Distal Dynamic Spatial Approximation Forest

Edgar Chávez[1], María E. Di Genaro[2], Nora Reyes[2], and Patricia Roggero[2]

[1] Centro de Investigación Científica y de Educación Superior de Ensenada, México
elchavez@cicese.mx
[2] Departamento de Informática, Universidad Nacional de San Luis, Argentina
{mdigena,nreyes,proggero}@unsl.edu.ar

Abstract. Querying large datasets by proximity, using a distance under the metric space model, has a large number of applications in multimedia, pattern recognition, statistics, etc. There is an ever growing number of indexes and algorithms for proximity querying, however there is only a handful of indexes able to perform well without user intervention to select parameters. One of such indexes is the *Distal Spatial Approximation Tree* (DiSAT) which is parameter-less and has demonstrated to be very efficient outperforming other approaches. The main drawback of the DiSAT is its static nature, that is, once built, it is difficult to add or to remove new elements. This drawback prevents the use of the DiSAT for many interesting applications.
In this paper we overcome this weakness. We use a standard technique, the Bentley and Saxe algorithm, to produce a new index which is dynamic while retaining the simplicity and appeal for practitioners of the DiSAT. In order to improve the DiSAF performance, we do not attempt to directly apply the Bentley and Saxe technique, but we enhance its application by taking advantage of our deep knowledge of the DiSAT behavior.

Keywords: similarity search, dynamism, metric spaces, non-conventional databases

1 Introduction

The metric space approach has become popular in recent years to handle the various emerging databases of complex objects, which can only be meaningfully searched for by similarity [3,11,12,5]. This approximation has applications in a vast number of fields. Some examples are non–traditional databases, text searching, information retrieval, machine learning and classification, image quantization and compression, computational biology, and function prediction. These problems can be mapped into a *metric space model* [3] as a metric database. That is, there is a universe $\mathbb{X}$ of objects, and a non negative real valued distance function $d : \mathbb{X} \times \mathbb{X} \longrightarrow \mathbb{R}^+ \cup \{0\}$ defined among them. This distance satisfies the three axioms that make the set a *metric space*: *strict positiveness* ($d(x, y) \geq 0$ and $d(x, y) = 0 \Leftrightarrow x = y$), *symmetry* ($d(x, y) = d(y, x)$), and *triangle inequality* ($d(x, z) \leq d(x, y) + d(y, z)$). We have a finite *database* $\mathbb{U} \subseteq \mathbb{X}$, $|\mathbb{U}| = n$, which is a subset of the universe.

Thereby, "proximity" or "similarity" searching is the problem of looking for objects in a dataset, that are "close" or "similar enough" to a given query object, under a certain (expensive to compute in time and/or resources) distance. The smaller the distance between two objects, the more "similar" they are. The database can be preprocessed to

build a *metric index*, that is, a data structure to speed up similarity searches. There are two typical similarity queries: *range queries* and k*-nearest neighbors queries*.

A large number of metric indices have flourished [3, 12, 11]. The *Distal Spatial Approximation Tree* (DiSAT) is an index based on dividing the search space and then approaching the query spatially. DiSAT is algorithmically interesting by itself, it has been shown that it gives an attractive trade-off between memory usage, construction time, and search performance. Besides, the great advantage of DiSAT compared to other indices is that it does not require any parameter tuning. However, DiSAT is a static index, that is, the index has to be rebuilt from scratch or it requires an expensive updating when the set of indexed objects undergoes insertions or deletions.

Although for some applications a static scheme may be acceptable, many relevant ones do require dynamic capabilities. Actually, in many cases it is sufficient to support insertions, such as in digital libraries and archival systems, versioned and historical databases, and several other scenarios where objects are never updated or deleted. In this paper we introduce a new dynamic version of DiSAT, by using the *Bentley-Saxe method* (BS)[1]. This method allows to transform a static index into a dynamic one, if on this index the search problem is *decomposable*. In [8] some static indexes are analyzed in combination with the BS method, obtaining certains acceptable results, but DiSAT in a static scenario has shown to outperform all these index. Now, we are focused only on supporting insertion and range searches, and we left deletions, k-NN searches and other improvements as future works.

The rest of this paper is organized as follows. In Section 2 we describe some basic concepts, and the BS method. Next, in Section 3 we detail the *Distal Spatial Approximation Trees* (DiSAT), and some notions of its close relatives: *Spatial Approximation Trees* (SAT) and the *Dynamic Spatial Approximation Trees* (DSAT). Section 4 introduces our dynamic variant of DiSAT. In Section 5 we show the experimental evaluation of our proposal. Finally, we draw some conclusions and future work directions in Section 6.

2 Previous Concepts

The metric space model can be formalized as follows. Let $\mathbb{X}$ be a universe of *objects*, with a nonnegative *distance function* $d : \mathbb{X} \times \mathbb{X} \longrightarrow \mathbb{R}^+$ 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 ($d(x, z) \leq d(x, y) + d(y, z)$). We handle a finite *dataset* $\mathbb{U} \subseteq \mathbb{X}$, which can be preprocessed (to build an index). Later, given a new object from $\mathbb{X}$ (a *query* $q \in \mathbb{X}$), we must retrieve all similar elements found in $\mathbb{U}$. There are two typical queries of this kind:

Range query: Retrieve all elements in $\mathbb{U}$ within distance r to q.
k*-nearest neighbors query (*k*-NN):* Retrieve the k closest elements to q in $\mathbb{U}$.

In this paper we are devoted to range queries. Nearest neighbor queries can be rewritten as range queries in an optimal way [6, 7], so we can restrict our attention to range queries. The distance is assumed to be expensive to compute. 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 $|\mathbb{U}| = n$ objects, queries can be trivially answered by performing n distance evaluations.

There exist a number of methods to preprocess the database in order to reduce the number of distance evaluations. (See [11,12,3] for more complete surveys.) Most of those structures work on the basis of discarding elements using the triangle inequality, and most use the classical divide-and-conquer approach. Algorithms to search in general metric spaces can be divided into two large areas: *pivot-based* and *clustering* algorithms. However, there are also algorithms that combine ideas from both areas.

Bentley and Saxe Method

The Bentley-Saxe method allows to transform a static index into a dynamic one, if on this index the search problem is *decomposable*, based on the binary representation of the integers [1]. A search problem with a query operation $\mathcal{Q}$ is *decomposable* if there exists an efficiently computable binary operator $\square$ satisfying the condition:

$$\mathcal{Q}(q, \mathbb{X}_1 \cup \mathbb{X}_2) = \square[\mathcal{Q}(q, \mathbb{X}_1), \mathcal{Q}(q, \mathbb{X}_2)]$$

where the $\square$ operation has to be associative and conmutative [1,8]. That is, the answer to a query on a dataset $\mathbb{X}_1 \cup \mathbb{X}_2$ has to be computed efficiently from the answer to a queries for each $\mathbb{X}_1$ and $\mathbb{X}_2$. In the particular case of range queries on $\mathbb{X}$, the $\square$ operation is the union of the sets obtained with the query operation $\mathcal{Q}$.

The main idea of BS method is to partition the indexed set $\mathbb{X}$ in certain subsets $\mathbb{X}_0, \mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_m$ (if $|\mathbb{X}| = n$, $m = \lfloor \log n \rfloor$) to reduce the size of the index of each subset that need to be rebuilt when an object is inserted or deleted [8]. This partition satisfies that $\bigcup_{0 \le i \le m} \mathbb{X}_i = \mathbb{X}$ and $\mathbb{X}_i \cap \mathbb{X}_j = \emptyset$ for $i \neq j$, and $|\mathbb{X}_i| = 2^i$. Then, the main data structure of BS is composed by a set of data strutures $T_0, T_1, \ldots, T_m$, where T_i is an empty data structure if $\mathbb{X}_i = \emptyset$, otherwise T_i is a static data structure that contains 2^i objects. Observe that for any value of n, there is a unique collection of subsets that must be non-empty. When a new object is inserted into the index, the algorithm proceeds with the same principle used for incrementing a binary counter. At query time, the search is solved independently by searching on each non-empty T_i and then the results of all individual searches are combined.

3 Distal Spatial Approximation Trees

The *Distal Spatial Approximation Tree* (DiSAT) [2] is a variant of the *Spatial Approximation Tree* (*SAT*) [9], both are data structures aiming at approaching the query spatially by starting at the root and getting iteratively closer to the query navigating the tree. In both cases the trees are built as follows. An element a is selected as the root, and it is connected to a set of *neighbors* $N(a)$, defined as a subset of elements $x \in \mathbb{U}$ such that x is closer to a than to any other element in $N(a)$. The other elements (not in $N(a) \cup \{a\}$) are assigned to their closest element in $N(a)$. Each element in $N(a)$ is recursively the root of a new subtree containing the elements assigned to it. For each node a the covering radius is stored, that is, the maximum distance $R(a)$ between a and any element in the subtree rooted at a. The starting set for neighbors of the root a, $N(a)$ is empty. Therefore we can select *any* database element as the first neighbor. Once this element

is fixed the database is split in two halves by the hyperplane defined by proximity to a and the recently selected neighbor. Any element in the a side can be selected as the second neighbor. While the zone of the root (those database elements closer to the root than the previous neighbors) is not empty, it is possible to continue with the subsequent neighbor selection. The SAT considers the elements of $\mathbb{U} - \{a\}$ in increasing order of distance tho a, but DiSAT considers exactly the opposite order.

The main difference between them is that DiSAT tries to increase the separation between hyperplanes, which in turn decreases the size of the covering radius; the two parameters governing the performance of these trees. The performance improvement consists in selecting distal nodes instead of the proximal nodes selected in the original algorithm. Considering an example of a metric database illustrated in Fig. 1, the Fig. 2 shows the SAT (Fig. 2(a)) and the DiSAT (Fig. 2(b)) obtained by selecting p_6 as the tree root. In both cases we also depict the covering radii for the neighbors of the tree root. It is possible to obtain completely different trees (SATs or DiSATs) if we select different roots, and each tree probably may have different search costs.

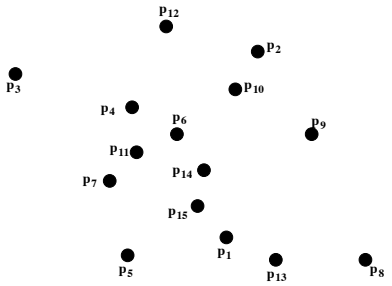
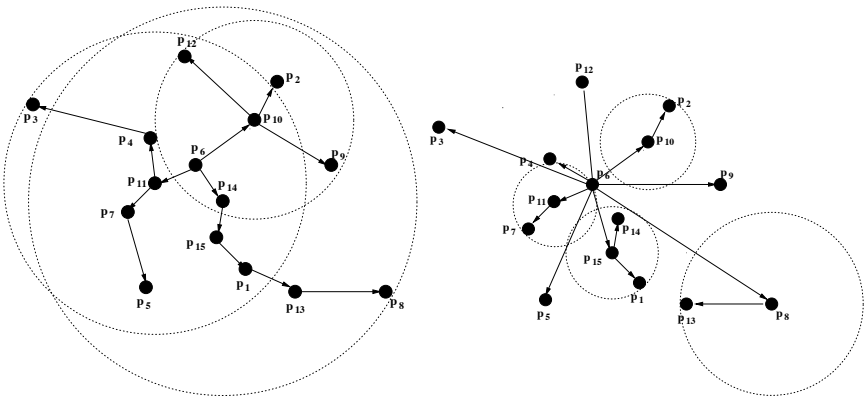


Fig. 1. Example of a metric database in $\mathbb{R}^2$.

Algorithm 1 gives a formal description of the construction of DiSAT. Range searching is done with the procedure described in Algorithm 2. This process is invoked as `RangeSearch(`a`,`q`,`r`,`$d(a,q)$`)`, where a is the tree root, r is the radius of the search, and q is the query object. One key aspect of DiSAT (SAT too) is that a greedy search will find all the objects previously inserted. For a range query of q with radius r, and being c the closest element between $\{a\} \cup N(a) \cup A(a)$ and $A(a)$ the set of the ancestors of a, the same greedy search is used entering all the nodes $b \in N(a)$ such that $d(q,b) \leq d(q,c) + 2r$ because any element $x \in (q,r)_d$, can differ from q by at most r at any distance evaluation, so it could have been inserted inside any of those b nodes [12, 9]. In the process, all the nodes x founded close enough to q are reported.

Dynamic Spatial Approximation Tree

The *Dynamic Spatial Approximation Tree* (DSAT) [10] is an online version of the SAT. It is designed to allow dynamic insertions and deletions without increasing the construction cost with respect to the SAT. A very surprising and unintended feature of the DSAT is the boosting in the searching performance. The DSAT is faster in searching even if at construction it has less information than the static version of the index. For the DSAT



(a) SAT. (b) DiSAT.

Fig. 2. Example of the SAT and DiSAT obtained if p_6 were the root.

the database is unknown beforehand and the objects arrive to the index at random as well as the queries. A dynamic data structure cannot make strong assumptions about the database and will not have statistics about all the database.

4 Our Proposal: Distal Dynamic Spatial Approximation Forest

As we mention previously, the BS method can be applied on any static data structure to transform it into a dynamic one. We select the DiSAT because it has shown that is a very competitive index and it do not need to set any parameter. Most of the more efficient indexes that need the setting of the value of at least one parameter can become inefficient at a bad choice of it.

In this particular case each T_i that considers the BS method is a tree, particularly a DiSAT, so our new dynamic data structure is named *Distal Dynamic Spatial Approximation Forest* (DiSAF), because we have a *forest* of DiSATs. The i-th DiSAT in the forest will have 2^i elements.

Considering the example illustrated in Fig. 1, the Fig. 3 and Fig. 4 illustrate the two dynamic data structures, based on spatial approximation, obtained by inserting the objects $p_1, \cdots, p_{15}$ one by one: DSAT with maximum arity of 6 (Fig. 3) and DiSAF (Fig. 4). In the DSAT the root will be p_1, because it is the first element arrived. On the other hand, as we have 15 elements, DiSAF will build four DiSATs: T_0, T_1, T_2, and T_3. As it is aforementioned, each T_i will have 2^i elements. As the insertion order is from p_1 to p_{15}, the final situation will have: T_0 with the dataset $\{p_{15}\}$, T_1 with $\{p_{13}, p_{14}\}$, T_2 with $\{p_9, \ldots, p_{12}\}$, and T_3 with $\{p_1, \ldots p_8\}$. We also depict the covering radii for the neighbors of the tree roots, some covering radii are equal to zero. On one hand, it is possible to obtain different DSATs if we consider different maximum arities or different

Algorithm 1 Process to build a DiSAT for $\mathbb{U} \cup \{a\}$ with root a.

```
BuildTree(Node a, Set of nodes U)
1.  N(a) ← ∅     /* neighbors of a */
2.  R(a) ← 0     /* covering radius */
3.  For v ∈ U in increasing distance to a Do
4.     R(a) ← max(R(a), d(v,a))
5.     If ∀b ∈ N(a),  d(v,a) < d(v,b) Then
6.        N(a) ← N(a) ∪ {v}
7.  For b ∈ N(a) Do S(b) ← ∅
8.  For v ∈ U − N(a) Do
9.     c ← argmin_{b∈N(a)} d(v,b)
10.    S(c) ← S(c) ∪ {v}
11. For b ∈ N(a) Do BuildTree(b, S(b))
```

Algorithm 2 Searching of q with radius r in a DiSAT with root a.

```
RangeSearch(Node a, Query q, Radius r, Distance d_min)
1. If d(a,q) ≤ R(a) + r Then
2.    If d(a,q) ≤ r Then Report a
3.    d_min ← min {d(c,q),  c ∈ N(a)} ∪ {d_min}
4.    For b ∈ N(a) Do
5.       If d(b,q) ≤ d_min + 2r Then
6.          RangeSearch(b, q, r, d_min)
```

insertion orders, and they will likely have different search costs. On the other hand, as DiSAF has not any parameter, the only way to obtain different forests is by considering different insertion orders.

The insertion process of a new element x in a DiSAF is described in the Algorithm 3. Initially, the DiSAF has an only DiSAT $T_0 = null$. Then, the index can be built via succesive insertions. **`Retrieve`**`(Tree T)` return all the elements that compose the tree T. The range search process is detailed in the Algorithm 4.

5 Experimental Results

For the empirical evaluation of the indices we consider three widely different metric spaces from the SISAP Metric Library (`www.sisap.org`) [4].

Dictionary: 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[3]. Any quadratic form can be used as a distance; we chose Euclidean as the simplest meaningful distance.

NASA images: a set of 40,700 20-dimensional feature vectors, generated from images downloaded from NASA[4]. The Euclidean distance is used.

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

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

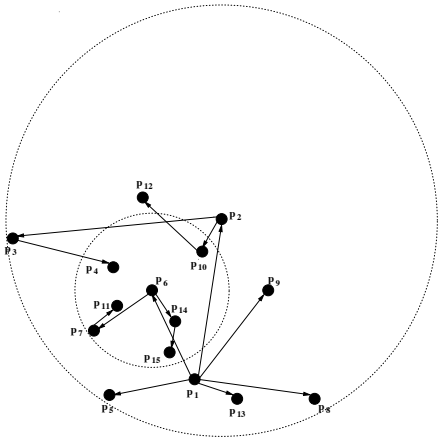


Fig. 3. Example of the DSAT with maximum arity of 6, inserting from p_1 to p_{15}.

Algorithm 3 Insertion of a new element x in a DiSAF with at most m trees.

```
Insert(Element x)
1. S ← ∅, k ← min_{0≤i≤m} i, such that T_i = null
2. For i from 0 to k − 1 Do
3.    S ← S ∪ Retrieve (T_i)  /* retrieve all the elements of T_i */
4.    T_i ← null  /* T_i is a new empty tree */
5. T_k ← BuildTree(x, S)
6. If k = m Then
7.    T_{k+1} ← null, m ← k + 1
```

When we evaluate construction costs, we build the index with the complete database. If the index is dynamic, the construction is made by inserting one by one the objects, otherwise the index knows all the elements beforehand. In order to evaluate the search performance of the indexes, we build the index with the 90% of the database elements and we use the remaining 10%, randomly selected, as queries. So, the elements used as query objetcs are not in the index. We average the search costs of all these queries. All results are averaged over 10 index constructions with different datasets permutations.

We consider range queries retrieving on average $0.01\%, 0.1\%$ and 1% of the dataset. This corresponds to radii 0.051768, 0.082514 and 0.131163 for the Color Histograms; and 0.605740, 0.780000 and 1.009000 for the NASA images. The Dictionary 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. As we mention previously, given the existence of range-optimal algorithms for k-nearest neighbor searching [6, 7], we have not considered these search experiments separately.

We show the comparison between our dynamic DiSAF, the DSAT, and the static alternatives SAT and DiSAT. The source code of the different SAT versions (SAT and DSAT) is available at`www.sisap.org`. A final note in the experimental part is the arity parameter of the *DSAT* which is tunable and is the maximum number of neighbors

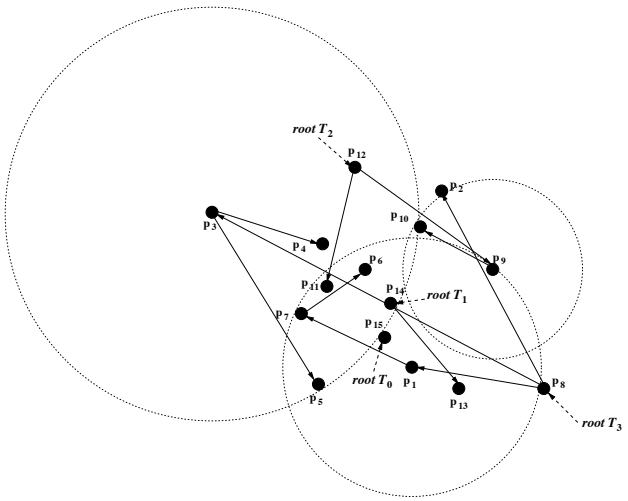


Fig. 4. Example of the DiSAF, inserting from p_1 to p_{15}.

Algorithm 4 Searching of q with radius r in a DiSAF with at most m trees.

```
RangeSearchNew(Query q, Radius r)
1. A ← ∅
2. For i from 0 to m − 1
3.   If T_i ≠ null Then
4.     Let x be the root of T_i
5.     A ← A ∪ RangeSearch(x,q,r,d(x,q))
6. Report A
```

of each node of the tree. In our experiments we used the arity suggested by authors in [10]. The Figure 5 illustrates the construction costs of the all indices, on the three metric spaces. As it can be seen, DiSAF is surpassed for the other three indexes, because it has to rebuild the trees too many times. On the other hand, DSAT do not make any reconstruction while it builds the tree via insertions. It has to be considered that SAT and DiSAT are built with all the elements known at the same time, not dynamically.

We analyze search costs in Figure 6. As it can be noticed, DiSAF surpasses DSAT in most of spaces. The only index that is always better than DiSAF is the DiSAT, but as we already mention it is static. Therefore, we can affirm that the heuristic of construction of DiSAT allows to surpasse in searches the other strategies used in SAT and DSAT. Besides, we have obtained a dynamic index that overcomes DSAT at searches and it do not have any parameter to tune.

6 Conclusions

We have presented a dynamic version of the DiSAT, which at this time is able of handling insertions without affecting significantly its search quality. Very few data structures for searching metric spaces are dynamic. Furthermore, we have shown that the

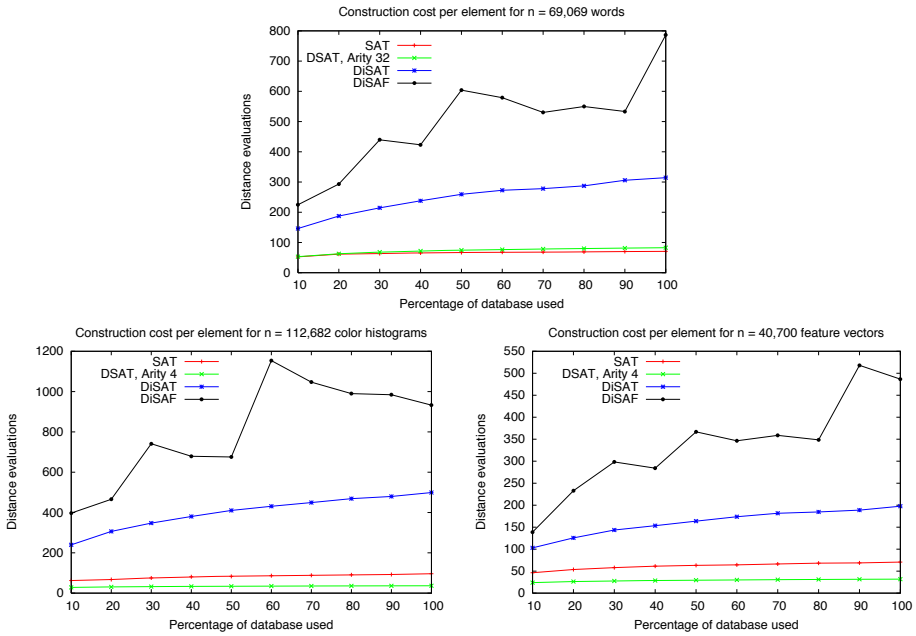


Fig. 5. Construction costs for the three metric spaces considered.

heuristic used in DiSAT and DiSAF to partition the metric space is better than that used in SAT and DSAT: distal nodes produce more compact subtrees, which in turn give more locality to the underlying partitions implicitly defined by the subtrees.

The DiSAT was a promising data structure for metric space searching, with several drawbacks that prevented it from being practical: high construction cost and inability to accommodate insertions and deletions. We have addressed one of these weaknesses. Despite of we have obtained worse construction costs, it is still possible to improve it, for example by providing a bulk-loading algorithm to create initially the DiSAF if we know aforehand a subset of elements, avoiding unnecessary rebuildings when we insert elements one by one, or with *lazy insertion* that do not always rebuild trees.

We are currently pursuing in the direction of making the DiSAF fully dynamic; that is, that it also supports deletions, and designing an efficient bulk-loading algorithm, which amortizes the insertion costs between several elements. Other topic of future work is to design a more efficient alternative of k-NN search that do not apply the basic solution of decomposable search, but that it applies a smart solution by taking advantage of all distances calculated in order to shrink, as soon as possible, the radius from q that encloses k elements.

References

1. Jon L. Bentley and James B. Saxe. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.
2. Edgar Chávez, Verónica Ludeña, Nora Reyes, and Patricia Roggero. Faster proximity searching with the distal sat. *Information Systems*, 59:15 – 47, 2016.

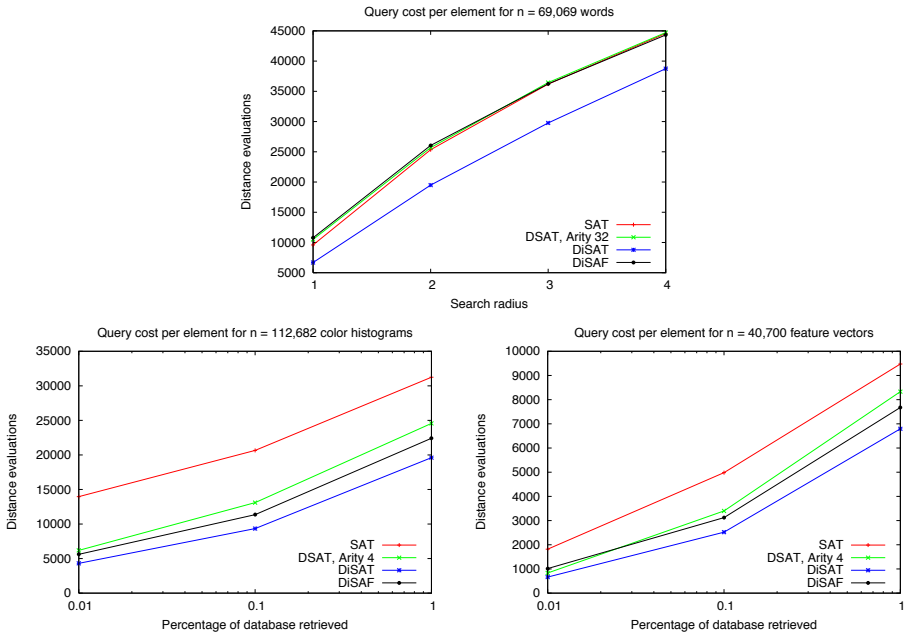


Fig. 6. Search costs for the three metric spaces considered.

3. Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.
4. Karina Figueroa, Gonzalo Navarro, and Edgar Chávez. Metric spaces library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.
5. Magnus Hetland. The basic principles of metric indexing. In Carlos Coello, Satchidananda Dehuri, and Susmita Ghosh, editors, *Swarm Intelligence for Multi-objective Problems in Data Mining*, volume 242 of *Studies in Computational Intelligence*, pages 199–232. Springer Berlin / Heidelberg, 2009.
6. Gísli R. Hjaltason and Hanan Samet. *Incremental Similarity Search in Multimedia Databases*. Number CS-TR-4199 in Computer science technical report series. Computer Vision Laboratory, Center for Automation Research, University of Maryland, 2000.
7. Gísli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, 2003.
8. Bilegsaikhan Naidan and Magnus Lie Hetland. Static-to-dynamic transformation for metric indexing structures (extended version). *Information Systems*, 45:48 – 60, 2014.
9. Gonzalo Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.
10. Gonzalo Navarro and Nora Reyes. Dynamic spatial approximation trees. *Journal of Experimental Algorithmics*, 12:1.5:1–1.5:68, June 2008.
11. Hanan 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.
12. Pavel Zezula, Giussepe Amato, Vlatislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.