A log square average case algorithm to make insertions in fast similarity search

Luisa Micó*, Jose Oncina*

Dept. Lenguajes y Sistemas Informáticos
Universidad de Alicante, E-03071 Alicante, Spain

Abstract

To speed up similarity based searches many indexing techniques have been proposed in order to address the problem of efficiency. However, most of the proposed techniques do not admit fast insertion of new elements once the index is built. The main effect is that changes in the environment are very costly to be taken into account.

In this work, we propose a new technique to allow fast insertions of elements in a family of static tree-based indexes. Unlike other techniques, the resulting index is exactly equal to the index that would be obtained by building it from scratch. Therefore there is no performance degradation in search time.

We show that the expected number of distance computations (and the average time complexity) is bounded by a function that grows with $\log^2(n)$ where n is the size of the database.

In order to check the correctness of our approach some experiments with artificial and real data are carried out.

Keywords: similarity search, metric space, dynamic index, insertions

1. Introduction

The similarity search problem can be stated as follows: given a finite data set of objects D, a dissimilarity measure d and a query object q find the set of elements in the data set ($P \subset D$)that is the most similar to the query (minimise a dissimilarity measure). Depending on the amount and type of information

*Corresponding Author
Email addresses: `mico@dlsi.ua.es` (Luisa Micó), `oncina@dlsi.ua.es` (Jose Oncina)

required, several similarity search techniques can be stated: nearest neighbour search (only the nearest object in the database is retrieved: $p \in P \iff \forall r \in D, d(q,p) \leq d(q,r)$), range search (all the objects in the database nearest to the query than a value h are retrieved: $p \in P \iff d(q,p) \leq h$), reverse nearest neighbour search (the elements in the dataset that have the query as their nearest element: $p \in P \iff \forall r \in D, d(p,q) \leq d(p,r)$), etc.

Over the time, these techniques have been applied to databases increasingly large making their execution times become real bottlenecks.

In order to speed up these techniques, fast similarity search methods have to exploit some property of the search space. Metric space searching techniques assume that the dissimilarity function ($d(\cdot,\cdot)$) defines a metric over the representation space E, that is:

1. $\forall x, y \in E, d(x,y) \geq 0$ non-negativity
2. $\forall x, y \in E, d(x,y) = d(y,x)$ symmetry
3. $\forall x \in E, d(x,x) = 0$ identity
4. $\forall x, y, z \in E, d(x,z) \leq d(x,y) + d(y,z)$ triangle inequality

One of the main characteristics of metric space searching is that no assumption about the structure of the objects (points) is necessary. Some examples of objects can be: protein sequences (represented by strings) (Lundsteen et al., 1980), skeleton of images (trees or graphs)(Carrasco and Forcada, 1995)(Escolano and Vento, 2007), histograms of images(Cha and Srihari, 2002), etc.

At present, many communities have paid great attention to these techniques because of the need for handling large amounts of data. Then, many metric space indexes designed to speed up searches have been proposed (some reviews can be found in (Chávez et al., 2001)(Hjaltason and Samet, 2003)(Zezula et al., 2006)). These indexes have proved to be very effective in many applications such as content based image retrieval (Giacinto, 2007), person detection or automatic image annotation (Torralba et al., 2008), texture synthesis, image colourisation or super-resolution (Battiato et al., 2007).

Unfortunately, most of these indexes are static (Yianilos, 1993)(Brin, 1995)(Micó et al., 1994)(Navarro, 2002). That is, the insertion or deletion of an object requires a complete rebuilding of the index. This is very expensive and discourages its use in interactive or on-line training systems.

In this work, we propose technique to allow fast insertions. The performance, in search time, of the index does no degrade with the insertions and we show that the expected number of distance computations is bounded by $\log^2(n)$ where n is the size of the database. This result compares very favourably with the number of distance computations needed in a whole rebuild ($n \log(n)$).

In order to check the correctness of our approach some experiments with artificial data (Euclidean distance in 5, 10 and 15 dimensional spaces) and real data (Euclidean distance in an image database and edit distance in handwritten digits contour strings and English words) have been carried out.

Section 2 describes related work and introduces the main ideas in our approach. Section 3 introduces the static index in wich our approach is based, and Section 4 describes our inserting algorithm. Section 5 is devoted to analyse the insertion cost. This analysis is followed by experimental results using artificial and real data in Section 6. Finally Section 7 describes the conclusions drawn from the results and summarises our contribution.

2. The approach

A number of proposals to allow object insertion/deletion operations have been made for metric space indexes (Fu et al., 2000)(Navarro and Reyes, 2008). In some cases dynamic approaches were proposed as a completely new algorithm to allow cheap insertions and deletions such as the M-tree (Ciaccia et al., 1997), and, in other cases, as a modification of previously existing static indexes (Fu et al., 2000)(Navarro and Reyes, 2002)(Procopiuc et al., 2003)

Usually, static search methods are faster searching that dynamic indexes and static methods degrades when they are adapted to allow insertions.

The main problem when adapting static methods to allow insertions is the need of a reorganization when an insertion is performed. To avoid this overhead, some authors (Navarro and Reyes, 2008) propose the use of buckets in selected places of the index to store the new objects in such a way they can be located easily and does not harm very much the performance of the index. Despite of that, the nearest neighbour search performance is degraded as the size of the buckets increases. To avoid such degradation a rebuilding of the index is forced when the size of a bucket exceeds a threshold. A trade-off between insertion performance and search performance should be established.

In our proposal the index obtained after the insertion is the same as the (static) one obtained if a complete rebuild would be made, without adding buckets or any type of additional information to the index. As a consequence, no insertion/search performance ratio should be adjusted and there is no degradation of search performance.

The idea of the strategy is quite simple: go ahead with the insertion unless a modification in the index is necessary; otherwise, rebuild completely the affected part of the index.

Although this strategy can be applied to many indexing techniques, it is specially effective when is applied to Most Distant to the Father (MDF) tree index. This tree based indexing is used in some state of the art searching techniques (Micó et al., 1996)(Gómez-Ballester et al., 2006).

The properties that make this structure so effective are:

1. the structure is based on the use of objects in very low probability regions
2. the rebuilding of the index section corresponding to one branch of the tree is independent of the other branches.

3. The Most Distant to the Father tree index

One of the most successful methods for reducing the search time (by reducing the average number of distance computations) is the mb-tree (or monotonous bisector tree). This method was originally intended to be used with vector-data and Minkowski metrics although it can be used with arbitrary metrics and then, with complex objects. The mb-tree was proposed by Noltemeier et

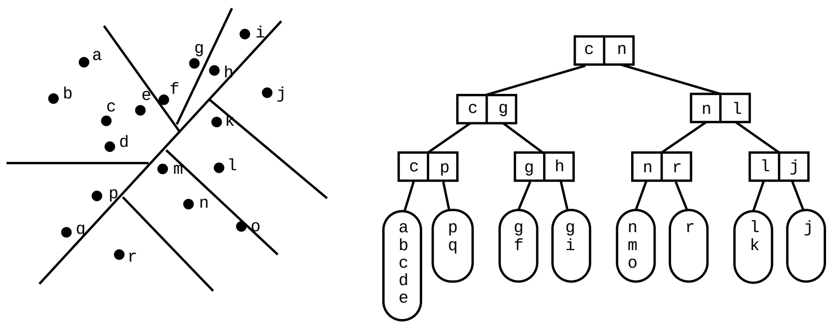

Figure 1: Example of space partitioning produced by a mb-tree in a two-dimensional space (left) and the mb-tree derived from it (right).

al. (Noltemeier et al., 1992) to modify the definition of the bisector tree (a tree that uses generalized hyperplane partitioning augmented by including for each pivot the maximum distance to an object in its subtree, (Kalantari and McDonald, 1983)) so that one of the two pivots in each nonleaf node is inherited from its parent (see Figure 3). This strategy allows to reduce the number of distance computations during the search (only a new distance, instead of two, is necessary to compute every time a new level is explored in the tree). But this is at the cost of a worse partitioning, obtaining a deeper tree. This general approach allows many different configurations in the selection of pivots.

The MDF tree is a binary indexing structure based on a hyperplane partitioning approach (Micó et al., 1996)(Gómez-Ballester et al., 2006) with similar properties to the mb-trees. The main difference is related to the selection of the representatives (pivots) for the next partition (branch of the tree).

In the MDF-tree firstly a pivot is randomly selected as the root of the tree (first level). Secondly, the most distant point from the root is selected as a new pivot, and the remaining points are distributed according to to the closest pivot. This procedure is recursively repeated until each leaf node has only one object (see Figure 3).

For each node, the covering radius (the distance from the pivot to the most distant point in the subspace) is computed and stored in the respective node.

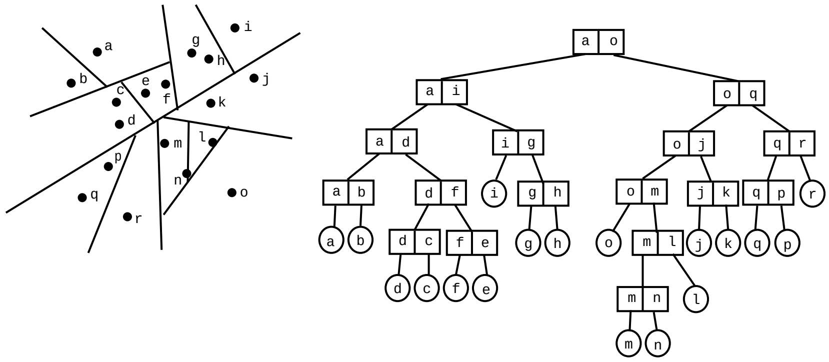

Figure 2: Example of space partitioning produced by a MDF-tree in a two-dimensional space(left) and the MDF-tree derive from it (right)

This procedure is described in algorithm 1.

The function `build_tree(`ℓ`,`S`)` takes as arguments the future representative of the root node (ℓ) and the set of objects to be included in the tree (excluding ℓ) and returns the MDF-tree that contains $S \cup \{\ell\}$. The first time that `build_tree(`ℓ`,`S`)` is called, ℓ is selected randomly among the data set. In the algorithm, M_T is the pivot corresponding to T, r_T is the covering radius, and T_L (T_R) is the left (right) subtree of T.

It is easy to see that the space complexity of the index is $O(n)$, with n being the number of points, and the time complexity is $O(hn)$ where h is the depth of the tree.

4. Incremental tree

In this work we focus on a procedure to obtain, each time an insertion is performed, the exact index that will be obtained if a complete rebuild of the tree was made (a preliminary version of this idea, with no theoretical guarantees, was presented in Micó and Oncina (2009)). Note that in such case the performance of the search algorithm that uses the MDF index is exactly the same as in the case when the index is build from scratch. Then no further research or experiments in search degradation performance is needed.

Algorithm 1: build_tree(ℓ, S)

Data:

$S \cup \{\ell\}$: set of points to include in T;

ℓ: future left representative of T

create MDF-tree T

if S *is empty* **then**

  $M_T = \ell$ // New representative of T

  $r_T = 0$

else

  $r = \text{argmax}_{x \in S}\, d(\ell, x)$

  $r_T = d(\ell, r)$

  $S_\ell = \{x \in S | d(\ell, x) < d(r, x)\}$

  $S_r = \{x \in S | d(\ell, x) \geq d(r, x)\} - \{r\}$

  $T_L =$ `build_tree(`ℓ`,`S_ℓ`)`

  $T_R =$ `build_tree(`r`,`S_r`)`

end

return T

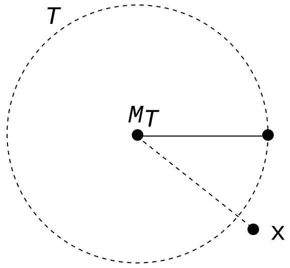

Figure 3: Case when a complete rebuilt is needed

The main idea consists on comparing the object to be inserted with the pivot (representatives) on every recursive call to check if it is farther than the farthest so far. If it is farther (it has to be a pivot in the resulting tree), the affected part of the tree is completely rebuilt. Otherwise, the insertion is made in the subtree whose pivot is nearest.

This may seems a quite expensive strategy, but as the pivots are very unusual objects (the farthest of its sibling pivot), and the sizes of the subtrees decrease very quickly, big reconstructions of the tree seldom happens. The high cost of big rebuilds is compensated by its low probability.

Let T be the MDF tree built using a database D. Let x be the new object to be inserted in the index, and let T' the MDF tree built using the data set $D \cup \{x\}$. The algorithm detects and rebuilds the subtree of T that is different from T'.

Let we denote by M_T the representative of the root node of a subtree T of the MDF tree, let r_T be its covering radius, and let T_L (T_R) be the left (right) MDF subtree of T.

We have several cases:

C 1. If $d(M_T, x) > r_T$, T' differs from T in the root node because the object x is selected in T' as the representative of the right node. Then the whole tree T is rebuilt in order to include x (see fig. 3).

C 2. Otherwise, the roots of the trees T and T' are identical. Then we have two cases (see fig. 4) :

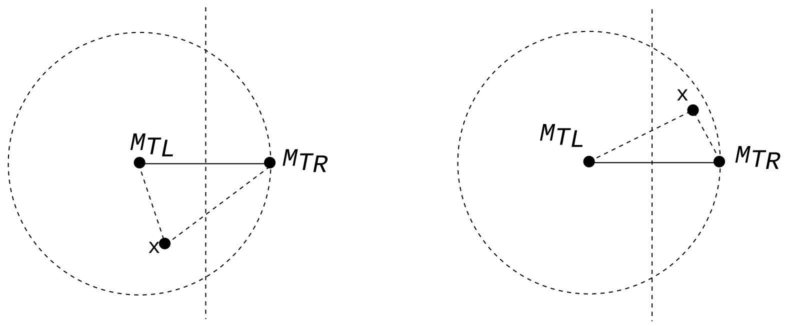

Figure 4: Inserting in the left (right) subtree

C 2.1. if $d(M_{T_L}, x) < d(M_{T_R}, x)$ the object x should be inserted in the left tree T_L and then the tree T'_R is identical to T_R.

C 2.2. Conversely if $d(M_{T_L}, x) \geq d(M_{T_R}, x)$ the object should be inserted in T_R and the tree T'_L is identical to T_L.

Algorithm 2 shows the insertion procedure.

5. Average time complexity

An MDF tree is generally unbalanced and, in the worst case, it can be fully degenerated.

We introduce a parameter α to measure the inbalance of a tree. Let $\alpha \in [0.5, 1.0[$ be defined such that for all node T in a MDF tree, where T_1 and T_2 are its two children and where $|T_1| \leq |T_2|$

$$|T_1| \leq \alpha\ |T|$$
$$|T_2| \geq (1-\alpha)\ |T|$$

An upper bound to the depth (h) of a α-unbalanced tree can be easily computed taking into account that, in the worst case, the size of the bigger child decreases at least a factor α in each level until we arrive to a leave (size 1).

Algorithm 2: **insert_tree(**T**,** x**)**

Data:

T: MDF-tree

x: object to be inserted in T

```
if d(M_T, x) > r_T then
|   T = build_tree(M_T, {s|s ∈ T} ∪ {x} − {M_T})
else if T_L is empty then
|   T = build_tree(M_T, {x})
else
|   d_ℓ = d(M_{T_L}, x)        // this distance has been previously computed
|   d_r = d(M_{T_R}, x)
|   if d_ℓ < d_r then
|   |   insert_tree(T_L, x)
|   else
|   |   insert_tree(T_R, x)
|   end
end
```

That is, $\alpha^h n = 1$, and then $h = -\frac{\log(n)}{\log(\alpha)}$. For example, if the tree is balanced $(\alpha = 0.5)$ $h = \log_2(n)$.

Then, the number of distance computations required to build a α-unbalanced MDF tree of size n is upper bounded by $nh = n(-\frac{log(n)}{log(\alpha)})$.

In the following, we are going to obtain an upper bound of this function. Let we denote by $E(n)$ the expected number of distance computations when inserting an object in a α-unbalanced MDF tree of size n.

Let x be the object to be inserted and assume that all the elements in $D \cup \{x\}$ where extracted i.i.d. from an unknown probability distribution. Following alg. 2 we have four possibilities:

1. if $n = 1$ then $E(n) = 1$
2. if $d(M_T, x) > r_T$ a rebuilt of the subtree is necessary. Its cost is upper bounded by $n(-\frac{\log(n)}{\log(\alpha)})$
3. if $d(M_{T_L}, x) < d(M_{T_R}, x)$, x is inserted in the left subtree
4. if $d(M_{T_L}, x) \geq d(M_{T_R}, x)$, x is inserted in the right subtree

Note that since we are assuming that all the points are extracted i.i.d., all the points have the identical probability of being the new pivot (fulfilling condition 2) and then, the probability of this event is $\frac{1}{n}$. Therefore, the probability of event 3 or event 4 is $\frac{n-1}{n}$.

Moreover, the action taken by the algorithm in such cases is to make an insertion in one of its children. Since the tree is α-unbalanced the cost of each of this actions are bounded by worst case: $E(\alpha n)$.

Now, expressing all that in an equation we have the upper bound:

$$E(n) \leq 1 + \frac{1}{n}\, n \left(-\frac{\log(n)}{\log(\alpha)} \right) + \frac{n-1}{n}\, E(\alpha n) \tag{1}$$

This equation is composed by three terms. The first term takes into account the distance computation needed to know the distance from the sample to the pivot. Second an third terms takes into account the possibility of the new sample being farther (or not) than the present representative in the right node. Second term is the probability that the new sample is farther than the present representative ($\frac{1}{n}$), multiplied by the cost of rebuilding the subtree. Third term is the probability that the new sample is not the farther representative, multiplied by the expected number of distance computations of inserting in the bigger of the two children (size at most αn).

If we unfold equation 1 we have:

$$E(n) \leq h - \frac{1}{\log(\alpha)} \left(\log(n) + \log(n\alpha) + \log(n\alpha^2) + \ldots^{h\ times}\right)$$

where we have taken into account that $\frac{n-i}{n} < 1, \forall i < n$.

$$E(n) \leq -\frac{\log(n)}{\log(\alpha)} - \frac{1}{\log(\alpha)} \log\left(\prod_{i=0}^{h} n\alpha^i\right)$$

and using some properties of the log function:

$$E(n) \leq \frac{\log^2(n)}{2\log^2(\alpha)} - \frac{3\log(n)}{2\log(\alpha)} \tag{2}$$

This upper bound shows that, in the worst case, the expected number of distance computations grows with $\log^2(n)$. Very far of the worst case ($n\log(n)$).

6. Experimental results

The experiments were done using artificial and real data represented as vectors or strings. For artificial data, the datasets were generated using a uniform distribution in the 5, 10 and 15 dimensional unit hypercube.

Three real data databases are used:

NASA: is a collection of 40 150, 20-dimensional vectors obtained from NASA video and image archives. The authors of the database (Katayama and Satoh, 1999) divided each image in four regions, nine color histograms for each region were computed. The features are a PCA projection to a 20 dimensional space. The Euclidean distance was used as dissimilarity measure (more details can be found in `http://www.sisap.com`).

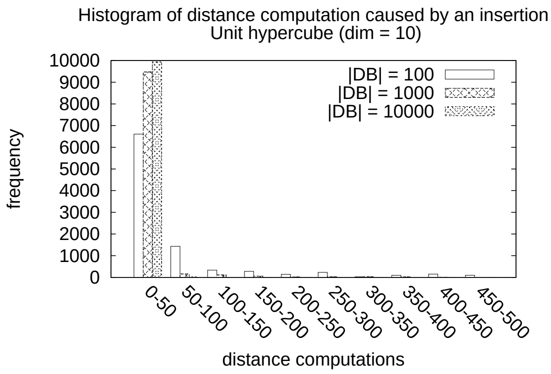

Figure 5: Histogram of the distance computations caused by an insertion for 100, 1000, and 10 000 artificial database sizes in dimension 10. Only the first 10 bins are showed

English: is an English dictionary of 69 069 words extracted from the dictionary of the GNU spell checker (ftp://ftp.gnu.org/gnu/aspell/dict/0index.html). In this case the edit distance was used as dissimilarity measure.

Contour: is a set of 10 000 8-directions contour strings extracted from the NIST Special Database 3 (Garris and Wilkinson, 1994). NIST database contains 128×128 black and white (bilevel) images of handwritten digits that was collected among Census Bureau employees.

First, in order to study the distribution of the number of distance computations needed to rebuild the index when an object is inserted, 10 000 insertions of an object over a fixed MDF-tree with sizes 100, 1000, and 10 000 was made. The number of distance computations were counted and its histogram is depicted in Figure 5. Only the case for the uniform distribution in a 10 dimensional space is plotted, the other cases show a similar behaviour.

It can be observed that independently of the size of the tree, almost all the insertions cause very few distance computations (left side). On the other hand, there are very few insertions that cause a large number of distance computations (right side). In all the experiments, rarely more than a hundred distance computations have been made in one insertion.

Next experiments are intended to study the behaviour of the algorithm when inserting objects in a database. For that, 10 000 different databases were generated for each size varying from 100 to 10 000 in steps of 100. Each point in the plots is the average number of the distance computations provoked by an insertion in each of the 10 000 MDF indexes build from these databases. The 95% percentile was also computed and plotted in the following experiments. That means that the 95% of the cases make a number of distance computations under this curve.

Moreover, the theoretical upper bound was plotted to check experimentally its validity. In order to do that, for each MDF tree, the values of the unbalance factor α was computed for each node of the tree. In order to meet the definition, the α for a tree should be the maximum α of its nodes. Instead of that, the 95% percentile of the node α's was computed to avoid pathological high values of α. Note that doing so the predicted values for the upper bound are going to be lower than if the maximum would be computed. The figures also show the value of the α factor for the corresponding experiment. The results are showed in Figure 6 for the artificial data and Figure 7 for the real data experiments.

The experimental results fits very well with the theoretical prediction. It can be seen that in all the cases the distance computations caused by an insertion seems to grow very slowly with the database size. Moreover, the 95% percentile decreases as the database size increases. This effect is due to the fact that the cost of the worst case increases much faster that the average case. Then, in order to compensate for the few worst case events, many events have to be very cheap.

7. Conclusions

In this work we have proposed a simple but efficient algorithm to insert objects in a MDF-tree. This algorithm, unlike others, has the property that the index obtained after the insertion is the same as the one obtained if a complete rebuild would be made. Then, the search efficiency does not degrade when

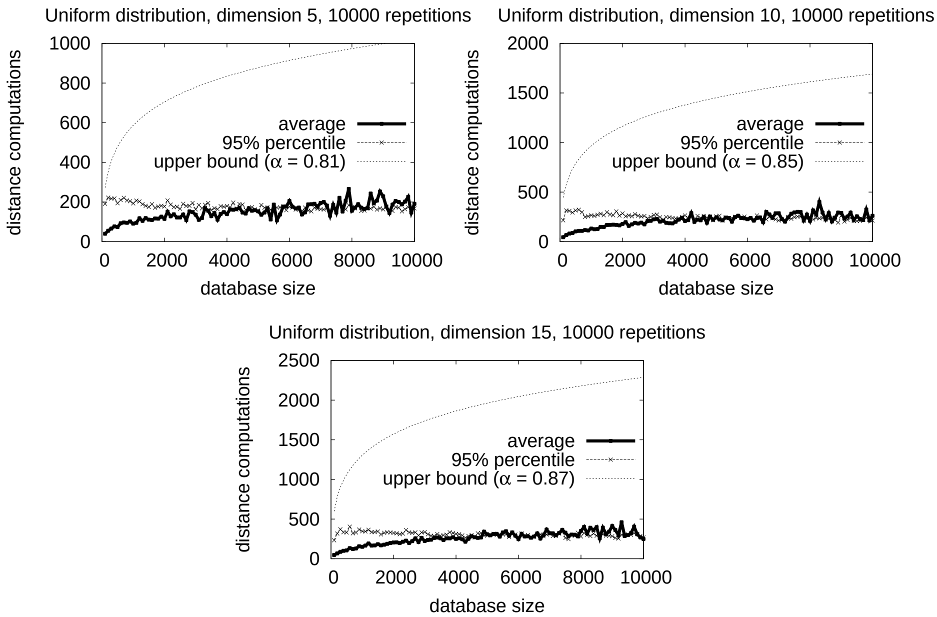

Figure 6: Experimental average number and theoretical upper bound of the distance computations caused by an insertion for increasing size database sets and for dimensions 5, 10 and 15.

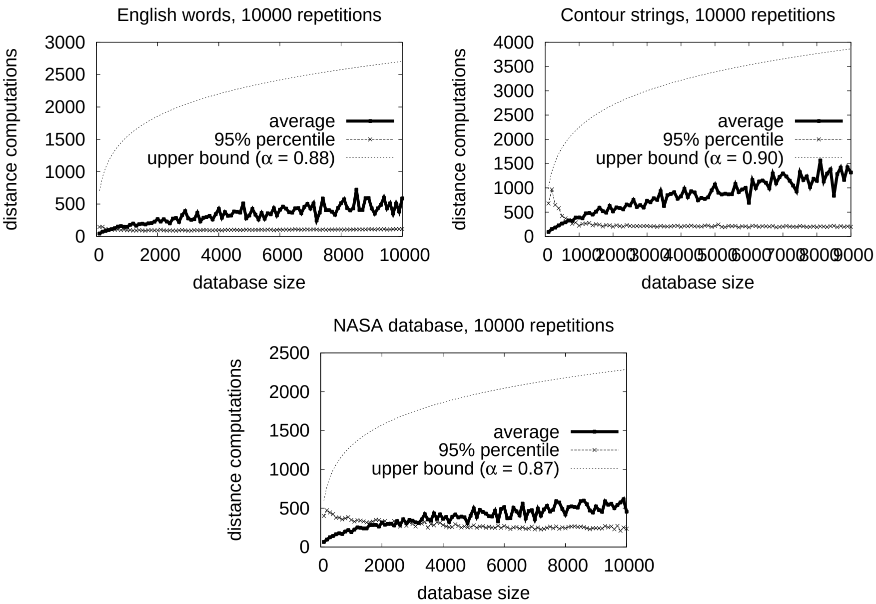

Figure 7: Experimental average number and theoretical upper bound of the distance computations caused by an insertion for increasing size database sets and for English, NASA and Contour databases.

insertions are incrementally done.

We have shown that the average number of distance computations (and the average complexity) is bounded by a function that grows with the square of the logarithm of the size of tree ($\log^2(n)$). This is a big improvement if compared with the "naïve" approach that grows with $n \log(n)$.

Moreover, we have tested this upper bound with several artificial and real data experiments. These experiments, as well as confirming the theoretical results, also shows that the 95% percentile decreases when the database size increases. That means that when the database size increases, pathological insertions, which provokes wide reconstructions of the tree, becomes very uncommon.

Acknowledgements

This work has been supported in part by grants TIN2009-14205-C04-01 from the Spanish CICYT (Ministerio de Ciencia e Innovación), the IST Programme of the European Community, under the Pascal Network of Excellence, IST-2002-506778, and the program CONSOLIDER INGENIO 2010 (CSD2007-00018).

Battiato, S., Blasi, G. D., Reforgiato, D. Advanced indexing schema for imaging applications: three case studies. Image Processing, IET 1 (3), 249–268.

Brin, S., 1995. Near neighbor search in large metric spaces. In: Proceedings of the 21^{st} International Conference on Very Large Data Bases. pp. 574–584.

Carrasco, R. C., Forcada, M. L., 1995. A note on the nagendraprasad-wang-gupta thinning algorithm. Pattern Recognition Letters 16, 539–541.

Cha, S.-H., Srihari, S. N., 2002. On measuring the distance between histograms. Pattern Recognition 35, 1355–1370.

Chávez, E., Navarro, G., Baeza-Yates, R., Marroquin. Searching in metric spaces. ACM Computing Surveys 33 (3), 273–321.

Ciaccia, P., Patella, M., Zezula, P., 1997. M-tree: An efficient access method for similarity search in metric spaces. In: Proceedings of the 23rd International Conference on VLDB. Morgan Kaufmann Publishers, Athens, Greece, pp. 426–435.

Escolano, F., Vento, M. (Eds.), 2007. Graph-Based Representations in Pattern Recognition, 6th IAPR-TC-15 International Workshop, GbRPR 2007, Alicante, Spain, June 11-13, 2007, Proceedings. Vol. 4538 of Lecture Notes in Computer Science. Springer.

Fu, A. W., Chan, P. M., Cheung, Y., Moon, Y. S., 2000. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. VLDB Journal 9, 154–173.

Garris, M., Wilkinson, R., 1994. NIST special database 3: Handwritten segmented characters.

Giacinto, G., 2007. A nearest-neighbor approach to relevance feedback in content based image retrieval. In: CIVR '07: Proceedings of the 6th ACM international conference on Image and video retrieval. ACM, New York, NY, USA, pp. 456–463.

Gómez-Ballester, E., Micó, L., Oncina. Some approaches to improve tree-based nearest neighbour search algorithms. Pattern Recognition 39 (2), 171–179.

Hjaltason, G., Samet, H., 2003. Index-driven similarity search in metric spaces. ACM Trans. Database Syst. 28 (4), 517–580.

Kalantari, I., McDonald, G., 1983. A data structure and an algorithm for the nearest point problem. IEEE Trans. Software Eng. 9 (5), 631–634.

Lundsteen, C., Philip, J., Granum, E., 1980. Quantitative analysis of 6985 digitized trypsin g-banded human metaphase chromosomes. Clin Genet 18 (5), 355–70.

Micó, L., Oncina, J., Carrasco, R., 1996. A fast branch and bound nearest neighbor classifier in metric spaces. Pattern Recognition Letters 17, 731–73.

Micó, L., Oncina, J., Vidal, E., 1994. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. Pattern Recognition Letters 15, 9–17.

Micó, L., Oncina, J., 2009. Experimental analysis of insertion costs in a naïve dynamic mdf-tree. Lecture Notes in Computer Science 5524, 402–408.

Navarro, G., 2002. Searching in metric spaces by spatial approximation. VLDB Journal 11 (1), 28–46.

Navarro, G., Reyes, N., 2002. Fully dynamic spatial approximation trees. In: Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002), LNCS 2476. Springer, pp. 254–270.

Navarro, G., Reyes, N., 2008. Dynamic spatial approximation trees. J. Exp. Algorithmics 12, 1–68.

Noltemeier, H., Verbarg, K., Zirkelbach, C., 1992. Monotonous bisector* trees – a tool for efficient partitioning of complex scenes of geometric objects. In: Data Structures and Efficient Algorithms. pp. 186–203.

Procopiuc, O., Agarwal, P. K., Arge, L., Vittery, J. S., 2003. Bkd-tree: A dynamic scalable kd-tree. In: In Proc. International Symposium on Spatial and Temporal Databases. pp. 46–65.

Katayama, N., Satoh, S., 1999. Analysis and Improvement of the SR-tree: an Index Structure for Nearest Neighbor Searches. 6th DIMACS Implementation Challenge Workshop: Near Neighbor Searches, Baltimore (Maryland) (USA) (1999)

Torralba, A., Fergus, R., Freeman, W., 2008. 80 million tiny images: A large data set for nonparametric object and scene recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence 30 (11), 1958–1970.

Yianilos, P., 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms. pp. 311–321.

Zezula, P., Amato, G., Dohnal, V., Batko, M., 2006. Similarity Search: The Metric Space Approach. Springer.