of marking the element as deleted without actually deleting it. As explained, this is likely to be unacceptable in most applications. It is assumed that the element has to be physically deleted. It may, if desired, keep its node in the tree, but not the object itself. It should be clear that a tree leaf can always be deleted without any complication, so the focus is on how to remove internal tree nodes.

There are several proposed methods to delete an element from a *DSAT*, but in [8] the authors showed that the best option is based in *ghost hyperplanes*. This technique is inspired on an idea presented in [10] for dynamic *gna–trees* [2], called *ghost hyperplanes*. This method replaces the element being deleted by a leaf, which is easy to

Algorithm 3 Deleting x from a *DSAT*, finding a substitute in the leaves of its subtree.

```
DeleteGH1(Node $x$)
  $b \leftarrow parent(x)$
  If $N(x) \neq \emptyset$ Then
     $y \leftarrow$ FindSubstituteLeaf ($x$)
     $d_g(x) \leftarrow d_g(x) + d(x,y)$
     Copy object of $y$ into node $x$
  Else $N(b) \leftarrow N(b) - \{x\}$
```

```
FindSubstituteLeaf(Node $x$): Node
  $y \leftarrow x$
  While $N(y) \neq \emptyset$ Do
     $x \leftarrow y$
     $y \leftarrow \mathrm{argmin}_{c \in N(b)} d(c,x)$
  $N(x) \leftarrow N(x) - \{y\}$
  Return $y$
```

delete. This way rebuilding is not necessary, but in exchange some tolerance must be considered when entering the replaced node at search time.

Remind that the neighbors of a node b in the *DSAT* partition the space in a Voronoi-like fashion, with hyperplanes. If element y replaces a neighbor x of b, the hyperplanes will be shifted (slightly, if y is close to x). We can think of a “ghost” hyperplane, corresponding to the deleted element x, and a real one, corresponding to the new element y. The data in the tree is initially organized according to the ghost hyperplane, but incoming insertions will follow the real hyperplane. A search must be able to find all elements, inserted before or after the deletion of x.

For this sake, we have to maintain a tolerance $d_g(x)$ at each node x. This is set to $d_g(x) = 0$ when x is first inserted. When x is deleted and the content of its node is replaced by y, we will set $d_g(x) = d_g(x) + d(x,y)$ (the node is still called x although its object is that of y). Note that successive replacements may shift the hyperplanes in all directions so the new tolerance must be added to previous ones.

At search time, we have to consider that each node x can actually be offset by $d_g(x)$ when determining whether or not we must enter a subtree. Therefore, we wish to keep $d_g()$ values as small as possible, that is, we want to find replacements that are as close as possible to the deleted object. When node x is deleted, we have to look for a substitute in its subtree to ensure that we reduce the problem size.

Choosing a leaf substitute We descend in the subtree of x by the children closest to x all the time. When it reach a leaf y, it disconnect y from the tree and put y into the node of x, retaining the original timestamp of x. Then, the d_g value of the node is updated.

Choosing a neighbor substitute We select y as the closest to x among $N(x)$ and copy object y into the node of x as above. If the former node of y was a leaf it delete it and finish. Otherwise we recursively continue the process at that node. So, we turn to *ghost* all the nodes in the path from x to a leaf of its subtree, following the closest neighbors. In exchange, the $d_g()$ values should be smaller.

Choosing the nearest-element substitute We select y as the closest element to x among all the elements in the subtree of x and copy object y into the node of x as above. If the former node of y was a leaf we delete it and finish. Otherwise, we recursively continue the process at that node. Therefore, we turn to *ghost* some nodes in the path from x to a leaf of its subtree, following the nearest elements. The $d_g()$ values should be smaller than with the other alternatives.

Algoritms 3, 4, and 5 detail these three deletion methods.

Algorithm 4 Deleting x from a *DSAT*, choosing its replacement among its neighbors.

```
DeleteGH2(Node $x$)
  $b \leftarrow parent(x)$
  If $N(x) \neq \emptyset$ Then
    $y \leftarrow \mathrm{argmin}_{c \in N(x)} d(c,x)$,  $d_g(x) \leftarrow d_g(x) + d(x,y)$
    Copy object of $y$ into node $x$
    DeleteGH2 ($y$)
  Else $N(b) \leftarrow N(b) - \{x\}$
```

Algorithm 5 Deleting x from a *DSAT*, choosing its replacement as its nearest element.

```
DeleteGH3(Node $x$)
  $b \leftarrow parent(x)$
  If $N(x) \neq \emptyset$ Then
    $y \leftarrow$ NNsearch($x$,$x$,1),  $d_g(x) \leftarrow d_g(x) + d(x,y)$
    Copy object of $y$ into node $x$
    DeleteGH3 ($y$)
  Else $N(b) \leftarrow N(b) - \{x\}$
```

Thus, for a permanent regime that includes deletions, we must periodically get rid of ghost hyperplanes and reconstruct the tree to delete them. Just as with fake nodes [8], when we rebuild the subtree we get rid of all the ghost hyperplanes that are inside it. We set a maximum allowable proportion α of ghost hyperplanes, and rebuild the tree when this limit is exceeded.

3 A New Deletion Method

In [8] it is concluded that the methods with the best performance during deletions use ghosts hyperplanes. Moreover, these methods have the possibility of using the parameter α to control the deletion average cost. Our new proposal to delete an element x is based on the best strategies presented in [8]. Therefore, this new proposed method is also based on the idea presented in [10], which use *ghost hyperplanes*.

We believe that the way to achieve a good tradeoff between the number of hyperplanes and the displacement of each d_f can be obtained by replacing the deleted element x with the leaf of his subtree whose distance is minimal; i. e. *the closest leaf in the complete subtree of* x. Therefore, with each deletion only one new ghost hyperplane appears and the displacement of this ghost hyperplane, although it is not necessarily the smallest one possible, is expected to be fairly close to it.

It is possible to notice, considering the presented algorithms in Section 2.2, that it is only needed to change the process invoked as **FindSubstituteLeaf**(x). This new algorithm has to choose the closest element to x between all the leaves in the subtree of x. Therefore, **DeleteGH1**(Node x) is similar to **DeleteGH4**(Node x) because only one ghost hyperplane appears after deletion. The Algorithm 6 shows this situation completely. In the function **FindSubstituteNNLeaf** all the leaves of the subtree of x are recovered in the set L of pairs (z,t), where z is a leaf of the subtree of x and t is his father. Q is a queue of elements to be used as an auxiliary data structure in a *traversal*

Algorithm 6 Deleting x from a *DSAT*, finding a substitute as the *closest leaf*.

```
DeleteGH4(Node $x$)
  $b \leftarrow parent(x)$
  If $N(x) \neq \emptyset$ Then
      $y \leftarrow$ FindSubstituteNNLeaf($x$)
      $d_f(x) \leftarrow d_f(x) + d(x, y)$
      Copy object of $y$ into node $x$
  Else $N(b) \leftarrow N(b) - \{x\}$

FindSubstituteNNLeaf(Node $x$): Node
  $Q \leftarrow \emptyset$, $L \leftarrow \emptyset$
  For $v \in N(x)$
      If $N(v) = 0$ Then $L \leftarrow \{(v, x)\}$
      Else $Q \leftarrow \{v\}$
  While $Q$ not empty
      $b \leftarrow$ first element of $Q$, $Q \leftarrow Q - \{b\}$
      For $v \in N(b)$
          If $N(v) = 0$ Then $L \leftarrow L \cup \{(v, b)\}$
          Else $Q \leftarrow Q \cup \{v\}$
  $(y, v) \leftarrow \mathrm{argmin}_{(z,t) \in L} d(x, z)$, $N(v) \leftarrow N(v) - \{y\}$
Return $y$
```

in level-order. Finally, when the full set of leaves of the subtree of x is determined, we select the leaf y that satisfies: $d(x, y) < d(x, z), \forall (z, t) \in L - \{(y, v)\}$, then y is returned after it is disconnected from its father.

This new method is based on the idea to obtain a better deletion strategy by considering the best characteristics of GH1 and GH3: only one ghost hyperplane appears after each deletion, and its displacement is nearby to the possible best one. Clearly, we can also set a maximum allowable proportion α of ghost hyperplanes, and rebuild the tree when this limit is exceeded.

4 Experimental Results

As it is aforementioned, we do not consider the cost to locate the element as part of the deletion problem, then the deletion costs obtained represent only the necessary work to effectively delete the element from the *DSAT*. Thus, we can directly compare our experimental results with those presented in [8]. To study the behavior and performance of this new deletion algorithm for *DSAT*, we need to evaluate the proper deletion costs and also the searching performance after that.

In order to make a fairly comparison between the previous deletion methods and the new one, we use the same metric spaces considered in [8] to evaluate the performance of *DSAT*, available from [4]. We use the best arity for each space, as it is described in [8]. They are four real-life metric spaces with widely different histograms of distances:

Strings: a dictionary of 69,069 English words. The distance is the *edit distance*, that is, the minimum number of character insertions, deletions and substitutions needed to make two strings equal.

NASA images: a set of 40,700 20-dimensional feature vectors, generated from images downloaded from NASA. [1] The Euclidean distance is used.

Color histograms: a set of 112,682 8-D color histograms (112-dimensional vectors) from an image database. [2] Any quadratic form can be used as a distance, so we chose Euclidean distance.

Documents: a set of 1,265 documents under the Cosine similarity, heavily used in Information Retrieval [1]. In this model the space has one coordinate per term and documents are seen as vectors in this high dimensional space. The distance we use is the angle among the vectors. The documents are the files of the TREC-3 collection.[3]

There are two types of experiments:

1. We build the index with the 90% of the database elements, the other 10% is used as queries for range searches. After the index is built, we delete a 10% of elements randomly selected.
2. We use the 60%, 70%, 80%, and 90% of the database elements to build the index. Then we delete the 10%, 20%, 30%, and 40% respectively, in order to leave 50% of the elements into the tree in each index. It can be noticed that in each case the 50% of the database, that remains after deletions into the tree, is not necessarily the same set of elements, as the elements deleted are randomly selected. Then, we perform queries with the non-inserted 10% of database elements.

We have tested several options in our experiments. For the parameter α we consider: 0% (without any ghost hyperplane), 1%, 3%, 10%, 30%, and 100% (without any rebuilding). In all cases, if $\alpha = 0\%$, as is pure rebuilding, costs are higher. Then, as the proportion α of allowed ghost hyperplanes grows, deletion costs decrease. For range search we consider three radii for the spaces with continuous distance, and four radii for Strings space (with discrete distance). For lack of space we only show some examples: for the first type of experiment, the comparison of deletion costs for $\alpha = 1\%$ when the 10% of elements is deleted; for the second one, the comparison of search costs obtained after 40% of elements is deleted with $\alpha = 1\%$, considering the 10% of reserved elements as queries. Figure 1 shows, for the first type of experiment, the average deletion costs obtained per element when 10% of the elements is deleted. Figure 2 illustrates, for the second type of experiment, the average search costs per element, after 40% of the database is deleted using $\alpha = 1\%$, when we search with the reserved 10% of elements as queries. As it can be noticed, our deletion method (GH4) obtains very good performance, both in deletion and search costs, for all metric spaces considered.

5 Conclusions

We have designed a new algorithm for efficient deletion in *DSAT*. This new algorithm has a low cost of deletion and allows that subsequent searches have a performance similar to the best algorithm proposed in [8]. On the other hand, efficient searches are still maintained: it is possible to apply the same algorithms of *DSAT* for range search and *k* closest neighbors. Our deletion algorithm kept, as a parameter, the proportion of

[1] At `http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html`

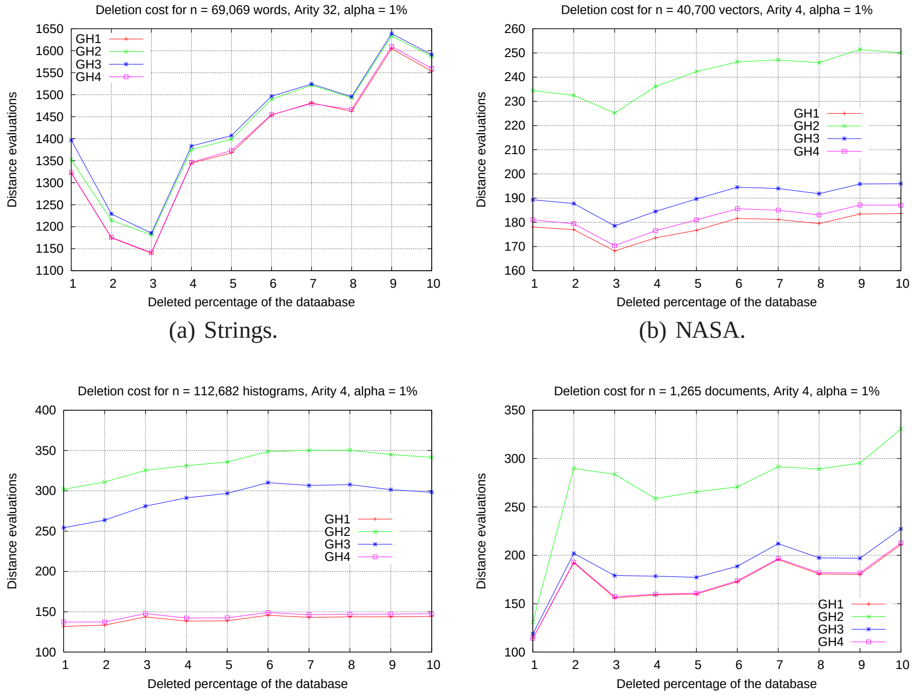
[2] At `http://www.dbs.informatik.uni-muenchen.de/~seidl/DATA/histo112.112682.gz`

[3] At `http://trec.nist.gov`

allowed nodes with ghost hyperplanes in the tree, which permits us to tune search cost versus deletion cost.

The outcome is a much more practical data structure that can be useful in a wide range of applications. We expect the *DSAT*, with the new deletion algorithm, to replace the static version in the developments to come.

As future work we plan to add our new deletion algorithm to the existing version of *DSAT* for secondary memory [9], since it has the advantage that only one ghost hyperplane is created, so only two nodes have to be changed, for each deletion. In this case will be relevant both number of distance evaluations and number of I/O operations.



(a) Strings.

(b) NASA.

(c) Color Histograms.

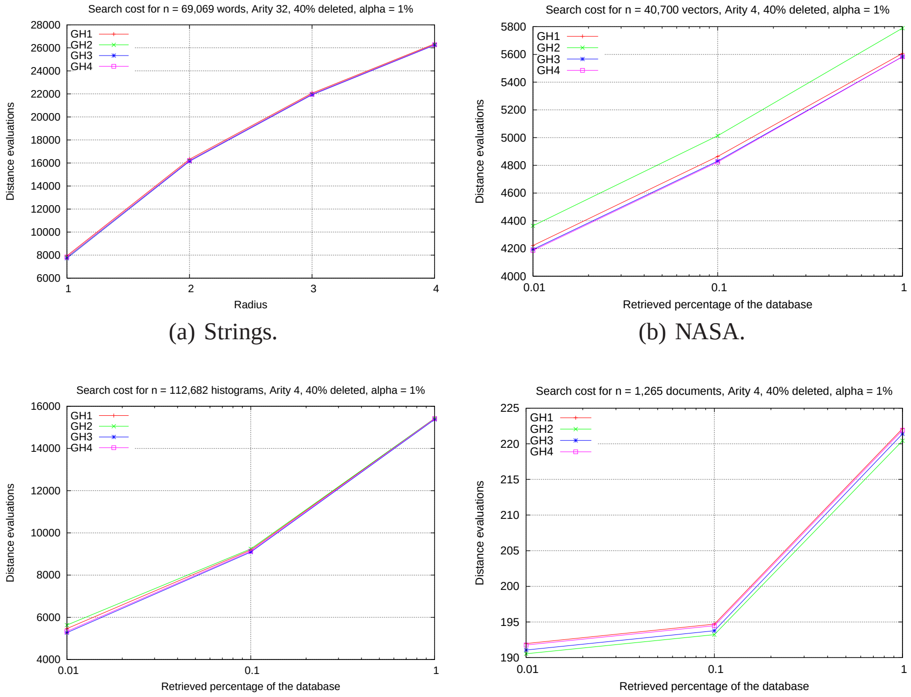
(d) Documents.

Fig. 1. Comparison of deletion costs, for all deletion algorithms using $\alpha = 1\%$.

References

1. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.
2. S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.
3. E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

4. K. Figueroa, G. Navarro, and E. Chávez. Metric spaces library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.
5. G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report CS-TR-4199, University of Maryland, Computer Science Department, 2000.
6. G. Navarro. Searching in metric spaces by spatial approximation. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, pages 141–148. IEEE CS Press, 1999.
7. G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.
8. G. Navarro and N. Reyes. Dynamic spatial approximation trees. *ACM Journal of Experimental Algorithmics (JEA)*, 12:article 1.5, 2008. 68 pages.
9. G. Navarro and N. Reyes. Dynamic spatial approximation trees for massive data. In T. Skopal and P. Zezula, editors, *SISAP*, pages 81–88. IEEE Computer Society, 2009.
10. R. Uribe and G. Navarro. Una estructura dinámica para búsqueda en espacios métricos. In *Actas del XI Encuentro Chileno de Computación, Jornadas Chilenas de Computación*, Chillán, Chile, 2003. In Spanish. In CD-ROM.



(a) Strings.

(b) NASA.

(c) Color Histograms.

(d) Documents.

Fig. 2. Comparison of search costs, after 40% of elements are deleted using $\alpha = 1\%$.