# An Efficient Dynamic Version of the Distal Spatial Approximation Trees

Edgar Chávez[1], María E. Di Genaro[2], and Nora Reyes[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}@unsl.edu.ar

**Abstract.** Metric space indices make searches for similar objects more efficient in various applications, including multimedia databases and other repositories which handle complex and unstructured objects. Although there are a plethora of indexes to speed up similarity searches, the *Distal Spatial Approximation Tree* (DiSAT) has shown to be very efficient and competitive. Nevertheless, for its construction, we need to know all the database objects beforehand, which is not necessarily possible in many real applications. The main drawback of the DiSAT is that it is a static data structure. That means, once built, it is difficult to insert new elements into it. This restriction rules it out for many exciting applications. In this paper, we overcome this weakness. We propose and study a dynamic version of DiSAT that allows handling lazy insertions and, at the same time, improves its good search performance. Therefore, our proposal provides a good tradeoff between construction cost, search cost, and space requirement. The result is a much more practical data structure that can be useful in a wide range of database applications.

**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 and unstructured objects- On these kinds of databases, it is only meaningfully searching for similar objects [4, 11, 12, 6]. Similarity searches have 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, among others. These problems can be mapped into a *metric space model* [4] as a metric database. In this model, there is a universe $\mathbb{U}$ of objects, and a non negative real-valued distance function $d : \mathbb{U} \times \mathbb{U} \longrightarrow \mathbb{R}^+ \cup \{0\}$ defined among them. This distance function, called also a metric, satisfies the three axioms that make the pair$\mathbb{U}, d)$ 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* $\mathbb{X} \subseteq \mathbb{U}$, $|\mathbb{X}| = n$.

Thereby, "proximity" or "similarity" searching is the problem of looking for objects in a dataset $\mathbb{X}$, similar enough to a given query object $q \in \mathbb{U}$, under a specific distance function. 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* [4].

There are a large number of metric indexes for metric spaces [4, 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, and it has been shown to give an attractive tradeoff between memory usage, construction time, and search performance. The DiSAT has a significant advantage over other indices because 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 we insert a new element into the database.

For several applications, a static scheme may be acceptable. However, 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. The *Distal Spatial Approximation Forest* (DiSAF) [2] is a dynamic index, based on the DiSAT. It uses the *Bentley-Saxe method* (BS)[1], that allows to transform a static index into a dynamic one only if on this index the search problem is *decomposable*. However, although the DiSAF admits insertions and DiSAF and DiSAT obtain similar search performance, its construction costs are very high. Therefore, in this paper, we are focused on a new dynamic version of the DiSAT that takes advantage of all our knowledge on the DiSAT and other metric space indexes. This new version significantly reduces the construction costs and obtains better search costs than DiSAT. 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. Next, in Section 3 we detail the *Distal Spatial Approximation Trees* (DiSAT), the *Distal Spatial Approximation Forest*, and some notions of their close relatives: *Spatial Approximation Trees* (SAT) and the *Dynamic Spatial Approximation Trees* (DSAT). Section 4 introduces our new dynamic version of DiSAT. In Section 5 we experimentally evaluate the performance of our proposal. Finally, we draw some conclusions and future works 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 $q$. That is, $\{x \in \mathbb{U}, d(x,q) \leq r\}$.

*k-nearest neighbors query (k-NN):* Retrieve the $k$ closest elements to $q$ in $\mathbb{U}$. That is, a set $A \subseteq \mathbb{U}$ such that $|A| = k$ and $\forall x \in A, y \in \mathbb{U} - A, d(x,q) \leq d(y,q)$.

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. In this scenario, the goal is to preprocess the dataset such that queries can be answered with as few distance evaluations as possible. In this paper, we are devoted to the most basic type of queries; range-queries. $k$-nearest neighbor queries can be obtained from range queries in an optimal way [7, 8], so we can restrict our attention to range queries.

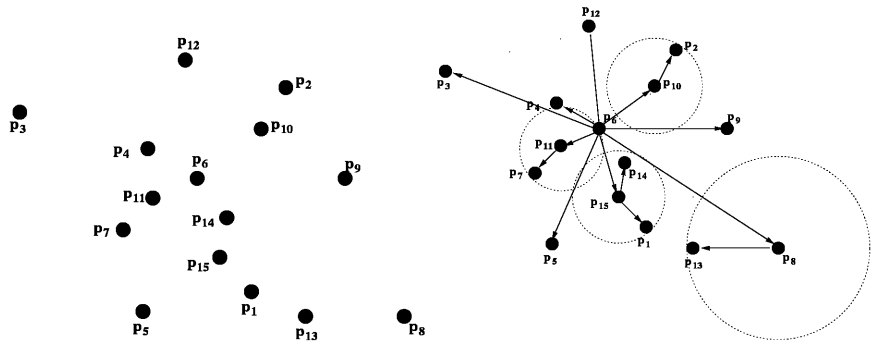
There are a plethora of indexes to speed up similarity searches [11, 12, 4] Algorithms to search in general metric spaces can be divided into two large areas: *pivot-based* and *clustering* algorithms. However, some algorithms combine ideas from both areas.

## 3 Distal Spatial Approximation Trees

The *Distal Spatial Approximation Tree* (DiSAT) [3] is a variant of the *Spatial Approximation Tree* (*SAT*) [9]. DiSAT and SAT are data structures that use a spatial approximation approach. They are iteratively getting closer to the query by starting at the root as navigating the tree. The DiSAT is 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 $x \in \mathbb{X}$ 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, the maximum distance $R(a)$ between $a$ and any element in the subtree rooted at $a$, is stored. 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 into 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 root zone (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 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(a) and Fig. 1(b) shows the DiSAT obtained by selecting $p_6$ as the tree root. We depict the covering radii for the neighbors of the tree root. It is possible to obtain completely different trees (DiSATs) if we select different roots, and each tree may have different search costs.

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




(a) Example of a space. (b) DiSAT.

**Fig. 1.** Example of a metric database in $\mathbb{R}^2$, and DiSAT obtained if $p_6$ were the root.

`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 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.

### 3.1 Distal Dynamic Spatial Approximation Forest

The Bentley-Saxe method (BS) allows transforming 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]. The Distal Spatial Approximation Forest (DiSAF) [2] applies the BS method to a DiSAT to transform it into a dynamic one. In this case, we use the BS method to have several subtrees $T_i$, particularly DiSATs. For this reason, this index is called as *Distal Dynamic Spatial Approximation Forest* (DiSAF), because we have a *forest* of DiSATs. Each subtree $T_i$ into the DiSAF is a DiSAT in the forest that will have $2^i$ elements.

Considering the example illustrated in Fig. 1(a), the Fig. 2 illustrates the DiSAF obtained by inserting the objects $p_1, \cdots, p_{15}$ one by one, in this order. As we have 15 elements, DiSAF will build four DiSATs: $T_0, T_1, T_2$, and $T_3$. 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 depict the covering radii for the neighbors of the tree roots; some

**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)
```

covering radii are equal to zero. As the DiSAF has not any parameter, the only way to obtain different forests is by considering different insertion orders.

### 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 for the SAT. An astounding and unintended feature of the DSAT is boosting 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, the database is unknown beforehand, and the objects arrive at the index at random and the queries. A dynamic data structure cannot make strong assumptions about the database and will not have statistics about all of the database.

## 4 Dynamic Distal Spatial Approximation Trees

As we mentioned, the DiSAT is a static index that must be rebuilt from scratch or requires an expensive updating when we insert a new element into the database. On the other hand, DiSAF allows to insert elements and obtains a similar search performance as DiSAT, but its construction costs are very high because each insertion has to rebuild some subtrees. Therefore, using our deep knowledge of DiSAT and its relatives

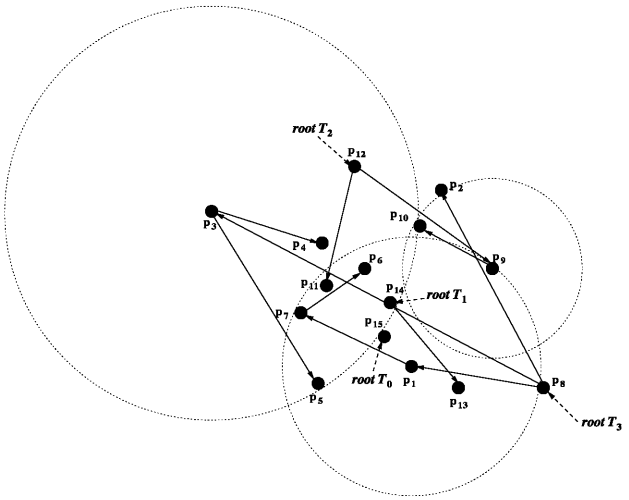



**Fig. 2.** Example of the DiSAF, inserting from $p_1$ to $p_{15}$.

and also taking advantage of storing one distance per element, we propose a new dynamic version of DiSAT that can be built by inserting the elements individually. The *Dynamic Distal Spatial Approximation Tree* (DDiSAT) reduces the construction costs significantly with respect DiSAF and obtains better search performance than DiSAT.

We want to avoid reconstruction at each insertion to reduce construction costs. Therefore, we consider using lazy insertions; we need to ensure that several insertions do not need to do any rebuilding and that only some of them require rebuilding the index. Each DDiSAT node can store an element $a$, its covering radius $rc(a)$, its set of neighbors $N(a)$, and a bag $B(a)$ of pairs of (element, distance), that are new elements into the database and the distance is its distance from $a$. The main idea is only to rebuild the DDiSAT when the new insertion in a bag makes the number of elements in the bags (pending insertion in the DiSAT) equal to the number of nodes in the DDiSAT. The above means, the DDiSAT reaches twice of the original elements inside its nodes. We have two cases to consider during insertions into the DDiSAT:

- If the DDiSAT has $i$ nodes and less to $i$ elements into their bags, we insert the new element $x$ into a node bag and do not need to rebuild the DDiSAT.
- Otherwise, we retrieve all the elements into the DDiSAT (in nodes and bags), and we rebuild the tree as a DiSAT.

Therefore, most of the insertions will proceed as follows. When we insert a new element $x$ into the DDiSAT, we search its insertion point. This search begins at the tree root. At any DDiSAT node, let be $b$ its element, if $b$ is closer to $x$ than any neighbors in $N(b)$ we insert the pair $(x, distance(b, x))$ into the bag $B(b)$ of this node. Otherwise, we go down by the node of the nearest element to $x$ in $N(b)$. As the new element $x$ insertion goes down through the tree nodes, we have to update the covering radii. This

**Algorithm 3** Searching of $q$ with radius $r$ in a DDiSAT 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.    For any pair (x, d_x) ∈ B(a)
4.       If |d(a,q) − d_x| ≤ r Then
5.          If (d(x,q) ≤ r Then Report x
6.    d_min ← min {d(c,q), c ∈ N(a)} ∪ {d_min}
7.    For b ∈ N(a) Do
8.       If d(b,q) ≤ d_min + 2r Then
9.          RangeSearch (b, q, r, d_min)
```

way, we avoid several rebuilding of the tree and ensure to do not degrade the search performance. As it can be observed, as the DDiSAT grows in elements, the number of insertions needed to double the number of elements also increases. Thus the reconstructions will be more sporadic. However, they will involve more elements.

During searches, we take advantage of all the information from the tree. As in a search on a DiSAT (Algorithm 2), we also use the distances stored in the buckets. The Algorithm3 illustrates the new search process. 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.

## 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`) [5].

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

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. 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

[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`

objects 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 has 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 mentioned previously, given the existence of range-optimal algorithms for $k$-nearest neighbor searching [7, 8], we have not considered these search experiments separately.

We show the comparison between our dynamic DDiSAT, the DiSAF and 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 of each tree node. In our experiments, we used the arity suggested by authors in [10]: the best arity for the NASA images and for Color histograms is of 4, and arity 32 for the Dictionary. Figure 3 illustrates the construction costs of all indices on the three metric spaces. As it can be seen, DDiSAT surpasses DiSAF on construction costs. On the other hand, DSAT does not make any reconstruction while it builds the tree via insertions. It must be considered that SAT and DiSAT are built with all the elements known simultaneously, not dynamically.

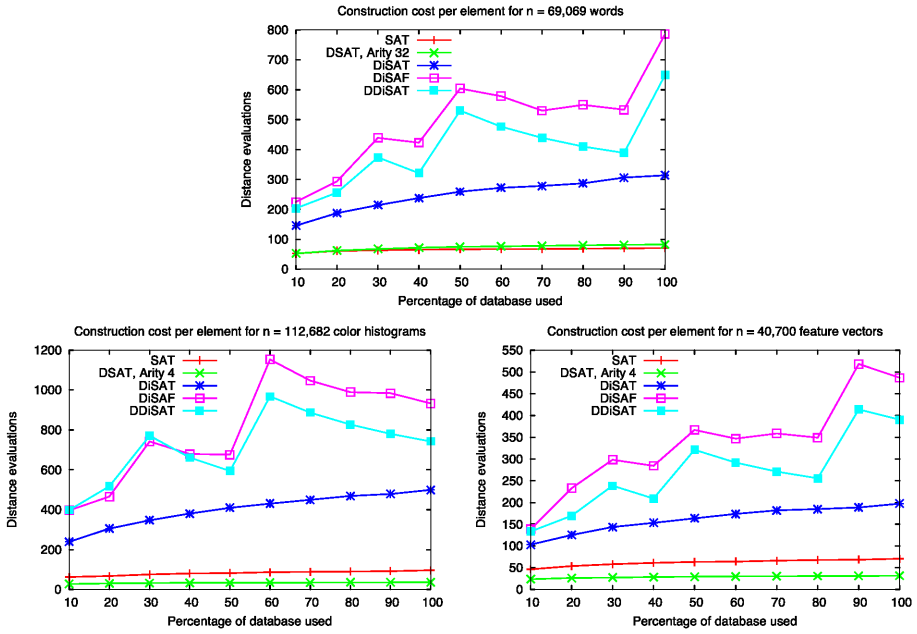



**Fig. 3.** Construction costs for the three metric spaces considered.

We analyze search costs in Figure 4. As can be noticed, DDiSAT surpasses the dynamic indexes DiSAF and DSAT in all the spaces. Moreover, DDiSAT obtains the best search performance concerning the other four indexes (static and dynamic ones). Therefore, we can affirm that the heuristic of construction of DiSAT allows surpassing in searches the other strategies used in SAT and DSAT, and combining it with the bags into the nodes that store new elements near them, it is possible to obtain even better results. Besides, we have obtained a dynamic index that overcomes DiSAT at searches. Moreover, DDiSAT has the advantage over DSAT, which does not have any parameters to tune.

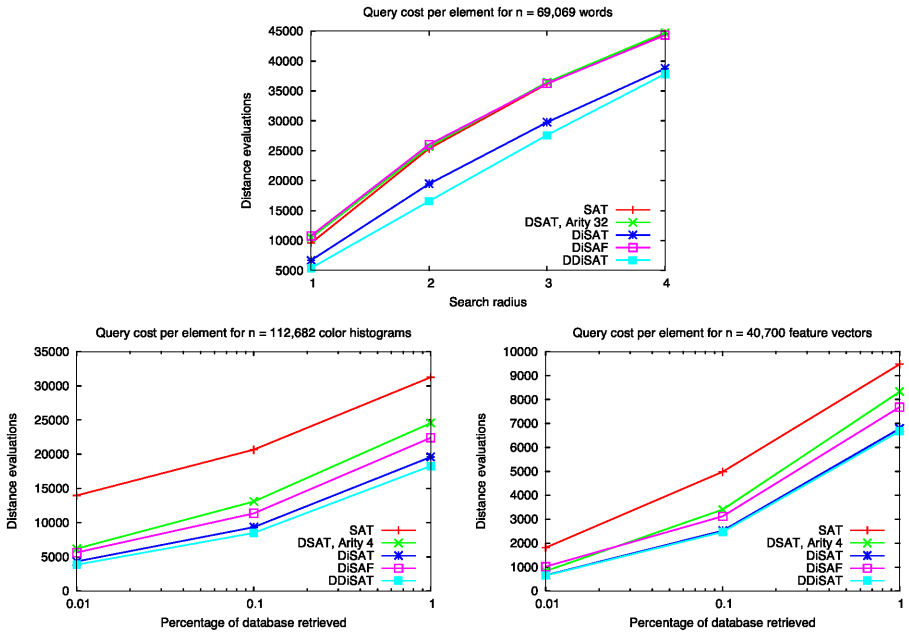



**Fig. 4.** Search costs for the three metric spaces considered.

## 6 Conclusions

We have presented a new dynamic version of the DiSAT, which at this time can handle insertions and improve its search quality. As we mentioned, there are few data structures for searching metric spaces that are dynamic and efficient. Furthermore, we have shown that we can take advantage of the heuristic used in DiSAT even more. As the distal nodes produce more compact subtrees, which in turn give more locality to the underlying partitions implicitly defined by the subtrees, we can use these partitions over the metric space to assign each new element to its closest object in the tree while it is waiting to be actually inserted as a DiSAT node.

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 some of these weaknesses. We have obtained reasonable construction costs, and it is still possible to improve it. For example by providing a bulk-loading algorithm to initially create the DDiSAT if we know aforehand a subset of elements, avoiding some unnecessary rebuildings when we insert elements and combining with *lazy insertion* that do not always rebuild trees.

In future works, we consider making the DDiSAT fully dynamic; that is, supporting deletions and designing an efficient bulk-loading algorithm, which allows for reducing more the insertion costs. We also consider to design an efficient alternative of $k$-NN search that 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, María E. Di Genaro, Nora Reyes, and Patricia Roggero. Decomposability of disat for index dynamization. *Journal of Computer Science and Technology*, pages 110–116, 2017.
3. Edgar Chávez, Verónica Ludeña, Nora Reyes, and Patricia Roggero. Faster proximity searching with the distal sat. *Information Systems*, 2016.
4. Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.
5. Karina Figueroa, Gonzalo Navarro, and Edgar Chávez. Metric spaces library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.
6. 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.
7. 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.
8. Gísli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, 2003.
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.