Journal of Computer Science and Technology
ISSN: 1666-6046
journal@lidi.info.edu.ar
Universidad Nacional de La Plata
Argentina

Chávez, Edgar; Di Genaro, María E.; Reyes, Nora; Roggero, Patricia
Decomposability of DiSAT for Index Dynamization
Journal of Computer Science and Technology, vol. 17, núm. 02, octubre, 2017, pp. 110-116
Universidad Nacional de La Plata
La Plata, Argentina

- ORIGINAL ARTICLE -

Decomposability of DiSAT for Index Dynamization

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

The *Distal Spatial Approximation Tree* (DiSAT) is one of the most competitive indexes for exact proximity searching. The absence of parameters, the most salient feature, makes the index a suitable choice for a practitioner. The most serious drawback is the static nature of the index, not allowing further insertions once it is built. On the other hand, there is an old approach from Bentley and Saxe (BS) allowing the dynamization of decomposable data structures. The only requirement is to provide a decomposition operation. This is precisely our contribution, we define a decomposition operation allowing the application of the BS technique. The resulting data structure is competitive against the static counterparts.

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 [1, 2, 3, 4]. 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* [1] 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*, *symmetry*, and *triangle inequality*. We have a finite *database* $\mathbb{U} \subseteq \mathbb{X}$,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 been presented along the years [1, 3, 2]. The *Distal Spatial Approximation Tree* (DiSAT) is a recent index derived from a simple modification of the *SAT* [5]. The main drawback of (DiSAT) is its static nature. 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) [6]. This method allows to transform a static index into a dynamic one, if on this index the search problem is *decomposable*. In [7] 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 insertions, bulk-loading and range searches. A preliminary version of this paper appeared in [8].

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.

Citation: E. Chávez, M. E. Di Genaro, N. Reyes and P. Roggero. *"Decomposability of DiSAT for Index Dynamization"*. Journal of Computer Science & Technology, vol. 17, no. 2, pp. 110–116, 2017.

Received: February 28, 2017. **Revised:** May 31, 2017. **Accepted:** August 30, 2017.

2 Previous Concepts

The metric space model can be formalized as follows. Let $\mathbb{X}$ be a universe of *objects*, with a non-negative *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 $\mathbb{U}$. Two queries are:

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 [9, 10], 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 [2, 3, 1] 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 any static data structure into a dynamic counterpart if it is *decomposable* [6]. Our data structure is an index for proximity searching. 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 commutative [6, 7]. 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 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 [7]. This partition satisfies that $\bigcup_{0 \leq i \leq m} \mathbb{X}_i = \mathbb{X}$ 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) [11] is a variant of the *Spatial Approximation Tree* (*SAT*) [5], 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 SAT and DiSAT the separation between hyperplanes (more separated in the DiSAT), 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 and Fig. 3 the DiSAT obtained by selecting p_6 as the tree

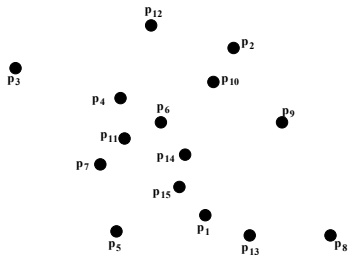


Figure 1: Example of a metric database in $\mathbb{R}^2$.

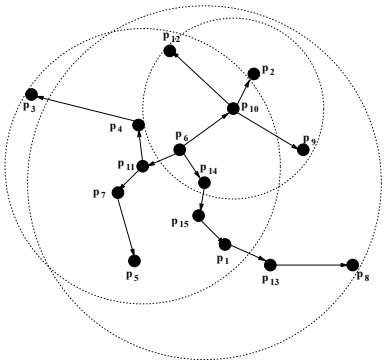


Figure 2: Example of the SAT obtained if p_6 were the root.

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.

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 [3, 5]. In the process, all the nodes x founded close enough to q are reported.

Dynamic Spatial Approximation Tree

The *Dynamic Spatial Approximation Tree* (DSAT) [12] is an online version of the SAT. It

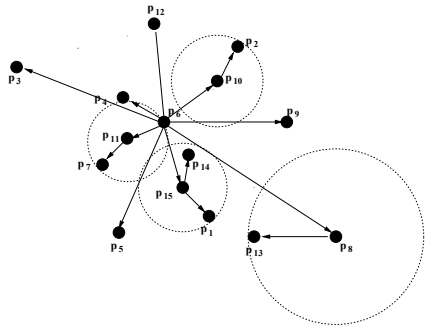


Figure 3: Example of the DiSAT obtained if p_6 were the root.

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

BuildTree (Node a, Set of nodes U)
1. $N(a) \leftarrow \emptyset$ /* neighbors of a */
2. $R(a) \leftarrow 0$ /* covering radius */
3. For $v \in U$ 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
6. $N(a) \leftarrow N(a) \cup \{v\}$
7. For $b \in N(a)$ Do $S(b) \leftarrow \emptyset$
8. For $v \in U - N(a)$ Do
9. $c \leftarrow \text{argmin}_{b \in N(a)} d(v,b)$
10. $S(c) \leftarrow S(c) \cup \{v\}$
11. For $b \in N(a)$ Do **BuildTree** $(b, S(b))$

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 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. However, it has a parameter to tune: the maximum arity of the tree. The thumb rule for tuning this parameter is that low arities are good for "easy" metric spaces and large ones for "difficult" metric spaces. Nevertheless, if an incorrect arity is chosen, it is possible to affect significantly the tree efficiency.

4 Our Proposal

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

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)
```

shown that is a very competitive index and it do not need to set any parameter, unlike other competitive indexes in the literature which depend critically on certain parameters for its efficiency.

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. 4 and Fig. 5 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. 4) and DiSAF (Fig. 5). 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 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.

Figure 6 illustrates the before (Figure 6(a)) and the after (Figure 6(b)) of one insertion of an element into a DiSAF.

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.

Bulk-Loading As we mentioned, we can build a DiSAF via successive insertions if the elements arrive at any time. However, if we know beforehand a subset of objects, we can avoid unnecessary rebuildings when we insert elements one by one by

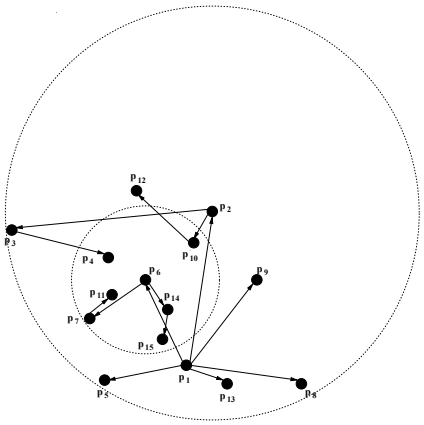


Figure 4: Example of the DSAT with maximum arity of 6, inserting from p_1 to p_{15}.

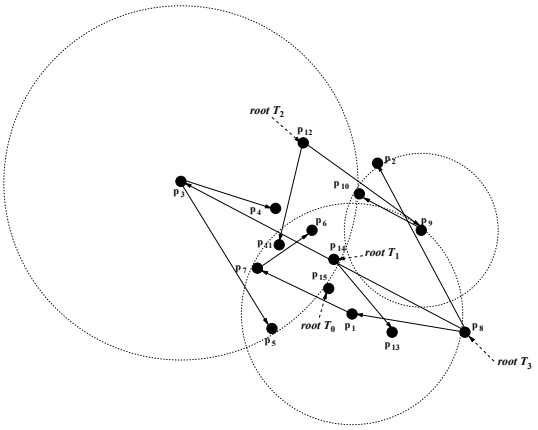


Figure 5: Example of the DiSAF, inserting from p_1 to p_{15}.

building a DiSAF with a bulk-loading algorithm. If we have a subset $Y \subseteq \mathbb{U}$, with $s = |Y|$ elements, the bulk-loading process partitionates Y in subsets $Y_0, Y_1, Y_2, \ldots, Y_{\lfloor \log s \rfloor + 1}$. Then, a static DiSAT is built with each subset and all the trees obtained form together the DiSAF with the whole Y. Later, we can insert elements one by one into this DiSAF as they arrive (by using Algorithm 3).

The Algorithm 5 describes the process of a bulk-loading with a set of elements S into a DiSAF. It can be noticed that if $|S| = n$ we firstly need to calculate in *bin* the binary representation of n (at line 1). We named as bin_i the i-th bit of this representation. If the bit bin_i is equal to zero the corresponding DiSAT T_i in the DiSAF structure will be null (line 10). Otherwise, if bin_i is equal to one, we need to build a DiSAT T_i with 2^i elements (at line 8) by using the original **BuildTree** algorithm (Algorithm 1) of DiSAT. Finally, the DiSAF structure will have some trees, each one with the adequate quantity of elements. Then, it will be possible to continue inserting the elements one by one if we want.

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

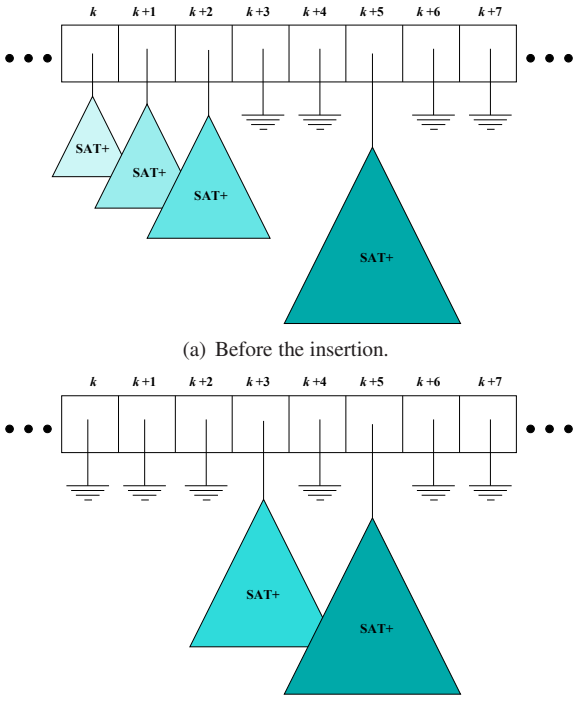
```
Insert (Element x)
1. S ← ∅, k ← min_{0≤i≤m} i, such that T_i = null
2. For i from 0 to k−1 Do
     /* retrieve all the elements of T_i */
3.   S ← S ∪ Retrieve (T_i)
4.   T_i ← null   /* T_i is a new empty tree */
5. T_k ← BuildTree (x, S)
6. If k = m Then
7.   T_{k+1} ← null, m ← k+1
```

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
```

Algorithm 5 Bulk-loading of a set of elements S in a DiSAF.

```
BulkLoading (Set of elements S)
     /* the representation base-2 of n = |S| */
1.   bin ← n_2
2.   For i from 0 to ⌊log n⌋ + 1 Do
3.     If bin_i ≠ 0 Then
         /* T_i will be a DiSAT with 2^i objects */
4.       Y_i ← ∞
5.       For j from 1 to 2^i Do
           /* let be an element x ∈ S */
6.         Y_i ← Y_i ∪ {x}
7.         S ← S − {x}
         /* let be an element a ∈ Y_i */
8.       T_i ← BuildTree (a, Y_i − {a})
9.     Else
         /* T_i will be an empty DiSAT */
10.      T_i ← null
```



(a) Before the insertion.

(b) After the insertion.

Figure 6: Example of the insertion into a DiSAF.

Lazy Rebuilding It is possible to reduce the high reconstruction costs as consequence of each insertion by amortizing them between several insertions. It suffices with delaying the reconstruction of the DiSAF until enough insertions amortize their cost. In [13, 14] different lazy rebuilding techniques are presented to delay reconstruction during insertion, because in many data structures for decomposable searching problems each insertion do not actually need to restore the "shape" of the structure immediately when it occurs, as long as the structure remains "in reasonable shape".

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`) [15].

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. 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. 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. 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 [9, 10], 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 of each node of the tree. In our experiments we used the arity suggested by authors in [12]. The Figure 7 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 8. 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. From this, it is clear that DiSAT surpass the strategies used in SAT and DSAT. We retain the parameterless nature of DiSAT overcoming DSAT.

6 Conclusions and Future Work

We presented a dynamic version of the DiSAT, accepting insertions. Deletions can be simulated by marking the object as deleted. The resulting searching times are not significantly impacted. We have to stress

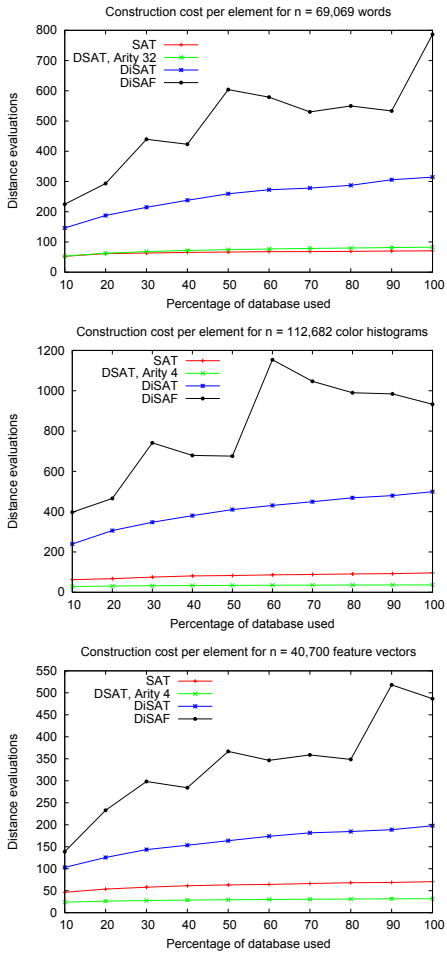


Figure 7: Construction costs for the three metric spaces considered.

out that very few data structures for searching metric spaces are dynamic. Furthermore, we have shown that the 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.

We are currently pursuing a fully dynamic DiSAF addressing implementation details. There are many open problems ahead to offer a practitioner a robust, all purpose index for proximity search under the metric space model.

Acknowledgements

The source codes of SAT and DSAT have been downloaded from the SISAP Metric Library (`www.sisap.org`).

Competing interests

The authors have declared that no competing interests exist.

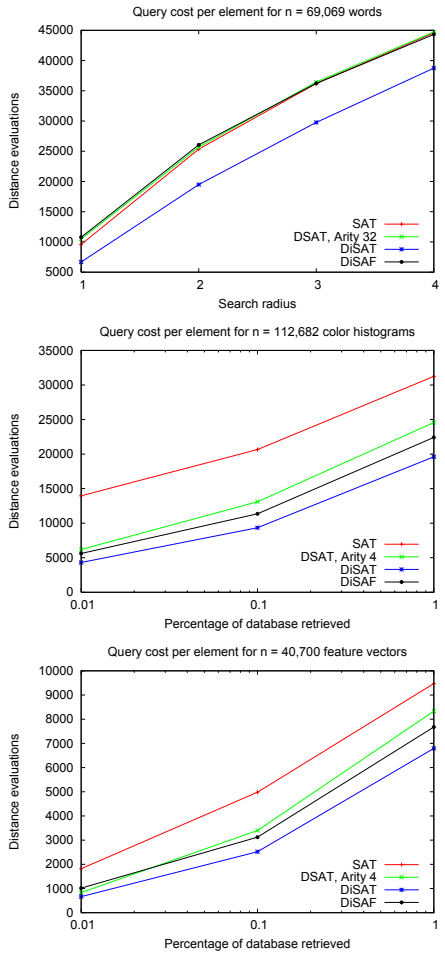


Figure 8: Search costs for the three metric spaces considered.

References

[1] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Computing Surveys*, vol. 33, pp. 273–321, Sept. 2001.

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

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

[4] M. Hetland, "The basic principles of metric indexing," in *Swarm Intelligence for Multi-objective Problems in Data Mining* (C. Coello, S. Dehuri, and S. Ghosh, eds.), vol. 242 of *Studies in Computational Intelligence*, pp. 199–232, Springer Berlin / Heidelberg, 2009.

[5] G. Navarro, "Searching in metric spaces by spatial approximation," *The Very Large Databases Journal (VLDBJ)*, vol. 11, no. 1, pp. 28–46, 2002.

[6] J. L. Bentley and J. B. Saxe, "Decomposable searching problems i. static-to-dynamic transformation," *Journal of Algorithms*, vol. 1, no. 4, pp. 301–358, 1980.

[7] B. Naidan and M. L. Hetland, "Static-to-dynamic transformation for metric indexing structures (extended version)," *Information Systems*, vol. 45, pp. 48 – 60, 2014.

[8] E. Chávez, M. E. D. Genaro, N. Reyes, and P. Roggero, "Distal spatial approximation forest," *Libro de Actas del XXII CACIC 2016*, pp. 804–813, 2016.

[9] G. R. Hjaltason and H. Samet, *Incremental Similarity Search in Multimedia Databases*. No. CS-TR-4199 in Computer science technical report series, Computer Vision Laboratory, Center for Automation Research, University of Maryland, 2000.

[10] G. R. Hjaltason and H. Samet, "Index-driven similarity search in metric spaces," *ACM Transactions on Database Systems*, vol. 28, no. 4, pp. 517–580, 2003.

[11] E. Chávez, V. Ludueña, N. Reyes, and P. Roggero, "Faster proximity searching with the distal sat," *Information Systems*, vol. 59, pp. 15 – 47, 2016.

[12] G. Navarro and N. Reyes, "Dynamic spatial approximation trees," *Journal of Experimental Algorithmics*, vol. 12, pp. 1.5:1–1.5:68, June 2008.

[13] M. H. Overmars, *The design of dynamic data structures*. Lecture notes in computer science, Berlin, New York: Springer-Verlag, 1983.

[14] M. H. Overmars and J. van Leeuwen, "Worst-case optimal insertion and deletion methods for decomposable searching problems," *Information Processing Letters*, vol. 12, no. 4, pp. 168 – 173, 1981.

[15] K. Figueroa, G. Navarro, and E. Chávez, "Metric spaces library," 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.