Faster Proximity Searching with the Distal SAT

Edgar Chávez[1], Verónica Ludueña[2],
Nora Reyes[2], and Patricia Roggero[2]

[1] CICESE, Ensenada, México
elchavez@cicese.mx
[2] Departamento de Informática Universidad Nacional de San Luis, Argentina
{vlud,nreyes,proggero}@unsl.edu.ar

Abstract. In this paper we present the *Distal Spatial Approximation Tree* (*DiSAT*), an algorithmic improvement of *S*AT. Our improvement increases the discarding power of the *SAT* by selecting distal nodes instead of the proximal nodes proposed in the original paper. Our approach is parameter free and it was the most competitive in an extensive benchmarking, from two to forty times faster than the *SAT*, and faster than the List of Clusters (*LC*) which is considered the state of the art for main memory, linear sized indexes in the model of distance computations.

In summary, we obtained an index more resistant to the curse of dimensionality, establishing a new benchmark in performance, faster to build than the *LC* and with a small memory footprint. Our strategies can be used in any version of the *SAT*, either in main or secondary memory.

1 Introduction

Proximity searching consists in finding objects from a collection near a given query. The literature is vast and there are many specializations of the problem. We will fix our attention in *exact* queries under metric distances. A metric database is a finite subset $S \subseteq \mathbb{U}$. Distances are computed with a function $d : \mathbb{U} \times \mathbb{U} \to \mathbb{R}$, such that for any $x, y, z \in \mathbb{U}$, $d(x, y) > 0$, $d(x, y) = 0 \iff x = y$, $d(x, y) = d(y, x)$ (symmetry), and obeying the triangle inequality: $d(x, z) + d(z, y) \geq d(x, y)$. For a query $q \in \mathbb{U}$ and $r \in \mathbb{R}^+$, $(q, r)_d = \{x \in S \mid d(q, x) \leq r\}$ denote a *range query*. $kNN_d(q)$ denote the K-nearest neighbors of q, say $R \subseteq S$ such that $|R| = k$ and $\forall u \in R, v \in S - R$, $d(q, u) \leq d(q, v)$. If the database S is large and/or the distance function is expensive to compute, than a sequential scan to answer queries does not scale and an index should be used.

Complexity Model. The problem at hand has been elusive for the analysis. A cost model allowing worst case guarantees for known indexing techniques is still pending in the literature. The folklore among specialists sustains that metric axioms are too weak to produce even a usable notion of complexity for the problem. However, it is well documented the existence of instances of metric databases hard to index, all data algorithms will end up reviewing the entire database even for selective queries. This is known as the *curse of dimensionality* even if a proper notion of dimensionality is elusive [1]. Complementarily, more progress have been done in the approximate setup, where probabilistic guarantees have been provided for the accuracy, when the memory, the speed and a notion of dimensionality are bounded, as in [2] and references therein. In view of the above discouraging panorama, our algorithmic improvement proposal for indexing will be tested experimentally. In this regard, only a few tricks are known and used for indexing. In a way those tricks are derived from the triangle inequality. Surveying all of them is beyond the scope of this paper. Much more details are found in surveys and books on the topic, such as [3–5].

Pivot tables are well known, generic approaches to indexing. Another alternative is to partition the space in compact zones, usually in a recursive manner, storing a representative object (a "center") c_i for each zone plus a few extra data that permits quickly discarding the zone at query time. The general idea is to have coherent clusters of objects. During search, entire zones can be discarded depending on the distance from their cluster center c_i to the query q. Two criteria can be used to delimit a zone. Representative techniques are: *Geometric Near-neighbor Access Tree* (*GNAT*) [6], *List of Clusters* (*LC*) [7] , the *Spatial Approximation Tree* (*SAT* and *DSAT*) [8,9].

Some data structures combine both ideas by dividing the space into compact partitions, and at the same time storing distances to pivots. The *D–index* [10,11] 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 and it is dynamic. Adapting the *D–index* to particular applications requires a non-trivial parameterization process. Another example in this group is obtained by adding pivots to some clustering-based data structure, as the *PM–tree* [12] does on top of the *M–tree* [13].

2 The Spatial Approximation Tree

Since our approach is an improvement of all the versions of *SAT* we will include a detailed discussion of this data structure. The *Spatial Approximation Tree* (*SAT*) [8] is a data structure aiming at approaching the query spatially, that is, start at the root and get iteratively closer to the query navigating the tree. The *SAT* is build 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. Fig. 1 shows an example *SAT* and the search path for a query.

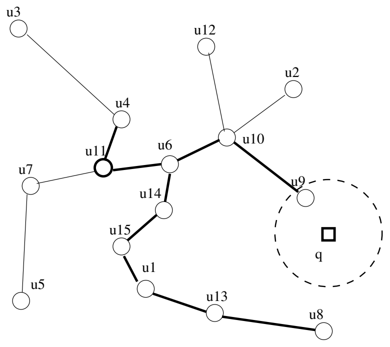


Fig. 1. Example of a *SAT* and the traversal towards a query q, starting at u_{11}. From [5].

```
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
6.        $N(a) \leftarrow N(a) \cup \{v\}$
7.  For $b \in N(a)$ Do $S(b) \leftarrow \emptyset$
8.  For $v \in S - N(a)$ 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($b$,$S(b)$)
```

Algorithm 1. Algorithm to build a *SAT* for $S \cup \{a\}$ with root a

Algorithm 1 depicts the construction process.

It is first invoked as `BuildTree(`a`,`$S - \{a\}$`)` where a is a random element of S selected as its root.

Note the construction process do not enforce a balanced data structure. While it is a disadvantage in exact searching, it seems that unbalancing does speed up searching in metric data structures [7]. In fact, the most competitive indexing algorithm in high dimensions, is precisely the *List of Clusters* (*LC*) which can be seen as an extremely unbalanced tree. The *LC* is considered the state of the art for indexing. We will see in the experimental part that our data structure outperforms the *LC* both in construction and searching time in all but a few cases, establishing a new benchmark.

One key aspect of *SAT* is that a greedy search will find all the objects previously inserted. For a query $(q, r)_d$, in each node a it is determined the closest element c of q among $a \cup N(a)$, then we use the same greedy search

```
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
6.          RangeSearch($b$,$q$,$r$,$d_{min}$)
```

Algorithm 2. The algorithm to search for $(q,r)_d$ in a *SAT* with root a

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. In the process, we report all the nodes x founded close enough to q.

Algorithm 2 `RangeSearch(`a`,`q`,`r`,`$d(a,q)$`)` describes the process. Here a is the tree root, r the range of the search and q the query object.

2.1 Dynamic Spatial Approximation Trees

If the objects to be indexed are not known beforehand, the *SAT* cannot be built with Algorithm 1. Instead of examining all possible objects to decide which of them fulfill the near condition, the neighbors are selected in a first-come-first-serve basis. There are several strategies to maintain an arbitrary arity in the tree, and to support also deletions as described in [9]. The arity was thought to play the lead role in the efficiency of searching, in this paper we have found a different factor accounting for the efficiency.

It has been shown that *DSAT* outperforms the static version for certain arity combinations. In [9] the authors proposed a couple of practical rules based on experiments: a) Low arities are good for low intrinsic dimensions or small search radii, and b) Large arities can be used for high intrinsic dimensions. From an algorithmic perspective this is an odd behavior, because a *static* data structure may have *all* the information of the data instance, while a dynamic data structure have *limited knowledge* about the data. In this paper we have found the underlying reason of this behavior. We describe our findings below.

3 The Distal Spatial Approximation Tree

From the definition of the *SAT* in algorithm 1, the starting set for neighbors of the root a, $N(a)$ is empty. This implies 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 one of the elements 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.

```
**BuildTree**(Node $a$, Set of nodes $S$)
 1. $N(a) \leftarrow \emptyset$       /* neighbors of $a$ */
 2. $R(a) \leftarrow 0$       /* covering radius */
 3. Fix an order $\pi$ in the set $S$
 4. For $v \in S$ according to order $\pi$ Do
 5.    $R(a) \leftarrow \max(R(a), d(v,a))$
 6.    If $\forall b \in N(a),\ d(v,a) < d(v,b)$ Then
 7.       $N(a) \leftarrow N(a) \cup \{v\}$
 8. For $b \in N(a)$ Do $S(b) \leftarrow \emptyset$
 9. For $v \in S - N(a)$ Do
10.    $c \leftarrow \mathrm{argmin}_{b \in N(a)} d(v,b)$
11.    $S(c) \leftarrow S(c) \cup \{v\}$
12. For $b \in N(a)$ Do **BuildTree**($b$,$S(b)$)
```

Algorithm 3. Algorithm to build a SAT^{+} for $S \cup \{a\}$ with root a

Sorting the elements in increasing order of distance to the root is just one of the $n!$ possible permutations of the database elements. Each database permutation can be used as an order for the *SAT* construction. Each insertion order will produce a *correct* version of the *SAT*, and the same searching algorithm can be used. It is very likely that the performance at search time will be different for each permutation, one natural question is: *What is the best permutation for a given database?* Instead of blindly trying every permutation we try to optimize the discarding rules of the *SAT*. A subtree is avoided using two rules, hyperplanes and covering radius. The key aspect in the hyperplane discarding rule is the separation between the two defining points, because the query ball is more likely to fall completely in either side of the hyperplane and all the objects in the opposite side can be discarded. A good hyperplane separation in the upper levels of the tree also implies small covering radius in the lower levels of the tree. We exploit this two observations using several heuristics in our *DiSAT* data structure. Interestingly enough, the original policy for *SAT* works exactly in the opposite direction of this improvement strategy. Even a *random* selection of the insertion order outperform the original *SAT*; this explains the dynamic version being better than the static version.

3.1 The SAT^{+} Strategy

Algorithm 3 gives a formal description of the construction of our data structure. The difference is in selecting the insertion order π in line 3. We tried farthest-to-nearest order from the root . Searching is done with the standard procedure described in Algorithm2.

A random permutation, or equivalently a random order, for the construction of the *SAT* is similar to inserting elements online in the *DSAT*. The difference will be to have a *natural* number of neighbors instead of an arbitrary arity to be tuned up. We call this the SAT^{Rand} in the experiments. We tested this construction mainly to explain the behavior of the *DSAT*.

When working with hyperplanes to perform data separation it is advisable to use object pairs far from each other as documented in [5] for the *GNAT* and *GHT* data structures. Using the above observations, we can ensure a good separation of the implicit hyperplanes by selecting the first neighbor as the farthest element to the root. Clearly it is advisable to do this recursively, at every node of the tree. Please note that this heuristic is the exact opposite of the original ordering in the construction of the *SAT*.

3.2 The SAT^{Glob} Strategy

Sorting elements by distance at every level can be time consuming. We tried a fixed insertion order π by sorting elements for distance to the tree root, farthest first. This fixed order π is used in all the following levels. Therefore, SAT^{Glob} and SAT^{+} are similar only at the first level of the tree, on the following levels the order π already determined is used without performing any new sort. This also serves to probe for the recursive need to select good hyperplanes at each tree level.

3.3 The SAT^{Out} Strategy

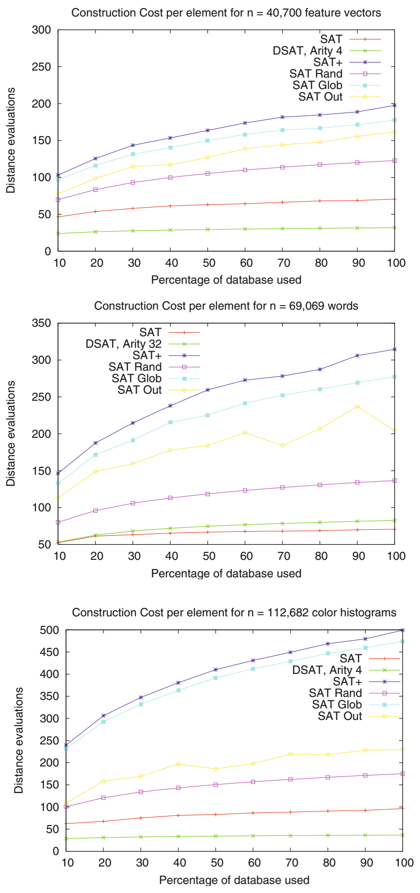
So far we have selected a random element as the tree root. Since we are aiming at maximizing the hyperplane separation, it makes a lot of sense to select the fathest pair as the root and the first neighbor respectively. This way there will be a lot of room for farthest pair selection in the lower levels of the tree.

The "farthest pair problem" is well known. We want objects $x, y \in S$, such that $d(x, y) \geq d(z, v), \forall z, v \in S$. This can be doing by comparing all against all the elements of the database, this is prohibitively expensive since it involves $O(n^2)$ operations. A randomized version is very effective and uses only $O(n)$ operations. The idea is to select a random starting point u_0, locate its farthest neighbor u_1 and repeat to find u_2, etc. A few iterations will get a good approximation of the farthest pair.

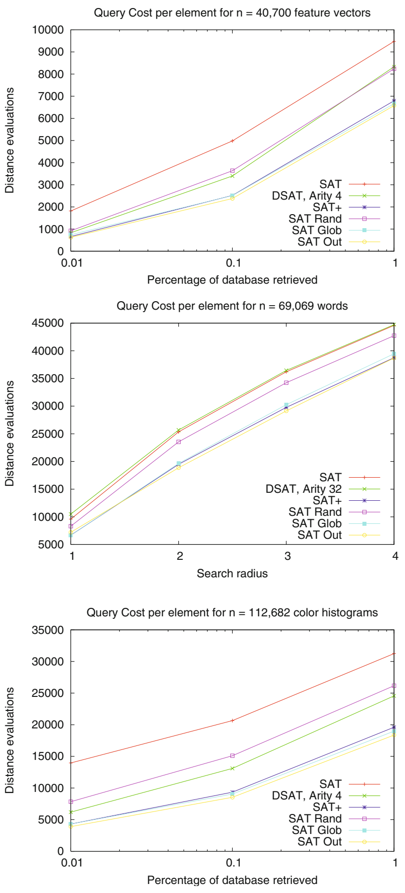
4 Experimental Results

For our first experiment we selected three widely used benchmark databases, all from the SISAP Metric Library `www.sisap.org`, NASA images, Strings and Color Histograms. We use euclidean distance for NASA images and Color Histograms, and edit distance for Strings. In all cases, we built the indexes with 90% of the points and used the other 10% (randomly chosen) as queries. All 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. Given the existence of range-optimal algorithms for k-nearest neighbor searching it is enough to consider only range searches in the experimental part. The source code *SAT* and *DSAT* is available in `www.sisap.org`, we submitted the code for *DiSAT*. The arity parameter of the *DSAT* was selected using the recommendation in [9][1] .

Fig. 2 (Subfigs. 2(a)) contains the results of construction costs obtained in the experiments for the three metric spaces. We show the comparison of the construction costs for the original *SAT*, for the *DSAT*, and for the new SAT^{Rand}, SAT^{+}, SAT^{Glob}, and SAT^{Out} built using the new construction criterions. As it can be seen, the SAT^{+} gets the worst construction costs. It can be explained because the arity in this case is the largest, and as it was shown in [9] construction cost grows with the tree arity. Moreover, despite of SAT^{Out} uses the same neighbor selection policy as SAT^{+}, SAT^{Out} achieves better construction costs



(a) Construction costs.



(b) Search costs.

Fig. 2. Comparison of construction and search costs

[1] The best arity for the NASA images and for Color histograms is of 4, and arity 32 for the Strings.

because the maximum arity tree is significantly lower than SAT^{+} because we selected the root more properly. Fig. 2 (Subfigs. 2(b)) depicts that the new SAT^{+}, SAT^{Glob} and SAT^{Out} significantly improve searching costs with respect to other ones, and they are very similar between them. However, SAT^{Out} achieves the lowest search costs.

We postulate that in the new indexes the neighbors of the root represent a more accurate sample of the different zones in the metric space and produce better hyperplane separation in two senses, the inter-sibling separation and the root-node separation. These two conditions also imply small covering radii. This in turn produces a more compact partition of the space, improving the search cost.

5 Comparison with Other Indexes

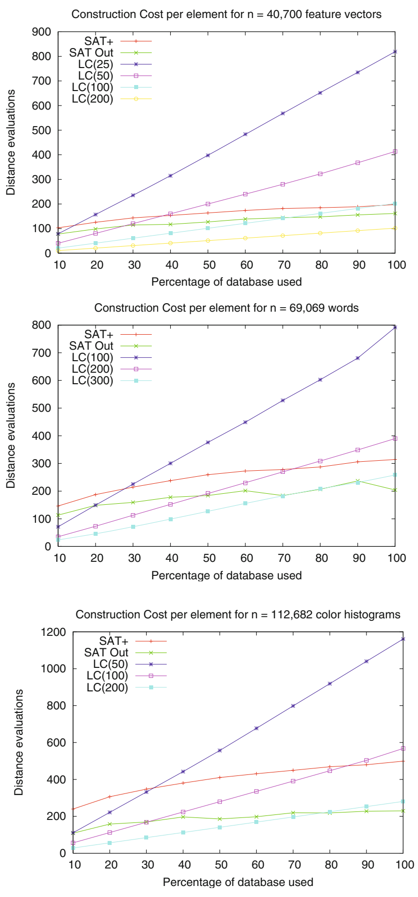
Among all the exact indexes AESA [14] stands as the lower bound in distance computations; however, it uses a quadratic amount of space. In this version of the paper we will only compare with linear size indexes. We have performed an exhaustive comparison with other approaches and confirmed the *DiSAT* as a competitive, standing as a new efficiency benchmark. Due to space restrictions, in this version of the paper we only compare with the *List of Clusters* (em LC), as this data structure currently holds the benchmark for exact searching. In [8,9] *SAT* and *DSAT* were compared with several competitive indexes, so transitively we show that *DiSAT* is a very efficient index because is a better option than *SAT* and *DSAT*.

5.1 List of Clusters

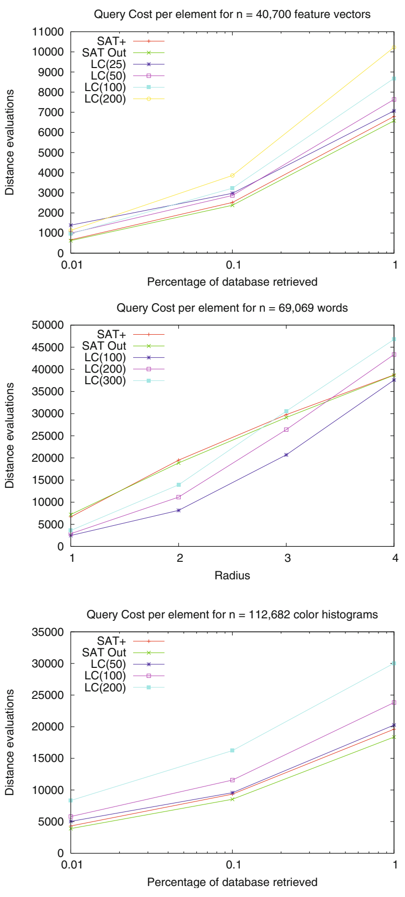
The *List of Clusters* (*LC*) [7], with a proper parameter selection stands as the most competitive exact index when counting distance computations as the complexity measure. As we have improved the original *SAT*, with our SAT^{+} and SAT^{Out}, we want to test how competitive is our approach against the state of the art. One drawback of the *LC* is the construction cost, another is the manually selected cluster size.

Fig. 3 compares construction and search costs, Subfigs. 3(a) and Subfigs. 3(b) respectively, of the SAT^{+}, SAT^{Out}, and *LC*. We test different values of m (*LC(m)*), some of them are presented in this comparison. We select values that allow us to show the behavior of *LC* at a similar construction cost and a similar or even better search cost with respect to our indexes.

For the NASA images database, SAT^{+} and SAT^{Out} beat *LC* for all search radii, even with a value of $m = 25$ for *LC* that implies approximately 5 times our construction costs. Moreover, in this database we could not get any cluster size that would enable *LC* to be superior to our indexes, , even if we disregard construction costs. Nevertheless, *LC* outperforms our indexes with $m = 100$ in all radii considered for Strings database, but it needs 2.5 to 3 times our construction costs. Moreover, in this database *LC* with similar construction costs



(a) Construction costs.



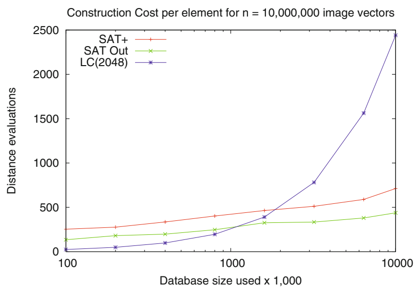
(b) Search costs.

Fig. 3. Comparison of construction and search costs against the LC, considering different cluster sizes

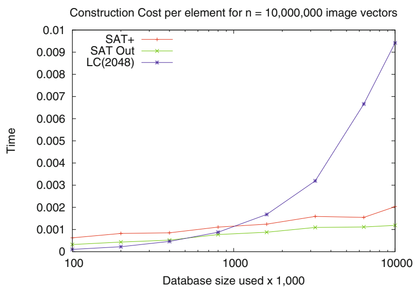
(with $m = 200, 300$) beats us, but for large radii. For Color histograms database, again it can be seen that our our SAT^+ and SAT^{Out} surpass LC for all radii considered. However, for $m = 50$ LC achieves slightly higher search costs than ours, but it needs to pay almost 6 times more than our cost of construction.

Please notice that as the size of the database grows, the increase in construction cost per element is not significant. It is also apparent that SAT^+ and SAT^{Out} have a good tolerance to large radii without needing parameter tuning. In the case of the LC a wrong parameter imply poor performance and/or large construction cost.

Scalibility. We also experimented with a larger database to test the scalability of our approach, and at the same time to compare with the LC. For this experiments, we use a 10 million images subset of the **COPHIR** database. For the *List of Clusters* we use a cluster size of 2048. We build the indexes on increasing sizes of the database in order to evaluate how is the behavior of all indexes as the database size grows. We started in 100,000 objects, doubling the size of the database up to 10 million objects. We reserved 200 objects, which would not be indexed, to be used as queries. In all sizes we use the same threshold r of 200 for the range queries, with $r = 200$ we retrieve in average more than 100 objects. Please notice that the items retrieved decrease with the database size, not retrieving any object in the sizes range of 100,000 to 400,000. Fig. 4 shows the construction costs obtained, and Fig. 5 depicts the search costs for the three indexes compared.

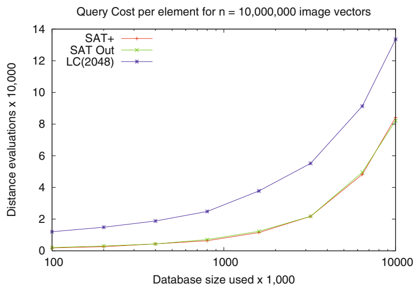


(a) Distance evaluations for building.

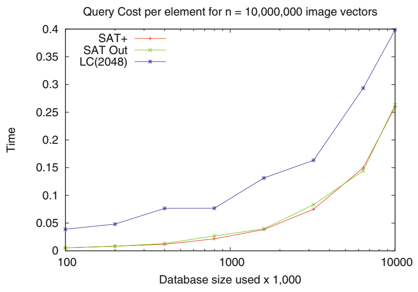


(b) Time for building.

Fig. 4. Comparison of construction costs for increasing subsets of COPHIR database



(a) Distance evaluations for searching.



(b) Time for searching.

Fig. 5. Comparison of search costs for increasing subsets of COPHIR database

As it can be noticed, our indexes outperform significantly the *List of Clusters*, in both construction and search time. Although construction costs are higher in lower sizes of the database on our indexes, our costs do not change too much as the database size grows, while with the *List of Clusters* it grows very quickly. During searches is even more remarkable that our indexes are better than the

List of Clusters obtaining better search costs for all sizes considered. Therefore, these experiments allow empirically to demonstrate that SAT^{+} and SAT^{Out} are very scalable indexes.

6 Conclusions and Future Work

We have presented a new heuristic for constructing the *SAT*. The rule is counterintuitive and consist in selecting distal instead of proximal nodes. With this approach our proposed index *DiSAT* stands as the new efficiency benchmark, supported by exhaustive experimentation. It improves the construction and searching times w.r.t. *LC* and other data structures.

Distal node selection can be used in static, dynamic and secondary memory versions of the *SAT* and produce more compact subtrees, inducing more locality to the implicit partitions of the subtrees. This factor will impact IO operations in secondary memory versions of *DiSAT*.

One possible consequence of a compact underlying partition, induced by a small covering radius is the possibility of producing coherent clusters suitable for statistics, mining, pattern recognition and machine learning purposes. One aspect of the putative clustering procedure is to produce a stable clustering (independent of the choice of the root, for example), or alternatively detecting natural, parameter free clusters.

References

1. Navarro, G.: Analyzing metric space indexes: What for? In: Second International Workshop on Similarity Search and Applications, SISAP 2009, pp. 3–10. IEEE (2009)
2. Houle, M.E., Nett, M.: Rank cover trees for nearest neighbor search. In: Brisaboa, N., Pedreira, O., Zezula, P. (eds.) SISAP 2013. LNCS, vol. 8199, pp. 16–29. Springer, Heidelberg (2013)
3. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach. Advances in Database Systems, vol. 32. Springer (2006)
4. Samet, H.: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling). Morgan Kaufmann Publishers Inc., San Francisco (2005)
5. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (2001)
6. Brin, S.: Near neighbor search in large metric spaces. In: Proc. 21st Conference on Very Large Databases (VLDB 1995), pp. 574–584 (1995)
7. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters 26(9), 1363–1376 (2005)
8. Navarro, G.: Searching in metric spaces by spatial approximation. The Very Large Databases Journal (VLDBJ) 11(1), 28–46 (2002)
9. Navarro, G., Reyes, N.: Dynamic spatial approximation trees. Journal of Experimental Algorithmics 12, 1–68 (2008)
10. Dohnal, V., Gennaro, C., Savino, P., Zezula, P.: D-index: Distance searching index for metric data sets. Multimedia Tools and Applications 21(1), 9–33 (2003)

11. Dohnal, V.: An access structure for similarity search in metric spaces. In: Lindner, W., Fischer, F., Türker, C., Tzitzikas, Y., Vakali, A.I. (eds.) EDBT 2004. LNCS, vol. 3268, pp. 133–143. Springer, Heidelberg (2004)
12. Skopal, T., Pokorný, J., Snásel, V.: PM-tree: Pivoting metric tree for similarity search in multimedia databases. In: ADBIS (Local Proceedings) (2004)
13. Ciaccia, P., Patella, M., Zezula, P.: M-tree: an efficient access method for similarity search in metric spaces. In: Proc. of the 23rd Conference on Very Large Databases (VLDB 1997), pp. 426–435 (1997)
14. Vidal Ruiz, E.: An algorithm for finding nearest neighbours in (approximately) constant average time. Pattern Recognition Letters 4, 145–157 (1986)