Dynamic Spatial Approximation Trees [*†]

Gonzalo Navarro
Center for Web Research
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

Metric space searching is an emerging technique to address the problem of efficient similarity searching in many applications, including multimedia databases and other repositories handling complex objects. Although promising, the metric space approach is still immature in several aspects that are well established in traditional databases. In particular, most indexing schemes are static, that is, few of them tolerate insertion or deletion of elements at reasonable cost over an existing index.

The Spatial Approximation Tree (*sa–tree*) [VLDBJ 2002] has been experimentally shown to provide a good tradeoff between construction cost, search cost, and space requirement. However, the *sa–tree* is static, which renders it unsuitable for many database applications.

In this paper we study different methods to handle insertions and deletions on the *sa–tree* at low cost. In many cases, the dynamic construction (by successive insertions) is even faster than the previous static construction, and both are similar elsewhere. In addition, the dynamic version significantly improves the search performance of *sa–trees* in virtually all cases. The result is a much more practical data structure that can be useful in a wide range of database applications.

1 Introduction

Similarity searching has applications in a vast number of fields [Sam05, ZADB06]. Some examples are non-traditional databases (for example, storing images, fingerprints or audio clips, where the concept of exact search is of no use and we search instead for similar objects) [ABH97, YI99]; text searching (to find words and phrases in a text database allowing a small number of typographical or spelling errors) [SK83, Kuk92]; information retrieval (to look for documents that are similar to a given query or document) [SM83, BYRN99]; machine learning and classification (to classify a new element according to its closest representative) [DH73]; image quantization and compression (where only some vectors can be represented and we code the others as their closest representable point, as in the MPEG standard); computational biology (to find homologous regions in a DNA or

[*]This work has been partially supported by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile (first author).

[†]Preliminary partial versions of this work appeared in [NR01, NR02a, NR02b, NR03].

protein sequence database) [Gus97, Wat95, SK83]; 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 share some common characteristics. There is a finite *dataset* of objects belonging to a *metric space*, where a *distance function* is used to assess similarity. *Similarity queries* are posed to this dataset. These consist basically in, given a new element of the space called the *query*, looking for elements of the dataset that are similar enough to the query. The dataset is preprocessed so as to build an *index* that reduces query time. This metric space approach to handle similarity search problems is becoming widely popular [Sam05, ZADB06] and a large number of indexing methods have flourished [CNBYM01, HS03b, Sam05], although mature solutions from the database point of view are a long way off.

Most of the existing indexes are *static*. This means that, once they are built for a given dataset, adding more elements to the dataset, or removing an element from it, requires an expensive updating of the index (in many cases equivalent to rebuilding it from scratch). Some indexes tolerate insertions in principle, but their quality degrades and require periodic rebuildings. Others tolerate deletions with the same quality problem. In several of them, some elements of the dataset can be deleted but others cannot, which is usually unacceptable as we are potentially dealing with very large objects (images, for example). Thus there are very few *dynamic* indexes.

We note that there are some applications where a static scheme may be acceptable, yet many of them require dynamic capabilities. Actually, in some cases it is sufficient to support insertions, as in digital libraries and archives where documents are never updated or deleted. Yet, there are many cases where deletions must be supported, for example Web engines that must avoid links to pages that do not exist anymore, and in general any multimedia or document database where up-to-date information must be kept (individuals in an organization, current projects in a manufacturer company, and a long so on).

From those few dynamic indexes, even fewer work well in secondary memory. That is, most of them need the data structure in main memory in order to operate properly. Although this issue is important, we are not focusing on it in this paper. There are many interesting databases for similarity searching where either (*i*) the similarity computation is so expensive (e.g., taking several CPU seconds) that one can disregard other costs, (*ii*) the objects are so large that they must stay on disk, but there are not that many of them, so the index itself fits perfectly well in main memory. For example, databases with terabytes of images usually mean that there are a few million images of a few megabytes each, so the index needs just a few megabytes of main memory.

In this paper we focus on obtaining a dynamic index that performs well in main memory. We base our work on the Spatial Approximation Tree (*sa–tree*) [Nav99, Nav02, HS03a]. It has been shown that the *sa–tree* gives an attractive tradeoff between memory usage, construction time, and search performance.

The *sa–tree*, however, has some important weaknesses. The first is that, in some spaces, it is relatively costly to build and not very efficient to search compared to other simple indexes. The second is that it is a markedly static data structure: Modifying it is extremely difficult. These weaknesses make the *sa–tree* unsuitable for important applications such as multimedia databases.

We study several insertion and deletion techniques to make the *sa–tree* dynamic. We show that the resulting dynamic *sa–tree* (*dsa–tree* for short) can be built incrementally (that is, by successive insertions) at least at the same cost of its static version in all cases, and much faster on some spaces. In addition, the search performance largely improves in virtually all cases. We also show that one can remove any element from the structure at about the same cost of an insertion, with a very small penalty in the search performance.

Our contributions are two-fold. From an algorithmic point of view, we give new insights on the *sa–tree*, which is by itself an intriguing data structure. We show that its invariants can be relaxed in non-obvious ways while preserving its search capabilities, and this can be used to make the *sa–tree* fully dynamic at low cost. As a consequence, the dynamic version turns out to be even more efficient than the static version in most cases. From a more practical point of view, our contribution is a relevant data structure for metric space searching, which performs well in construction and search time, and which is fully dynamic. This result makes the *dsa–tree* useful in a much wider range of applications, as it not only supersedes the static *sa–tree* in functionality and efficiency, but we also show that it compares favorably against state-of-the-art alternatives.

2 Basic Concepts

Let $\mathbb{U}$ be a universe of *objects*, with a nonnegative *distance function* $d : \mathbb{U} \times \mathbb{U} \longrightarrow \mathbb{R}^+$ defined among them. This distance satisfies the three axioms that make $(\mathbb{U}, d)$ 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 handle a finite *dataset* $S \subseteq \mathbb{U}$, which is a subset of the universe of objects and can be preprocessed (to build an index). Later, given a new object from the universe (a *query* $q \in \mathbb{U}$), we must retrieve all similar elements found in the dataset. There are two typical queries of this kind:

Range query: Retrieve all elements within distance r to q in S. That 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 assumed to be expensive to compute (as in most of the examples we gave in the Introduction). 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 dataset of $|S| = n$ objects, queries can be trivially answered by performing n distance evaluations. The goal is to preprocess the dataset such that queries can be answered with as few distance evaluations as possible.

A particular case of this problem arises when the space is a set of D-dimensional points and the distance belongs to the Minkowski L_p family: $L_p = (\sum_{1 \leq i \leq D} |x_i - y_i|^p)^{1/p}$. For example $p = 2$ yields Euclidean distance. There are effective methods to search in those spaces [GG98, BBK01]. However, for roughly 20 dimensions or more those structures cease to work well. We focus in this paper on general metric spaces, although the solutions are well suited also for D-dimensional spaces. Moreover, regarding a D-dimensional space as a metric space reveals the true dimensionality of the dataset, which may be much lower than D, without the need of applying expensive dimensionality reduction techniques.

Measuring the difficulty of searching in a metric space is a difficult task. The search performance depends in a non-obvious way on the shape of the histogram of distances, and even on the histogram of subsets of the space. Although the concept of "dimensionality" has been extended to metric spaces (e.g. [Bri95, CNBYM01]), the existing estimates are still not adequate to replace thorough experimentation. In this paper we use four real-life metric spaces with widely different histograms of distances, so as to derive sufficiently general conclusions on performance. Still, we ignore extremely difficult metric spaces (where the histograms are very concentrated) as those are intractable with exact algorithms [CNBYM01, CN01, BBK01, BN02].

Experimental setup. Experiments are spread throughout this paper, thus we describe here the experimental setup. We have selected four widely different metric spaces.

NASA images: a set of 40,700 20-dimensional feature vectors, generated from images downloaded from NASA (`http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html`). The Euclidean distance is used.

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. This distance is useful in text retrieval to cope with spelling, typing and optical character recognition (OCR) errors.

Color histograms: a set of 112,682 8-D color histograms (112-dimensional vectors) from an image database (`http://www.dbs.informatik.uni-muenchen.de/~seidl/DATA/histo112.112682.gz`). Any quadratic form can be used as a distance, so we chose Euclidean distance as the simplest meaningful alternative.

Documents: a set of 1,265 documents under the Cosine similarity, heavily used in Information Retrieval [BYRN99]. 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 (arccos of inner product) among the vectors. The documents are the files of the TREC-3 collection (`http://trec.nist.gov`).

Figure 1 shows the distance histograms for each metric space considered. These make up an interesting sample of real-life metric spaces, with widely different histograms (Gaussian-like, discrete, heavy-tail, arbitrary).

In all cases, we built the indexes with 90% of the points and used the other 10% (randomly chosen) as queries. All our results are averaged over 10 index constructions using different permutations of the datasets.

We have considered range queries retrieving on average 0.01%, 0.1% and 1% of the dataset. This corresponds to radii 0.605740, 0.780000 and 1.009000 for the NASA images, 0.051768, 0.082514 and 0.131163 for the color histograms, and 0.140000, 0.150000 and 0.195000 for the documents. Strings have a discrete distance, so we used radii 1 to 4, which retrieved on average 0.00003%, 0.00037%, 0.00326% and 0.01757% of the dataset, respectively. The same queries were used for all the experiments on the same datasets. Given the existence of range-optimal algorithms for k-nearest neighbor searching (see Section 3.5), we have not considered these search experiments separately, as their search cost is exactly that of range searching with a radius that captures the k neighbors.

For the experiments of searching with deletions in an index of n elements, we select at random a given fraction of the elements and delete them from the index, so that *after* the deletions the index contains n elements. For example, if we search half the space of strings after 30% of deletions, it means that we inserted 49,335 elements and then removed 14,800, so as to leave 34,534 elements (half of the set).

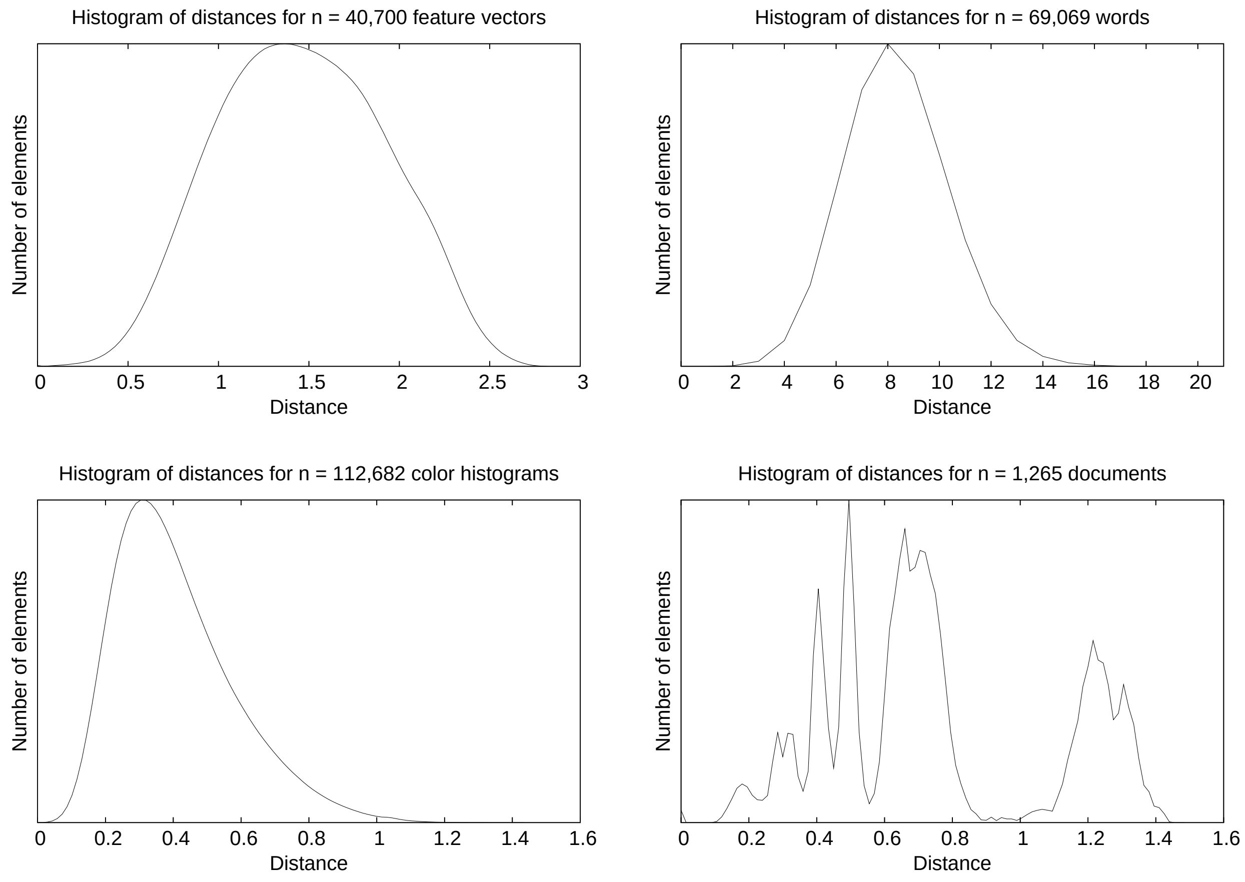


Figure 1: Distance histograms for each metric space considered in the experiments.

3 Previous Work

Algorithms to search in general metric spaces can be divided into two large areas: pivot-based and clustering algorithms. However, there are algorithms that combine ideas from both areas. (See [Sam05, ZADB06, CNBYM01, HS03b] for more complete surveys.)

3.1 Pivot-Based Algorithms

The idea is to use a set of k distinguished elements ("pivots") $p_1 \dots p_k \in S$ and storing, for each dataset element x, its distance to the k pivots $(d(x, p_1) \dots d(x, p_k))$. Given the query q, its distance to the k pivots is computed $(d(q, p_1) \dots d(q, p_k))$. 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 discarded using this rule are directly compared against the query.

Several algorithms [Vid86, MOV94, CMBY99, NN97, BYCMW94, CMN01] 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.

These indexes permit easy insertion/deletion of elements, by simply adding or removing rows to/from the table of kn distances. An element can be added with k distance computations and removed without any distance evaluation. Removing a pivot, however, is rather problematic, as it must be replaced by another pivot (at the cost of n distance computations) to avoid degrading the quality of the index. This may also require a large amount of extra CPU work. Finally, the optimal number of pivots k is related to the number of elements n, so after many insertions/deletions one should add/remove pivots anyway.

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 child corresponding to $d(a,b) \in [x_1, x_2]$, then we can avoid entering 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 [BK73, Uhl91b, MOC96, Yia93, BO97, Yia00].

In some of these trees, elements can be easily added at the leaves, although in others some global information (such as percentiles) is used to shape the tree from the root and therefore insertions either are too expensive or progressively degrade the quality of the structure. Deletions are always problematic because they require rebuilding all the subtree of the removed element.

3.2 Clustering Algorithms

The second trend consists of dividing the space into zones as compact as possible, usually in a recursive fashion, and storing a representative point ("center") for each zone plus a few extra data that permit quickly discarding the zone at query time. Two criteria can be used to delimit a zone.

The first one is the *Voronoi region*, where we select a set of centers and put every other point inside the zone of its closest center. The regions are bounded 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 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 use only hyperplanes [Uhl91b, NVZ92, DN87, Ver95], some use only covering radii [CPZ97, CN05], and some use both criteria [Bri95, Nav02].

In most of these structures, elements can be gracefully inserted at the leaves of the tree. There are some exceptions, such as [CN05, Nav02], where this progressively degrades the quality of the structure or is directly unaffordable.

Others that have been specifically designed to maintain their quality through insertions are [CPZ97, Ver95]. Deleting elements, on the other hand, is too expensive in all these structures (where a full subtree reconstruction is necessary), especially in Voronoi-based trees. A structure where a deletion algorithm seems feasible is the *M–tree* [CPZ97], but no such an algorithm has been yet proposed.

3.3 Combining Clustering with Pivots

There are some data structures that combine both ideas by dividing the space into compact zones, and at the same time storing distances to some distinguished elements (pivots).

The *D–index* [DGSZ03, Doh04] divides the space into *separable* partitions of data blocks and combines this with pivot-based strategies to decrease I/O costs and distance evaluations performed during searches. It supports disk storage, as well as insertions and deletions of elements but, as before, removing a pivot is problematic. Besides, adapting the *D–index* to particular applications requires a non-trivial parameterization process.

Another example is presented in [CFP$^+$05]. Although the authors claim that this structure supports insertions and deletions, it is not clear how to carry them out efficiently. Another complication is that the structure is not easy to parameterize, not to mention maintaining a good parameterization under a dynamic setting.

Another example in this group is obtained by adding pivots to some clustering-based data structure, as the *PM–tree* [SPS04] does on top of the *M–tree* [CPZ97].

3.4 Inexact Algorithms

In many applications it is acceptable to carry out an inexact similarity search, where accuracy or determinism is traded for improved performance [ZADB06, Sam05, CNBYM01]. This alternative to exact similarity searching is called approximate similarity searching, and encompasses approximate [KL04] and probabilistic algorithms [Cla99, KR02]. The general idea of approximation algorithms is to allow a relaxation on the query precision in order to obtain a speedup in the query time complexity. Additionally to the query one specifies a precision parameter ϵ to control how far away (in some sense) we want the outcome of the query from the correct result. A reasonable behavior for this type of algorithm is to asymptotically approach the correct answer as ϵ goes to zero, and complementarily to speed up the algorithm, losing precision, as ϵ moves in the opposite direction.

3.5 Nearest Neighbor Queries

Although we have considered only range searching up to now, all the indexes are capable of nearest neighbor searching. The technique was adapted from vector space data structures to metric trees in [Uhl91a], and later it was extended to work on most data structures and proved to be range-optimal [HS00]. Range optimality means that, if a k-NN query (q, k) retrieves elements $\{u_1 \dots u_k\}$, then the cost of the search is exactly that of a range query (q, r), with $r = max(d(q, u_1) \dots d(q, u_k))$. That is, in a range-optimal k-NN algorithm, there is no cost for not knowing in advance which is the distance to the k-th nearest neighbor. This idea was also adapted to the *sa–tree* in [Nav02].

The technique is described for any tree data structure that divides the space into zones and that can prove lower bounds on distances from each zone to q. This encompasses most existing data structures, and it is not hard to extend the technique to others, such as pivot-based tables [BN02]. The idea is to maintain a priority queue Q of subtrees, ordered by provable lower bound distance to q, as well as a priority queue A of at most k elements closest to q found so far. Initially, Q contains the whole tree and A is empty. At each step, the first subtree T (that is, the one with smallest lower bound distance to q) is extracted from Q, its root elements are inserted into A (which discards the farthest elements so as to maintain at most k of them), and the subtrees of T are inserted into Q. As soon as we have k elements in A and the farthest element in A is closer to q than the lower bound given by the first element in Q, we can safely stop because it is not possible that a tree contained in Q has an element closer to q than those k elements already in A.

4 The Spatial Approximation Tree

In this section we present the static data structure we build on, the *sa–tree* [Nav99, Nav02]. Unlike most other structures, based on dividing the search space, the *sa–tree* is based on the idea of approaching the query spatially, that is, starting at some point in the space and getting closer and closer to the query. The *sa–tree* is experimentally shown to offer better space-time tradeoffs than other data structures in several spaces.

4.1 Construction

We select a random element $a \in S$ to be the root of the tree. We then select a suitable set of neighbors $N(a)$ satisfying

Condition 1: (given a, S) $\quad \forall x \in S,\ x \in N(a) \quad \Leftrightarrow \quad \forall y \in N(a) - \{x\},\ d(x,y) > d(x,a).$

That is, the neighbors of a form a set such that any neighbor is closer to a than to any other neighbor. The "if" ($\Leftarrow$) part of the definition guarantees that, if from a 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 "only if" part ($\Rightarrow$) aims at putting as few neighbors as possible. Choosing nearest neighbors owes to the concept of getting spatially closer to the query, so that if we cannot get closer (with tolerance r) from a tree node then we can stop the search there.

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). A simple heuristic that adds more neighbors than necessary is used, and it works 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 into $N(a)$. 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 into $N(a)$ 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.

Figure 2 illustrates the construction process, showing **(a)** a set of points in a two-dimensional Euclidean space; **(b)** the first step of the construction process starting with a as the tree root (and thus $N(a) = \{b, c, d, e\}$), including the corresponding hyperplanes; and **(c)** the final tree obtained.

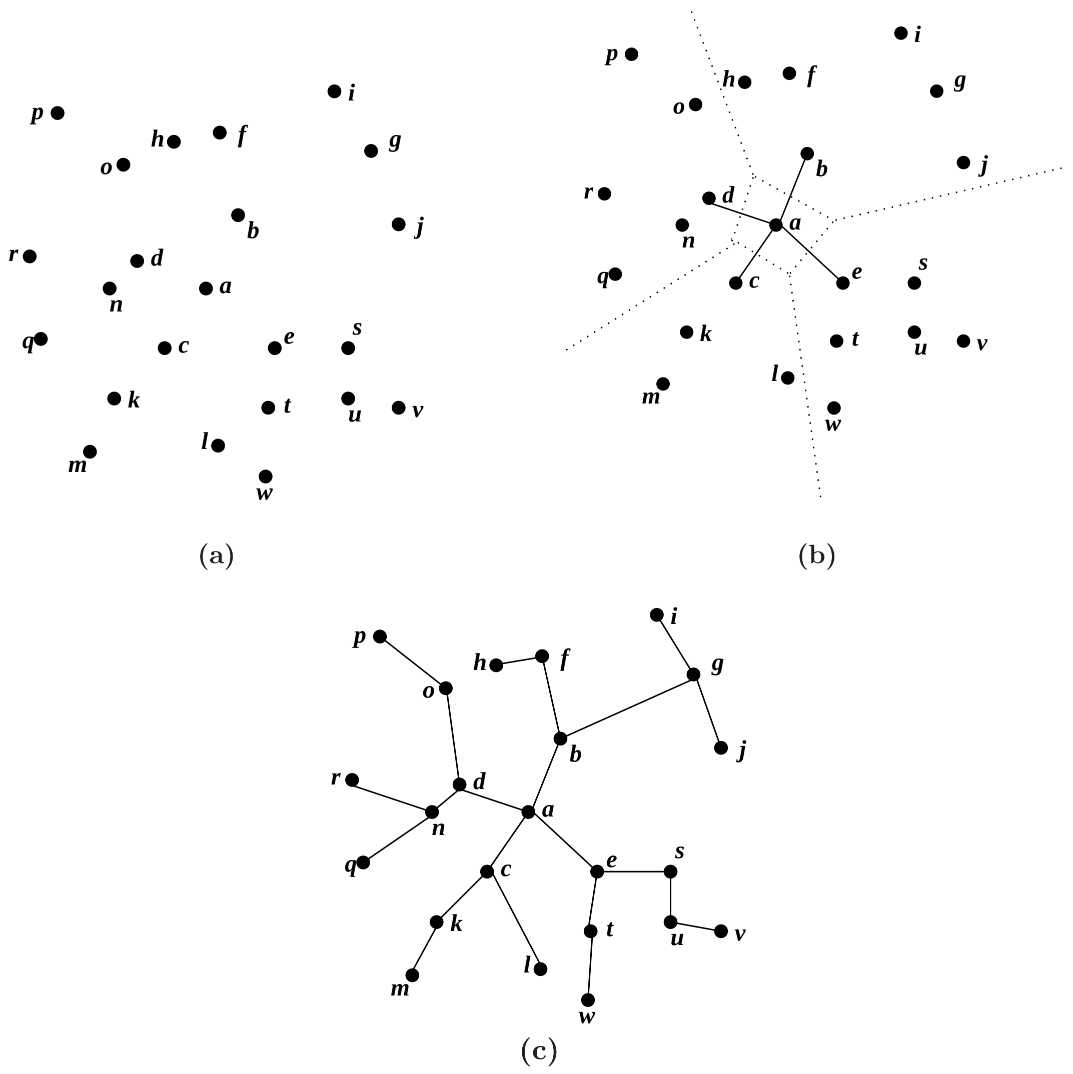


Figure 2: An example of the *sa–tree* construction process.

Together with each node a we also store its covering radius, that is, the maximum distance $R(a)$ between a and any element in the subtree rooted at a.

Algorithm 1 depicts the construction process. It is first invoked as `BuildTree(`a`,`$S-\{a\}$`)` where a is a random element of S. Note that, except for the first level of the recursion, we already know all the distances $d(v,a)$ for every $v \in S$ and hence do not need to recompute them. Similarly, some of the $d(v,b)$ distances at line 8 are already known from line 5. The information stored by the data structure is the root a and the $N()$ and $R()$ values of all the nodes.

```
BuildTree(Node $a$, Set of nodes $S$)

 1. $N(a) \leftarrow \emptyset$        /* neighbors of $a$ */
 2. $R(a) \leftarrow 0$        /* covering radius */
 3. For $v \in S$ in increasing distance to $a$ Do
 4.    $R(a) \leftarrow \max(R(a), d(v,a))$
 5.    If $\forall b \in N(a),\ d(v,a) < d(v,b)$ Then $N(a) \leftarrow N(a) \cup \{v\}$
 6. For $b \in N(a)$ Do $S(b) \leftarrow \emptyset$
 7. For $v \in S - N(a)$ Do
 8.    $c \leftarrow \mathrm{argmin}_{b \in N(a)} d(v,b)$
 9.    $S(c) \leftarrow S(c) \cup \{v\}$
10. For $b \in N(a)$ Do BuildTree($b$,$S(b)$)
```

Algorithm 1: Algorithm to build a *sa–tree* for $S \cup \{a\}$ with root a.

Note that the construction process does not guarantee that the structure is balanced. However, it is not clear at all that this is a desirable property for metric space searching, see [CN05].

4.2 Range Searching

Note that the structure that results from the above construction is a tree that can be searched for any $q \in S$ by spatial approximation using nearest neighbor queries. The reason why this works is that, at search time, we repeat exactly what happened to q during the construction process (that is, we enter the subtree of the neighbor closest to q), until we reach q. This is because q is present in the tree, that is, we are doing an exact search after all.

Of course it is of little interest to search only for elements $q \in S$. The tree we have described can, however, be used as a device to solve queries of any type for any $q \in \mathbb{U}$. We consider first range queries with radius r.

The key observation is that, even if $q \notin S$, the answers to the query are elements $q' \in S$. So we use the tree to pretend that we are searching for an element $q' \in S$. We do not know q', but since $d(q,q') \leq r$, we can obtain from q some distance information regarding q': by the triangle inequality it holds that for any $x \in \mathbb{U}$, $d(x,q) - r \leq d(x,q') \leq d(x,q) + r$.

Hence, instead of simply going to the closest neighbor, we first determine the closest neighbor c of q among $\{a\} \cup N(a)$. We then enter *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 into any of those b nodes. In the process, we report all the nodes q' we found close enough to q.

Therefore, we are pruning all subtrees rooted at those b such that $d(q,b) > d(q,c) + 2r$. A more sophisticated pruning criterion is obtained by noticing that all elements inserted into child c

of a are not only closer to c than to a and $N(a)$, but also closer to a than to the parent of a and any neighbor of the parent of a. Extending the argument transitively, we see that c is closer to a than to any ancestor of a and to any neighbor of any ancestor of a. Let us call $A(a)$ the set of ancestors of a in the *sa–tree* (we include a itself in $A(a)$), and $N(A(a))$ the set of neighbors of ancestors of a. Therefore, we can take c as the closest element to q among $N(A(a))$.

Finally, the covering radius $R(a)$ is used to further prune the search, by not entering subtrees such that $d(q,a) > R(a) + r$, since they cannot contain useful elements.

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

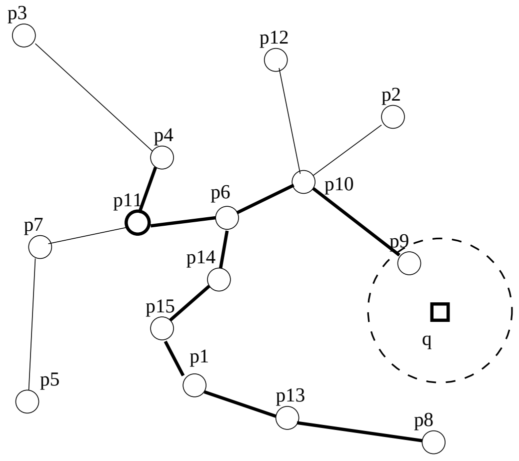


Figure 3: An example of the search process.

```
RangeSearch(Node $a$, Query $q$, Radius $r$, Distance $d_{min}$)

 1. If $d(a,q) \leq R(a) + r$ Then
 2.    If $d(a,q) \leq r$ Then Report $a$
 3.    $d_{min} \leftarrow \min\ \{d(c,q),\ \ c \in N(a)\} \cup \{d_{min}\}$
 4.    For $b \in N(a)$ Do
 5.       If $d(b,q) \leq d_{min} + 2r$ Then RangeSearch($b$,$q$,$r$,$d_{min}$)
```

Algorithm 2: The algorithm to search for q with radius r in a *sa–tree* with root a.

4.3 Nearest Neighbor Searching

As explained in Section 3.5, nearest neighbor searching can be done using a *sa–tree* by adapting the range search algorithm. The main issue is how to compute a lower bound to the distance between q and a subtree. Two lower bounds are given by the tree, and also the lower bounds of the ancestors are inherited.

1. Since we compute d_{min} and then enter any neighbor b such that $d(q,b) - d_{min} \leq 2r$, a lower bound distance from the subtree rooted at b to q is $(d(q,b) - d_{min})/2$.

2. Another lower bound to the distance between q and an element in the subtree is $d(q,b) - R(b)$.

3. Children nodes inherit the lower bound of their parents, if their own lower bounds are not better.

Note that the d_{min} value for each subtree inserted into Q must also be remembered. Algorithm 3 depicts the algorithm[1] `NNsearch(`a,q,k`)`, which performs a k-NN query for q on the tree rooted at a.

```
NNsearch(Tree $a$, Query $q$, Neighbors wanted $k$)

 1. $create(Q)$, $create(A)$
 2. $insert(Q, (a, \max\{0, d(q,a) - R(a)\}, d(a,q)))$
 3. $r \leftarrow \infty$
 4. While $size(Q) > 0$ Do
 5.    $(a, lbound, d_{min}) \leftarrow extractMin(Q)$
 6.    If $lbound > r$ Then Break
 7.    $insert(A, (a, d(q,a)))$
 8.    If $size(A) > k$ Then $extractMax(A)$
 9.    If $size(A) = k$ Then $r \leftarrow max(A)$
10.    $d_{min} \leftarrow \min\ \{d_{min}\} \cup \{d(b,q),\ b \in N(a)\}$
11.    For $b \in N(a)$ Do
12.       $insert(Q, (b, \max\{(d(q,b) - d_{min})/2, d(q,b) - R(b), t\}, d_{min})$
13. Return $A$
```

Algorithm 3: Algorithm to search for the k nearest neighbors of q in a *sa–tree* rooted at a. A is a priority queue of pairs *(node,distance)* sorted by decreasing *distance*. Q is a priority queue of triples *(node,lbound,dmin)* sorted by increasing *lbound*.

5 Incremental Construction

The construction of the *sa–tree* 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.

In this section we discuss and empirically evaluate different alternatives to permit insertion of new elements into an already built *sa–tree*. We start with naive and/or folklore alternatives (rebuilding the subtree and using overflow buckets).[2] Then we move to more sophisticated choices that are based on specific properties of the *sa–tree*: first-fit, timestamping, and inserting at the fringe. Finally, we consider combinations of the previous choices and propose the best alternative overall.

[1]The original algorithm [Nav02] has a mistake in line 12, as $(d(q,b) - d_{min})$ is not divided by 2 as it should.

[2]There exist classical alternatives to turn static into dynamic data structures [BS80, GR93]. Those are not considered here because, although they would not degrade much the insertion performance, our best alternatives indeed improve the insertion costs. Deletions, on the other hand, are handled by maintaining objects for some time after they are removed, and we considered this unacceptable in this application, as explained in Section 6.

5.1 Rebuilding the Subtree

The crudest approach is to collect all the set S again, and rebuild the *sa–tree* for $S \cup \{x\}$. This has the advantage of preserving exactly the same tree that is built statically, and therefore the search performance is as good as on the static tree. In particular, the tree will have the same arity as with the static construction, $O(\log n)$ at the root of a subtree of size n [Nav02].

Let us refine a bit this procedure in order to avoid unnecessary recomputation. If we built the tree on $S \cup \{x\}$, x would take some place after we sorted the set by increasing distance to a (there is no reason to choose $x = a$ as the root). This means that x would find some of the neighbors in $N(a)$ already inserted when its time came. Should x be closer to any of the already inserted neighbors than to a, x would be inserted into the subtree of that neighbor and the rest of the construction would be exactly the same.

This means that, instead of fully rebuilding the tree, we could first check whether there exists $b \in N(a)$ such that $d(b,a) \leq d(x,a)$ (so b would be in $N(a)$ when x was considered) and $d(x,b) \leq d(x,a)$ (so x would prefer to get into the subtree of b instead of being part of $N(a)$). If such a b exists, we pick the one minimizing $d(x,b)$ and continue the process of inserting x in the subtree rooted at b. If, at some point, no such b exists, then x should become a neighbor of the current tree node a. At this point we must fully rebuild the subtree rooted at a, since some nodes that went into neighbors could now prefer to get into the new neighbor x.

This process strictly guarantees that the resulting tree is exactly the same *sa–tree* for $S \cup \{x\}$. However, we can slightly relax the condition of becoming a neighbor so as to insert x as low as possible in the tree. Imagine that there is a neighbor $b \in N(a)$ such that $d(x,b) \leq d(x,a)$, but $d(x,a) < d(b,a)$. In rigor, x should become a neighbor of a and b should get into the subtree rooted at x. However, if we instead continue the insertion of x inside the subtree rooted at b, Condition 1 is still satisfied at node a. This is equivalent to assuming that x was appended at the end of the sorted list S.[3] The net result is that we will insert x into $N(a)$ only when $d(x,a) < d(x,b)$ for all $b \in N(a)$; otherwise we will insert x into the subtree of its closest neighbor in $N(a)$. The resulting tree is not exactly the same of the static *sa–tree* construction for $S \cup \{x\}$, but still a correct *sa–tree*, that can be built at lower cost. In particular, the same search algorithms of the static *sa–tree* can be used.

Algorithm 4 shows the insertion algorithm for element x into a tree rooted at a. We follow only one path from the tree root to the parent of the inserted element and then rebuild the whole subtree. The *dsa–tree* can be built by successive insertions into an initial tree formed by a single node a where $N(a) = \emptyset$ and $R(a) = 0$. Algorithm `BuildTree(`a`,`S`)`, invoked at line 4, is that used for the static construction (Algorithm 1).

Figure 14 (in the Appendix) shows the cost to build the *sa–tree* by successive insertions using this technique versus that of building it statically. As it can be seen, even with the improvement we have made, the incremental construction is prohibitively costly for this alternative to be considered seriously (20–40 times the static construction cost).

5.2 Using Overflow Buckets

We can have an overflow bucket per node with "extra" neighbors that should go in the subtree but have not yet been classified. We follow the same insertion mechanism of the previous section

[3]We saw that the sorting was necessary to ensure Condition 1 to hold, but this was used only to ensure that elements inserted into $N(a)$ were sorted; elements not inserted into $N(a)$ could be in any order.

```
InsertRB(Node $a$, Element $x$)

 1. $c \leftarrow \text{argmin}_{b \in N(a)} d(b,x)$
 2. If $d(a,x) < d(c,x)$ Then
 3.   Collect in $S$ the elements of the subtree rooted at $a$
 4.   BuildTree($a$,$S \cup \{x\}$) /* rebuild the subtree */
 5. Else
 6.   $R(a) \leftarrow \max\{R(a), d(a,x)\}$
 7.   InsertRB($c$,$x$)
```

Algorithm 4: Insertion algorithm of a new element x into a *dsa–tree* with root a, using the method of rebuilding the subtree.

until we determine that the new element x must become a neighbor of a. At this point, instead of rebuilding the subtree rooted at a, we put x in the overflow bucket of a. Each time we reach a at query time, we also compare q against all the elements in its overflow bucket and report any close enough element.

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 and query costs. As smaller overflow buckets are permitted, we rebuild the tree more often and improve query time, but construction time increases.

Algorithm 5 illustrates the insertion process of a new element x into a *dsa–tree* rooted at a, using overflow buckets. $MaxOB$ is the maximum size allowed for the overflow bucket $OB()$ of a node. We follow the path from the root to the parent of the inserted element, and then, if its overflow bucket is not full, we insert the new element there, otherwise we rebuild the whole subtree. We can build the *dsa–tree* by successive insertions over an initial tree formed by a single node a where $N(a) = \emptyset$, $OB(a) = \emptyset$ and $R(a) = 0$. Algorithm `BuildTree(`a`,`S`)`, invoked at line 6, is that used for the static construction (Algorithm 1). At line 5 we collect all the elements in the subtree rooted at a, which includes those not yet classified and hence allocated in the overflow buckets.

```
InsertOB(Node $a$, Element $x$)

 1. $c \leftarrow \text{argmin}_{b \in N(a)} d(b,x)$
 2. If $d(a,x) < d(c,x)$ Then
 3.   If $|OB(a)| < MaxOB$ Then $OB(a) \leftarrow OB(a) \cup \{x\}$ /* add to bucket */
 4.   Else
 5.     Collect in $S$ the elements of the subtree rooted at $a$
 6.     BuildTree($a$,$S \cup \{x\}$) /* rebuild the subtree */
 7. Else
 8.   $R(a) \leftarrow \max\{R(a), d(a,x)\}$
 9.   InsertOB($c$,$x$)
```

Algorithm 5: Insertion algorithm of a new element x into a *dsa–tree* rooted at a, using overflow buckets.

Figure 15 shows the construction cost using different maximum bucket sizes, which exhibits significant 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 space. The fluctuations appear because a larger bucket size may produce more rebuilds than a smaller one for a given set size n.

Algorithm 6 depicts the range search algorithm for a *dsa–tree* built using overflow buckets. It is initially invoked as `RangeSearchOB(`a`,`q`,`r`,`$d(a,q)$`)` where a is the tree root. Note that, in recursive invocations, the distance $d(a,q)$ is already computed. The only difference from the algorithm presented in Algorithm 2 is found at lines 3 and 4 of Algorithm 6, where we must compare the query q against all the elements in the overflow buckets (OB), reporting those that are close enough to q. The algorithm for k-NN queries is the same as that of Algorithm 3, except that all the elements in $OB(a)$ should be inserted into A after line 7.

```
RangeSearchOB(Node $a$, Query $q$, Radius $r$, Distance $d_{min}$)

1. If $d(a,q) \leq R(a) + r$ Then
2.    If $d(a,q) \leq r$ Then Report $a$
3.    For $b \in OB(a)$ Do
4.       If $d(b,q) \leq r$ Then Report $b$
5.    $d_{min} \leftarrow \min\ \{d_{min}\} \cup \{d(q,c), c \in N(a)\}$
6.    For $b \in N(a)$ Do
7.       If $d(b,q) \leq d_{min} + 2r$ Then RangeSearchOB($b$,$q$,$r$,$d_{min}$)
```

Algorithm 6: Algorithm to search for q with radius r in a *dsa–tree* rooted at a built using overflow buckets.

Figure 16 shows the search costs using overflow buckets. As it can be seen, this technique permits interesting tradeoffs between search and construction costs. In general, lower construction costs correspond to higher search costs. This has to do with the number of elements that stay in the overflow buckets. It is usually possible to find a bucket size so that construction and search costs are similar to those of the static version. The main problem with this method is its high sensitivity to the maximum bucket size, which makes it difficult to select a bucket size that achieves a good tradeoff.

5.3 A First-Fit Strategy

An alternative to the best-fit strategy is the *first-fit* strategy, which puts 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 we can easily add more elements by pretending that the new incoming element x was the last in the bag. This means that, when x becomes a neighbor of a, it can be simply appended at the end of $N(a)$, and there were no further elements in the bag that had the chance of getting into x. Hence, no reconstruction of the tree is necessary. This allows building the structure by successive insertions at low cost.

Algorithm 7 depicts the insertion algorithm of a new element x into a *dsa–tree* rooted at a, built using the first-fit strategy. Figure 17 shows that the construction using first-fit is much cheaper than the static construction using best-fit strategy (disregard for now the curve labeled “Timestamp”).

Range searching under the first-fit strategy is a bit different. We consider the neighbors $\{b_1, \ldots, b_k\}$ of a in order. We perform the minimization while we traverse the neighbors. That is, we enter the subtree of b_1 if $d(q,b_1) \leq d(q,a) + 2r$; the subtree of b_2 if $d(q,b_2) \leq \min(d(q,a), d(q,b_1)) +$

```
InsertFF(Node $a$, Element $x$)

 1. $R(a) \leftarrow \max\{R(a), d(a,x)\}$
 2. $c \leftarrow a$
 3. For $b_i \in N(a)$ Do /* taking the neighbors $b_i$ in order */
 4.    If $d(b_i,x) \leq d(a,x)$ Then
 5.       $c \leftarrow b_i$ /* the first closer neighbor */
 6.       Break
 7. If $c = a$ Then
 8.    $N(a) \leftarrow N(a) \cup \{x\}$
 9.    $N(x) \leftarrow \emptyset$, $R(x) \leftarrow 0$
10. Else InsertFF($c$,$x$)
```

Algorithm 7: Insertion algorithm of a new element x into a *dsa–tree* rooted at a, using first-fit strategy.

$2r$; and in general the subtree of b_i if $d(q,b_i) \leq \min(d(q,a), d(q,b_1), \ldots, d(q,b_{i-1})) + 2r$. This is because b_{i+j} can never take out an element from b_i, so even if there exists a closer neighbor later, the elements of interest will choose the first one.

Algorithm 8 shows the search algorithm for a *dsa–tree* built using the first-fit strategy. It is invoked as `RangeSearchFF(`a,q,r,$d(a,q)$`)`, where a is the tree root. Note that, in recursive invocations, the distance $d(a,q)$ is already computed. Line 5 performs the minimization while traversing the neighbors in order.

```
RangeSearchFF(Node $a$, Query $q$, Radius $r$, Distance $d_{min}$)

1. If $d(a,q) \leq R(a) + r$ Then
2.    If $d(a,q) \leq r$ Then Report $a$
3.    For $b_i \in N(a)$ Do   /* considering neighbors in order */
4.       If $d(b_i,q) \leq d_{min} + 2r$ Then RangeSearchFF($b_i$,$q$,$r$,$d_{min}$)
5.       $d_{min} \leftarrow \min\{d_{min}, d(q,b_i)\}$
```

Algorithm 8: Algorithm to search for q with radius r into a *dsa–tree* rooted at a built using the first-fit strategy.

The k-NN search algorithm for this version is easily obtained by considering that the three bounds mentioned in Section 4.3 still hold, as long as we understand that d_{min} is computed incrementally as we traverse the neighbors, instead of first computing it over all the neighbors and then using it to insert elements into Q. That is, line 10 of Algorithm 3 is removed and an instruction $d_{min} \leftarrow \min\{d_{min}, d(b,q)\}$ is executed after each insertion at line 12.

Figure 18 shows range search times. As it can be seen, the search performance using a first-fit construction is poor in the dictionary (except for $r = 1$) and in the space of feature vectors, at a point that it cannot compete against simpler alternatives. However, in documents and in the space of color histograms it obtains significantly better search performance than the static construction using best-fit strategy. Thus, first-fit turns out to be a very interesting alternative for those spaces, as in addition it provides much faster construction.

5.4 Timestamping

The first-fit strategy suffers from two shortcomings. On one hand, the tree has more elements in its first branches, and on the other hand, those branches are visited more frequently at search time. This kind of unbalancing is usually beneficial in "difficult" spaces [CN05], as shown by the experiments in the previous section, but not on most spaces. In this section we seek a more balanced structure.

An alternative that partially solves the first problem (producing more balanced trees) and still retains the low insertion cost of first-fit strategy, is based on 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 strategy, but 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.

Algorithm 9 depicts the insertion process for element x over a tree rooted at a. The *dsa–tree* can be built by successive insertions over an initial tree formed by a single node a where $N(a) = \emptyset$, $time(a) = CurrentTime = 1$ and $R(a) = 0$.

```
InsertTS(Node $a$, Element $x$)

1. $R(a) \leftarrow \max\{R(a), d(a,x)\}$
2. $c \leftarrow \text{argmin}_{b \in N(a)} d(b,x)$
3. If $d(a,x) < d(c,x)$ Then
4.     $N(a) \leftarrow N(a) \cup \{x\}$
5.     $N(x) \leftarrow \emptyset$, $R(x) \leftarrow 0$
6.     $time(x) \leftarrow CurrentTime$
7.     $CurrentTime \leftarrow CurrentTime + 1$
8. Else InsertTS($c$,$x$)
```

Algorithm 9: Insertion of a new element x into a *dsa–tree* rooted at a, using the timestamp technique.

As seen in Figure 17, this alternative can cost from a moderately more to much less than the static construction, depending on the case.

At search time, we consider the neighbors $\{b_1, \ldots, b_k\}$ of a from oldest to newest. We perform the minimization while we traverse the neighbors, exactly as in Section 5.3. This is because between the insertion of b_i and b_{i+j} there may have appeared new elements that preferred b_i just because b_{i+j} was not yet a neighbor, so we may miss an element if we do not enter b_i because of the existence of b_{i+j}. Note that, although the search process is the same as under first-fit strategy, 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 by them. Yet a more sophisticated scheme is to use the timestamps to reduce the work done inside older neighbors at search time. Say that $d(q, b_i) > d(q, b_{i+j}) + 2r$. We have to enter b_i because it is older. However, only the elements with timestamp smaller than that of b_{i+j} should be considered when searching inside b_i; younger elements have seen b_{i+j} and they cannot be interesting for the search if they chose b_i. As parent nodes are older than their descendants, as soon as we find a node inside the subtree of b_i with timestamp larger than that of b_{i+j} we can stop the search in that branch, because its subtree is even younger.

Algorithm 10 shows the range search algorithm considering a *dsa–tree* built using the timestamp strategy. The computation of d_{min} is carried out in line 7, as we traverse the neighbors in ascending timestamp order. The algorithm is initially invoked as `RangeSearchTS(`a`,`q`,`r`,`$d(a,q)$`,`$CurrentTime$`)` where a is the tree root. Note that the distance $d(a,q)$ is already computed in recursive invocations. Despite the quadratic nature of the loop implicit in lines 3 and 5, the query is of course compared only once against each neighbor.

```
RangeSearchTS(Node $a$, Query $q$, Radius $r$, Distance $d_{min}$, Timestamp $t$)

1. If $time(a) < t \;\wedge\; d(a,q) \leq R(a) + r$ Then
2.    If $d(a,q) \leq r$ Then Report $a$
3.    For $b_i \in N(a)$ Do   /* in ascending timestamp order */
4.       If $d(b_i,q) \leq d_{min} + 2r$ Then
5.          $t' \leftarrow \min\{t\} \cup \{time(b_j), j > i \wedge d(b_i,q) > d(b_j,q) + 2r\}$
6.          RangeSearchTS($b_i$,$q$,$r$,$d_{min}$,$t'$)
7.          $d_{min} \leftarrow \min\{d_{min}, d(b_i,q)\}$
```

Algorithm 10: Algorithm to search for q with radius r into a *dsa–tree* rooted at a built with the timestamp technique.

Let us now consider nearest neighbor searching. We have to manage to express our operational handling of timestamps as lower bounds on distances. Instead of thinking in terms of maximum allowed timestamp of interest inside y, let us think in terms of maximum search radius that permits entering y. Each time we enter a subtree y of b_i, we search for the siblings b_{i+j} of b_i that are older than y. Over this set, we compute the maximum radius that permits us not to enter y, namely $r_y = \max(d(q,b_i) - d(q,b_{i+j}))/2$. If it holds $r < r_y$, then we do not need to enter the subtree y.

Assume that we are currently processing node b_i and insert its children y into the priority queue. We compute r_y and insert it together with y into 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 child z of y, then we put it with value $\max(r_y, r_z)$. Algorithm 11 shows the algorithm.

Figure 18 compares this technique against the static one. As it can be seen, timestamping is a good alternative to the static construction in all the spaces except the dictionary, providing the same or much better construction cost and also better search performance than the static version. Timestamping also performs better than the first-fit strategy on some spaces.

5.5 Inserting at the Fringe

Let us imagine that we remove the "$\Leftarrow$" part of Condition 1 (Section 4.1), that is, there are some elements closer to a than to any element of $N(a)$, and yet those elements are not in $N(a)$. This part of Condition 1 guarantees that, if q is closer to a than to any neighbor in $N(a)$, then we can stop the search at that point because q should be in $N(a)$ and not inside any subtree. If we weaken Condition 1 as explained, then there is no such guarantee. Even if x is closer to a than to any neighbor in $N(a)$, x could be in the subtree of its closest neighbor in $N(a)$.

Hence, at search time, instead of finding the closest c among $\{a\} \cup N(a)$ and entering any $b \in N(a)$ such that $d(q,b) \leq d(q,c) + 2r$, we exclude the subtree root $\{a\}$ from the minimization. Therefore, we *always* continue to the leaves, by the closest neighbor and others close enough. This seems to degrade the search time, but the difference is marginal in practice.

```
NNsearchTS(Tree $a$, Query $q$, Neighbors wanted $k$)

 1. $create(Q)$, $create(A)$
 2. $insert(Q, (a, \max\{0, d(q,a) - R(a)\}, d(q,a)))$
 3. $r \leftarrow \infty$
 4. While $size(Q) > 0$ Do
 5.    $(a, lbound, d_{min}) \leftarrow extractMin(Q)$
 6.    If $lbound > r$ Then Break
 7.    $insert(A, (a, d(q,a)))$
 8.    If $size(A) > k$ Then $extractMax(A)$
 9.    If $size(A) = k$ Then $r \leftarrow max(A)$
10.    For $b_i \in N(a)$ Do   /* in increasing timestamp order */
11.       $maxr \leftarrow \max\ \{(d(q,b_i) - d(q,b_j))/2, j > i\}$
12.       $insert(Q, (b_i, \max\{maxr, (d(q,b_i) - d_{min})/2, d(q,b_i) - R(b_i), t\}, d_{min})$
13.       $d_{min} \leftarrow \min\{d_{min}, d(q,b_i)\}$
14. Return $A$
```

Algorithm 11: Algorithm to search for the k nearest neighbors of q in a *dsa–tree* rooted at a built using timestamp. A is a priority queue of pairs *(node,distance)* sorted by decreasing *distance*. Q is a priority queue of triples *(node,lbound,dmin)* sorted by increasing *lbound*.

The benefit is that, at insertion time, we are not forced anymore to put the new element x as a neighbor of a, even when Condition 1 would require that. 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 in $N(a)$. At search time we will reach x because the search and insertion processes are similar.

An immediate consequence of this freedom is that we can always insert at the leaves of the tree. That is, the tree is read-only in its top part and changes only at its fringe. However, we have to permit the reconstruction of small subtrees so that the tree does not degenerate into 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.

Algorithm 12 depicts the insertion algorithm of a new element x into a *dsa–tree* rooted at a. $MaxSize$ is the maximum tree size allowed to rebuild and $size(a)$ is the size of the subtree rooted at a. The *dsa–tree* can be built by successive insertions over an initial tree formed by a single node a with $N(a) = \emptyset$, $size(a) = 1$ and $R(a) = 0$. At line 4 we invoke algorithm `BuildTree(`a`,`S`)` used for the static construction (Algorithm 1).

Figure 19 shows the construction cost for different maximum tree sizes that can be rebuilt. As it can be seen, permitting a tree size of 50 yields similar construction cost as the static version, and reasonably close costs are achieved with tree sizes from 10 to 100.

Algorithm 13 depicts the range search algorithm considering that the *dsa–tree* was built using insertion at the fringe. It is invoked as `RangeSearchFR(`a`,`q`,`r`)` where a is the tree root, and at recursive invocations $d(a,q)$ is already known. The algorithm is very similar to the static version (Algorithm 2), but there is an important difference. The value d_{min} is not inherited, which means that the root a is not included in the minimization, as explained. It also means that the neighbors of ancestors of a, $N(A(a))$, are excluded from the minimization. The reason is that, given the relaxation to Condition 1, it is not guaranteed that $b \in N(a)$ is closer to a than to the parent of a, or to any ancestor of a.

```
InsertFR(Node $a$, Element $x$)

1. $c \leftarrow \mathrm{argmin}_{b \in N(a)} d(b,x)$
2. If $d(a,x) < d(c,x) \;\wedge\; size(a) < MaxSize$ Then
3.    Collect in $S$ the elements of the subtree rooted at $a$
4.    BuildTree($a$,$S \cup \{x\}$) /* rebuild the subtree */
5. Else
6.    $R(a) \leftarrow \max\{R(a), d(a,x)\}$
7.    InsertFR($c$,$x$)
```

Algorithm 12: Insertion algorithm of a new element x into a *dsa–tree* rooted at a, using insertion at the fringe.

```
RangeSearchFR(Node $a$, Query $q$, Radius $r$)

1. If $d(a,q) \leq R(a) + r$ Then
2.    If $d(a,q) \leq r$ Then Report $a$
3.    $d_{min} \leftarrow \min\ \{d(q,b_i),\ b_i \in N(a)\}$
4.    For $b_i \in N(a)$ Do
5.       If $d(b_i,q) \leq d_{min} + 2r$ Then RangeSearchFR($b_i$,$q$,$r$)
```

Algorithm 13: Algorithm to search for q with radius r into a *dsa–tree* rooted at a built by insertion at the fringe.

Nearest neighbor searching is also simplified from that of Algorithm 3. It is not necessary to store d_{min} together with the subtrees maintained in Q, and in line 10, element $\{d_{min}\}$ can be excluded from the minimization in the right hand of the assignment.

Figure 20 shows the search time using this technique. As it can be seen, using a tree size of 10 to 100 yields usually much better search time compared to the static version. The exception is the dictionary, where all the costs are very close anyway. This shows that it may be beneficial to move elements downward in the tree, which is an interesting result we study more in depth next.

5.6 Bounding the Arity

The relaxation used in the previous section can be used in several ways. It is particularly interesting how it can significantly reduce construction time while retaining a competitive search time. By analyzing the trees resulting from the above dynamic construction, we have found that, in the cases where dynamic construction improves most over static construction, the average arity (number of children) of the tree nodes is reduced most. This seems to indicate that the reason why the *sa–tree* performs poorly in some spaces is that its arity is too high. Even when the *sa–tree* automatically adapts its arity to the space, this mechanism is not optimal.

This gives us a motivation for a different way of controlling insertions in a *dsa–tree*. We directly control the tree arity by fixing a maximum admissible arity, $MaxArity$. Whenever a new inserted element wants to become a neighbor of a tree node a, we permit that only if $|N(a)| < MaxArity$, otherwise the element is forced to choose its closest $b \in N(a)$ and continue the insertion there. Algorithm 14 gives the insertion algorithm, which is very similar to `InsertFR`. Range and nearest neighbor searching are identical to the version for inserting at the fringe.

```
InsertBA(Node $a$, Element $x$)

1. $c \leftarrow \mathrm{argmin}_{b \in N(a)} d(b,x)$
2. If $d(a,x) < d(c,x) \ \wedge \ |N(a)| < MaxArity$ Then
3.   Collect in $S$ the elements of the subtree rooted at $a$
4.   BuildTree($a$,$S \cup \{x\}$) /* rebuild the subtree */
5. Else
6.   $R(a) \leftarrow \max\{R(a), d(a,x)\}$
7.   InsertBA($c$,$x$)
```

Algorithm 14: Insertion algorithm of a new element x into a *dsa–tree* rooted at a, using bounded arity.

Figure 21 shows the construction cost for different maximum arities. As it can be seen, permitting a maximum arity of 4 yields the same construction cost of the static version. The construction cost increases as the arity grows, and it becomes too large already for arity 8 in most cases.

Figure 22 shows the search time using this technique. Except on the dictionary, the lowest arity is the best, and the static search cost is reached for arity 8. In the dictionary, on the other hand, we need larger arities, reaching a search cost similar to the static version for arity at least 8. This shows again that it may be beneficial to move elements downward in the tree.

5.7 Combining Insertion Algorithms

The last two alternatives yielded better construction times than the static version. Although the initial idea was to limit the size of the tree to rebuild, a side effect was that the dynamic tree was better suited to some spaces. These restrictions on the insertion point can therefore be viewed as tuning parameters by themselves, unrelated to the goal of limiting the size of the tree to rebuild. Moreover, the reconstruction cost itself can be completely avoided by combining them with timestamping. This way, we would have trees that use timestamping to avoid any reconstruction, and at the same time limit the possible insertion points with the aim of obtaining a tree of better shape.

The variant combining timestamping with bounded arity works as follows. We fix a maximum tree arity, and also keep a timestamp of the insertion time of each element. The search for the insertion point is exactly as in Algorithm 14, except that in lines 3 and 4 we do not rebuild the subtree but rather add x as the last neighbor in the list. Algorithm 15 gives the algorithm. The variant combining timestamping with insertion at the fringe (`InsertTF`) is similar: Condition $|N(a)| < MaxArity$ in line 3 becomes $size(a) < MaxSize$.

Figure 23 shows the construction costs combining timestamping with insertion at the fringe. The costs are much better than without timestamping, and much better than the static construction cost as well. Figure 24 shows the construction cost for different maximum arities, using timestamping plus bounded arity. The results are very similar.

At search time we have to combine the considerations done for timestamping with those for bounded arity. Algorithm 16 shows the search algorithm (algorithm `RangeSearchTF` for insertion at the fringe is identical). Note that $d(a,q)$ is always known except in the first invocation.

Figures 25 and 26 compare the search costs of the methods that combine timestamping with insertion at the fringe and bounded arity, respectively, against the static version. Except on the

```
**InsertTBA**(Node $a$, Element $x$)

1. $R(a) \leftarrow \max(R(a), d(a,x))$
2. $c \leftarrow \mathrm{argmin}_{b \in N(a)} d(b,x)$
3. If $d(a,x) < d(c,x) \wedge |N(a)| < MaxArity$ Then
4.    $N(a) \leftarrow N(a) \cup \{x\}$
5.    $N(x) \leftarrow \emptyset$, $R(x) \leftarrow 0$
6.    $time(x) \leftarrow CurrentTime$
7.    $CurrentTime \leftarrow CurrentTime + 1$
8. Else **InsertTBA**($c$,$x$)
```

Algorithm 15: Insertion of a new element x into a *dsa–tree* with root a using timestamping plus bounded arity.

```
**RangeSearchTBA**(Node $a$, Query $q$, Radius $r$, Timestamp $t$)

1. If $time(a) < t \;\wedge\; d(a,q) \leq R(a) + r$ Then
2.    If $d(a,q) \leq r$ Then Report $a$
3.    $d_{min} \leftarrow \infty$
4.    For $b_i \in N(a)$ Do   /* in ascending timestamp order */
5.       If $d(b_i,q) \leq d_{min} + 2r$ Then
6.          $t' \leftarrow \min\{t\} \cup \{time(b_j), j > i \wedge d(b_i,q) > d(b_j,q) + 2r\}$
7.          **RangeSearchTBA**($b_i$,$q$,$r$,$t'$)
8.          $d_{min} \leftarrow \min\{d_{min}, d(b_i,q)\}$
```

Algorithm 16: Searching for q with radius r in a *dsa–tree* rooted at a, built with timestamping plus bounded arity.

dictionary, the performance of the static version is surpassed, and smallest trees or arities work better. On the dictionary, we must use large enough tree reconstruction sizes and arities to approach the performance of the static version.

Nearest neighbor searching is also a combination of both algorithms. It is almost like `NNsearchTS` (Algorithm 11) except that d_{min} is not stored in Q but initialized at ∞ just before line 10.

5.8 Choosing the Best Insertion Algorithm

We have proposed a number of techniques to build the *sa–tree* incrementally, each one giving us different tradeoffs between construction and search cost. Several of those have improved construction and search time simultaneously. Figures 4 and 5 illustrate this tradeoff, for every space and search radius. Observe that a difference in the x axis is more significant as that scale is logarithmic.

In all the insertion methods and metric spaces, the standard deviation of the search process is around 0.5 of the mean, and therefore the standard deviation of the estimations is 5% to 0.5% of the mean, depending on the size of the space.

The first conclusion is that the static version is never an interesting choice. In each case there is some other alternative that obtains better construction and search cost simultaneously. The only case where it is not superseded is in the dictionary and large search radius, but even there it is very close to other methods. Similarly, the use of timestamping (alone) and of overflow buckets is never the best choice.

A remarkable alternative that turns out to be relevant for its very low construction cost is first-fit. Although it is usually far from the best search time that can be achieved, no other technique can obtain the same search performance with as low construction cost.

With regard to search time, a remarkable method is insertion at the fringe, which in many cases obtains results far better than what can be achieved with any other choice. In some cases it is closely followed by bounded arity, but only on strings with $r = 3$ the latter is clearly better.

Finally, both combined alternatives are very similar and much more stable with respect to changing parameters. They usually offer a tradeoff between first-fit and insertion at the fringe or with bounded arity.

In order to continue our work with deletions, to keep the scope of the paper reasonably bounded, we will stick to one of the insertion techniques. Yet, any other technique can be easily adapted to support deletions. We have chosen the combination of timestamping with bounded arity, which is in most cases among the best choices and never a very bad choice. Although sometimes it is widely surpassed by bounded arity alone, its good performance is more consistent. For example, it works also well on strings, which is a discrete space that behaves very differently of the others. We could also have chosen the combination of timestamping with insertion at the fringe. However, the bounded arity has an extra plus if we have secondary memory as a future goal: Bounding the arity simplifies the task of packing subtrees into disk blocks, and deciding how many subtree levels fit in a block.

6 Deletions

To delete an element x, the first step is to find it in the tree. Unlike most classical data structures for traditional search problems, 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

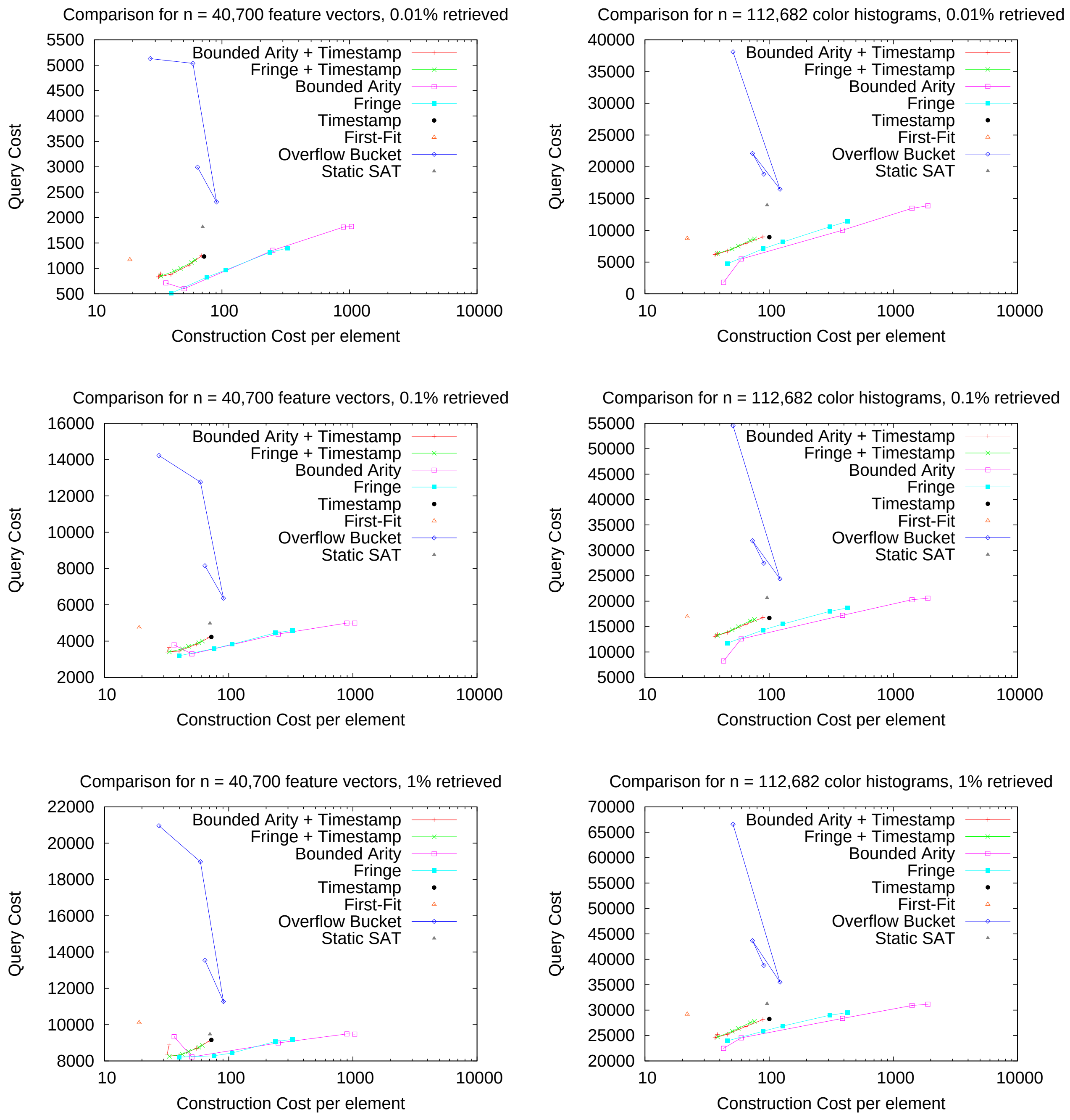


Figure 4: Comparison for all methods considered, for the space of feature vectors (left) and for the space of color histograms (right).

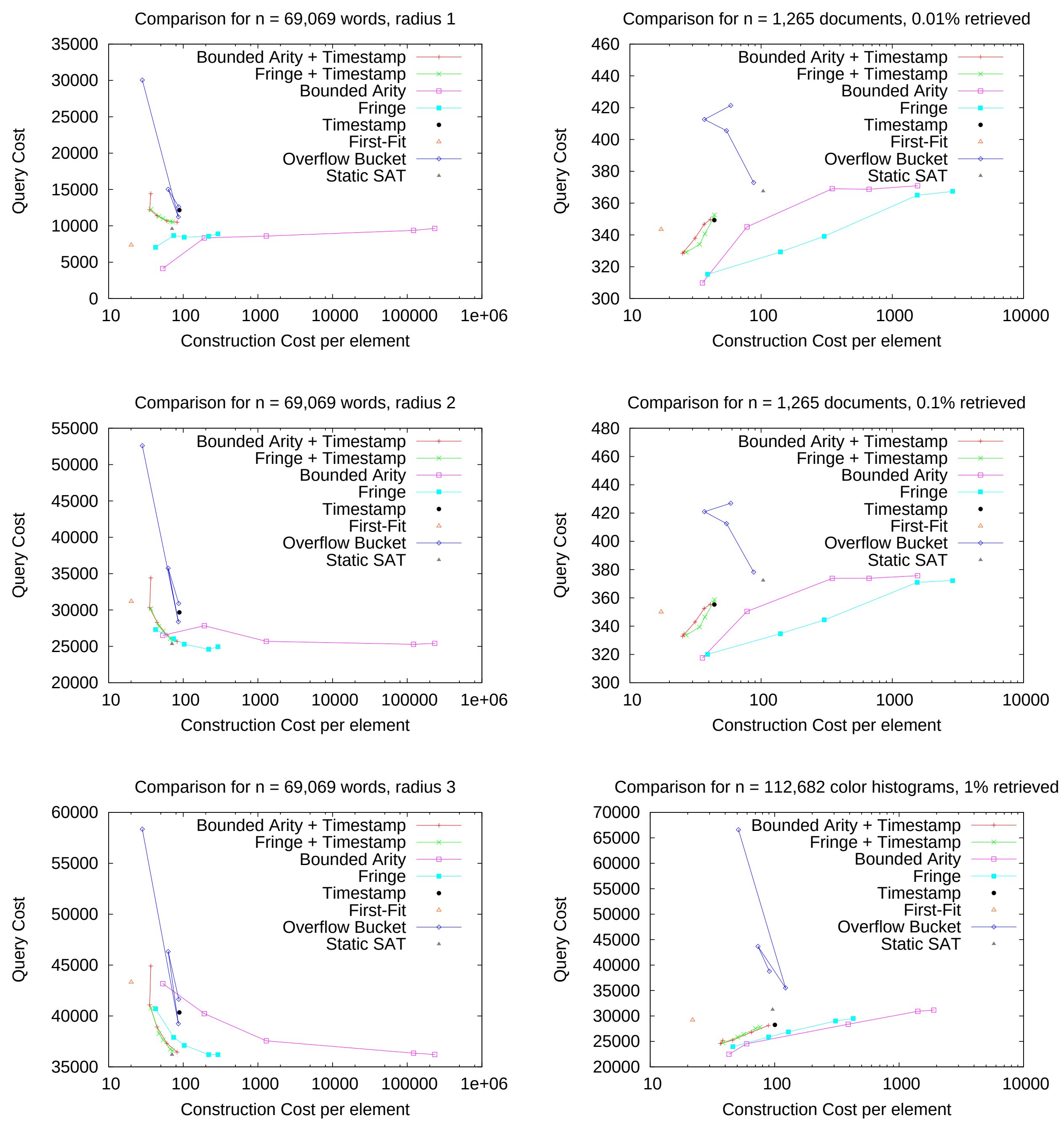


Figure 5: Comparison for all methods considered, for the dictionary (left) and for the space of documents (right).

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 dataset. This handle can contain a pointer to the corresponding tree node. Adding pointers to the parent in the tree would permit us to locate the path for free (in terms of distance computations). Henceforth we do not consider the location of the object as a part of the deletion problem, although we have shown how to proceed if necessary.

We have studied several alternatives to delete elements from a *dsa–tree*. From the beginning we have discarded the trivial option of marking the element as deleted without actually deleting it. As explained in the Introduction, this is likely to be unacceptable in most applications. We assume that the element has to be physically deleted. We 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 removed without any cost or complication, so we focus on how to remove internal tree nodes. Note, however, that most tree nodes are leaves, especially when the arity is higher. Thus, there will be a motivation to use higher arity when deletions are considered.

We present several deletion alternatives in this section. The first two disconnect the subtree of the deleted node and reinsert it (wholly or in parts) from the tree root again, with the hope of redistributing the tree better. The third choice manages to rebuild the affected subtree exactly as if x was never inserted, which guarantees the quality of the tree after successive deletions (this is not achieved by the first choices). As all the deletion costs turn out to be significant, we also give a way to amortize the cost of a reconstruction over many deletions, while maintaining a desired tree quality. The final method we present replaces the deleted element with another that occupies its place in the node, in order to avoid any rebuilding. Still, we must periodically rebuild the trees to avoid degrading query performance.

6.1 Reinserting Subtrees

A widespread idea in the Euclidean range search community is that reinserting the elements of a disk page may be beneficial because, with more elements in the tree, the space can be clustered better. We follow this principle now to obtain a method with costly deletions but good search performance.

When node x is deleted, we disconnect the subtree rooted at x from the main tree. This operation does not affect the correctness of the remaining tree, but we have now to reinsert the subtrees rooted at the nodes of $N(x)$. To do this efficiently we try to reinsert complete subtrees whenever possible.

In order to reinsert a subtree rooted at y, we follow the same steps as for inserting a fresh object y, so as to find the insertion point a. The difference is that we have to assume that y is a "fat" object with radius $R(y)$. That is, we can choose to put the whole subtree rooted at y as a new neighbor of a only if $d(y, a) + 2R(y)$ is smaller than $d(y, b)$ for any $b \in N(a)$. Similarly, we can choose to go down by neighbor $c \in N(a)$ only if $d(y, c) + 2R(y)$ is smaller than $d(y, b)$ for any $b \in N(a)$. When none of these conditions hold, we are forced to split the subtree rooted at y into its elements: one is the single element y, and the others are the subtrees rooted at $N(y)$. Once we split the subtree, we continue the insertion process with each constituent separately.

Every time we insert a node or a subtree, we pick a fresh timestamp for it. The elements inside the subtree should get fresh timestamps while keeping the relative ordering among the subtree elements. The easiest way to do this is to assume that timestamps are stored relative to those of their parent. In this way, nothing has to be done. We need, however, to store at each node the maximum differential time stored in the subtree, so as to update $CurrentTime$ appropriately when a whole subtree is reinserted. This is easily done at insertion time and omitted in the pseudocode for simplicity.

During reinsertion, we also modify the covering radii of the tree nodes a traversed. When inserting a whole subtree we have to include $d(y,a) + R(y)$, which may be larger than necessary. This involves at search time a price for having reinserted a whole subtree in one shot.

Algorithm 17 shows the algorithm to reinsert a tree with root y into a *dsa–tree* rooted at a, as well as to delete node x from the tree via subtree reinsertion.

```
**ReinsertT**(Node $a$, Node $y$)

 1. If $|N(a)| < MaxArity$ Then $M \leftarrow \{a\} \cup N(a)$    Else $M \leftarrow N(a)$
 2. $c_1 \leftarrow \mathrm{argmin}_{b \in M} d(b,y)$
 3. $c_2 \leftarrow \mathrm{argmin}_{b \in M - \{c_1\}} d(b,y)$
 4. If $d(c_1,y) + 2 \cdot R(y) \leq d(c_2,y)$ Then /* keep subtree together */
 5.    $R(a) \leftarrow \max(R(a), d(a,y) + R(y))$
 6.    If $c_1 = a$ Then /* insert it here */
 7.       $N(a) \leftarrow N(a) \cup \{y\}$
 8.       $time(y) \leftarrow CurrentTime$ /* subtree shifts automatically */
 9.    Else **ReinsertT**($c_1$, $y$) /* go down */
10. Else /* split subtree */
11.    For $z \in N(y)$ Do **ReinsertT**($a$, $z$)
12.    $N(y) \leftarrow \emptyset$, $R(y) \leftarrow 0$
13.    **ReinsertT**($a$, $y$)

**DeleteT**(Node $a$, Node $x$)

 1. $b \leftarrow parent(x)$
 2. $N(b) \leftarrow N(b) - \{x\}$
 3. For $y \in N(x)$ Do **ReinsertT**($a$,$y$)
```

Algorithm 17: Algorithm to delete x from a *dsa–tree* with root a, by reinserting subtrees.

Note that it may seem that, when searching for the place to reinsert the subtrees of a removed node x, one could save some time by starting the search at the parent of x. However, the tree has changed since the time the subtree of x was created, and new choices may exist now. So it might be that the subtree chooses a different path this time. Yet, we can make use of timestamps to take some advantage of this fact. Say that x will be deleted, and let $A(x)$ be the set of ancestors of x. When a node y was inserted into the subtree rooted at x, it was compared against all the elements in $N(A(x))$ whose timestamp was lower than that of y. Using this information we can avoid reevaluating distances to these nodes when revisiting them at the time of reinserting y. That is, when looking for the neighbor closest to y, we know that the one in $A(x)$ is closer to y than any older neighbor, so we have to consider only newer neighbors. Note that this is valid as long as we reenter the same path where c was previously inserted.

Figure 27 shows the deletion cost by reinsertion of subtrees, when we delete up to 10% of randomly chosen database elements, using different arities. The figures display a high cost and large variance, due to the fact that the deletion cost depends strongly on the subtree size of the deleted element. It is also interesting to notice that deletion costs are not monotonic with the arity. The reason is that, the higher the arity, the smaller is the subtree rooted by a random node chosen for deletion (in particular the probability of the node being a leaf increases), and thus the smaller is the number of nodes to reinsert. On the other hand, as the arity increases, the cost to reinsert each of those fewer subtree nodes increases. In all cases the optimum turned out to be arity 8.

Let us now consider how the search costs are affected by deletions. We search on an index built on half the database elements. This half is built by inserting more elements and then removing enough elements to leave 50% of the set in the index. So we compare the search on sets of the same size where a percentage of the elements has been deleted in order to leave the set in that size (recall the end of Section 2). Figure 28 compares search costs after deletions, using arity 32 for the dictionary and 4 for the other spaces. This gives a reasonable tradeoff between insertion, search, and deletion cost.

As it can be seen, the reinsertion of whole subtrees may significantly degrade the search performance. This could be argued to be a consequence of the overestimation of covering radii: If we have to reinsert a subtree rooted at y, we follow the path from the root to a node where we reinsert the whole subtree or we have to split it, and then reinsert y and every subtree rooted at $N(y)$. In every node a traversed in this path we have to update, inevitably, $R(a)$ to a value possibly greater than necessary, $d(a, y) + R(y)$. In the next section we look for a technique that gives tighter covering radii.

6.2 Reinserting Elements

In an attempt to reduce covering radii, we explore in this section the idea of reinserting all the subtree rooted at y element-wise. This will increase deletion cost but could improve search costs. The new deletion algorithm is shown in Algorithm 18.

Figure 29 shows the deletion cost by reinsertion of elements, when we delete up to 10% of random database elements, using different arities. Figure 30 compares search costs after deletions.

As it can be seen, deletion costs have increased slightly as expected, but we have not solved the degradation problem. A possible reason is that, even when reinserting element-wise, there is still an overestimation of covering radii due to the fact that no covering radius is reduced after deletion of x. That is, if x was farthest to its ancestor b among all the elements rooted at b, then $R(b)$ should be reduced when x is removed. This is not done because it is too expensive.

We have repeated the experiments by artificially recomputing the tight values of all covering radii after the deletions, and the results vary very little. This indicates that the problem is not the covering radii. There must be a more complex explanation for the degradation of a *dsa–tree* after successive deletions. Note that it is necessary to sort out this problem in order to have a data structure that can handle datasets for long periods of time upon insertions, deletions and searches.

We conjecture that the reason is of a geometric nature. In a *dsa–tree*, each subtree handles the points closer to it than to other subtrees. This is a kind of Voronoi partitioning of the space, where each subtree root acts as the center of the area. When one removes one such subtree, its elements might be inserted elsewhere, since the partitioning at higher levels of the tree may have

```
ReinsertE(Node $a$, Node $y$)

 1. If $|N(a)| < MaxArity$ Then $M \leftarrow \{a\} \cup N(a)$ Else $M \leftarrow N(a)$
 2. $N \leftarrow N(y)$ /* keep neighbors of $y$ */
 3. $N(y) \leftarrow \emptyset$, $R(y) \leftarrow 0$
 4. $c_1 \leftarrow \text{argmin}_{b \in M} d(b, y)$
 5. $R(a) \leftarrow \max\{R(a), d(a, y)\}$
 6. If $c_1 = a$ Then /* reinsert here */
 7.     $N(a) \leftarrow N(a) \cup \{y\}$
 8.     $time(y) \leftarrow CurrentTime$
 9. Else ReinsertE($c_1$, $y$) /* go down */
10. For $z \in N$ Do ReinsertE($a$, $z$)

DeleteE(Node $a$, Node $x$)

 1. $b \leftarrow parent(x)$
 2. $N(b) \leftarrow N(b) - \{x\}$
 3. For $y \in N(x)$ Do ReinsertE($a$,$y$)
```

Algorithm 18: Algorithm to delete x from a *dsa–tree* with root a, by reinserting elements.

changed. In this case, the emptied area is covered by other neighbors of x, which however have no elements in there. This reduces the accurateness of the search because those neighbors that cover those empty areas receive many useless searches for those areas. Alternatively, imagine that the elements in the subtree of x do fall again in the same subtree. They will be appended at the end of the neighbor list, losing their original place and producing a rightward asymmetry (i.e., towards younger neighbors). This means that any search will give priority to the first neighbors (which still are covering the emptied areas) and then, additionally, will enter the newer neighbors that actually contain the elements of that area. In a sense, x was acting as a stopper that prevented searches from unnecessarily entering its younger neighbors. After removing such a stopper, searches are more expensive.

6.3 Rebuilding Subtrees

Irrespective of whether the above conjecture holds, it seems clear that we should find a deletion method that does not degrade searches. The best way to ensure that is to ensure that the tree resulting from the deletion of x is exactly as if x had never been inserted. This is what we do in this section.

When node $x \in N(b)$ is deleted, we disconnect x from the main tree. Hence all its descendants must be reinserted. Moreover, elements in the subtree of b that are younger than x have been compared against x to determine 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 than x that descend from b (that is, those whose timestamp is greater, which includes the descendants of x) 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 fresh elements, that is, if they get new timestamps, then we must search for the appropriate reinsertion point beginning at the tree root. On the other hand, if we maintain their original timestamp, then we can begin reinsertion from b and save many comparisons. The reason is that we are reinserting them as if the current time was that of their original insertion, when all the newer choices that appeared later did not exist, and hence those elements should make the same choice as at that moment, arriving again at b. In order to leave the resulting tree exactly as if x had never been inserted, we must reinsert the elements in the original order, that is, in increasing order of their timestamps.

Therefore, when node $x \in N(b)$ is deleted, we retrieve all the elements younger than x from the subtree rooted at b, disconnect them from the tree, sort them in increasing order of timestamp, and reinsert them one by one, searching for their reinsertion point from b. Algorithm 19 shows the deletion algorithm.

```
DeleteR(Node $x$)

 1. $b \leftarrow parent(x)$
 2. Collect in $S$ the elements of the subtree rooted at $b$, younger than $x$
 3. Sort $S$ by increasing timestamps
 4. $N(b) \leftarrow N(b) - \{x\}$
 5. For $y \in S$ Do InsertTBA($b$,$y$) /* without changing its timestamp */
```

Algorithm 19: Algorithm to delete x from a *dsa–tree*, by rebuilding subtrees.

We make two optimizations to rebuilding subtrees. Say that x will be deleted from the subtree rooted at node b (that is $x \in N(b)$). The first optimization makes a more clever use of timestamps. We observe that there are some elements younger than x that will not change their insertion point when we reinsert them into the subtree rooted at b. These elements are those older than the first child of x and also than the last sibling of x. For those elements we can avoid computing their new insertion point. To see this, note that we refer to the first nodes inserted after x. Those nodes had already the choice of entering x, but they chose otherwise (as they came before the element that chose to be the first child of x). All those nodes have, at their reinsertion point, exactly the same options they had at their insertion time except for x, which was not preferred anyway. Thus they will choose the same again. The only possible exception is that, because of the bounded arity, they had been forced to enter some neighbor although they would have preferred to become a new neighbor of b. Now, the absence of x leaves them space to become a new neighbor of b. This is why we can ensure the property only until the insertion time of the last sibling of x.

A second optimization is similar to the one made in Section 6.1, that is, we know that the elements in $A(y)$ are closest to y than any older neighbor, so we need to compare y only against newer neighbors (as long as we repeat the same insertion path).

Figure 31 shows the deletion cost by rebuilding subtrees, when we delete up to 10% of random database elements, using different arities. As it can be seen, rebuilding subtrees is considerably more expensive than reinserting elements. The reason is that we reinsert not only the subtree of x but also all the younger descendants of its parent. The fact that we are reinserting from the parent of x and not from the tree root is not enough to counterweight the larger number of elements reinserted. We note that this time the best results are obtained with arity 4.

The reward comes at search time. As it can be seen in Figure 32, the search quality stays the same no matter how many deletions we make. This happens even when, under this deletion method, the covering radii can still become overestimated, because they are never reduced due to a deleted element. This confirms that overestimation is not really an issue[4].

6.4 Using Fake Nodes

We observe that the deletion costs obtained in the previous sections are rather high compared to insertion costs, as we have to rebuild whole subtrees. In this section we show how to amortize this cost over many deletions.

An alternative to delete an element x is to leave its node in the tree (without content) and mark it as deleted. Such a node is said to be *fake*. Although cheap and simple at deletion time, we must now figure out how to carry out a consistent search when some nodes do not contain any object.

Basically, if node $b \in N(a)$ is fake, we do not have enough information to avoid entering the subtree of b once we have reached a. So we cannot include b in the minimization and always have to enter its subtree (except if we can use the timestamp information of b to prune the search).

The search performed at insertion time, on the other hand, has to follow just one path in the tree. In this case, one is free to choose inserting the new element into any fake neighbor of the current node, or into the closest non-fake neighbor. A good policy is, however, trying not to increase the size of subtrees rooted at fake nodes, as they will have to be eventually rebuilt, and also because they are entered more frequently during searches.

Hence, although deletion is simple, the performance of the search process degrades. Therefore, we must periodically get rid of fake nodes and actually delete them. Note that the cost of rebuilding a subtree would not be much different if it contained many fake nodes, so we could remove all the fake nodes with a single reconstruction, therefore amortizing the high reconstruction cost over many deletions.

Our idea is to ensure that every subtree has at most a fraction α of fake nodes. We say that such subtrees are "dense", otherwise they are "sparse". When we mark a new node $x \in N(a)$ as fake, we check if we have not made its subtree sparse. In this case, x is actually deleted from the tree. In the process of reinserting elements, we also discard every other fake node we find.

This technique has a nice performance property. If the number of elements to reinsert is m, this is because αm of them are fake and we will only reinsert $(1-\alpha)m$ real nodes. Therefore, we only perform $(1-\alpha)m$ reinsertions for each group of αm deletions that have occurred. The reinsertions get rid of those αm fake nodes, so we are actually paying an amortized deletion cost which is $(1-\alpha)/\alpha$ times the cost of an insertion. Asymptotically, the tree works as if we permanently had a fraction α of fake nodes. Hence, we can control the tradeoff between deletion and search cost.

A small complication of this scheme is that deleting x may make sparse several ancestors of x, even if x is just a leaf that can be directly removed, and even if the ancestor is not rooted at a fake node. As an example, consider a unary tree of height $3n$ where all the nodes at distance $3i$ from the root, $i \geq 0$, are fake. The tree is dense for $\alpha = 1/3$, but removing the leaf or marking it as fake makes every node sparse.

We solve this problem incrementally. Upon marking x as fake, we follow the path from x upwards to the tree root, checking at each node a whether it becomes sparse or not. If we find a node a that becomes sparse, we rebuild the whole subtree from the parent of a, and continue checking upwards. Rebuilding a lower subtree makes it more probable that higher subtrees become dense again, at a lower cost compared to rebuilding the highest sparse subtree directly.

[4]The documents exhibit a strange behavior in this case, as the searches perform slightly *better* after 20% of deletions. Actually, the covering radii happen to be tighter than with other deletion percentages. We attribute this to variance, as the searched sets are different in all cases (although they are of the same size).

Figures 33 and 34 show deletion costs combining reinsertion of elements and rebuilding of subtrees, respectively, with fake nodes for different values of α. We maintain arity 4 for all spaces except the dictionary, which uses arity 32.

It can be seen that the deletion costs are largely reduced by using even moderate α values. For example, the average insertion cost in the space of color histograms is about 37 distance computations per element. Each deletion using reinsertion of elements costs about 250 distance computations, that is, almost 7 times the cost of an insertion. The combined method largely improves upon this: using α as low as 1% we have a deletion cost of 63 distance computations, and with α=3% this reduces to 37, the same cost of an insertion. If we consider rebuilding of subtrees, each deletion costs 800 distance computations, more than 20 times the cost of an insertion. By using α as low as 1% we have a deletion cost of 128 distance computations, and with α=3% this reduces to 72.

Figures 35 and 36 show the results of searching an index built on half the dataset, combining reinsertion of elements with fake nodes, deleting 10% and 40% of elements. As it can be seen, the search quality degrades as α grows. With 10% deleted the degradation is not so significant, but for 40% it is very noticeable. The reason is that, as α grows, the search needs to enter all the children of fake nodes. The degradation is noticeable even for α=1%, except on the dictionary. The worst space in this respect is that of the documents[5].

Figures 37 and 38 show the same results, combining rebuilding of subtrees with fake nodes. As expected, for a given α value, we obtain better search time, albeit we paid a higher deletion cost.

6.5 Ghost Hyperplanes

Our final technique is inspired on an idea presented in [UN03] for dynamic *gna–trees* [Bri95], called *ghost hyperplanes*. This method replaces the deleted element by a leaf, which is easy to delete. This way rebuilding is not necessary, but in exchange some tolerance must be exercised when entering the replaced node at search time.

Remind that the neighbors of a node b in the *sa–tree* 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. Algorithm 20 shows the pseudocode of the modified search algorithm.

[5]We note that the behavior in this case is not monotonic on α. These fluctuations are possible because the subtrees are not rebuilt over increasing subsets, and because the final searches are done over different subsets, even if they all have the same size.

```
**RangeSearchTBA-GH**(Node $a$, Query $q$, Radius $r$, Timestamp $t$)

1. If $time(a) < t \ \wedge \ d(a,q) - d_g(a) \leq R(a) + r$ Then
2.   If $d(a,q) \leq r$ Then Report $a$
3.   $d_{min} \leftarrow \infty$
4.   For $b_i \in N(a)$ Do   /* in ascending timestamp order */
5.     If $d(b_i,q) - d_g(b_i) \leq d_{min} + 2r$ Then
6.       $t' \leftarrow \min\{t\} \cup \{time(b_j), j > i \wedge d(b_i,q) - d_g(b_i) > d(b_j,q) + d_g(b_j) + 2r\}$
7.       **RangeSearchTBA-GH**($b_i$,$q$,$r$,$t'$)
8.       $d_{min} \leftarrow \min\{d_{min}, d(b_i,q) + d_g(b_i)\}$
```

Algorithm 20: Modified search algorithm for q with radius r in a *dsa–tree* rooted at a, so that ghost hyperplanes are considered.

When node x is deleted, we look for a substitute in its subtree to ensure that we reduce the problem size. In [UN03] they choose a leaf of the subtree by descending always to the child that is closest to x. Although this does not guarantee that y is the leaf closest to x, performing a true nearest-neighbor query in the subtree is argued to be too expensive. We consider this alternative of choosing the replacement among the leaves with the same policy. The *sa–tree*, however, has an interesting advantage over the *gna–tree* in the sense that the neighbors (i.e., children) of a node are chosen to be close to it, while in the *gna–tree* they are random or chosen to be far apart from each other [Bri95]. In a *sa–tree*, choosing the replacement among the neighbors of the deleted element could give a good candidate for replacement at very low cost. We explain now both methods in detail.

Choosing a leaf substitute: We descend in the subtree of x by the children closest to x all the time. When we reach a leaf y, we disconnect y from the tree and put y into the node of x, retaining the original timestamp of x. Then we update the d_g value of the node. Algorithm 21 depicts the algorithm.

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 we delete it and finish. Otherwise we recursively continue the process at that node. So, we turn to *ghost* all the nodes in a path from x to a leaf of its subtree, following closest neighbors. In exchange, the $d_g()$ values should be smaller. Algorithm 22 shows this deletion algorithm.

Choosing the nearest-element substitute: We select y as the nearest 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. So, we turn to *ghost* some nodes in a 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.

Algorithm 23 illustrates the algorithm. `NNsearch(`x`,`x`,1)` invokes the algorithm to perform a 1-NN search for query x in the subtree of x. Given the *dsa–tree* version we use, the algorithm corresponds to `NNsearchTBA`, described at the end of Section 5.7 as a modification of the algorithm in Algorithm 11.

```
**DeleteGH1**(Node $x$)

1. $b \leftarrow parent(x)$
2. If $N(x) \neq \emptyset$ Then
3.    $y \leftarrow$ **FindSubstituteLeaf** ($x$)
4.    $d_g(x) \leftarrow d_g(x) + d(x,y)$
5.    Copy object of $y$ into node $x$
6. Else $N(b) \leftarrow N(b) - \{x\}$
```

```
**FindSubstituteLeaf**(Node $x$): Node

1. $y \leftarrow x$
2. While $N(y) \neq \emptyset$ Do
3.    $x \leftarrow y$
4.    $y \leftarrow \text{argmin}_{c \in N(b)} d(c,x)$
5. $N(x) \leftarrow N(x) - \{y\}$
6. Return $y$
```

Algorithm 21: Algorithm to delete x from a *dsa–tree*, using ghost hyperplanes, and finding a substitute for x among the leaves of its subtree.

```
**DeleteGH2**(Node $x$)

1. $b \leftarrow parent(x)$
2. If $N(x) \neq \emptyset$ Then
3.    $y \leftarrow \text{argmin}_{c \in N(x)} d(c,x)$
4.    $d_g(x) \leftarrow d_g(x) + d(x,y)$
5.    Copy object of $y$ into node $x$
6.    **DeleteGH2** ($y$)
7. Else $N(b) \leftarrow N(b) - \{x\}$
```

Algorithm 22: Algorithm to delete x from a *dsa–tree*, using ghost hyperplanes, and choosing its replacement among its neighbors.

Figure 39 shows the deletion cost by ghost hyperplanes replacing by a leaf, when we delete up to 10% of random database elements, using different arities. Figure 40 shows the same, replacing by a neighbor. Figure 41 depicts also the same but substituting by the nearest element in its subtree. In all cases the deletion costs are very low, comparable with using fake nodes. Similarly, we expect successive deletions to degrade the quality of the trees.

Figures 42, 43, and 44 compare search costs after deletions, using arity 32 for the dictionary and 4 for the other spaces, for all the replacement options. As it can be seen, the search quality degrades almost as fast as with fake nodes. That is, even when we now have an element in the place of the deleted node which permits us not entering into its subtree at every search, the tolerance

```
**DeleteGH3**(Node $x$)

1. $b \leftarrow parent(x)$
2. If $N(x) \neq \emptyset$ Then
3.    $y \leftarrow$ **NNsearch**($x$,$x$,1)
4.    $d_g(x) \leftarrow d_g(x) + d(x,y)$
5.    Copy object of $y$ into node $x$
6.    **DeleteGH3** ($y$)
7. Else $N(b) \leftarrow N(b) - \{x\}$
```

Algorithm 23: Algorithm to delete x from a *dsa–tree*, using ghost hyperplanes, and choosing its replacement as its nearest element in its subtree.

introduced by $d_g(x)$ is also a significant factor in worsening the search quality.

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, when we rebuild the subtree we get rid of all the ghost hyperplanes that are inside it. Therefore, we can apply exactly the same mechanism used in Section 6.4 to control the amount of fake nodes. We set a maximum allowable proportion α of ghost hyperplanes, and rebuild the tree when this limit is exceeded.

Figures 45, 46, and 47 show deletion costs with all the replacement options. Figures 48 to 53 show the corresponding search costs, for 10% and 40% deleted elements. It can be seen that, using some intermediate α values, we can obtain a reasonable tradeoff between deletion and search time. We can also see that the alternative of replacing the deleted node by a neighbor performs slightly worse than the others. This is probably caused by the higher number of ghost hyperplanes introduced.

6.6 Choosing the Best Deletion Method

Figures 6 and 7 help illustrate the tradeoff (by varying α). In each case we show search versus deletion costs, when deleting 10% or 40% of the database. For brevity, we have included only one representative search radius per space. The point labeled "Insertion vs. Search" shows the insertion cost combined with the search cost when no deletions have occurred.

The best deletion method is different for each space, but most of them perform quite similarly. Among them, GH1 and GH3 are the ones that perform consistently well.

The beauty of the methods using α is that they permit controlling the expected deletion cost as a proportion of the insertion cost. For example, we could state that our deletion cost should be similar to the insertion cost. In this case, using the points "Insertion vs. Search" one can see that the search costs would be just 3%–13% higher after 10% of deletions (compared to no deletions at all), and 8%–23% after 40% of deletions.

7 Comparison with Previous Work

As explained in Section 3, only a few data structures provide full support for insertions, and even fewer support deletions. In the previous sections we have studied several alternatives to give the *dsa–tree* these insertion and deletion capabilities.

In order to evaluate how our *dsa–tree* compares to previous work in terms of distance evaluations for construction and searching, we have chosen a set of good representative data structures. Those include data structures that are actually dynamic, as well as those that can presumably be made dynamic with reasonable effort. In particular, we have included the *M–tree* as a dynamic data structure (yet not supporting deletions), because it is an important referent in the literature, even if the *M–tree* is in fact designed for secondary memory. Construction is made by successive insertions for the *M–tree* and *dsa–tree*, and therefore construction cost serves to compare insertion performance. The other structures are actually static, and thus their construction cost displays the best achievable insertion cost if the structures were made dynamic.

Our experiments show that the *dsa–tree* stands out as a practical and efficient dynamic data structure for metric space searching, being very competitive against existing alternatives.

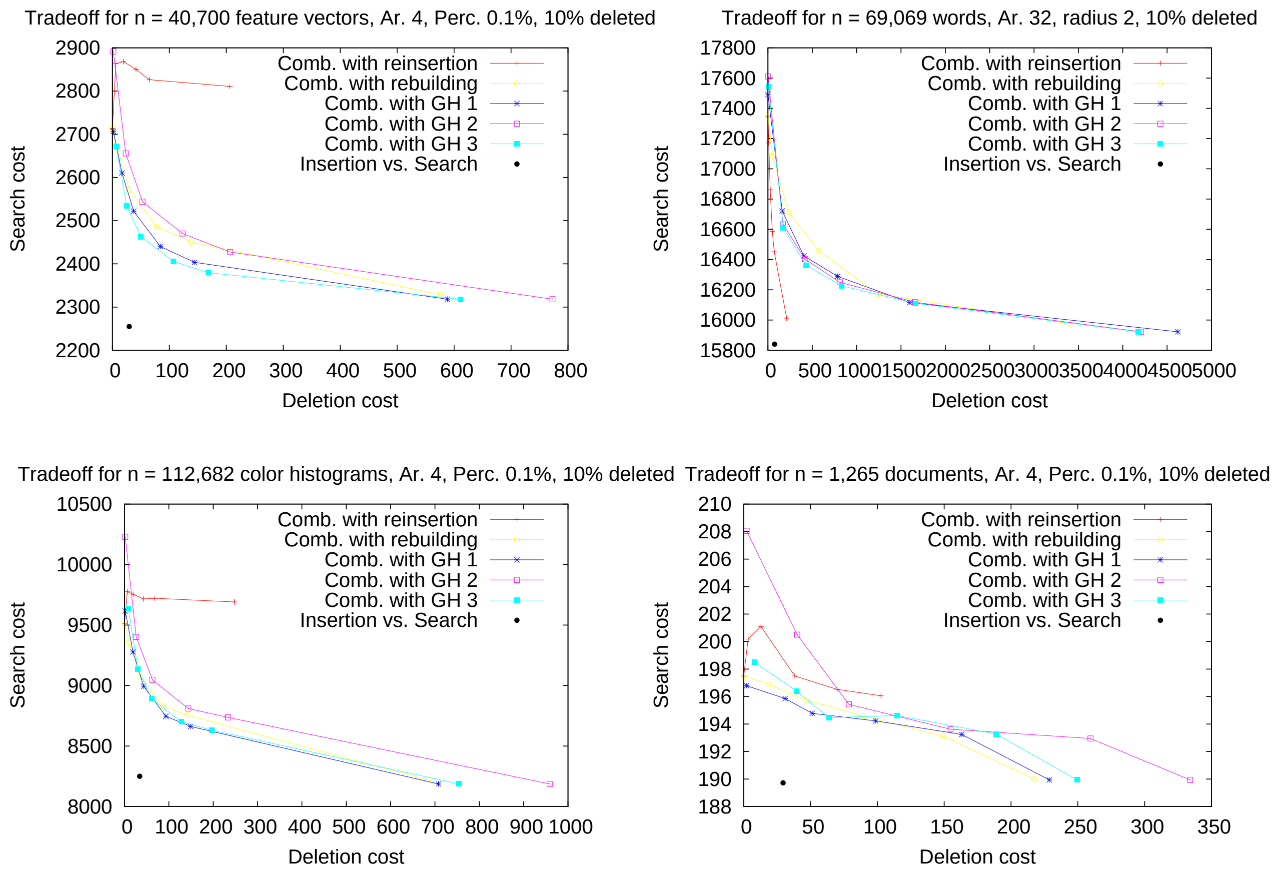


Figure 6: Tradeoffs for all the deletion methods proposed, when deleting 10% and retrieving 0.1% of database, or with radius 2 for strings.

7.1 M–tree

The *M–tree* [CPZ97] is probably the best-known existing dynamic data structure, and a baseline most of the newer developments compare with. A practical advantage of the *M–tree* is that its code is available online[6]. Another prominent alternative is the *D–index*, yet this has already been shown to perform similar to the *M–tree* [Doh04, DGSZ03].

The *M–tree* also performs well in secondary memory, although in this paper we are only interested in the number of distance evaluations. We have used the parameter setting suggested by the authors [CPZ97]. We have also checked several other parameterizations to make sure that the suggested values were indeed the best choices when considering number of distance evaluations at search time (it is possible to reduce construction costs by increasing search costs, but we chose to give more importance to searches). We do not compare deletion costs because the available version of the *M–tree* does not support deletions.

To show how the *dsa–tree* compares against its original static version, we have also included the *sa–tree* in the experiments. For the *dsa–tree* we have used arity 32 for the space of strings and 4 for the others, as this gives a good tradeoff between insertion, deletion, and search times.

[6]At `http://www-db.deis.unibo.it/research/Mtree/`

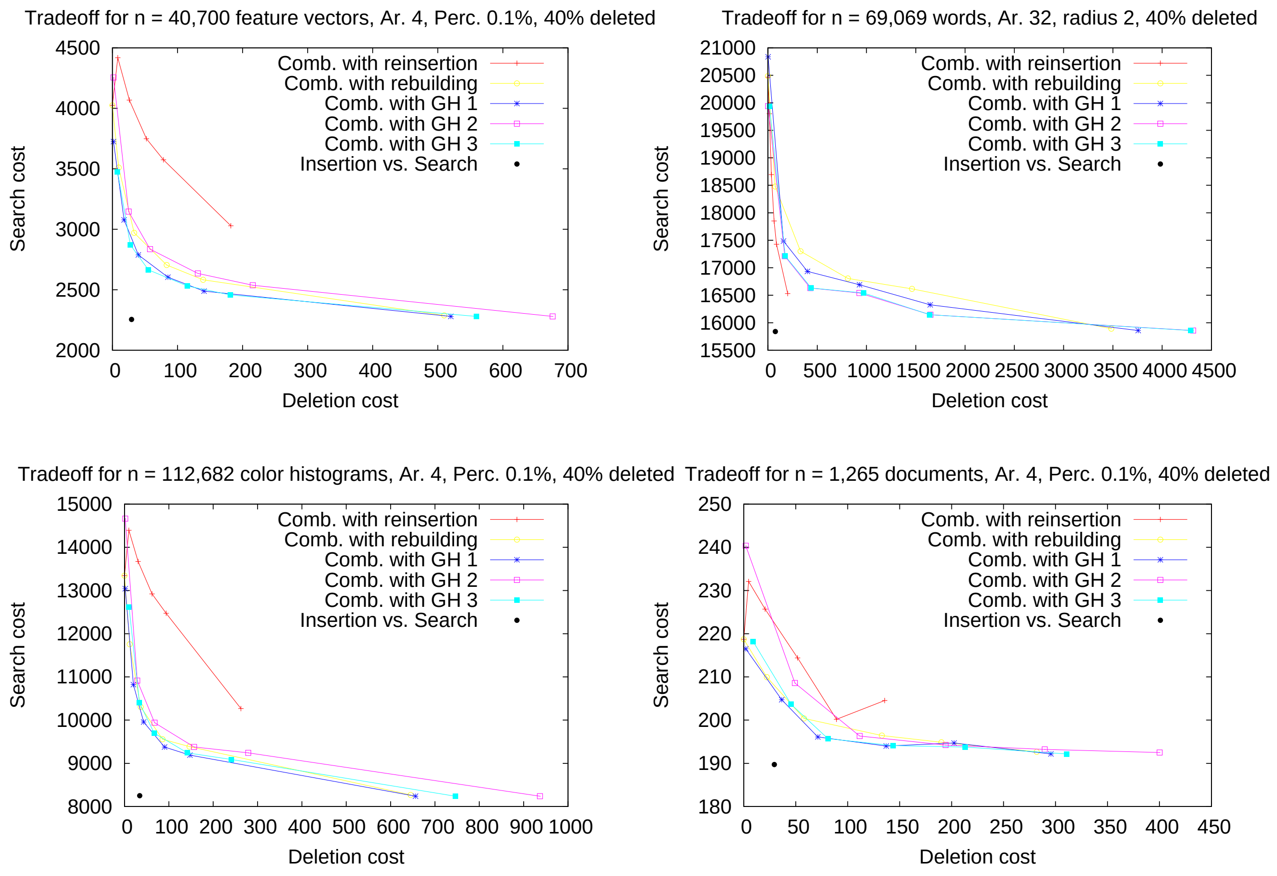


Figure 7: Tradeoffs for all the deletion methods proposed, when deleting 40% and retrieving 0.1% of database, or with radius 2 for strings.

Figure 8 shows the comparison of the construction costs over the four metric spaces, and Figure 9 depicts the results of the search experiments.

As it can be seen, our *dsa–tree* requires up to 4 times fewer distance evaluations than both alternatives for construction. If we consider the search performance, we have that the *dsa–tree* outperforms the *M–tree* in three of the considered metric spaces, reaching up to 3 times fewer distance evaluations. The only space where the *M–tree* is superior is that of the documents, yet it achieves only 5% fewer distance computations.

Another practical advantage of the *dsa–tree* over the *M–tree* is the number and types of parameters to be tuned. In the case of the *dsa–tree* this consists only of maximum arity allowed, whereas the parameterization is not trivial on the *M–tree*.

7.2 Pivots

A large number of metric space methods is based on pivots. We compare our *dsa–tree* against a generic pivoting algorithm. Pivot algorithms can improve their performance by using a (possibly impractical) amount of memory. In this section we compare the performance of the basic pivot algorithm when using s times the amount of memory used by the *dsa–tree*. We note that the pivot

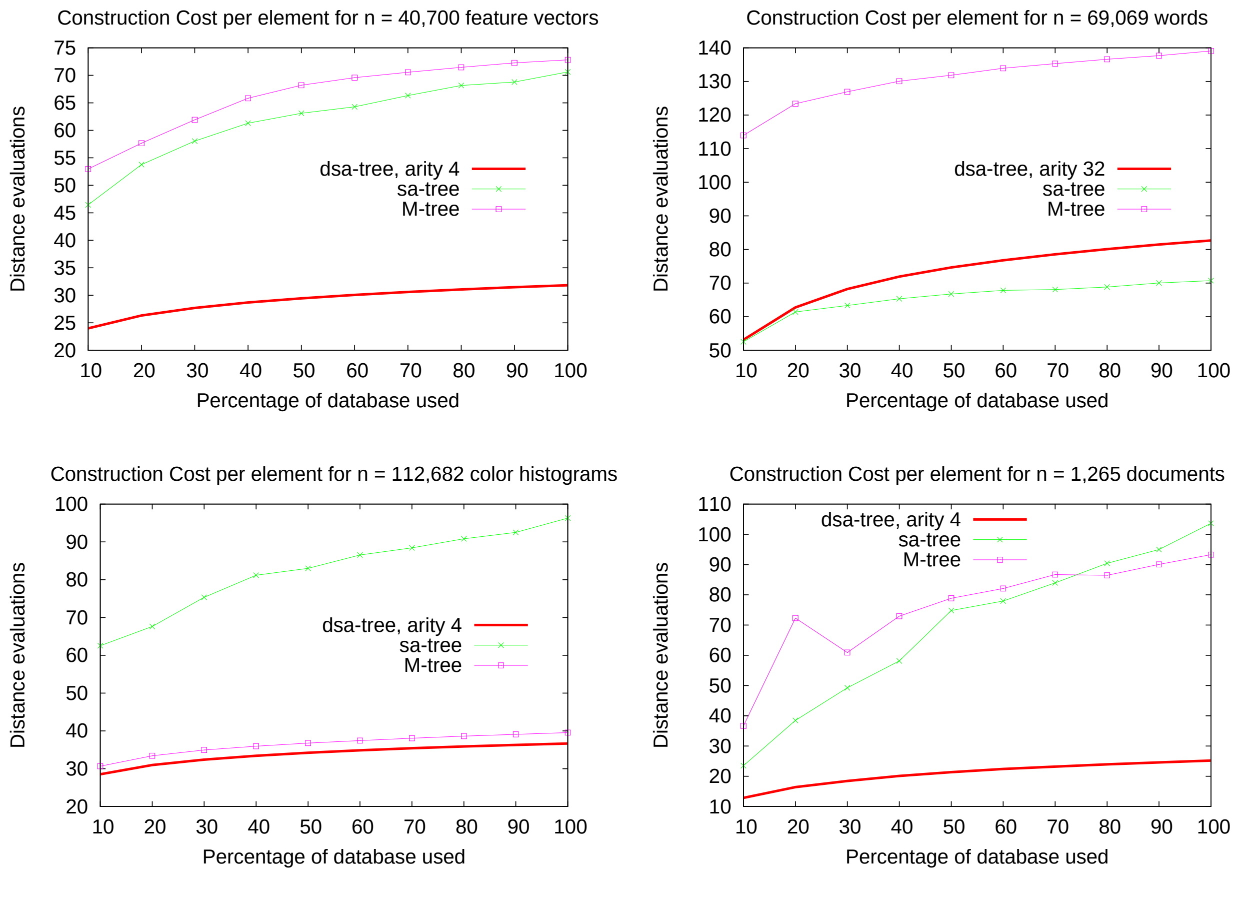


Figure 8: Comparison of construction costs against the *M–tree*.

algorithm we consider is not dynamic, yet there exist variants that could be made dynamic more or less straightforwardly [CMBY99]. Our generic pivot algorithm chooses k pivots at random. We assume that only the space to store the kn distances to the pivots is necessary. This implies a linear amount of extra CPU time at searching, but there exist practical alternatives to reduce the extra CPU time without significantly increasing the space [CMN01].

In a compact implementation of our data structure we could have in each node an array with its neighbors (not pointers to the neighbors but their records would be physically placed in the array), so we need a pointer to the array and the number of neighbors (the number of bits for the latter is limited by the logarithm of the maximum arity allowed). Therefore, we need 32 bits for the array pointer and 2 or 5 bits more for the number of neighbors (depending on the arity used); 1 byte is enough for most practical arities (up to 256). Also, we store the covering radius (32 bits suffices when distances are represented by a float or an integer) and the timestamp (32 bits is more than enough). Besides, leaves are distinguished for having zero neighbors, and they do not need to store the covering radius nor the neighbor array. It is easy to arrange leaves and non-leaves sharing the same neighbor array despite their different sizes, for example by putting all the leaves at the end and putting the "number of neighbors" field in the beginning of the record; a sequential scan can distinguish the place where leaves begin in the array. This slightly increases CPU times for insertions and deletions, but not significantly. Overall, according to our conservative computation,

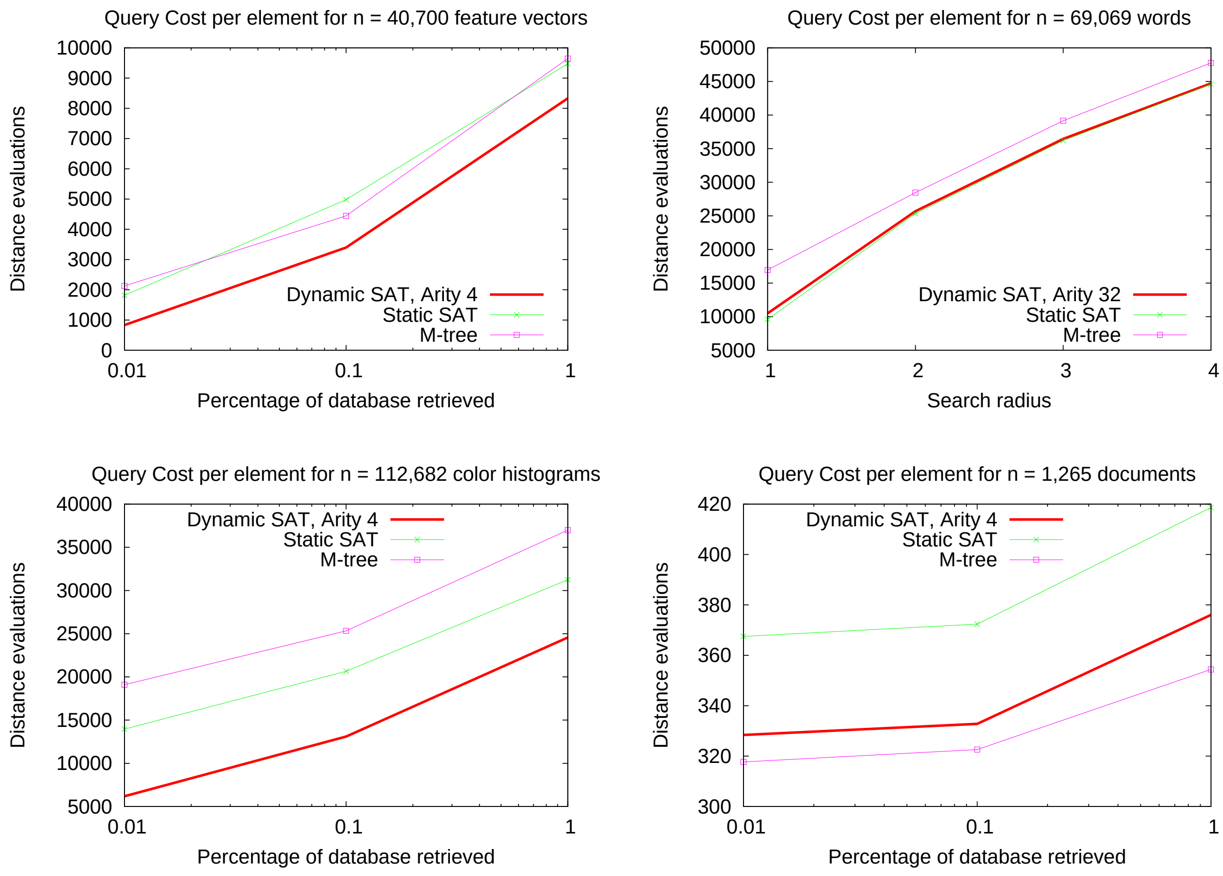


Figure 9: Comparison of search costs against the *M–tree*.

leaves require 5 bytes and internal nodes 13 bytes. We do not consider the metric space objects themselves, as they have to be stored in every kind of index.

The average percentage of leaves in our trees is above 57% in all the metric spaces used in the experiments. Hence, the space needed to store our *dsa–tree* is 69 bits per element. On the other hand, as 32 bits are needed to represent a distance, the minimal space to use k pivots is $32kn$ bits. In the sequel, $Pivot(s)$ is equivalent to using $k = 2s$ pivots, as this is a good approximation to using s times the amount of memory used by the *dsa–tree*.

Figure 10 compares the search costs of the *dsa–tree* and the generic pivot algorithm, considering values of s from 1 to 32. As it can be seen, if we bound the number of pivots at the same space needed for the *dsa–tree*, our data structure is always better. In order to outperform the *dsa–tree* for all considered radii, the pivot algorithm needs to use much more space, namely 8 times for feature vectors and the dictionary, 2 times for color histograms, and 4 times for documents. Besides, the *dsa–tree* tolerates large radii better than pivots.

7.3 List of Clusters

List of Clusters is another good example of clustering-based data structures [CN05]. The construction process chooses an element p and finds the m closest elements in S. This is the cluster of p.

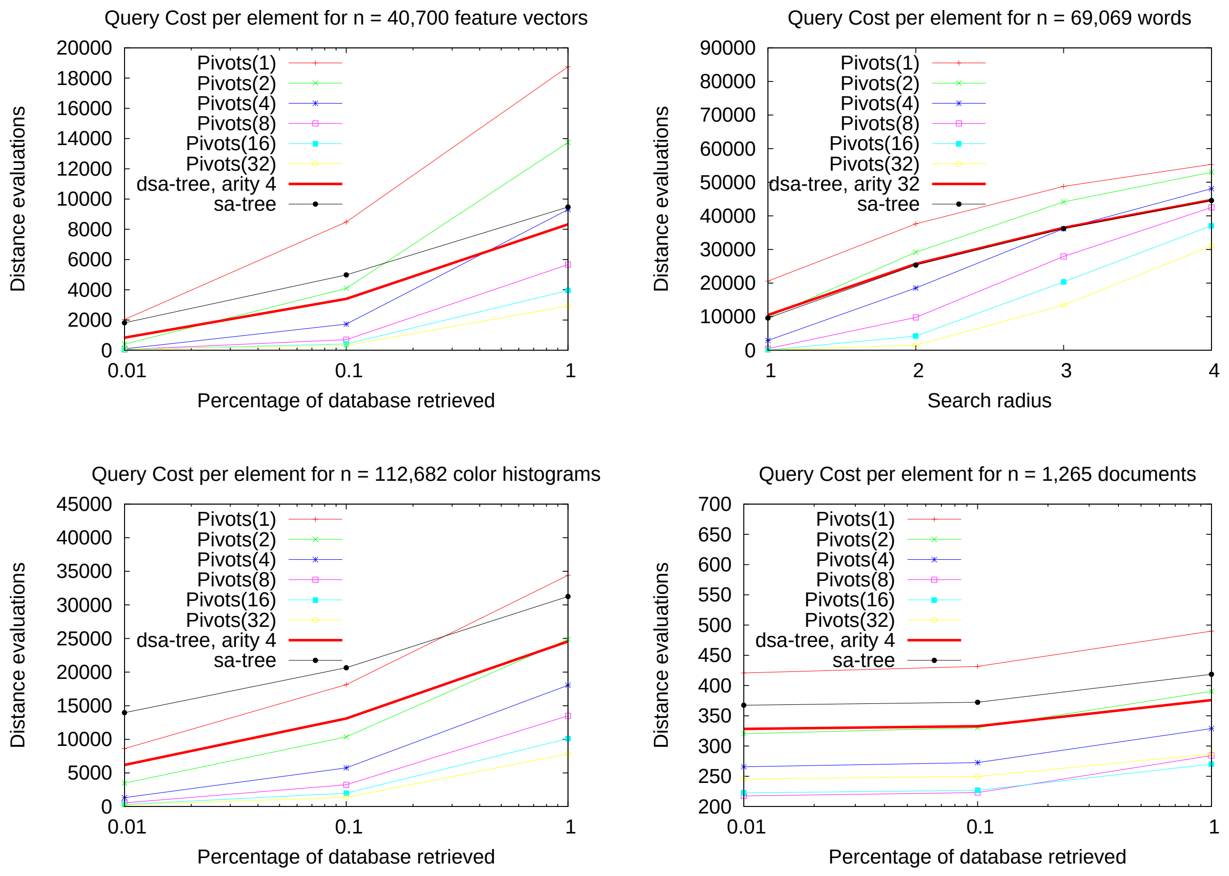


Figure 10: Comparison of search costs against a generic pivot algorithm, giving the pivots s times the space needed for the *dsa–tree*.

The process continues recursively with the remaining elements until a list of $n/(m+1)$ clusters is obtained. The covering radius $cr()$ of each cluster is stored. At search time the clusters are inspected one by one. If $d(q,p_i) - r > cr(p_i)$ we do not enter the cluster of p_i, otherwise we verify it exhaustively. If $d(q,p_i) + r < cr(p_i)$ we do not need to consider subsequent clusters. The authors report unbeaten performance on "difficult" spaces, at the cost of $O(n^2/m)$ construction time. The space required is linear in n.

Although *List of Clusters* is not a dynamic data structure, some hints to make it dynamic (with rather high insertion/deletion costs, $O(n/m)$) are given in [CN05]. This is a structure that can obtain very good search times at the price of a very high construction (and update) cost.

To make a fair comparison against the *dsa–tree* we consider the construction time required. We test different values of m ($LC(m)$), so as to obtain either similar construction time or similar search time compared with the *dsa–tree*.

Figure 11 compares the search costs of the *dsa–tree* and *List of Clusters*. In the dictionary, with a cluster size of 836 we obtain the same construction cost of *dsa–tree*, but our data structure beats *Lists of Clusters* on search radii 3 and 4. In order to beat the *dsa–tree* in all the radii considered, *List of Clusters* needs 4 times our construction cost, that is, a cluster size of 200. In the space of feature vectors, for similar construction time (achieved with cluster size 1279) our *dsa–tree* significantly outperforms *List of Clusters*; *Lists of clusters* needs 25 times the construction cost of the *dsa–tree* to outperform it (using cluster size 50). In the space of color histograms, *List of Clusters* (using cluster size 3073) obtains the same construction cost of the *dsa–tree*, but its search performance is significantly worse than ours. Only if we use cluster size 100, requiring 30 times the construction cost of the *dsa–tree*, *List of Clusters* achieves scarcely better search performance in all radii. For the space of documents, with cluster size 51 *List of Clusters* obtains similar construction cost as the *dsa-tree*, but the latter obtains better search costs. In this case we could not find a cluster size that allows *List of Clusters* outperform the *dsa-tree*, even if we disregard construction costs.

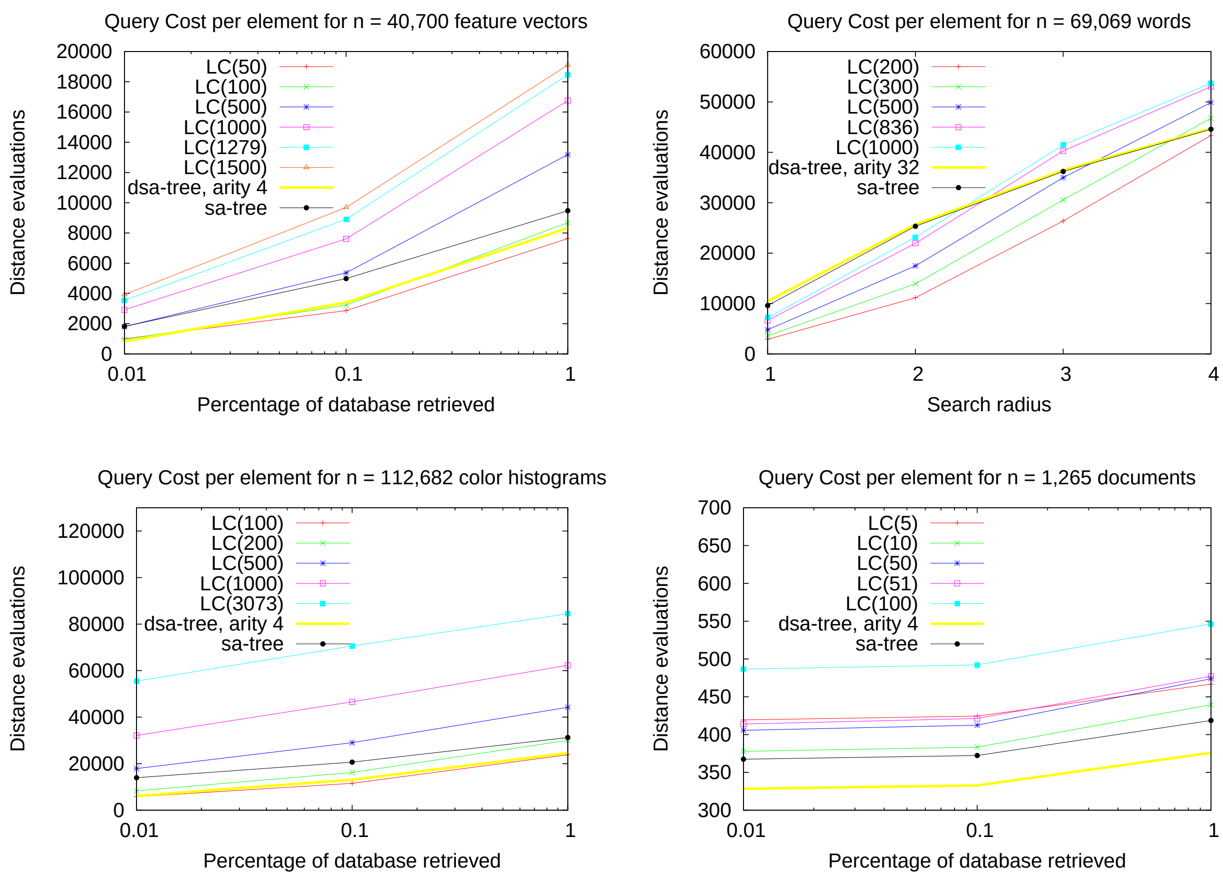


Figure 11: Comparison of search costs against *List of Clusters* considering different cluster sizes.

Thus, *dsa–trees* provide a better tradeoff between efficiency and construction cost than *List of Clusters*. It is necessary to pay much more construction time to beat *dsa–trees*, although in some cases this is not enough.

7.4 C–Tree

Another dynamic and balanced data structure is the *C–tree* [Ver95]. The *C–tree* is a clustering-based data structure inheriting from the *Monotonous Bisector tree* [NVZ92]. The *C–tree* supports insertion and deletion of elements, but the code is not available.

Therefore, in order to give an idea of the comparison between our *dsa–tree* and the *C–tree*, we select a simplified version of it, that is, we compare the *dsa–tree* against the *Bisector tree* (*bs–tree*). However, we strengthen the *bs–tree* with the hyperplane criterion (used by a close relative, the *gh-tree* [Uhl91b]), at search time.

Figure 12 compares construction costs and Figure 13 compares search performances. Except on the space of strings, where the *dsa–tree* is costlier to build and significantly (14%–37%) faster to search, in the other spaces the construction costs are almost identical. From those, the *dsa–tree* is 10%–18% faster to search on feature vectors and 6%–12% on histograms. It is slower than *bst–trees* to search on documents, albeit the difference is just 0.7%–1.3%.

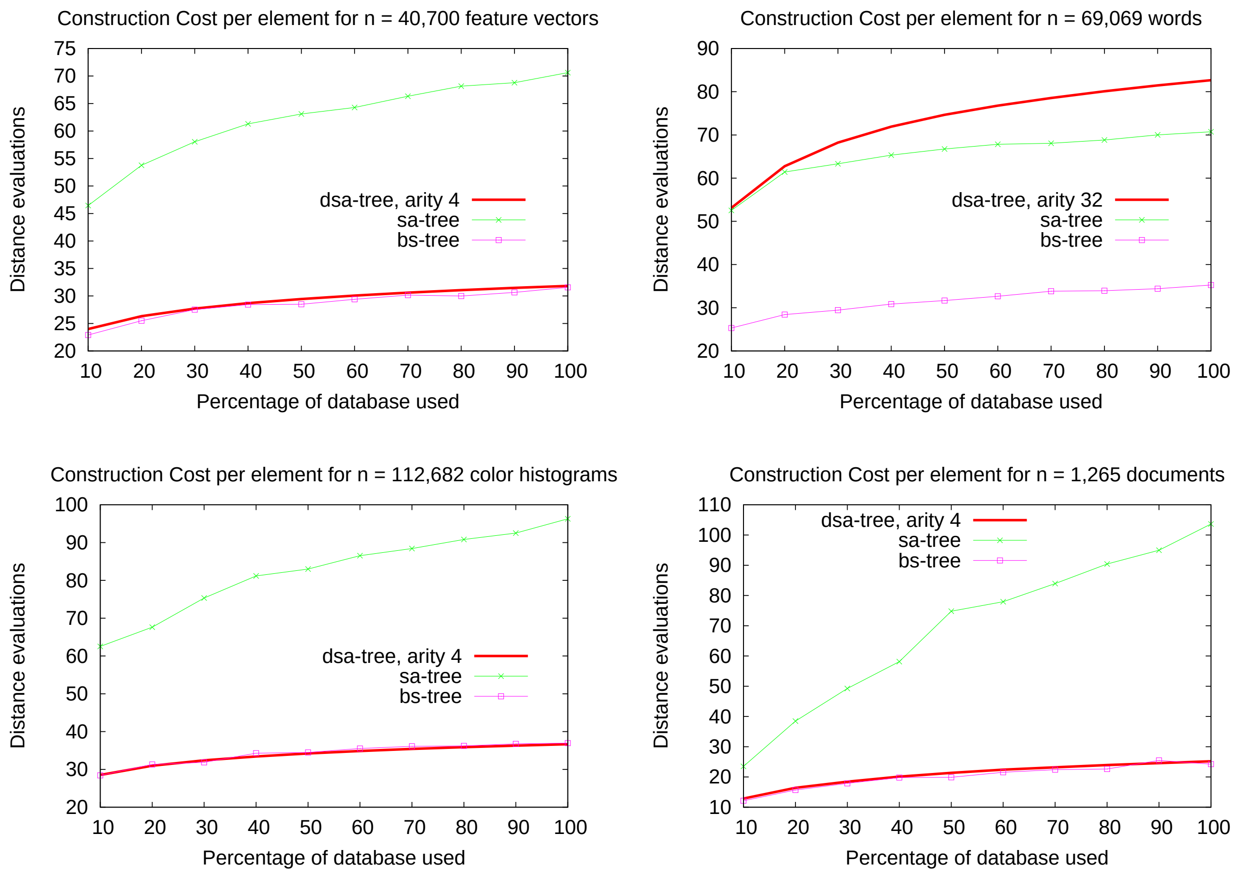


Figure 12: Comparison of construction costs against the *bs–tree*.

8 Conclusions

We have presented a dynamic version of the *sa–tree* data structure, which is able of handling insertions and deletions over arbitrarily long periods of time efficiently and without affecting its search quality. Very few data structures for searching metric spaces are fully dynamic. Furthermore, we have shown that our dynamic version can actually improve the static one both in construction and search performance.

The *sa–tree* was a promising data structure for metric space searching, with several drawbacks

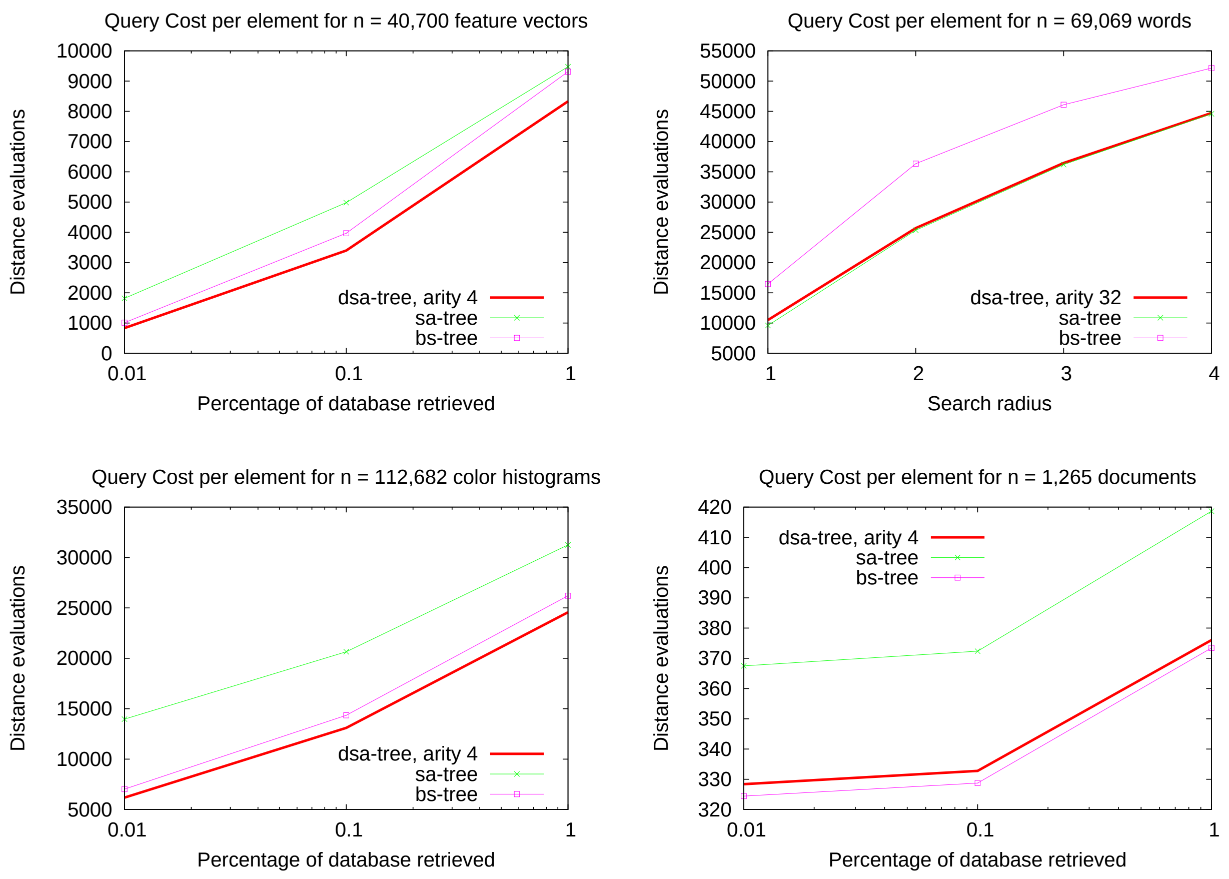


Figure 13: Comparison of search costs against the *bs–tree*.

that prevented it from being practical: high construction cost and poor search performance in some spaces, and inability to accommodate insertions and deletions.

We have addressed all these weaknesses. Our new *dsa–tree* stands out as a practical and efficient data structure that can be used in a wide range of applications, while retaining the good features of the original data structure.

We are currently pursuing in the direction of making the *dsa–tree* work efficiently in secondary memory. In that case both the number of distance computations and disk accesses are relevant. A simple solution to store the *dsa–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 our data structure because we can control the maximum arity of the tree so as to make the neighbors fit in a disk page. It will also be interesting to compare the inherently top-dowm construction of the *dsa–tree* with the bottom-up construction of the *M–tree*.

For deletions, we have considered unacceptable just to mark the deleted elements, because this is not space-effective in the long term. However, in some applications, we could permit a small fraction of deleted elements in the tree, without significantly increasing the overall storage cost. This can be regarded as going one step further from fake nodes, and as such it could amortize tree reconstructions over several deletions. Unfortunately the amount of deleted objects that could be maintained depends on their actual size, and thus on the application. This is another topic of future work.

References

[ABH97] P. Apers, H. Blanken, and M. Houtsma. *Multimedia Databases in Perspective.* Springer, 1997.

[BBK01] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[BK73] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Comm. of the ACM*, 16(4):230–236, 1973.

[BN02] B. Bustos and G. Navarro. Probabilistic proximity searching algorithms based on compact partitions. In *Proc. 9th String Processing and Information Retrieval (SPIRE'02)*, LNCS 2476, pages 284–297, 2002.

[BO97] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. ACM Conference on Management of Data (ACM SIGMOD'97)*, pages 357–368, 1997. Sigmod Record 26(2).

[Bri95] S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[BS80] J. Bentley and J. Saxe. Decomposable searching problems i: Static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[BYCMW94] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[CFP$^+$05] D. Cantone, A. Ferro, A. Pulvirenti, D. Reforgiato Recupero, and D. Shasha. Antipole tree indexing to support range search and k-nearest neighbor search in metric spaces. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):535–550, 2005.

[Cla99] K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete & Computational Geometry*, 22(1):63–93, 1999.

[CMBY99] 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.

[CMN01] 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.

[CN01] E. Chávez and G. Navarro. A probabilistic spell for the curse of dimensionality. In *Proc. 3rd Workshop on Algorithm Engineering and Experiments (ALENEX'01)*, LNCS 2153, pages 147–160, 2001.

[CN05] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[CNBYM01] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[CPZ97] 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.

[DGSZ03] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools and Applications*, 21(1):9–33, 2003.

[DH73] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* Wiley, 1973.

[DN87] F. Dehne and H. Noltemeier. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987.

[Doh04] V. Dohnal. An access structure for similarity search in metric spaces. In *EDBT Workshops*, pages 133–143, 2004.

[GG98] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GR93] I. Galperin and R. Rivest. Scapegoat trees. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[HS00] G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report CS-TR-4199, University of Maryland, Computer Science Department, 2000.

[HS03a] G. Hjaltason and H. Samet. Improved search heuristics for the sa-tree. *Pattern Recognition Letters*, 24(15):2785–2795, 2003.

[HS03b] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. on Database Systems*, 28(4):517–580, 2003.

[KL04] R. Krauthgamer and J. Lee. Navigating nets: simple algorithms for proximity search. In *SODA*, pages 798–807, 2004.

[KR02] D. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 741–750, New York, NY, USA, 2002. ACM Press.

[Kuk92] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[MOC96] L. Micó, J. Oncina, and R. Carrasco. A fast branch and bound nearest neighbour classifier in metric spaces. *Pattern Recognition Letters*, 17:731–739, 1996.

[MOV94] 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.

[Nav99] 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.

[Nav02] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[NN97] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, 1997.

[NR01] G. Navarro and N. Reyes. Dynamic spatial approximation trees. In *Proc. XXI Conference of the Chilean Computer Science Society (SCCC'01)*, pages 213–222. IEEE CS Press, 2001.

[NR02a] 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.

[NR02b] G. Navarro and N. Reyes. Improved dynamic spatial approximation trees. In *Proceedings of the XXVIII Latin American Conference on Informatics (CLEI'02)*, page 74, Montevideo, Uruguay, 2002. Abstract in print, complete papers in CD-ROM.

[NR03] G. Navarro and N. Reyes. Improved deletions in dynamic spatial approximation trees. In *Proc. XXIII Conference of the Chilean Computer Science Society (SCCC'03)*, pages 13–22, Chillán, Chile, 2003. IEEE CS Press.

[NVZ92] H. Noltemeier, 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. Springer-Verlag, 1992.

[Sam05] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[SK83] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.

[SM83] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[SPS04] T. Skopal, J. Pokorný, and V. Snásel. PM-tree: Pivoting metric tree for similarity search in multimedia databases. In *ADBIS (Local Proceedings)*, 2004.

[Uhl91a] J. Uhlmann. Implementing metric trees to satisfy general proximity/similarity queries. In *Proc. Command and Control Symposium*, Washington, DC, 1991. Also published as Code 5570 NRL Memo Report (1991) at the Naval Research Laboratory.

[Uhl91b] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[UN03] 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.

[Ver95] K. Verbarg. The C-Tree: a dynamically balanced spatial index. *Computing Suppl.*, 10:323–340, 1995.

[Vid86] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[Wat95] M. Waterman. *Introduction to Computational Biology.* Chapman and Hall, 1995.

[YI99] A. Yoshitaka and T. Ichikawa. A survey on content-based retrieval for multimedia databases. *IEEE Trans. on Knowledge and Data Engineering*, 11(1):81–93, 1999.

[Yia93] 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.

[Yia00] P. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, pages 361–370, 2000.

[ZADB06] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems.* Springer, 2006.

A Performance Plots

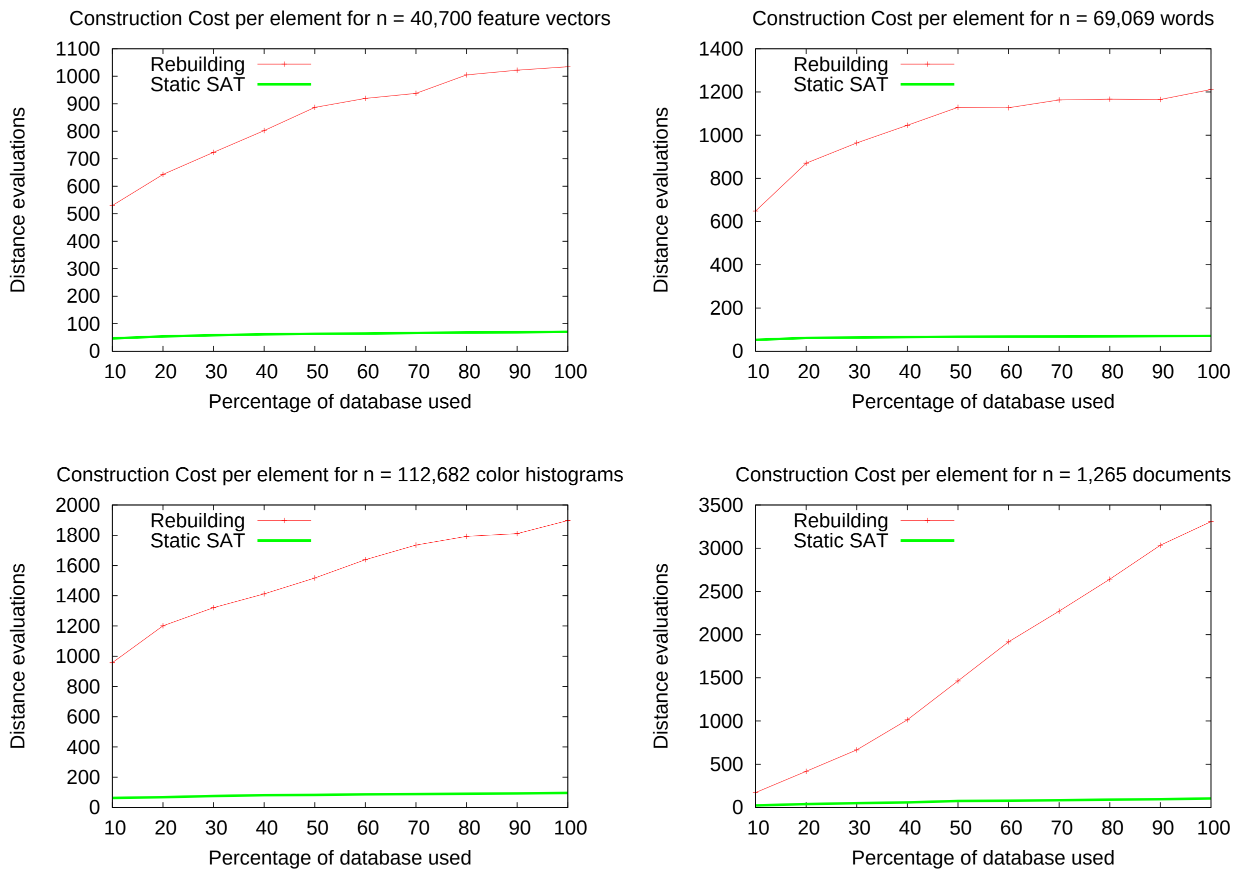


Figure 14: Construction costs per element by rebuilding the subtree.

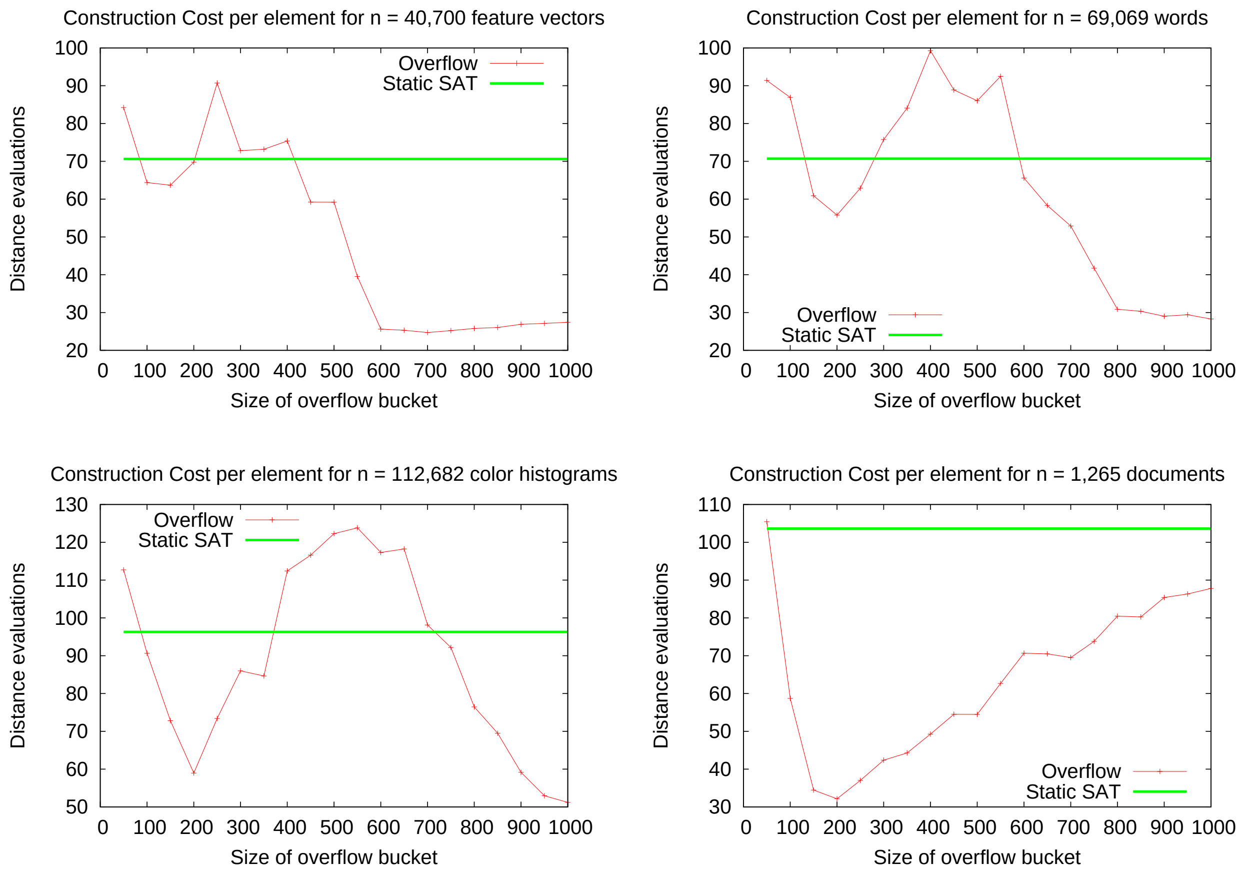


Figure 15: Construction costs per element using overflow buckets.

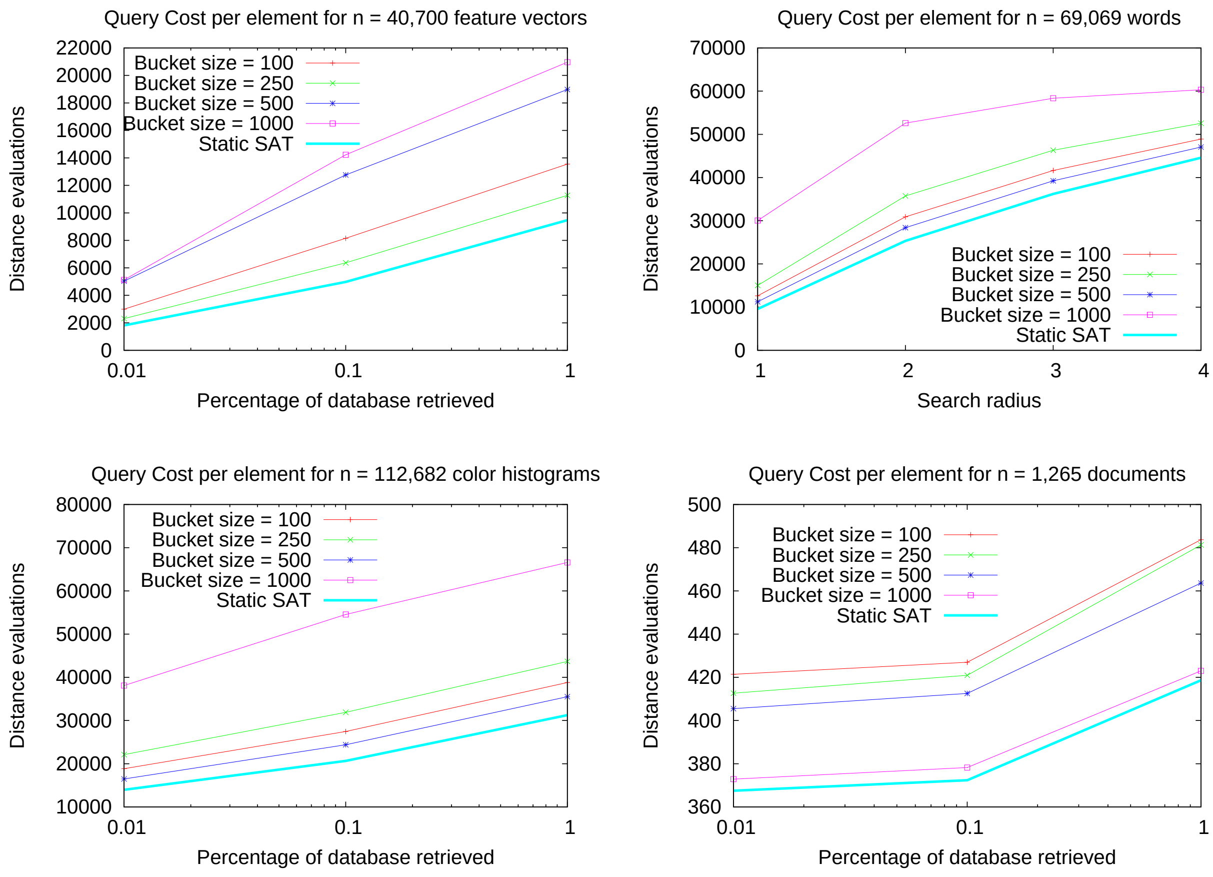
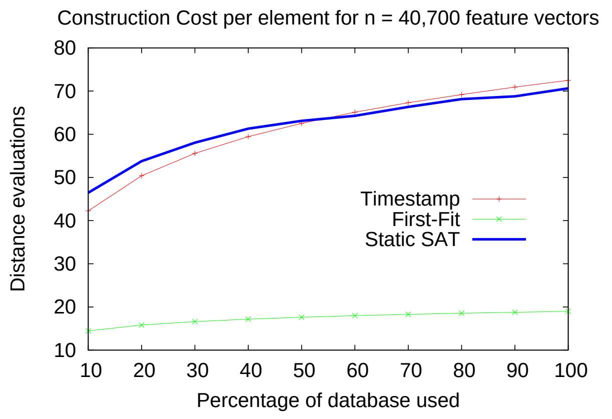
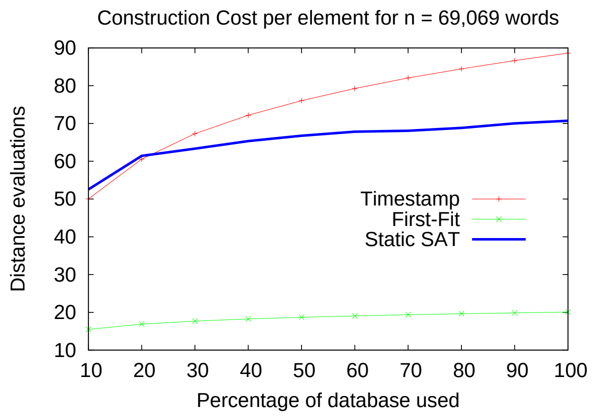
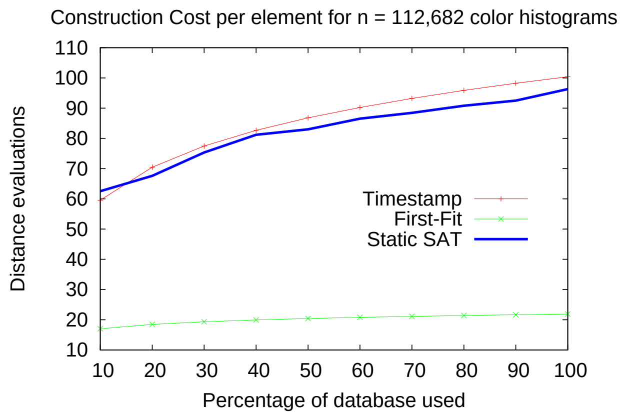


Figure 16: Search costs using overflow buckets.







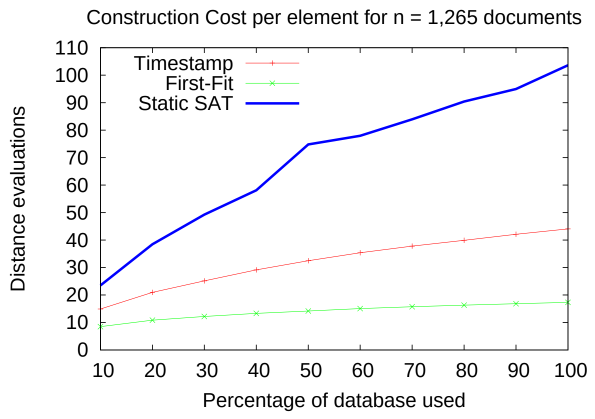
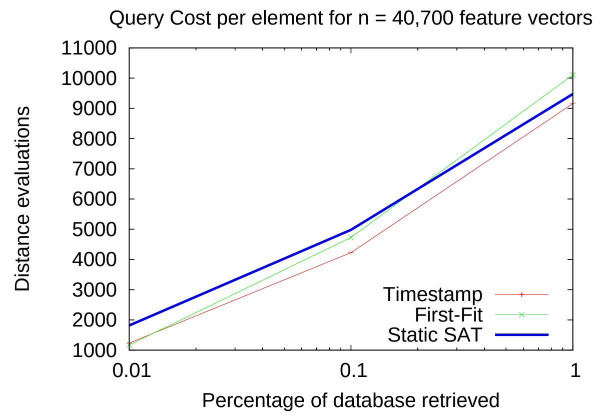
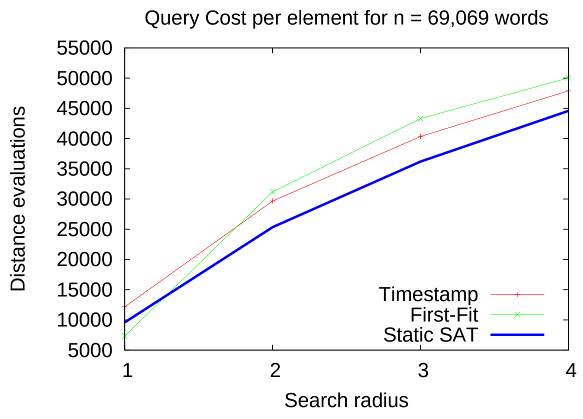
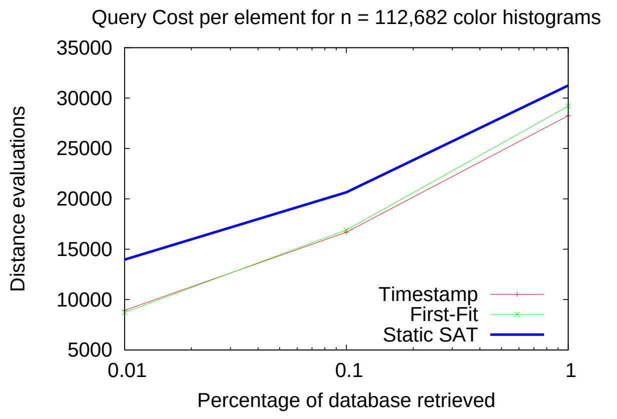


Figure 17: Construction costs per element using first-fit and using timestamps.







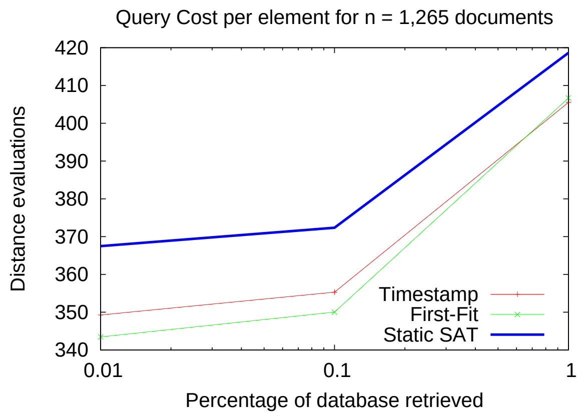


Figure 18: Search costs using first-fit and timestamping strategies.

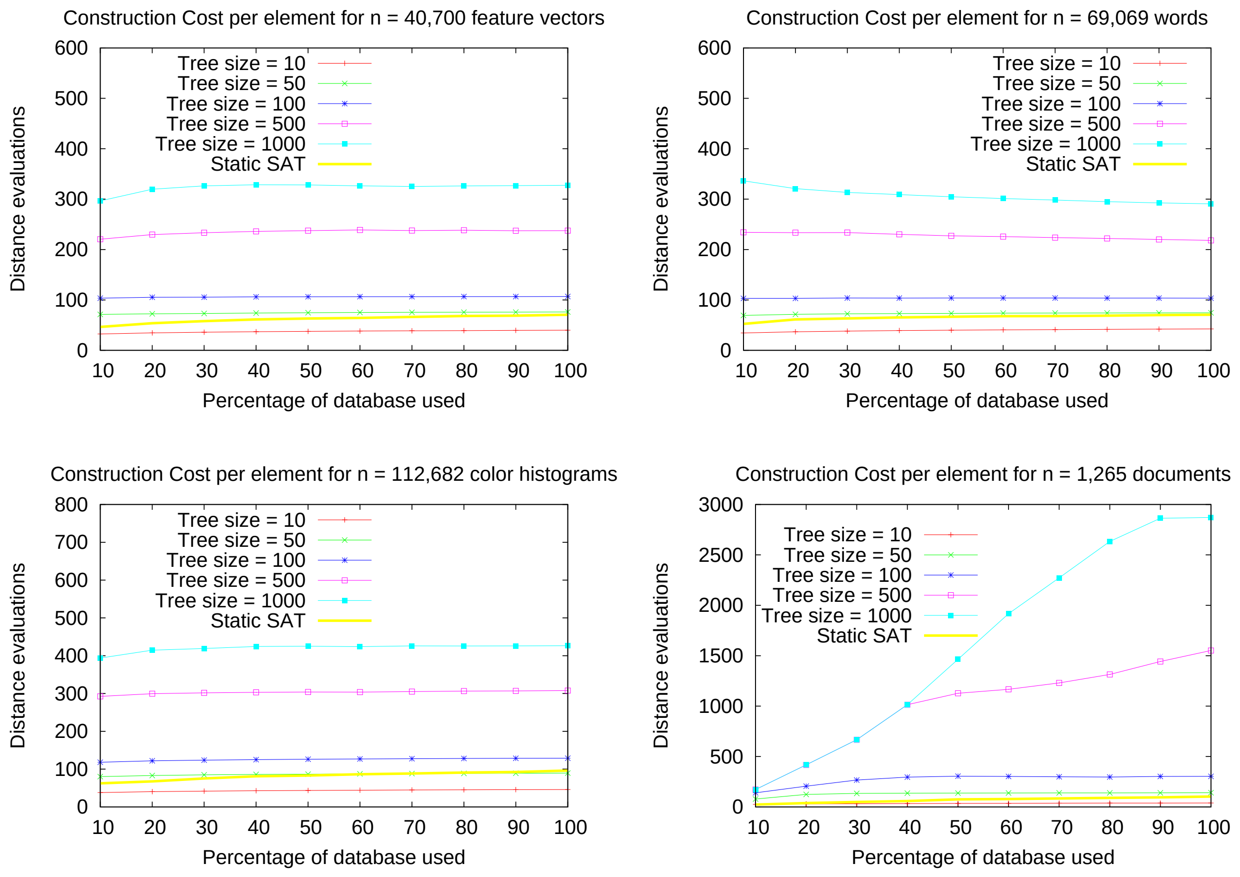


Figure 19: Construction costs per element inserting at the fringe.

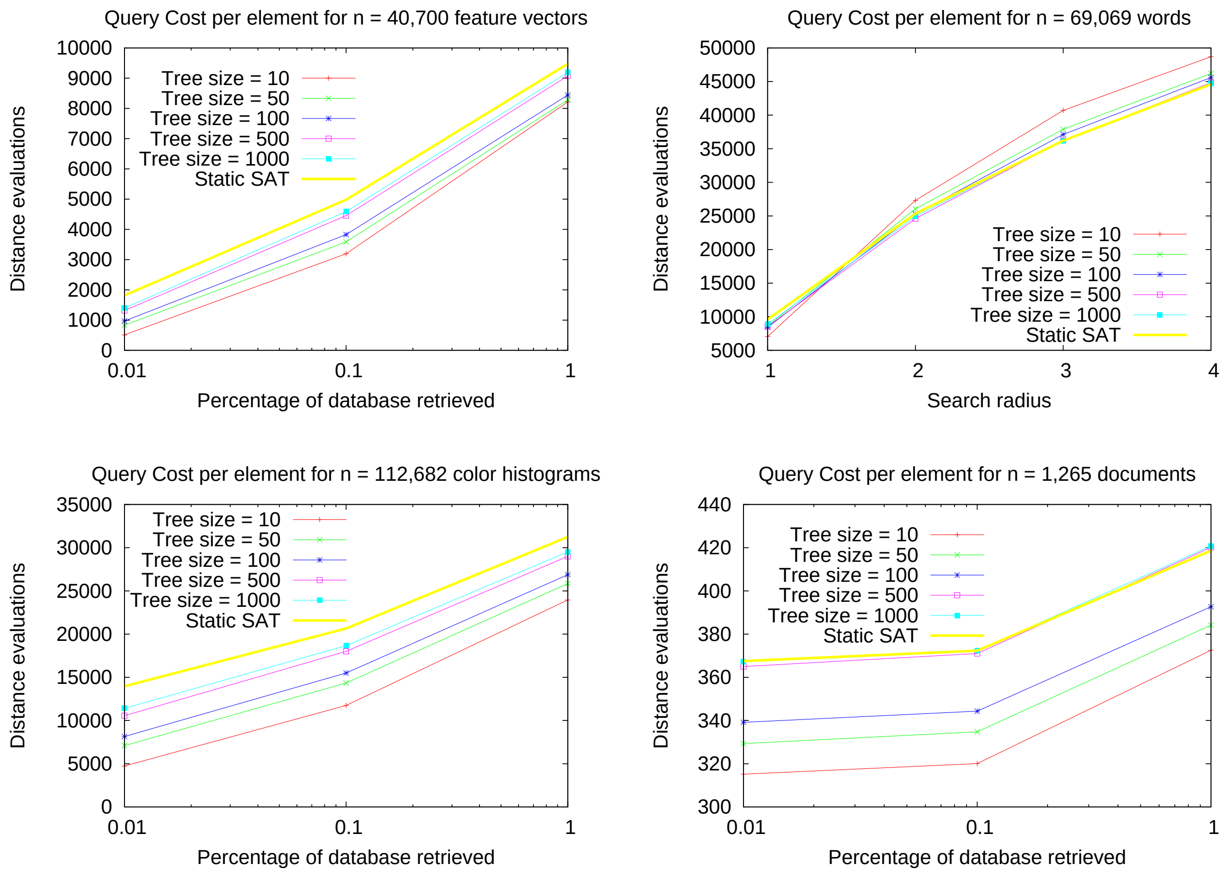


Figure 20: Search costs using insertion at the fringe.

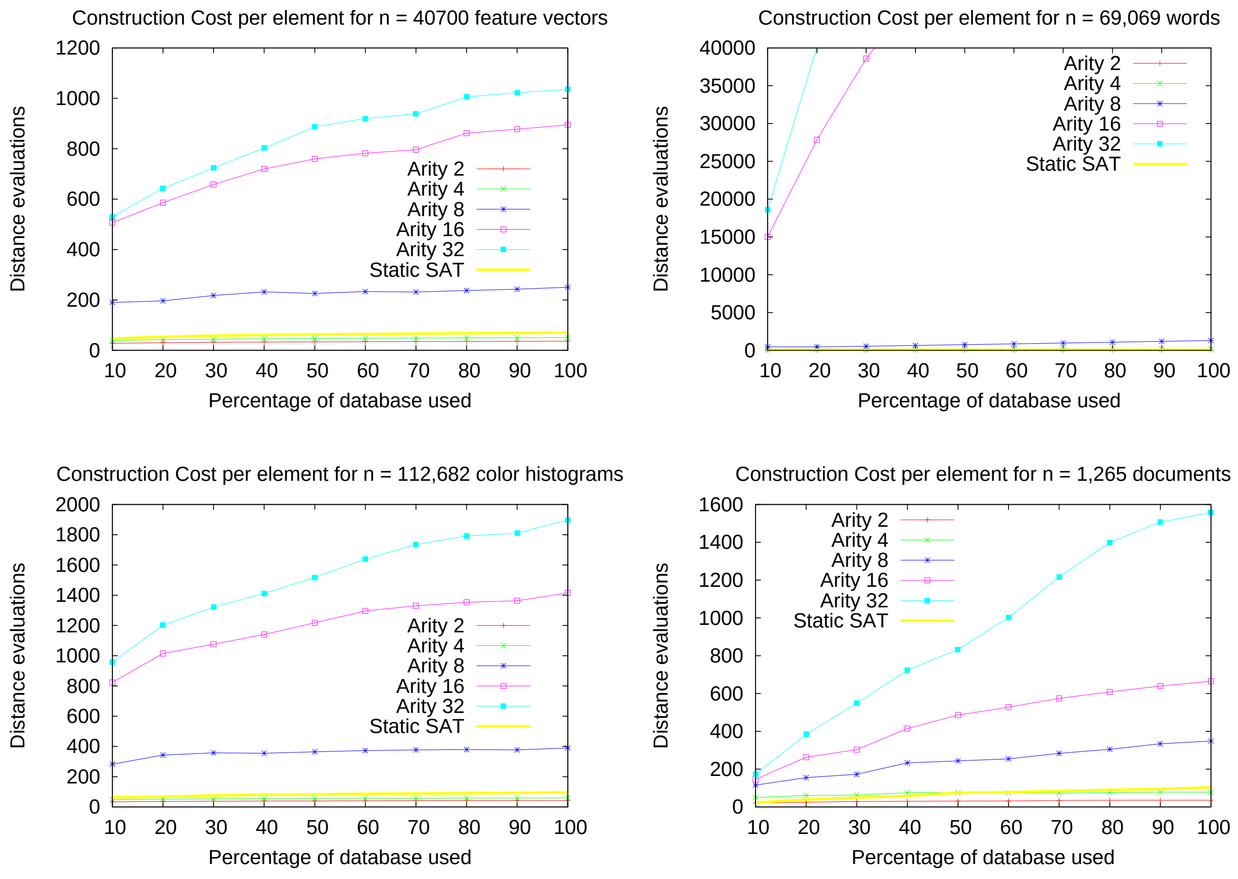


Figure 21: Construction costs per element using bounded arity.

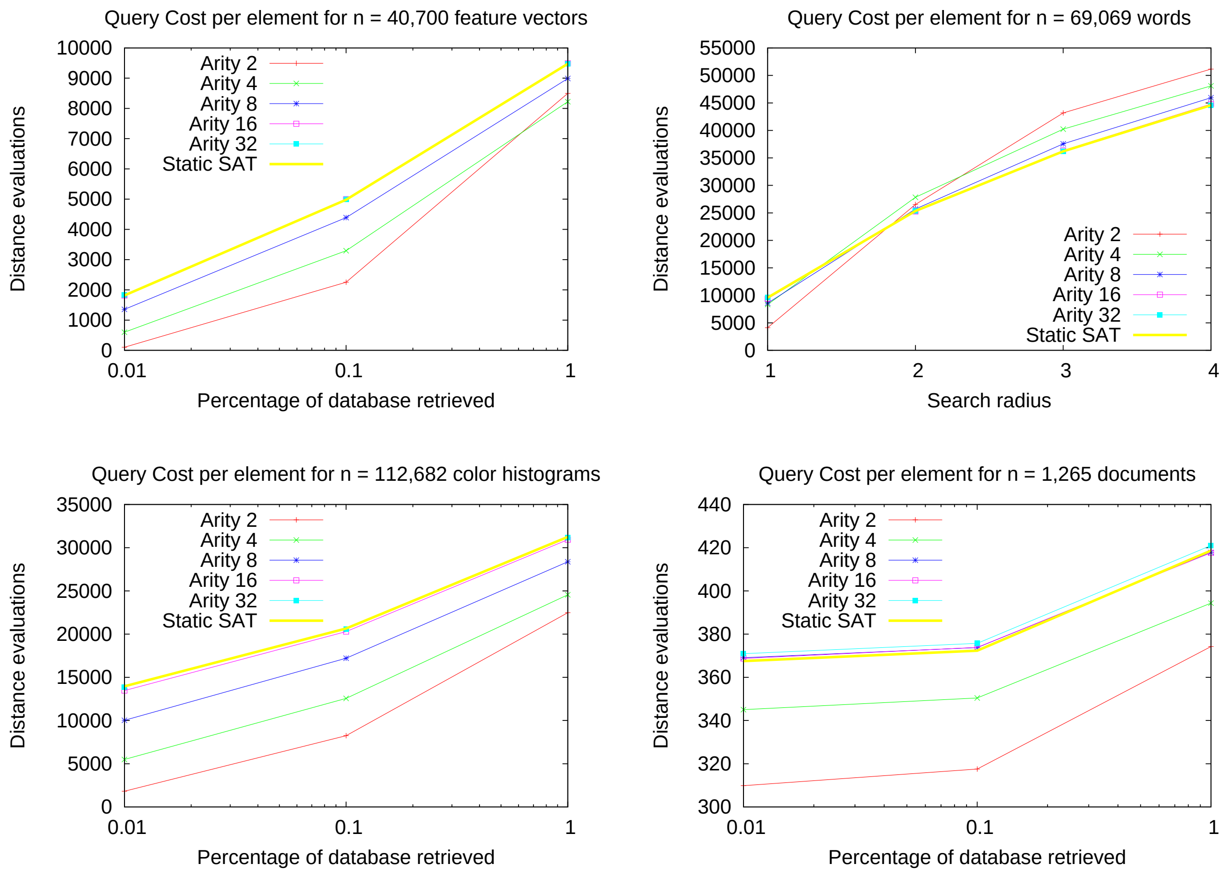


Figure 22: Search costs using bounded arity.

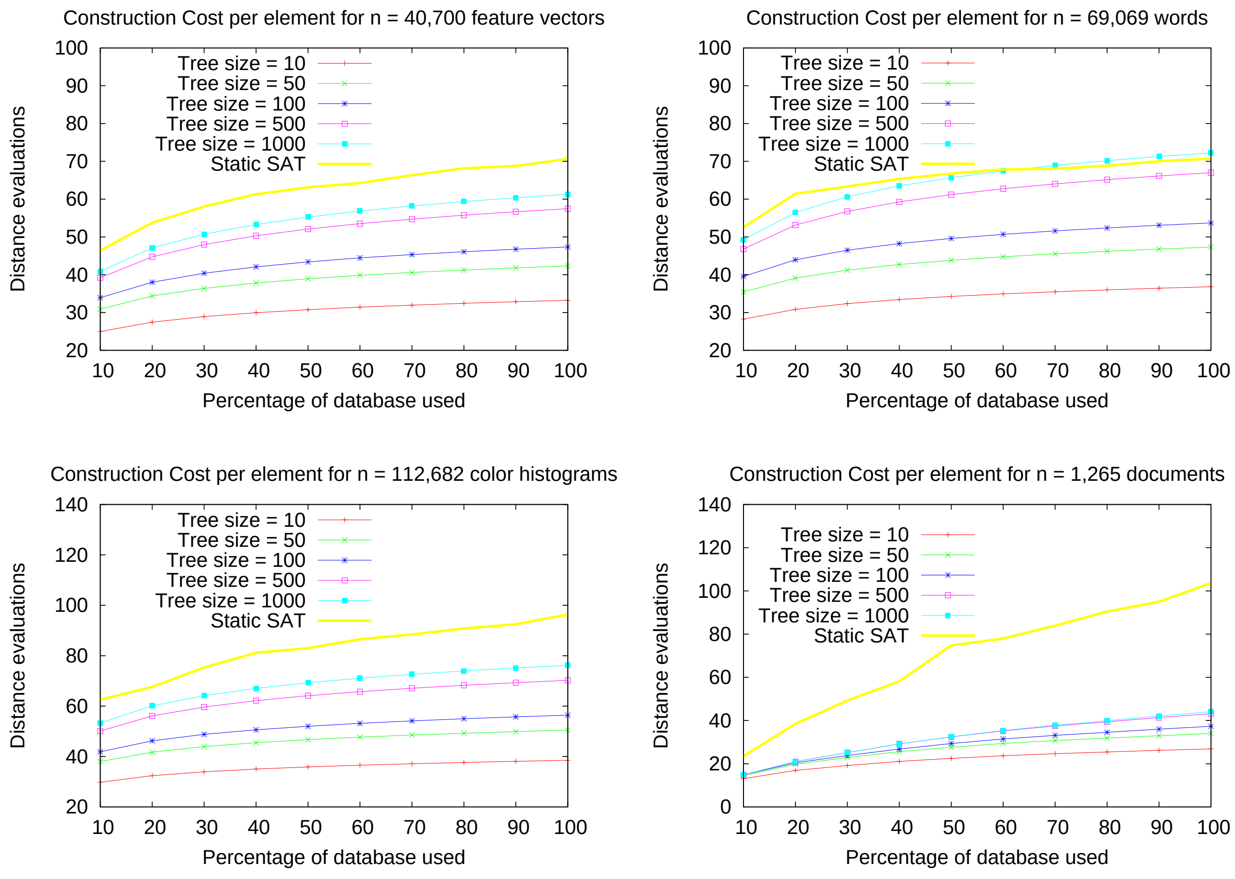


Figure 23: Construction costs per element combining timestamping with insertion at the fringe.

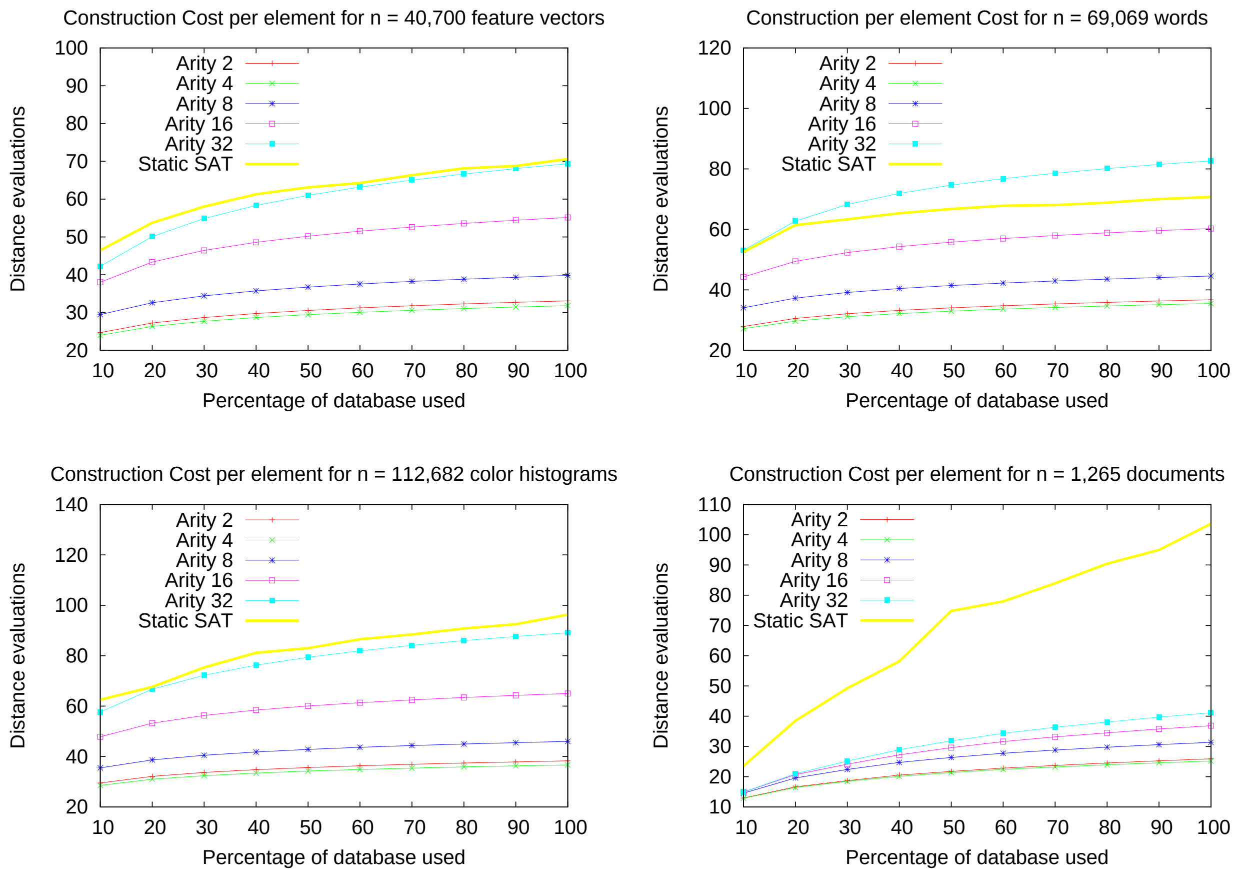


Figure 24: Construction costs per element using timestamping plus bounded arity.

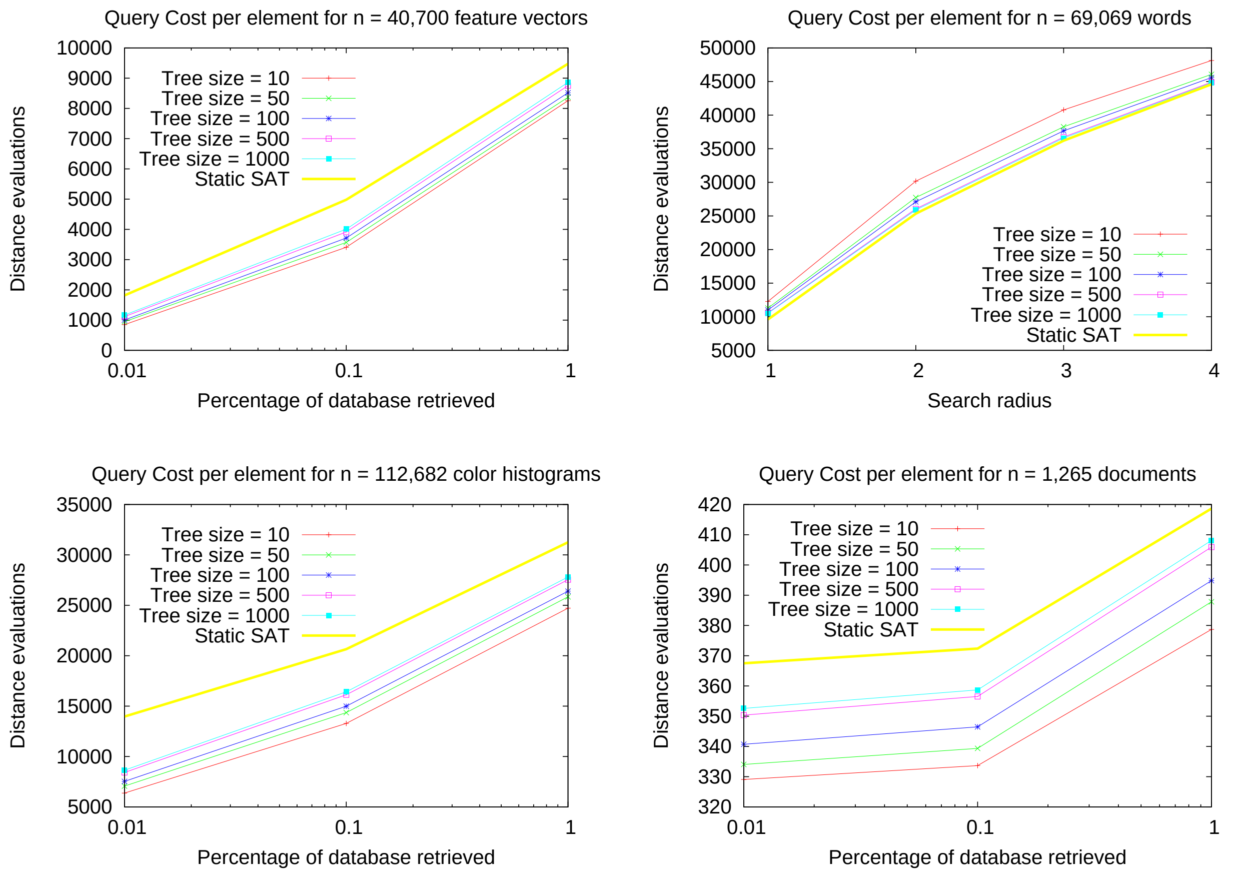
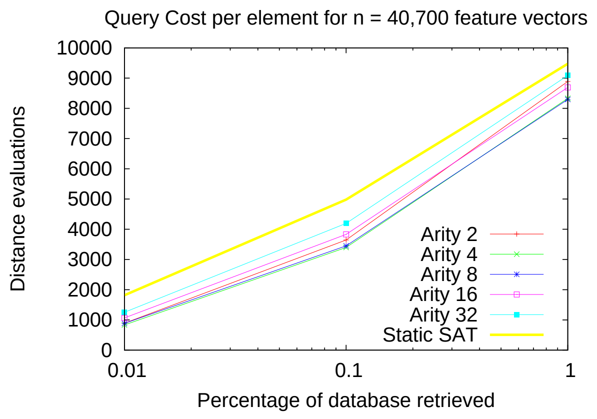
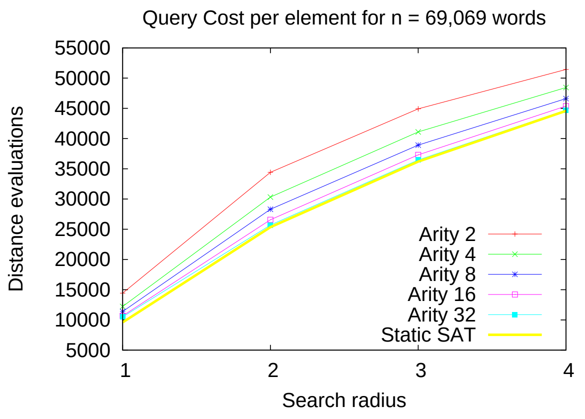
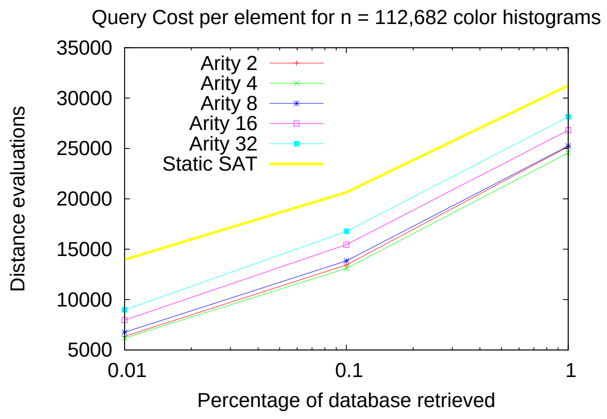


Figure 25: Search costs combining timestamping with insertion at the fringe.







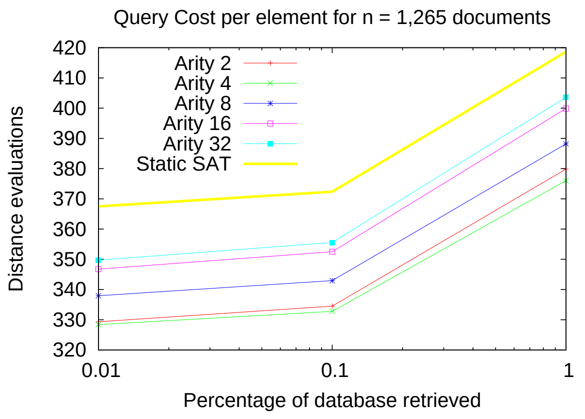


Figure 26: Search costs using timestamping plus bounded arity.

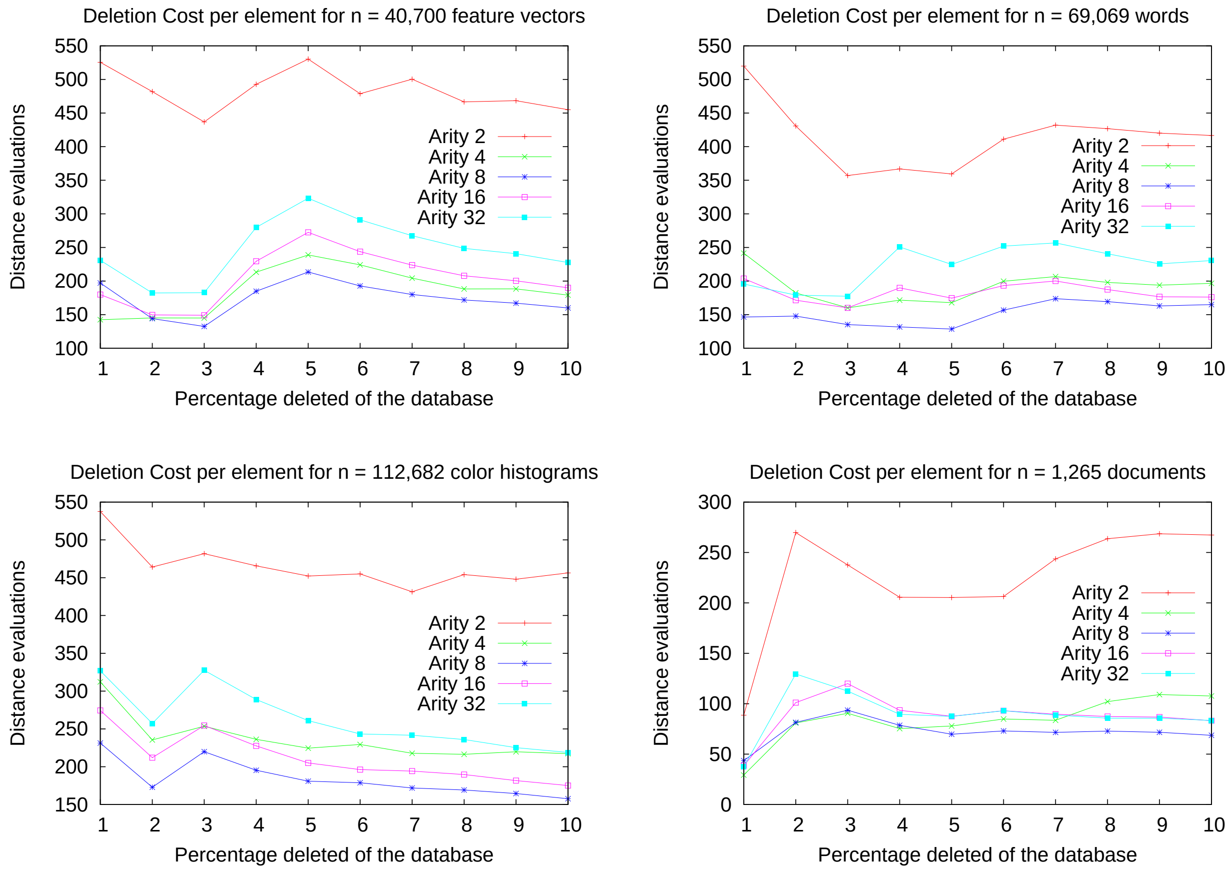


Figure 27: Deletion cost per element using reinsertion of subtrees.

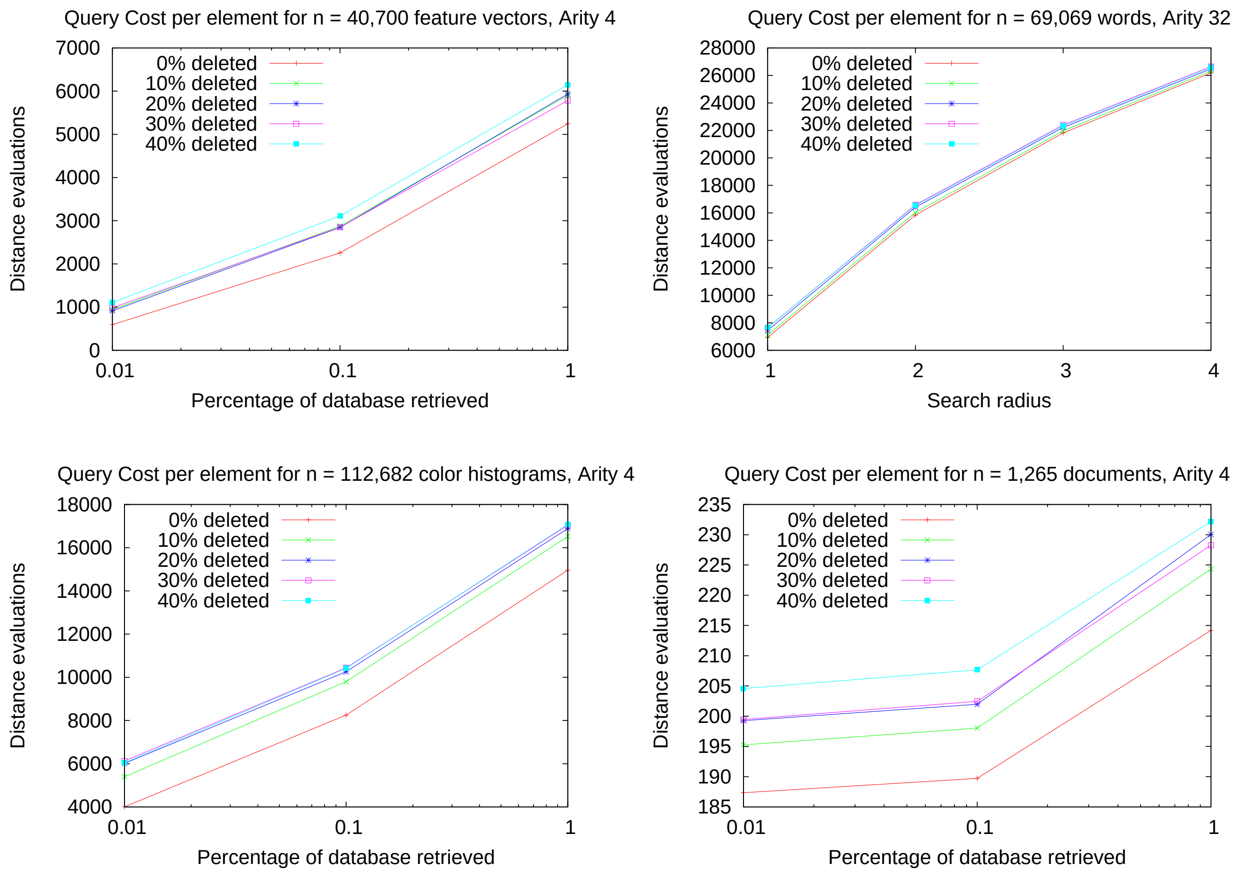


Figure 28: Search costs using reinsertion of subtrees.

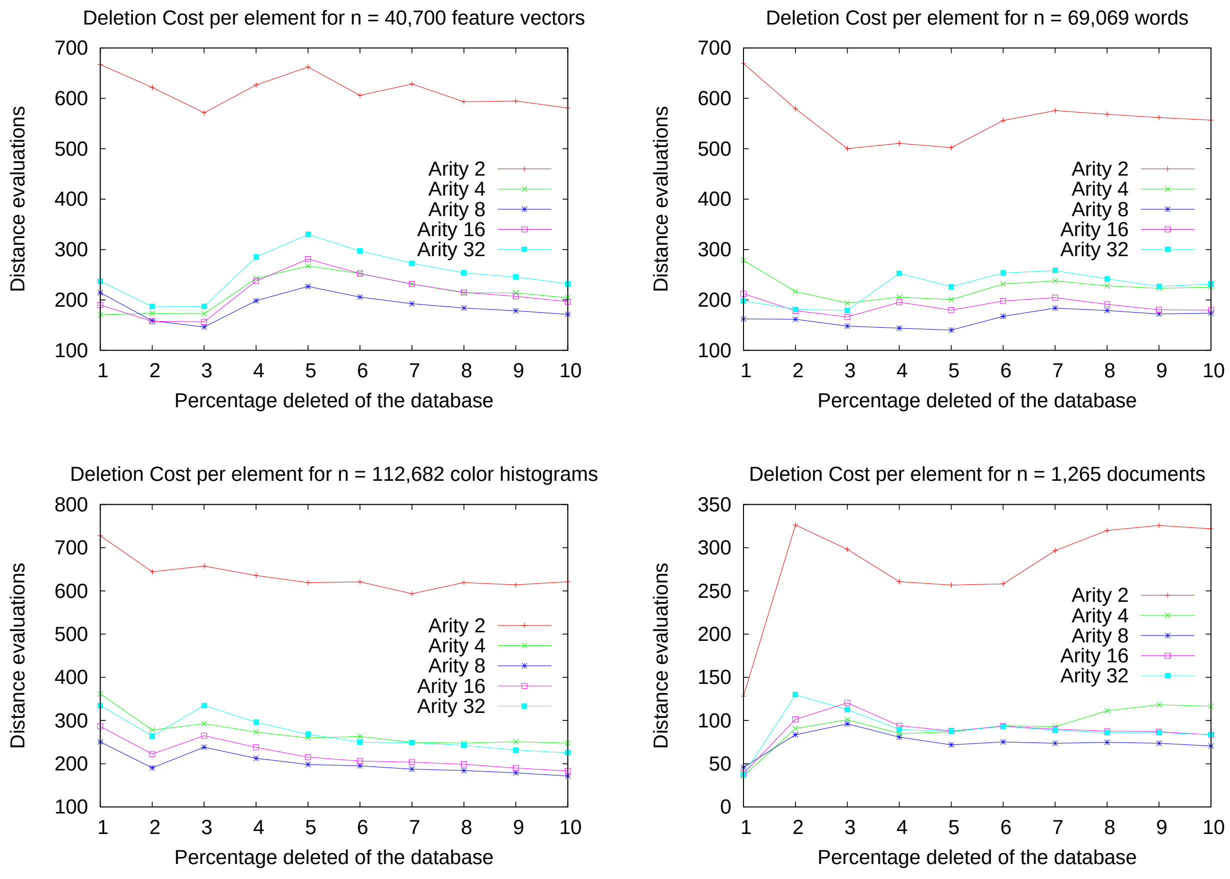


Figure 29: Deletion cost per element using reinsertion of elements.

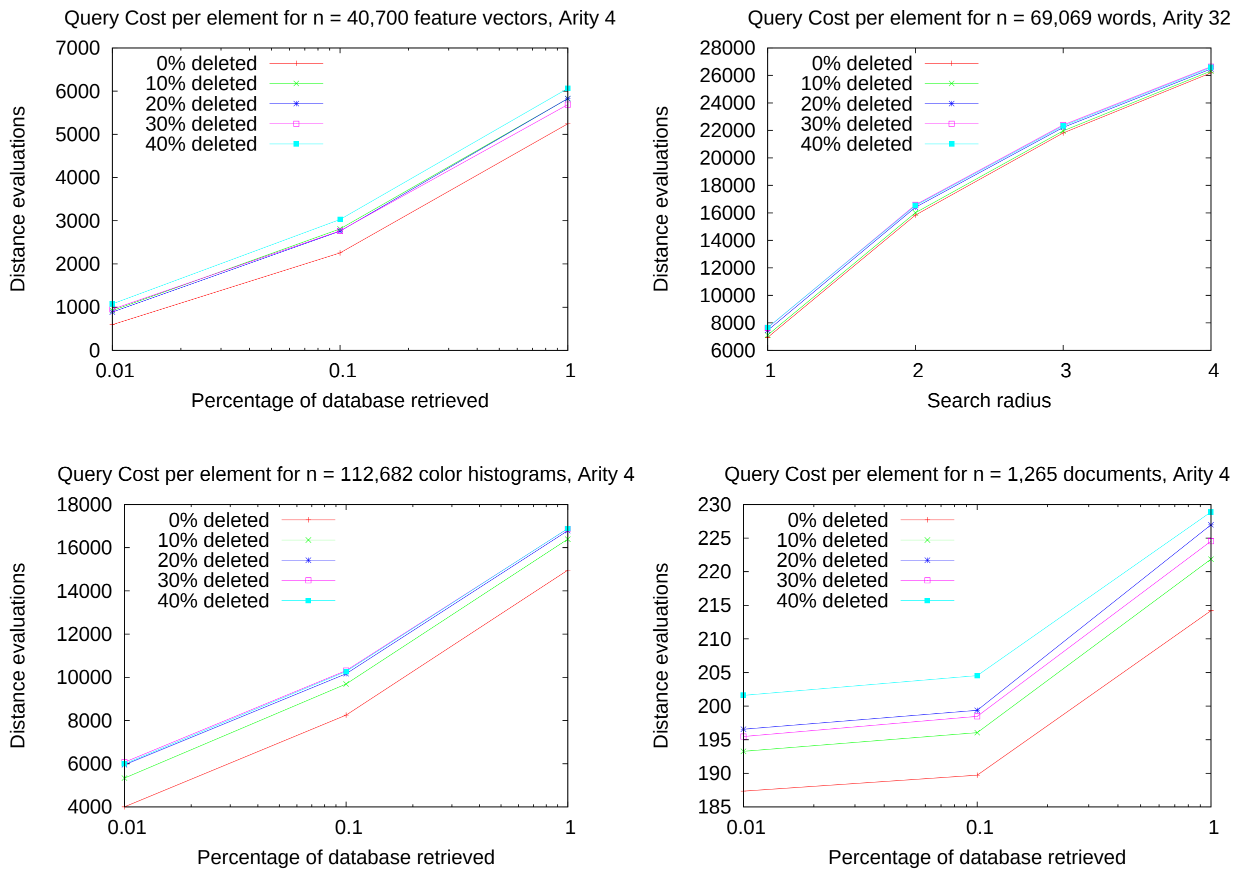


Figure 30: Search costs using reinsertion of elements

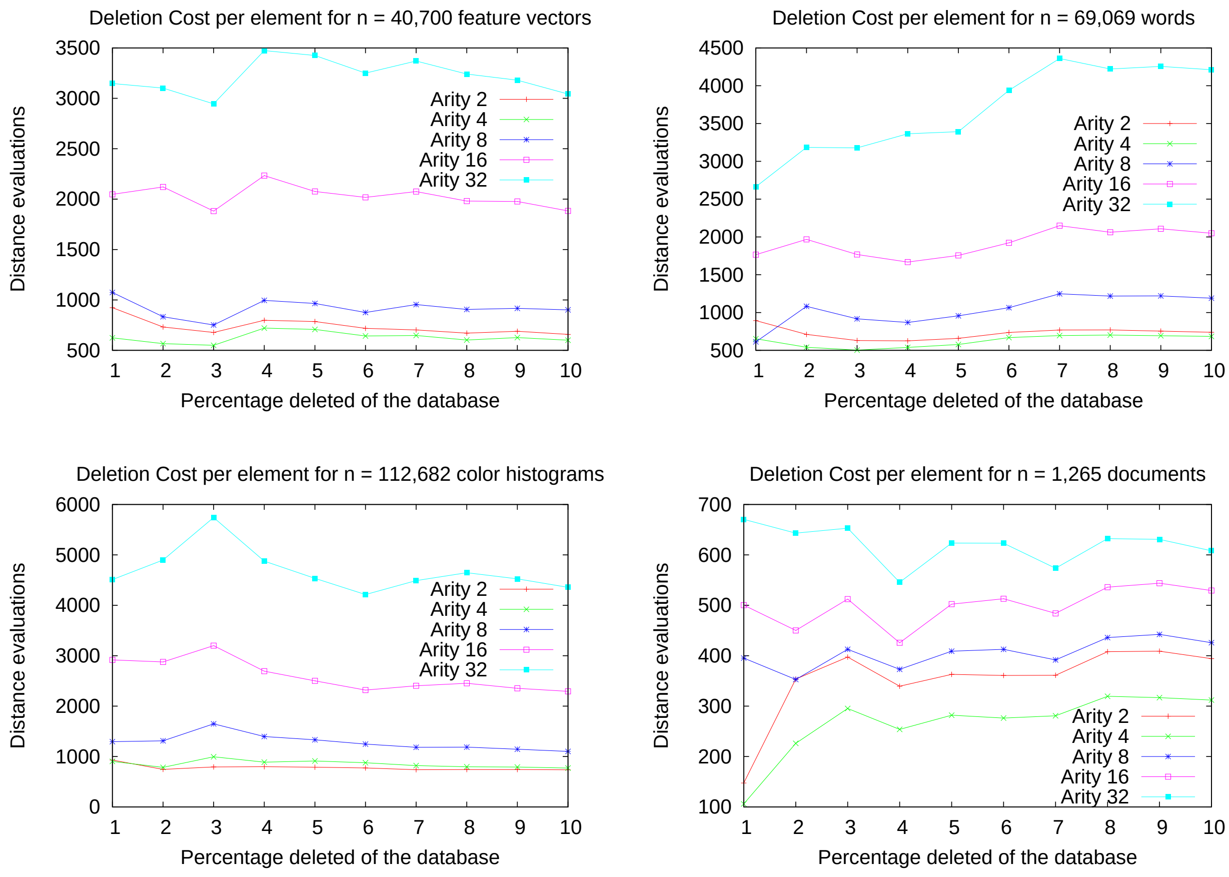
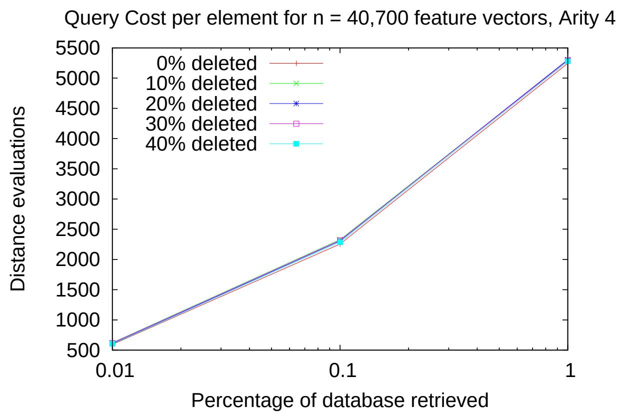
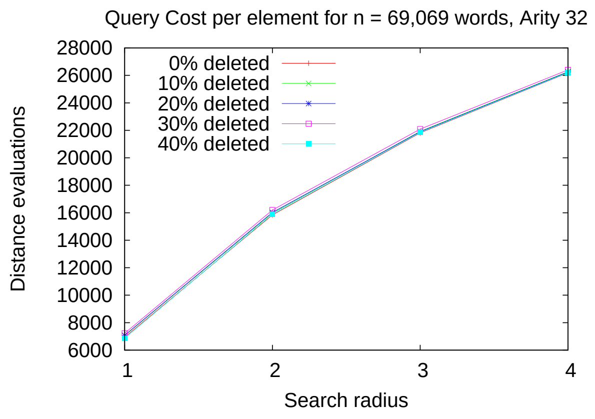
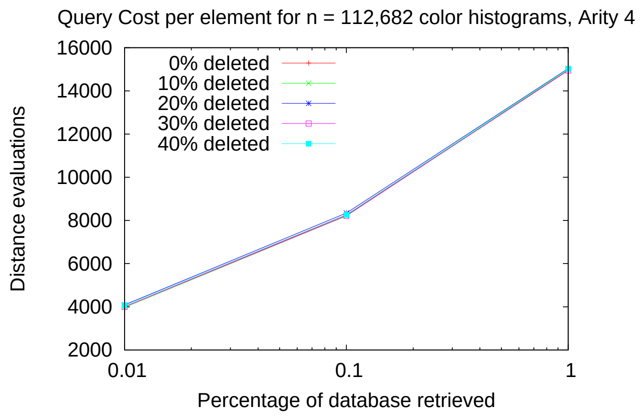


Figure 31: Deletion cost per element using rebuilding of subtrees.







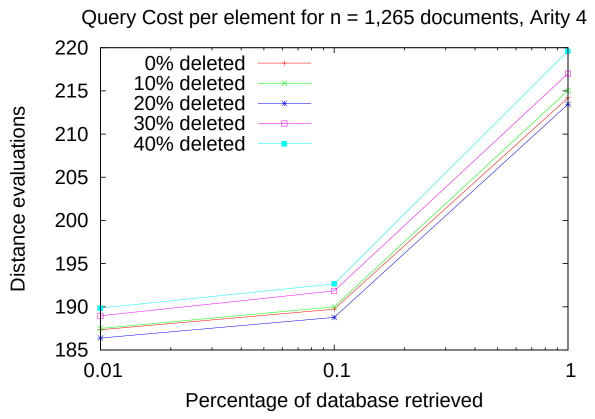
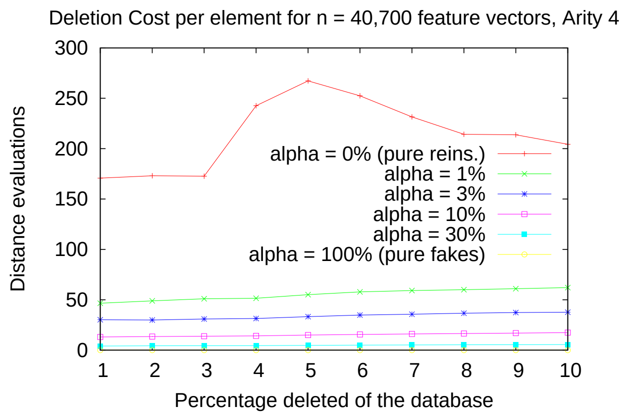
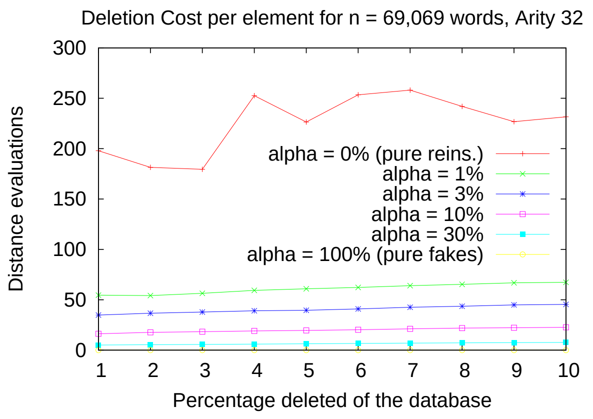
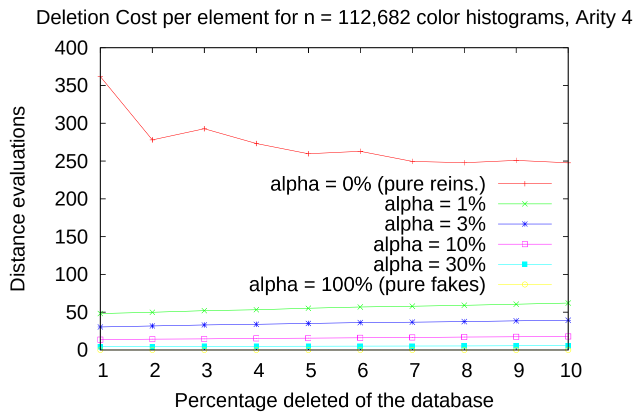


Figure 32: Search costs using rebuilding of subtrees.







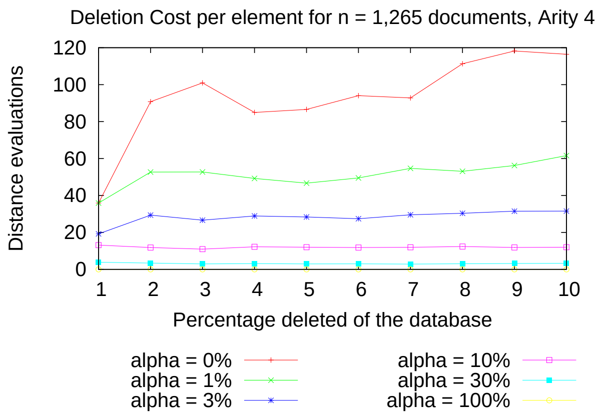


Figure 33: Deletion cost per element combining reinsertion of elements with fake nodes.

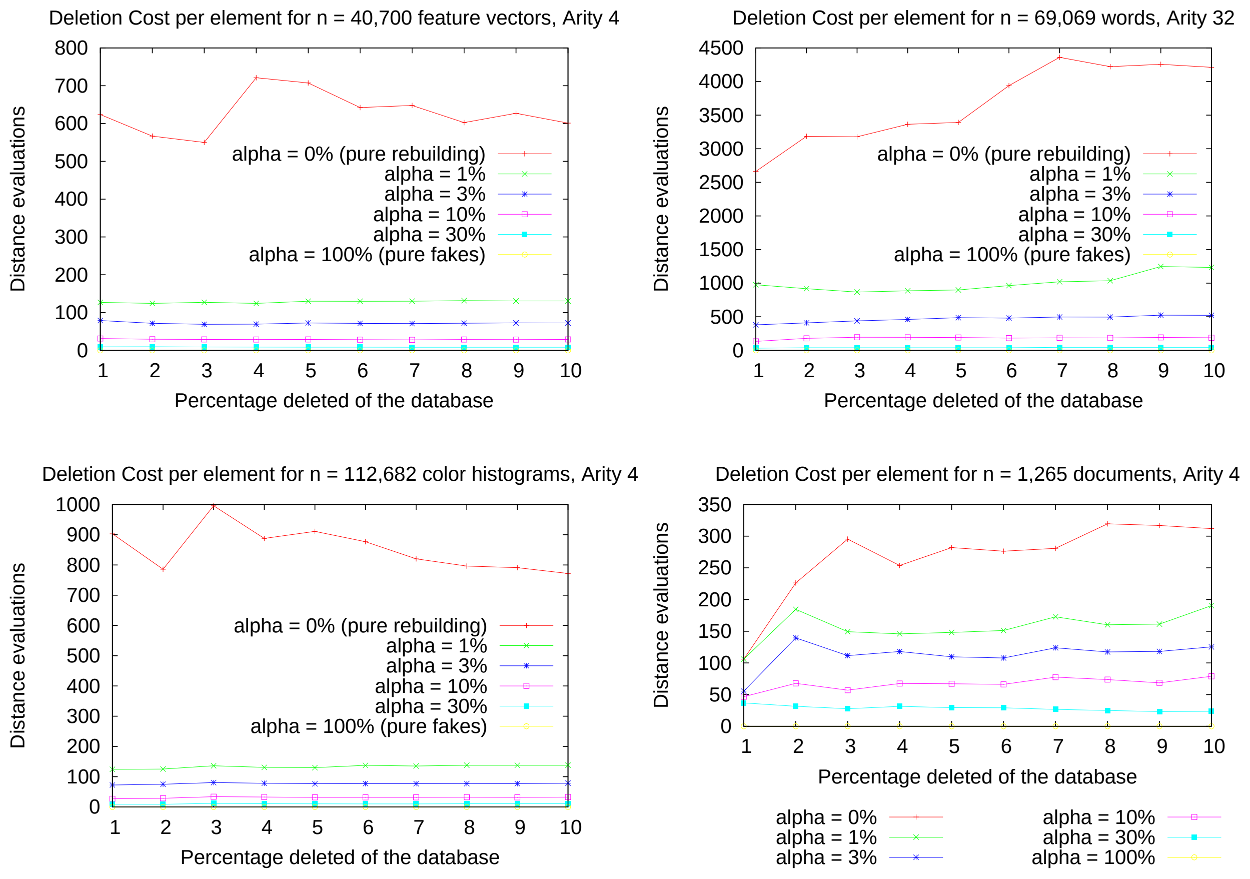


Figure 34: Deletion cost per element combining rebuilding of subtrees with fake nodes.

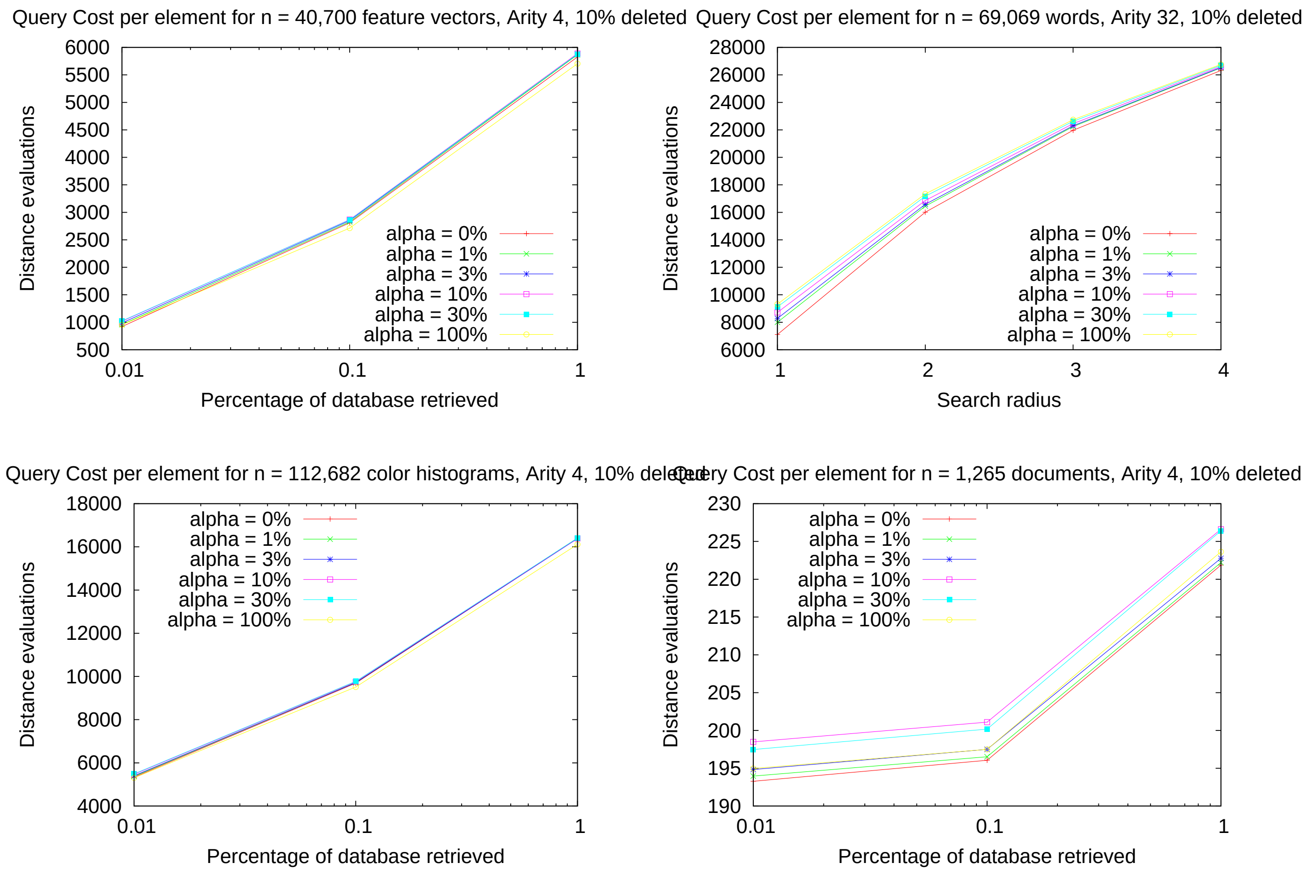
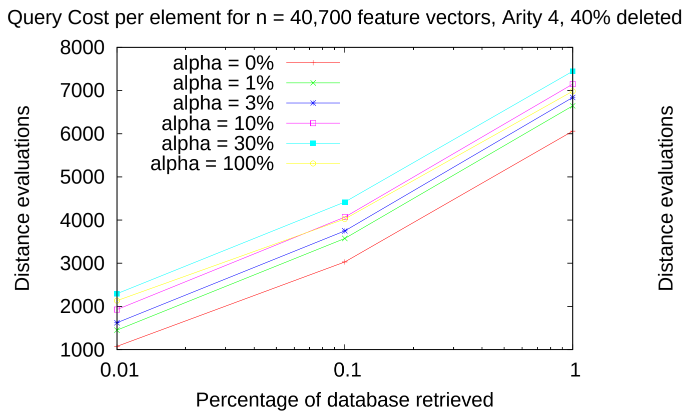
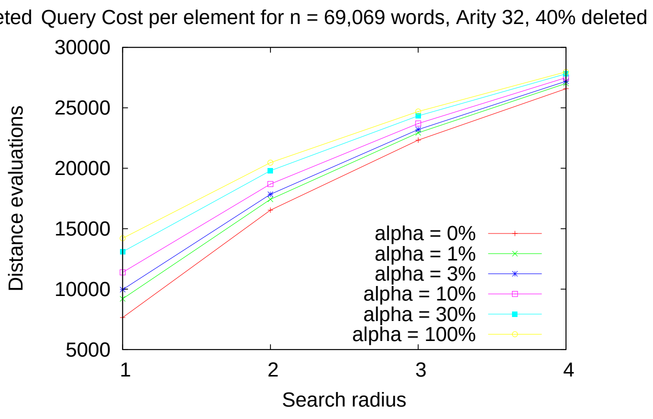
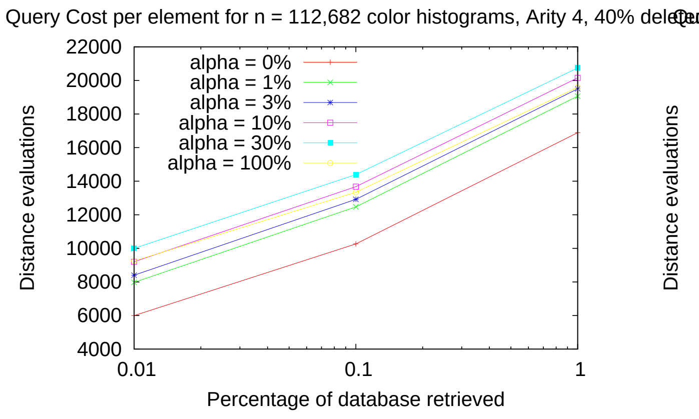


Figure 35: Search costs combining reinsertion of elements with fake nodes, for 10% of elements deleted.







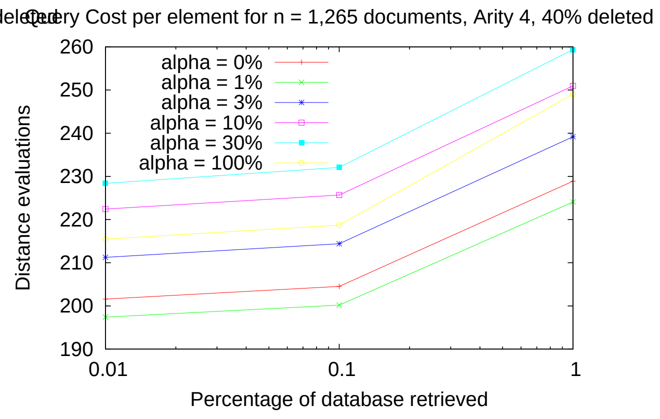


Figure 36: Search costs combining reinsertion of elements with fake nodes, for 40% of elements deleted. The search is done over half of the set.

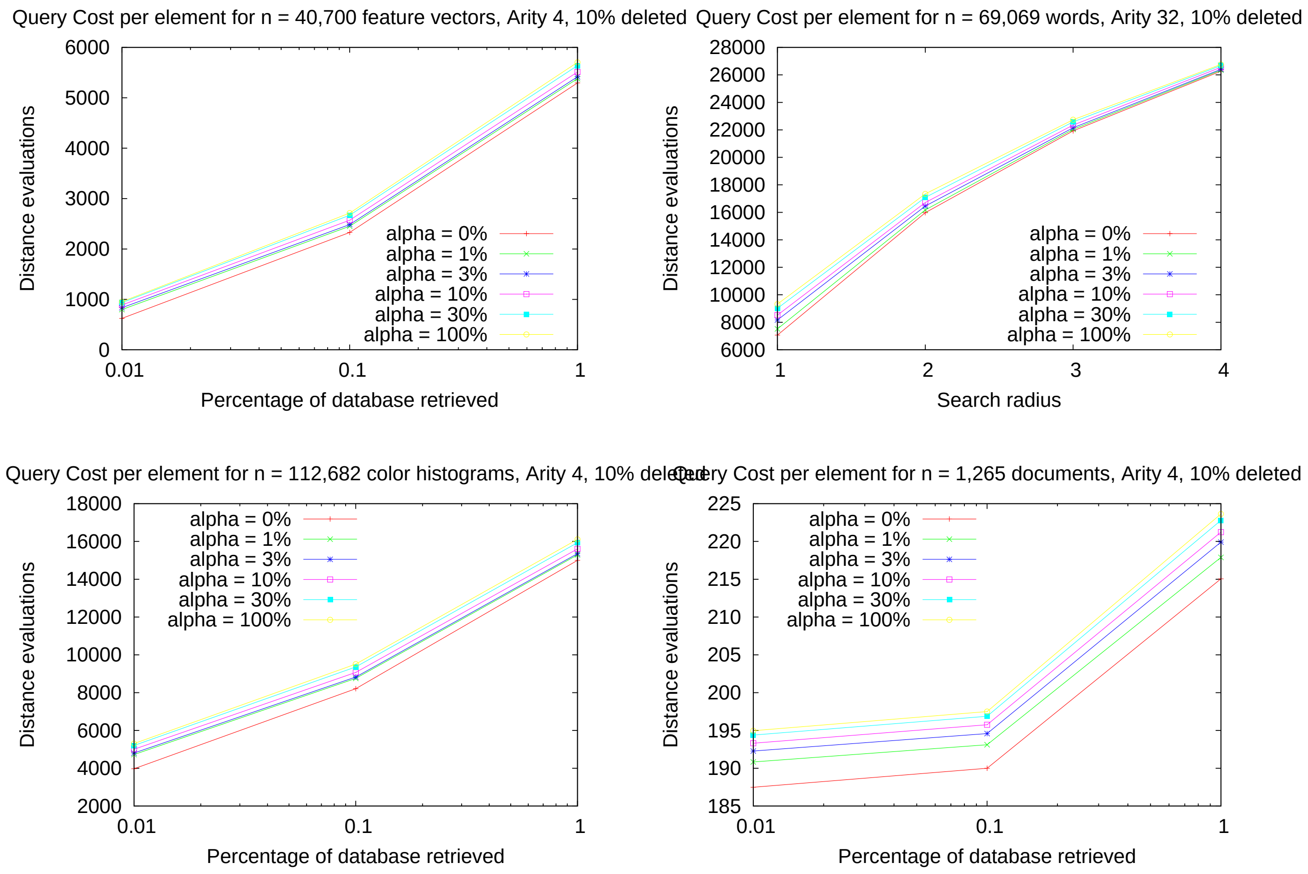


Figure 37: Search costs combining rebuilding of subtrees with fake nodes, for 10% of elements deleted. The search is done over half of the set.

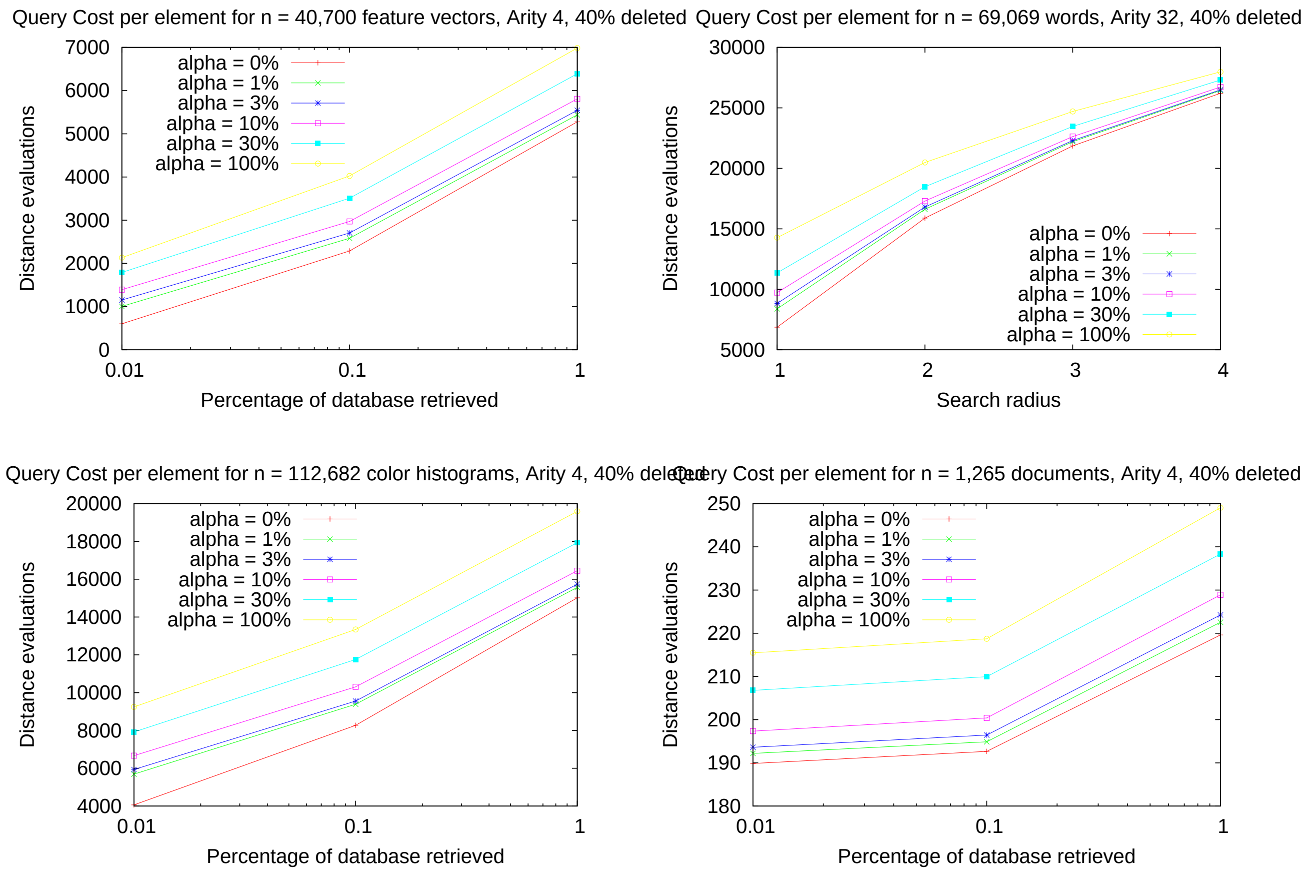


Figure 38: Search costs combining rebuilding of subtrees with fake nodes, for 40% of elements deleted. The search is done over half of the set.

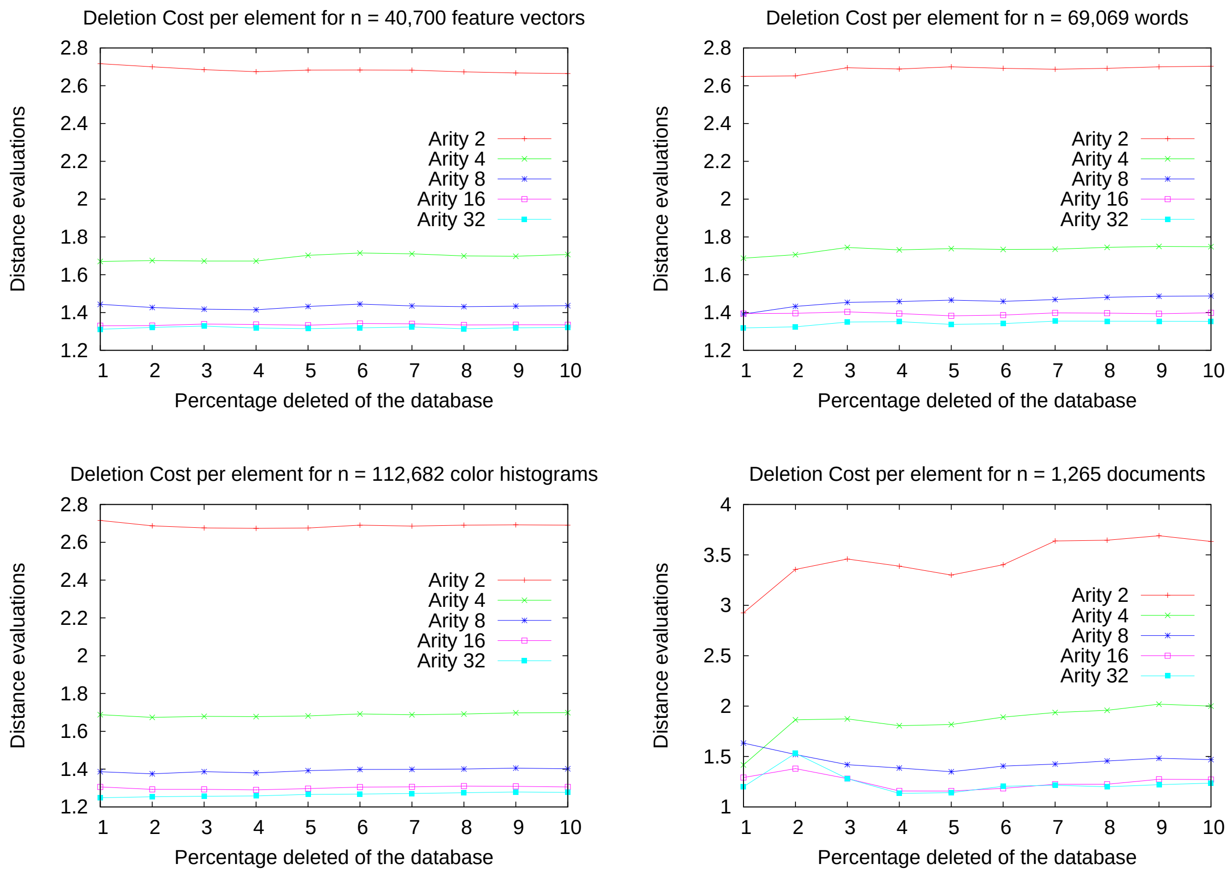


Figure 39: Deletion cost per element using ghost hyperplanes, replacing by a leaf.

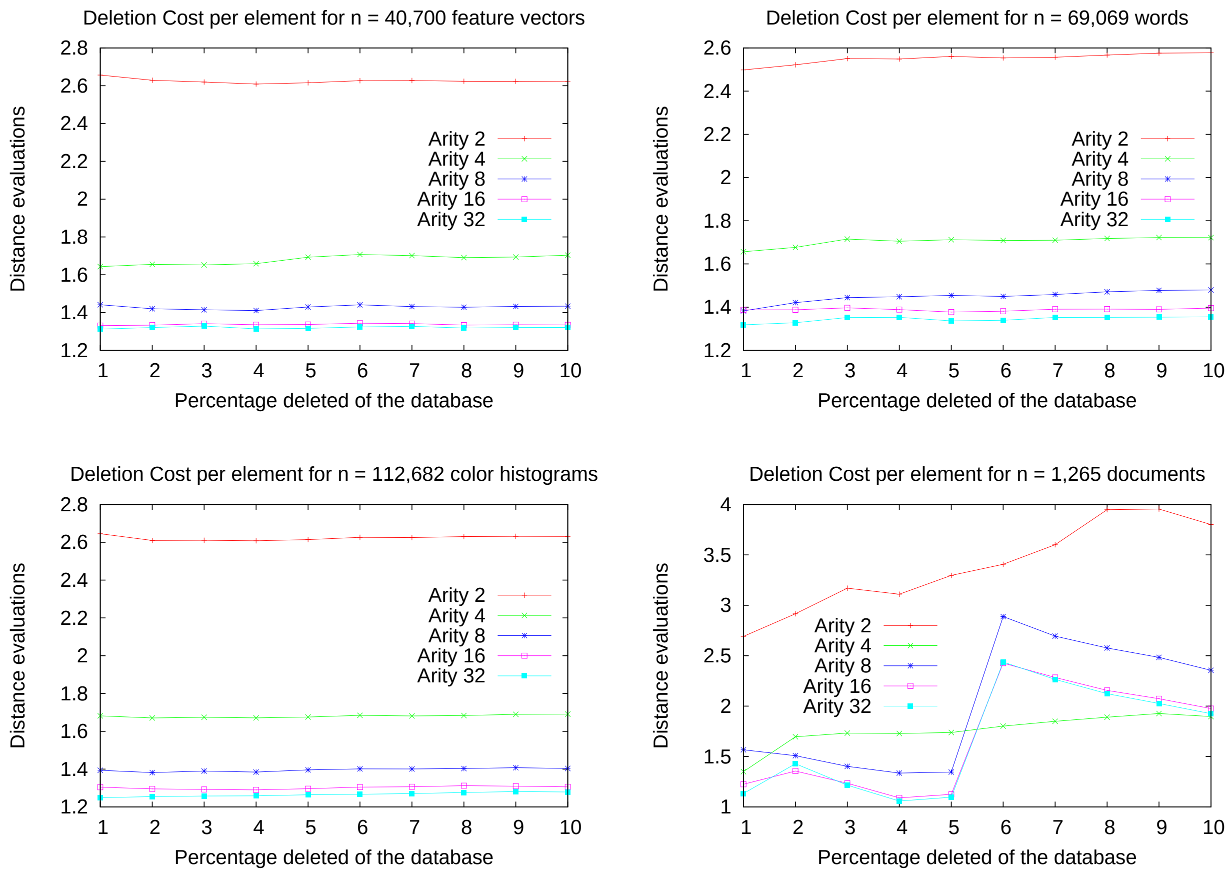


Figure 40: Deletion cost per element using ghost hyperplanes, replacing by a neighbor.

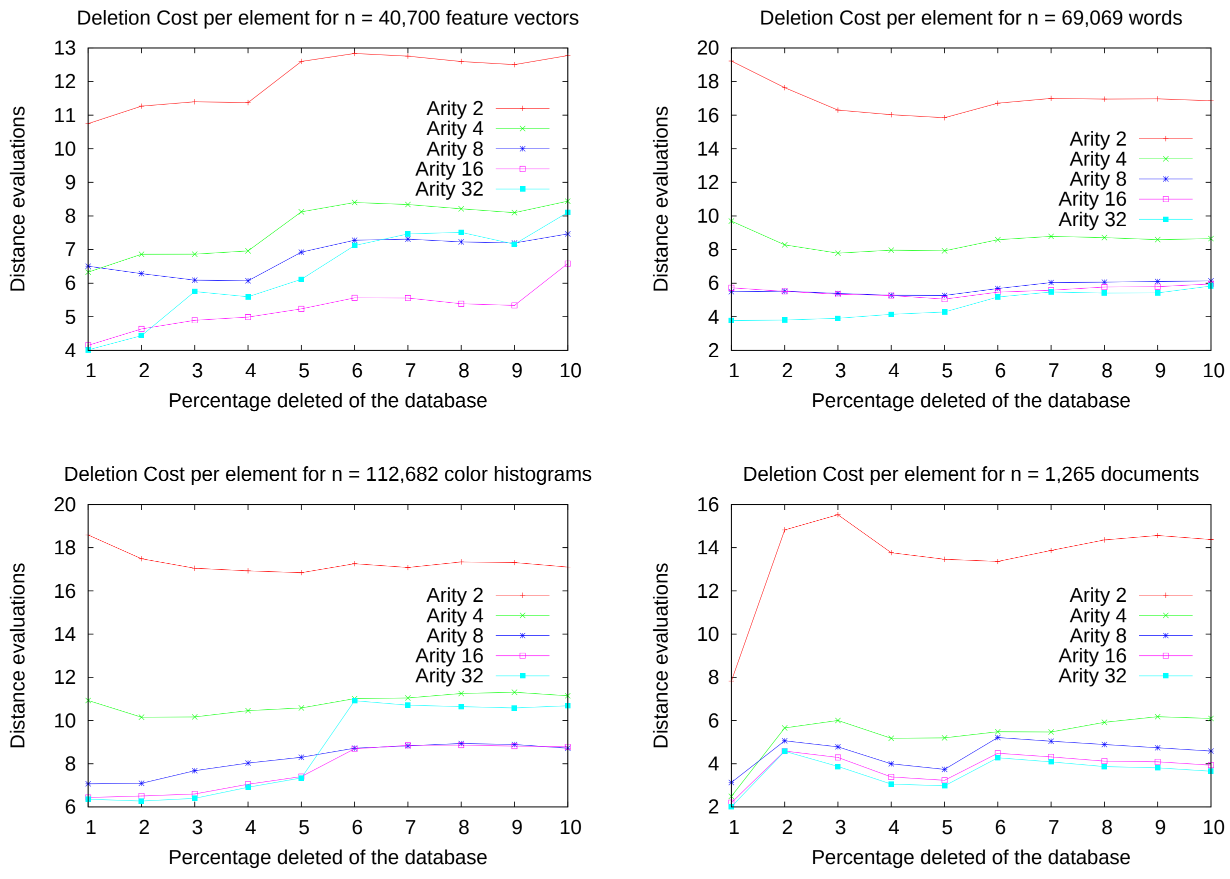


Figure 41: Deletion cost per element using ghost hyperplanes, replacing by the nearest element.

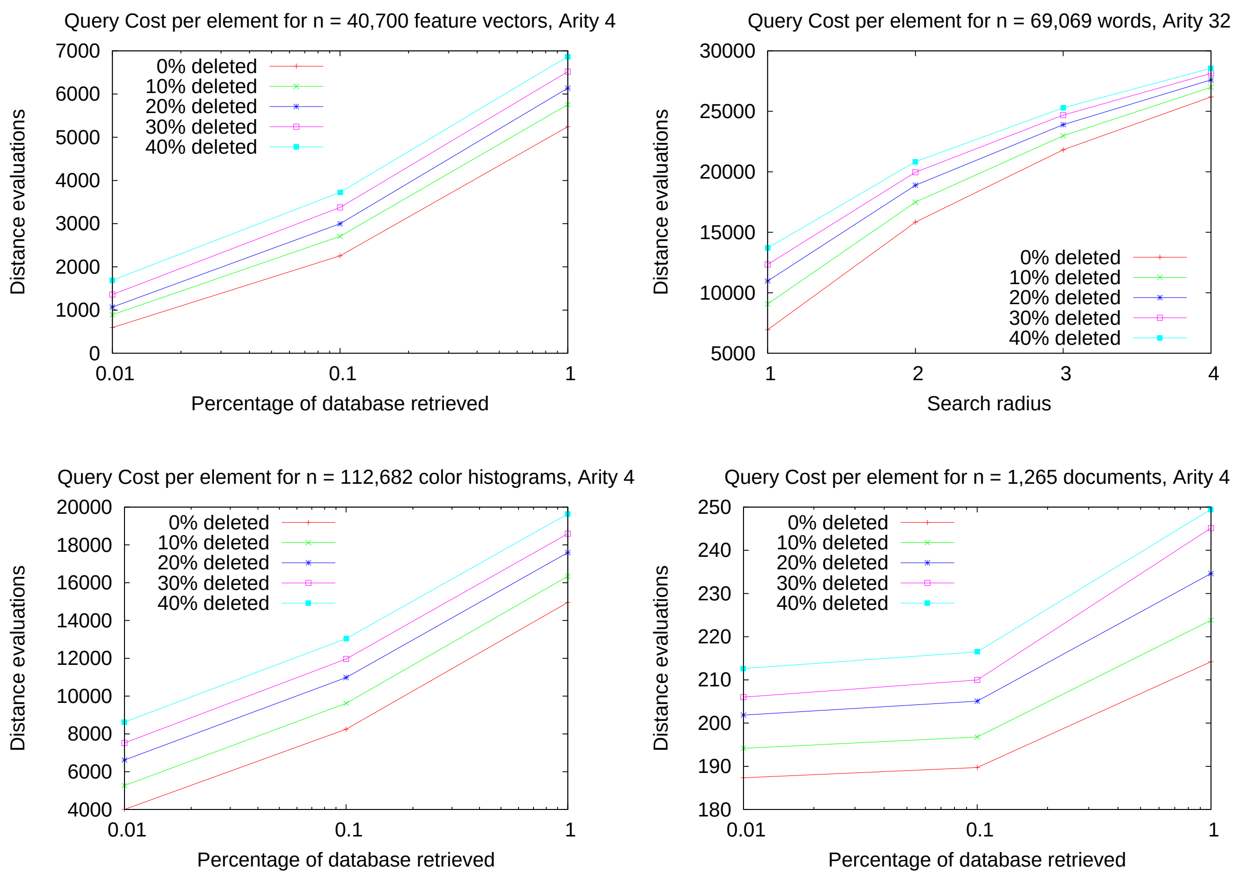


Figure 42: Search costs using ghost hyperplanes and replacing by a leaf. The search is done over half of the set.

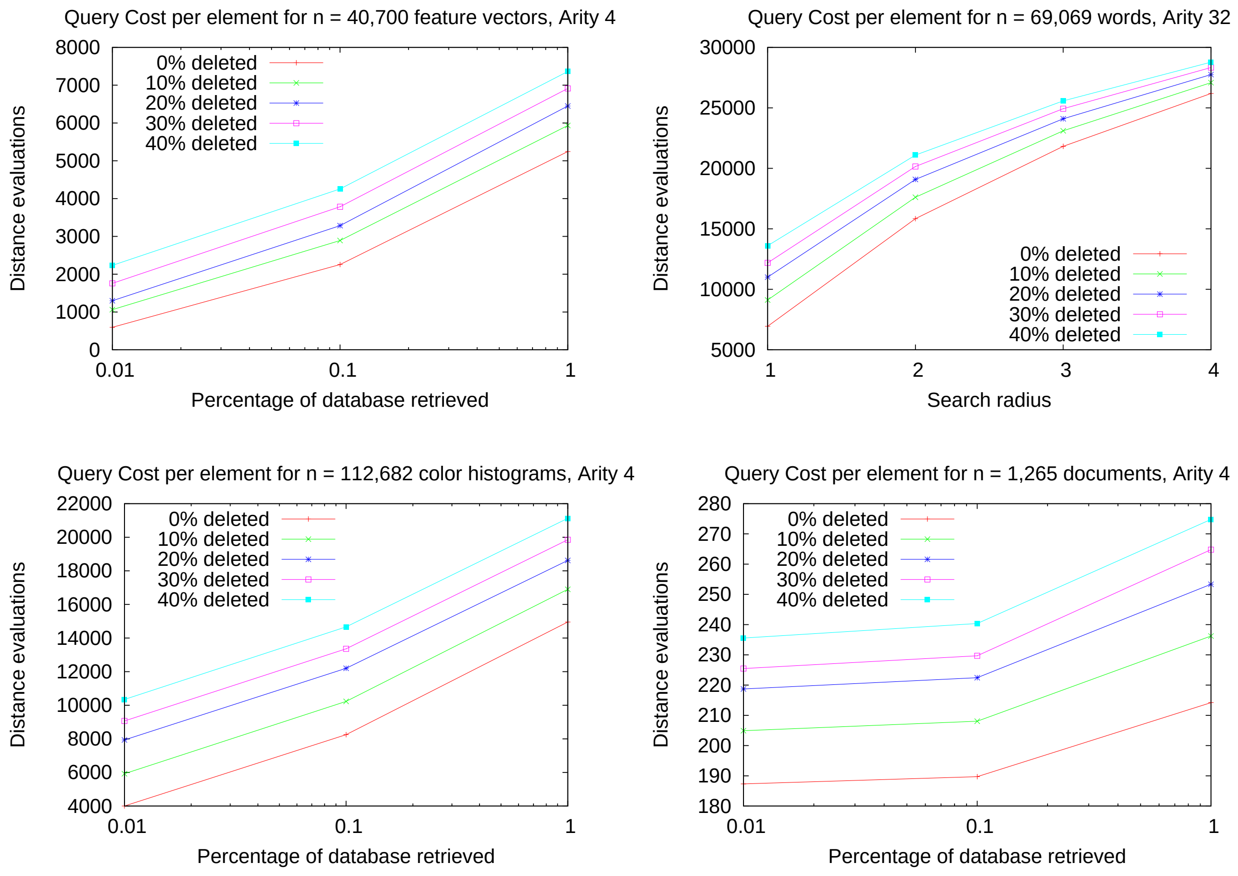


Figure 43: Search costs using ghost hyperplanes and replacing by a neighbor. The search is done over half of the set.

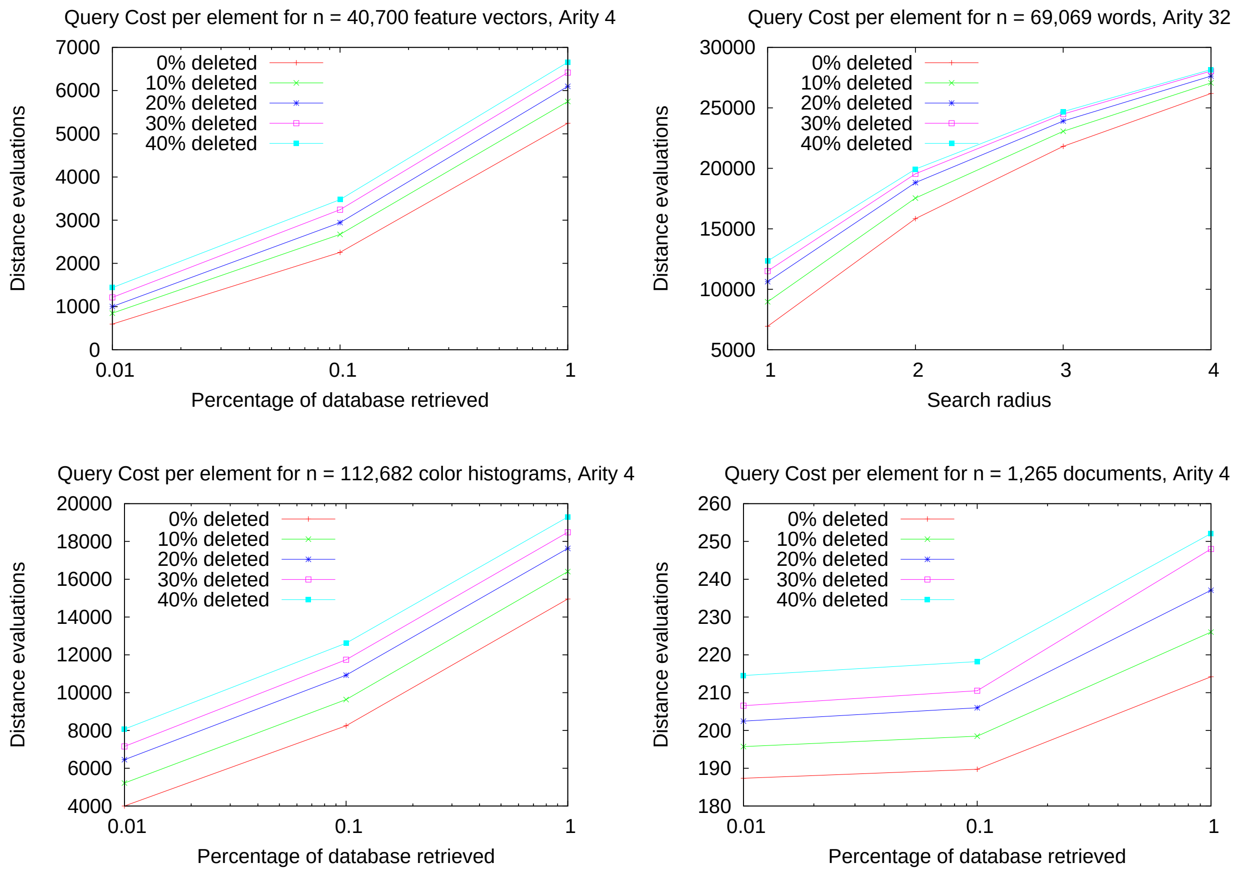


Figure 44: Search costs using ghost hyperplanes and replacing by the nearest element. The search is done over half of the set.

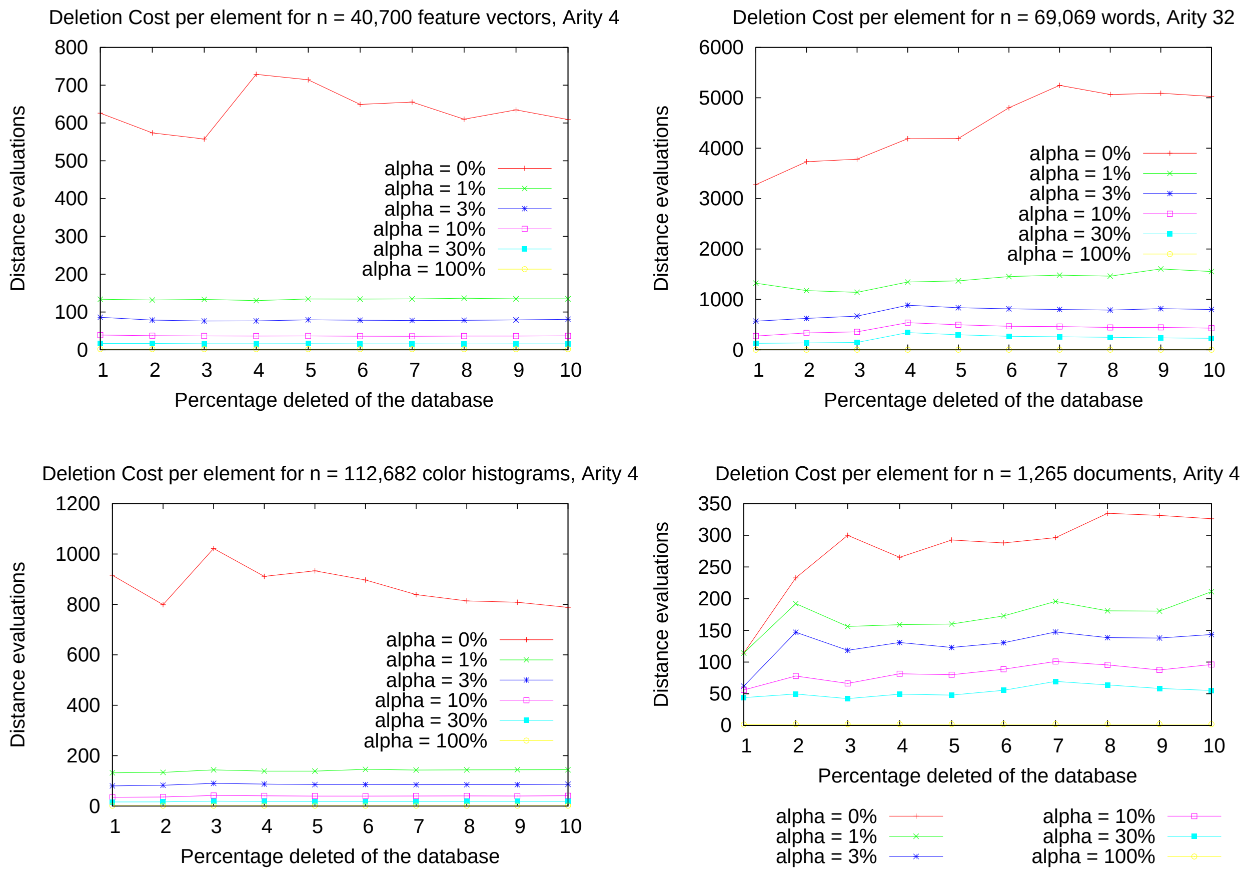


Figure 45: Deletion cost per element working with ghost hyperplanes and replacing by a leaf, for different values of α.

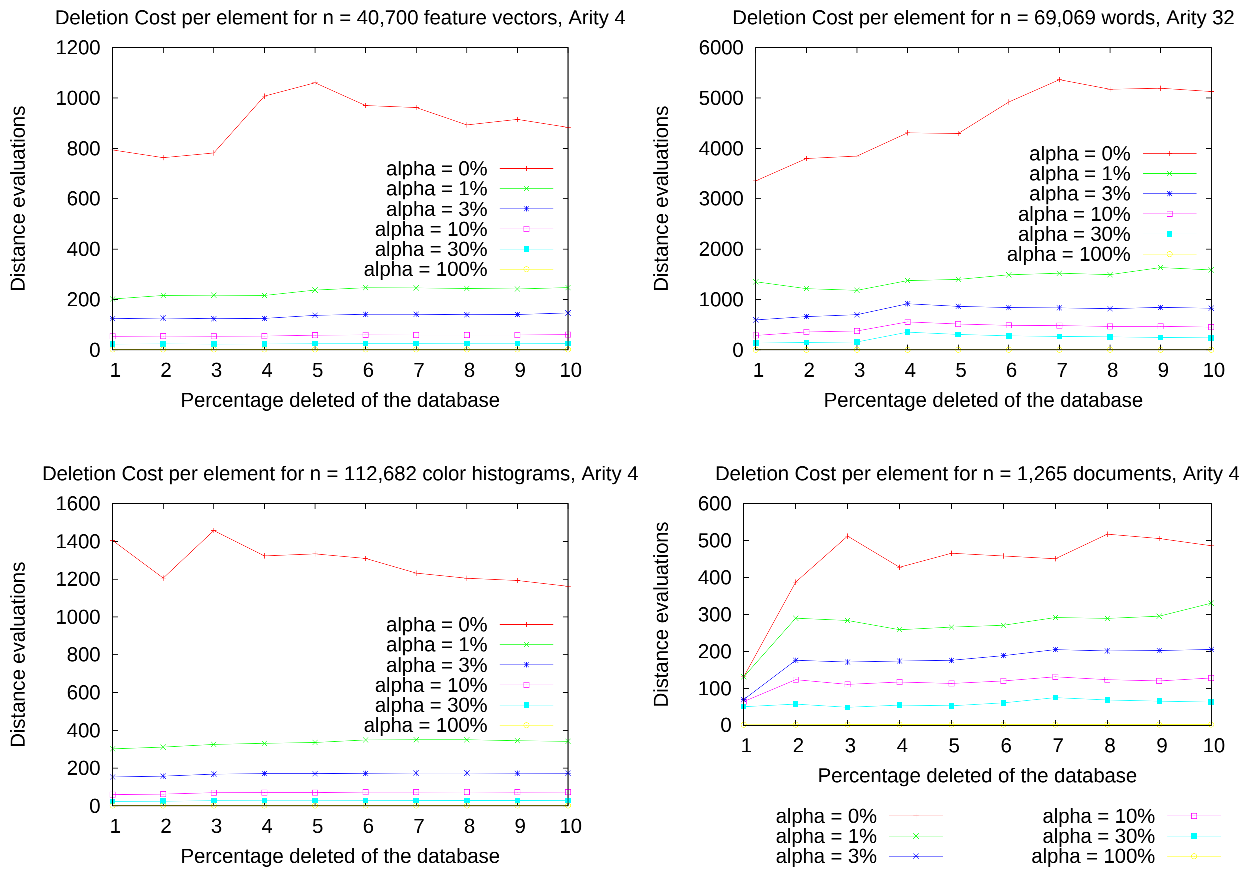


Figure 46: Deletion cost per element working with ghost hyperplanes and replacing by a neighbor, for different values of α.

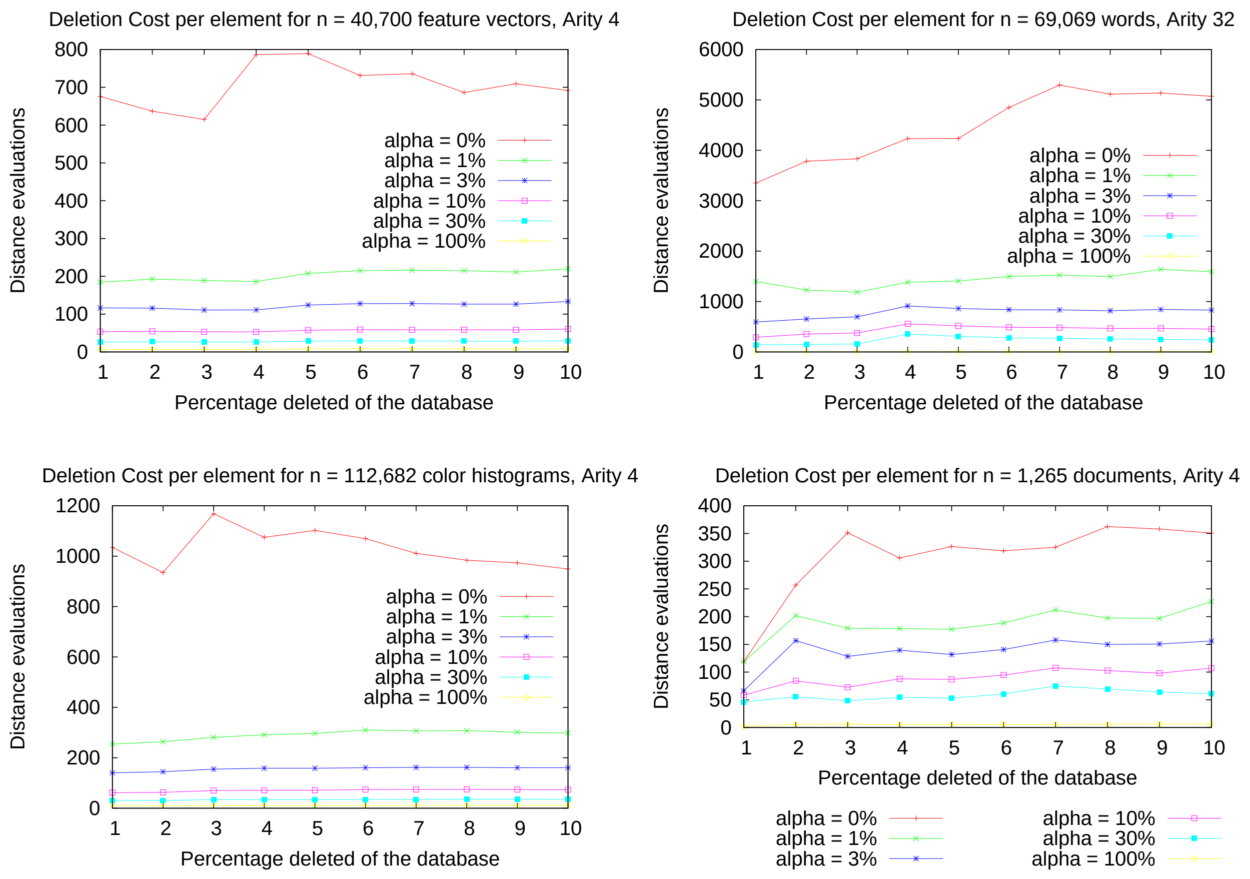


Figure 47: Deletion cost per element working with ghost hyperplanes and replacing by the nearest element, for different values of α.

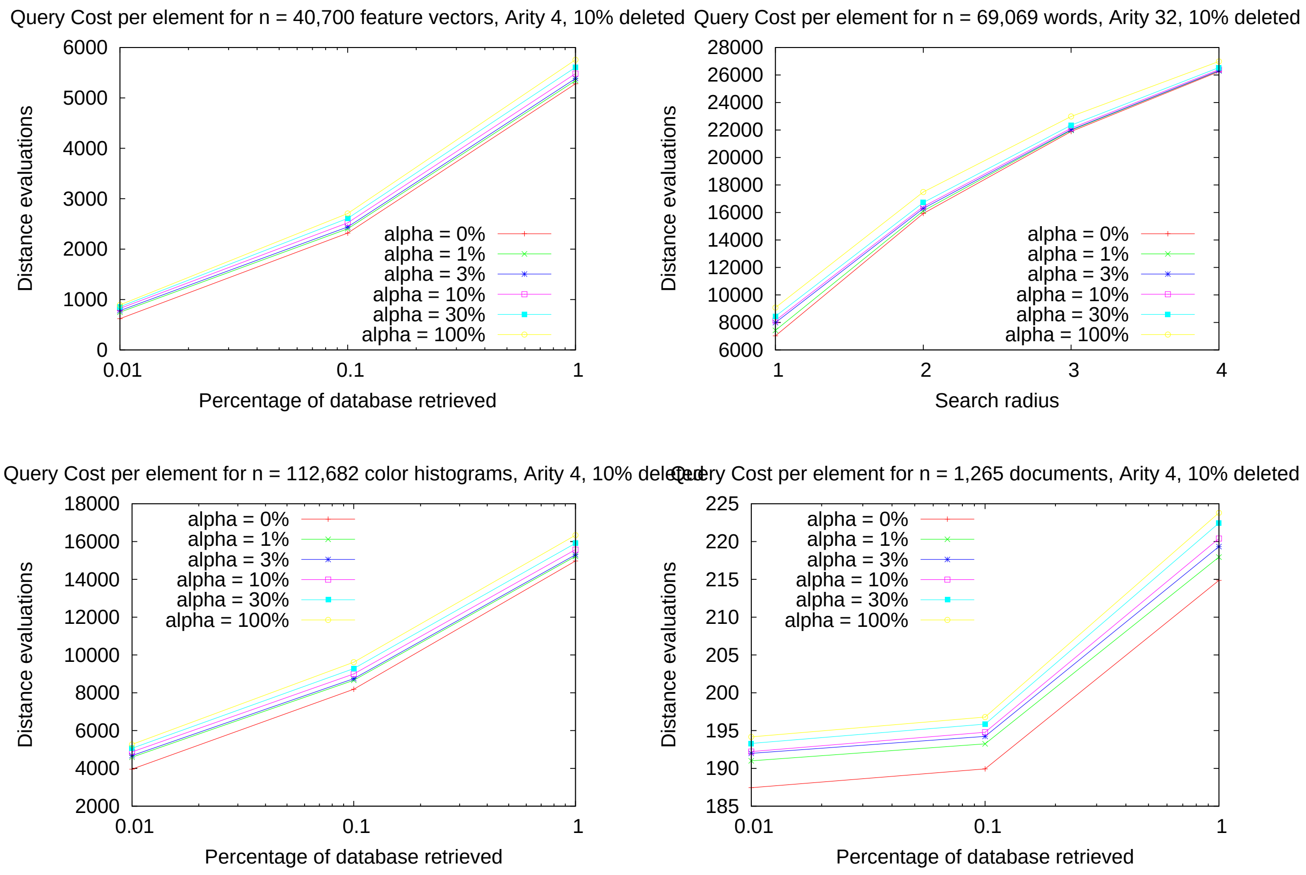


Figure 48: Search costs combining ghost hyperplanes that replace by a leaf, with a fraction α of ghost hyperplanes allowed, for 10% of elements deleted. The search is done over half of the set.

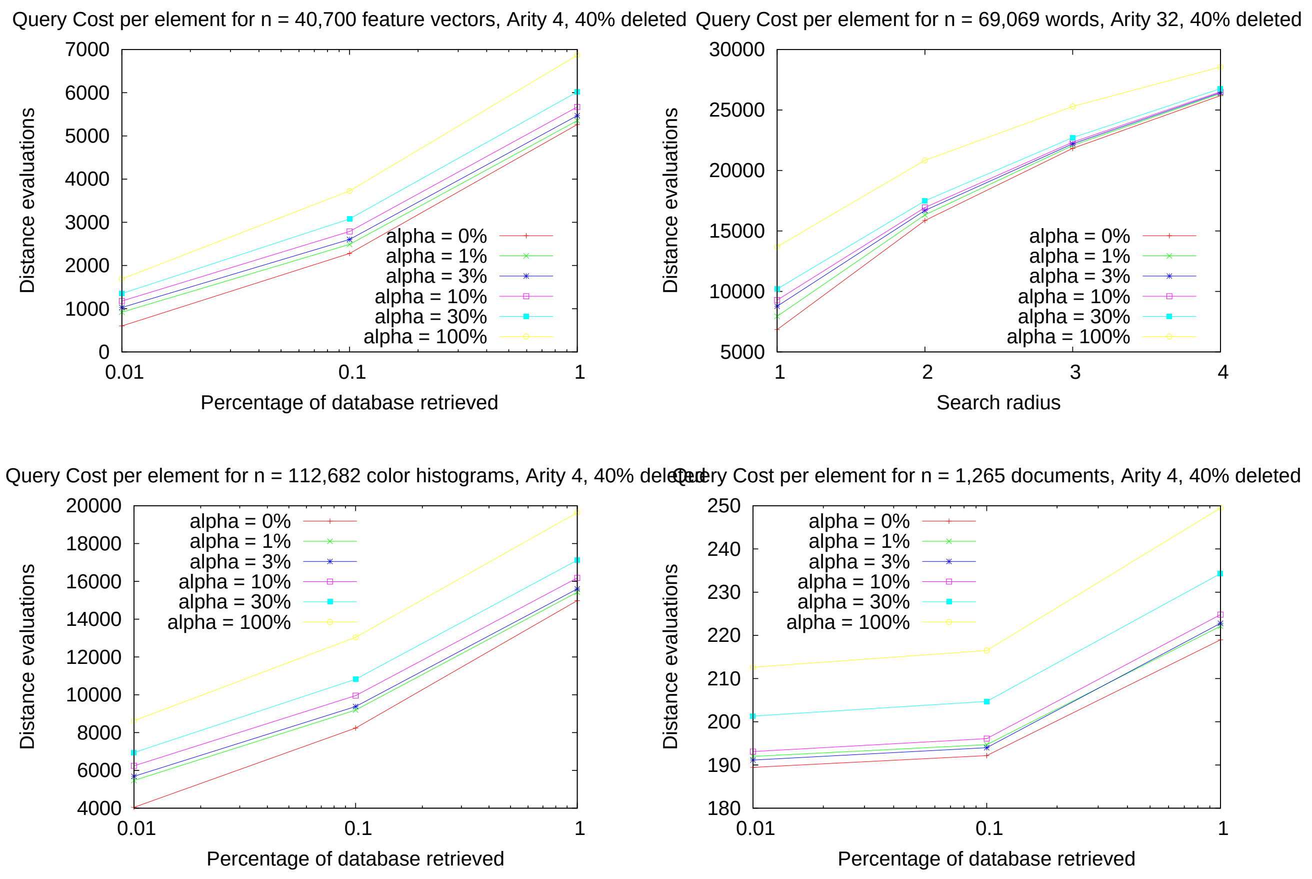


Figure 49: Search costs combining ghost hyperplanes that replace by a leaf, with a fraction α of ghost hyperplanes allowed, for 40% of elements deleted. The search is done over half of the set.

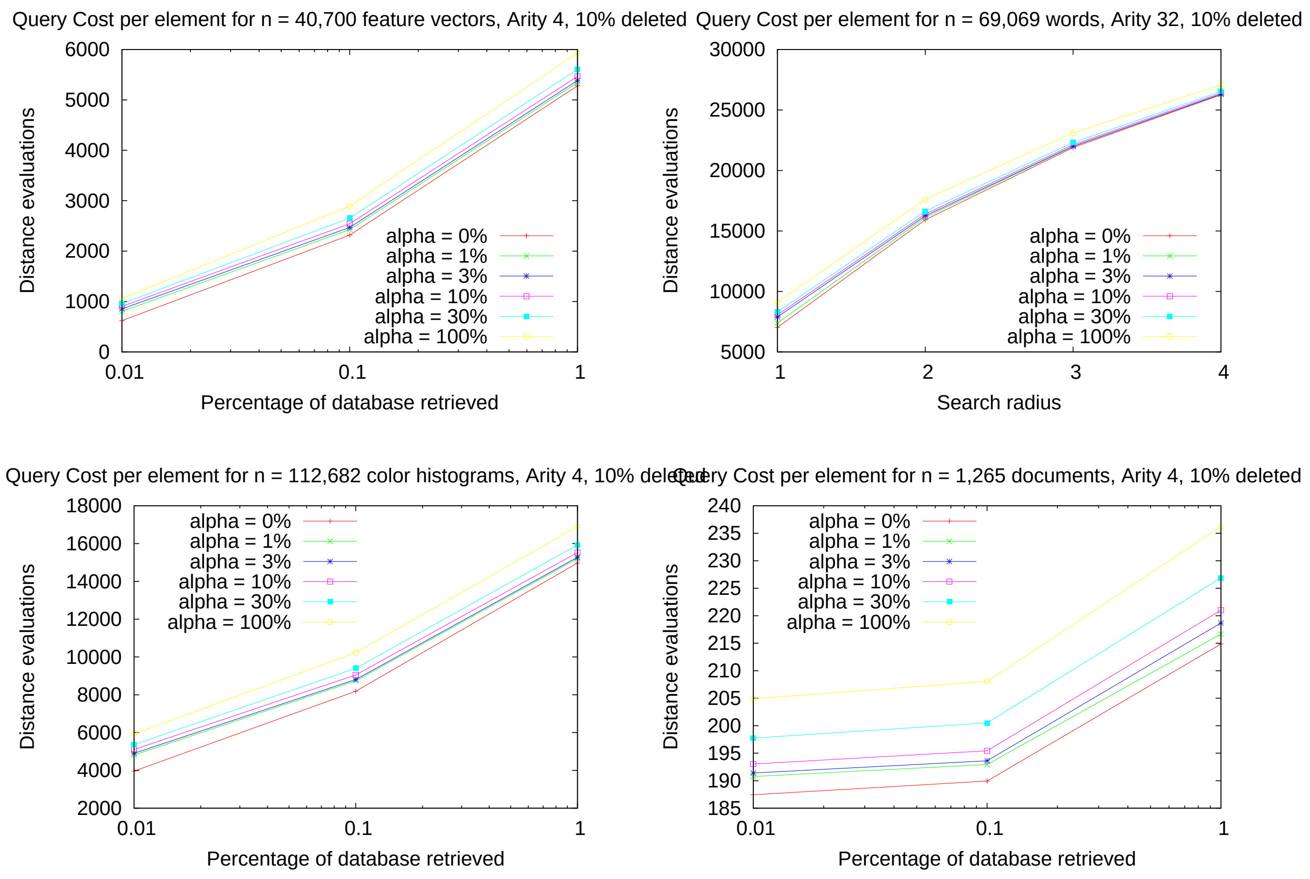


Figure 50: Search costs combining ghost hyperplanes that replace by a neighbor, with a fraction α of ghost hyperplanes allowed, for 10% of elements deleted. The search is done over half of the set.

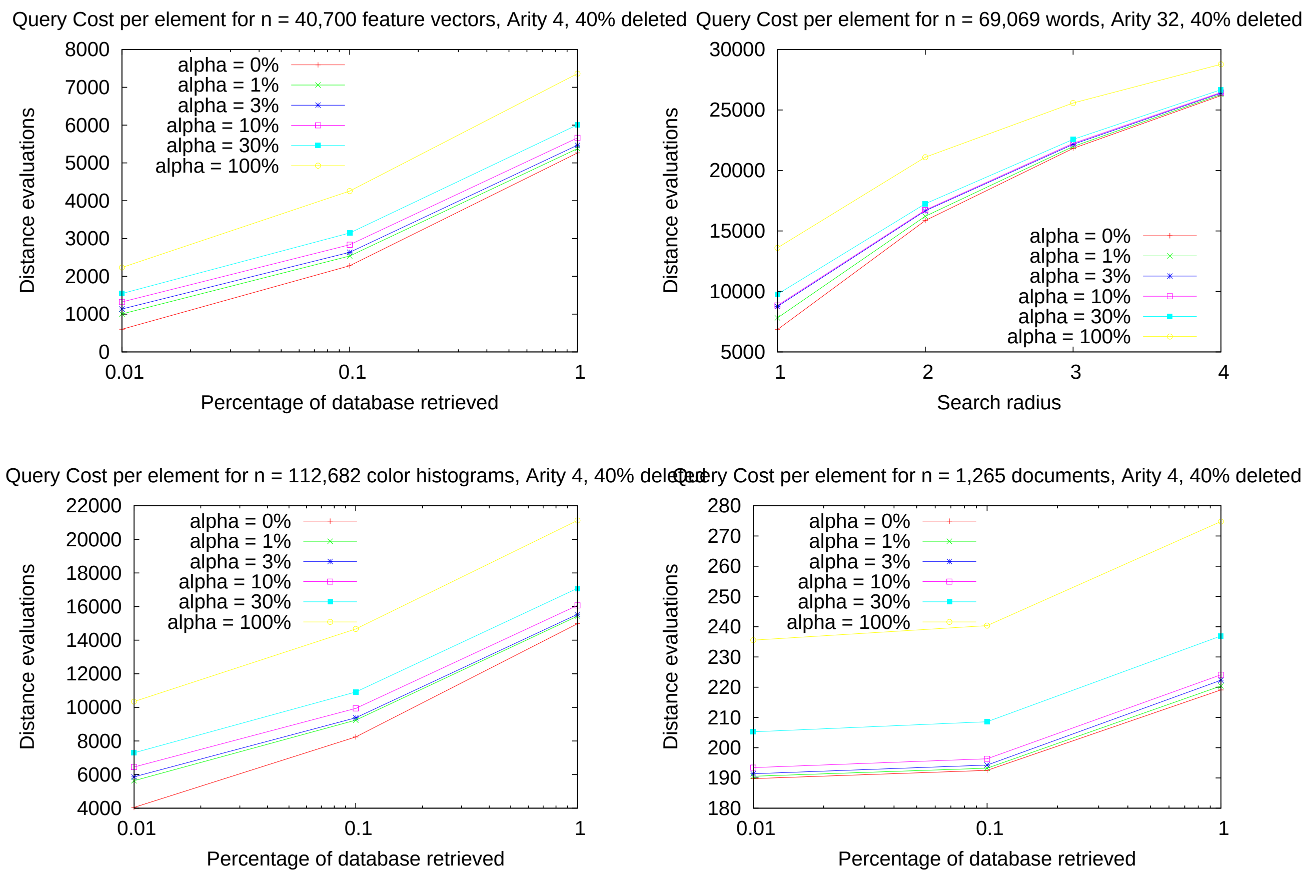


Figure 51: Search costs combining ghost hyperplanes that replace by a neighbor, with a fraction α of ghost hyperplanes allowed, for 40% of elements deleted. The search is done over half of the set.

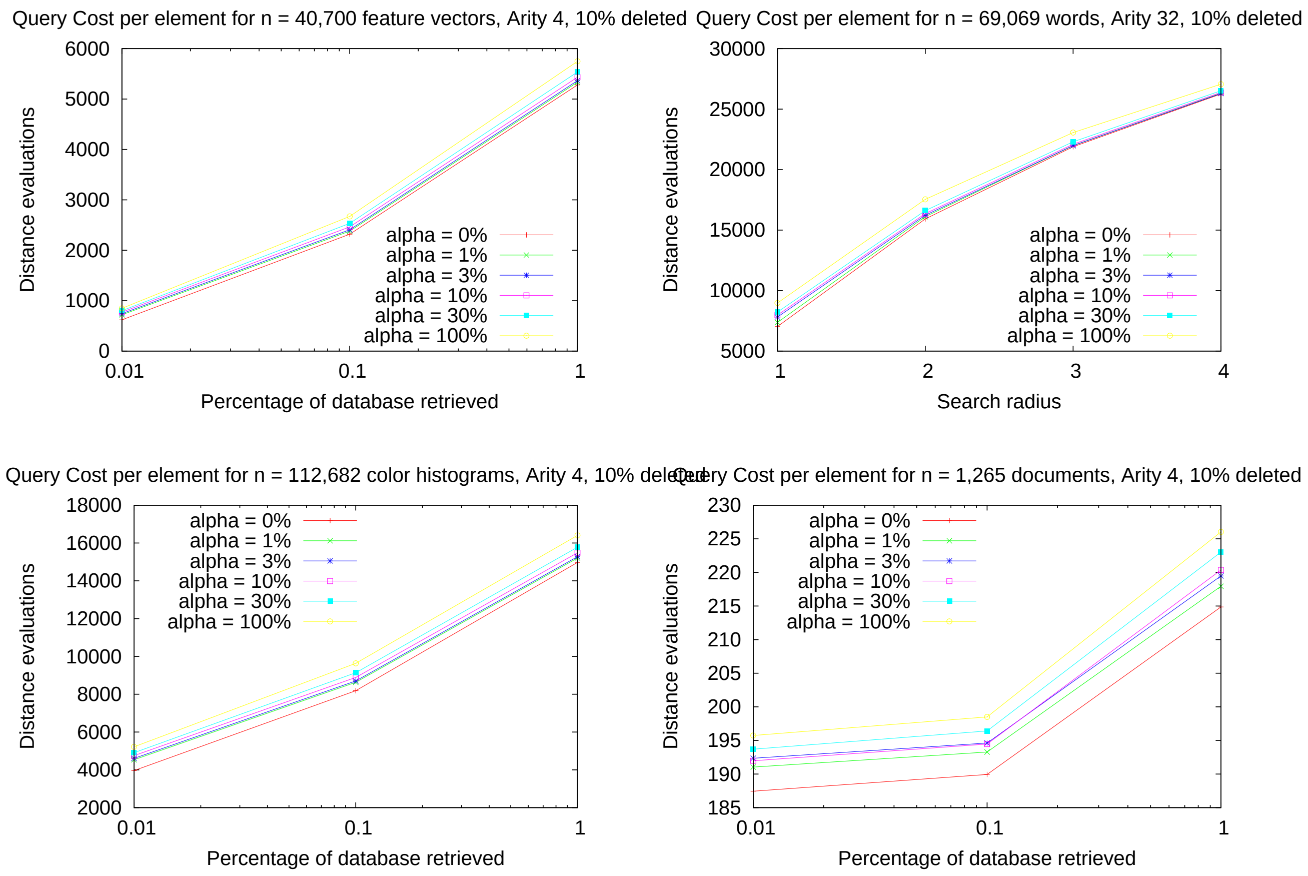
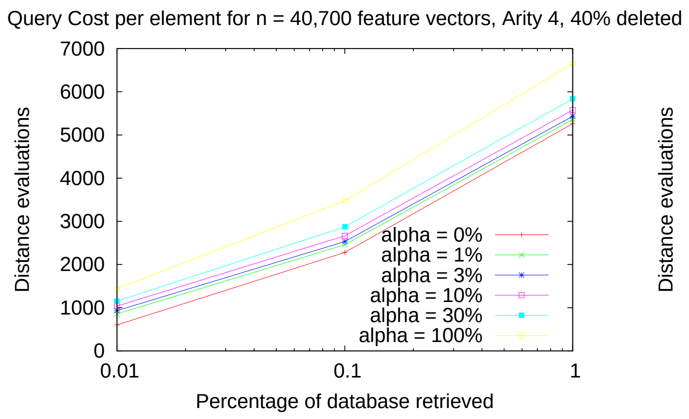
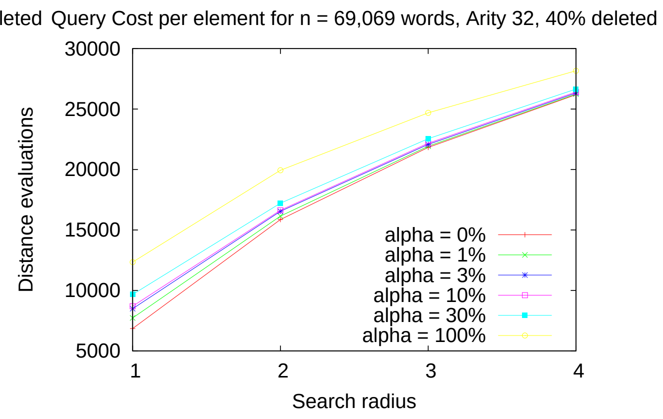
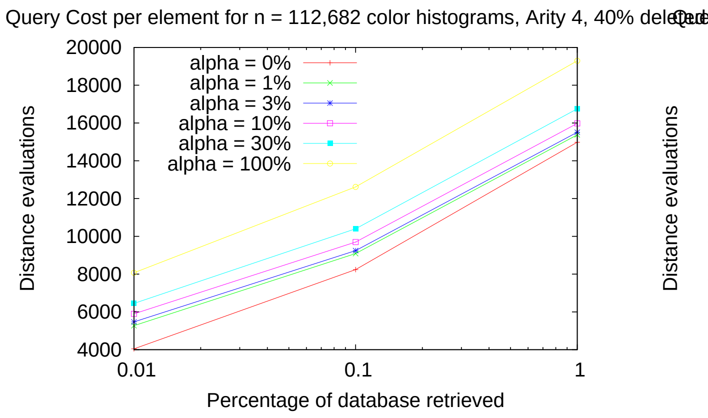


Figure 52: Search costs combining ghost hyperplanes that replace by the nearest element, with a fraction α of ghost hyperplanes allowed, for 10% of elements deleted. The search is done over half of the set.







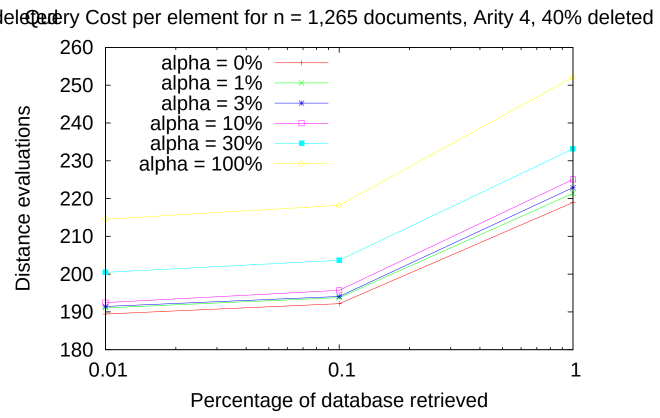


Figure 53: Search costs combining ghost hyperplanes that replace by the nearest element, with a fraction α of ghost hyperplanes allowed, for 40% of elements deleted. The search is done over half of the set.