RESTRUCTURING VERSUS NON RESTRUCTURING INSERTIONS IN MDF INDEXES

Aureo Serrano, Luisa Micó and Jose Oncina
Departamento de Lenguajes y Sistemas Informáticos, Universidad de Alicante, E-03080 Alicante, Spain

Keywords: Similarity search, Dynamic indexes, Tree insertions.

Abstract: MDF tree is a data structure (index) that is used to speed up similarity searches in huge databases. To achieve its goal the indexes should exploit some property of the dissimilarity measure. MDF indexes assume that the dissimilarity measure can be viewed as a distance in a metric space. Moreover, in this framework is assumed that the distance is computationally very expensive and then, counting distance computations is a good measure of the time complexity.

To tackle with a changing world, a problem arises when new points should be inserted in the index. Efficient algorithms should choose between trying to be efficient in search maintaining the "ideal" structure of the index or trying to be efficient when inserting but worsening the search time.

In this work we propose an insertion algorithm for MDF trees that focus on optimizing insertion times. The worst case time complexity of the algorithm only depends on the depth of the MDF tree. We compare this algorithm with a similar one that focuses on search time performance. We also study the range of applicability of each one.

1 INTRODUCTION

In the area of similarity based searches, the metric space searching is an arising general approach that has received special attention. The main feature of metric space searching is that no assumption about the structure of the objects to search (points) is made. 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), histograms of images (Cha and Srihari, 2002), etc.

At present, several techniques have been proposed based on this approach, however, many of them are static (Yianilos, 1993)(Brin, 1995)(Micó et al., 1994)(Navarro, 2002). That is, the insertion or deletion of an object of the database, requires an expensive complete rebuilding of the index that has been created to speed up the search. Some indexes that tolerate efficient insertions have been developed, but their quality, in search time, degrades as insertions goes on and requires periodic rebuilds.

In (Micó and Oncina, 2009) was proposed a technique that is free of this inconvenience. The produced indexes, based on MDF (Most Distant to the Father) trees (Micó et al., 1996)(Gómez-Ballester et al., 2006), do not degrades with insertions. By construction, the obtained indexes are exactly the same that would be obtained if all the inserted elements were available when the index was firstly built. Moreover, the proposed technique is shown to allow insertions making, on average, a number of distance computations bounded by $O(\log^2 n)$, where n is the number of objects in the database.

In some applications an average of $O(\log^2 n)$ distance computations can be too high. In this work, we are going to propose an algorithm to insert in MDF trees, that tights this bound at expenses of allowing some degradation. Now, the worst case is bounded by $O(\log n)$, the depth of the tree.

In the Experiments section we study the degradation and compare both strategies.

2 THE MDF-TREE

The MDF tree is a binary indexing structure based on a *hyperplane partitioning* approach.

The MDF building algorithm (Alg. 1) begins by randomly choosing a point in the database as representative of the root of the tree. Then, it searches the most distant point of the actual representative and

Serrano A., Micó L. and Oncina J. (2012).
RESTRUCTURING VERSUS NON RESTRUCTURING INSERTIONS IN MDF INDEXES.
In *Proceedings of the 1st International Conference on Pattern Recognition Applications and Methods*, pages 474-480
DOI: 10.5220/0003877204740480
Copyright © SciTePress

Algorithm 1: Building MDF.

```
function build($\ell$,$S$)
input  : $S \cup \{\ell\}$: set of points to include in the
         tree
         $\ell$: future left representative of the tree
output: $T$: tree
begin
    $T$ = emptyTree() ;
    if isEmpty($S$) then
        $M_T = \ell$ ;    // New representative
        $r_T = 0$ ;         // Assigns radius
    else
        $r = \text{argmax}_{x \in S} d(\ell,x)$;
        $r_T = d(\ell,r)$ ;    // Assigns Radius
        $S_\ell = \{x \in S | d(\ell,x) < d(r,x)\}$ ;
        $S_r = \{x \in S | d(\ell,x) \geq d(r,x)\} - \{r\}$ ;
        $T_L$ = build($\ell$,$S_\ell$);
        $T_R$ = build($r$,$S_r$);
    end
    return $T$
end
```

splits the database in two. The present representative will become the representative of the left node and its farthest point will become the representative of the right node. Each point in the database will be distributed depending on which representative is the closest. This procedure is repeated recursively while there are still points in the database. In each node of the tree the representative and the distance to the farthest point is stored. This information is used by a branch and bound algorithm to find the nearest neighbour in sublinear time.

The main difference with other tree based indexes is that, in the MDF, the representative of a left child is always the same that its father. This allows further savings in distance computations when searching.

2.1 Dynamic Insertion

When a point is going to be inserted in the tree, firstly the tree is depth-traversed in order to find the node where the point should be placed. The place to insert the new item in the tree is the node were the distance between the new element and the left representative exceeds the current radius.

In the restructuring algorithm the subtree is replaced by a new MDF subtree built with all the points in the old subtree plus the point to be inserted (Alg. 2). It was shown (Micó and Oncina, 2009) that the insertions usually happens in very deep levels of the tree, then rebuildings are usually not very costly.

In the non restructuring algorithm (Alg. 3),the radius of the node is updated and the procedure is recursively called to insert the point in the corresponding subtree. Then finally, the point is always inserted as a new leaf of the tree. It is easy to see that the time complexity is, in the worst case, the depth of the tree

Algorithm 2: Restructuring insertion.

```
function insert($T$,$x$)
input  : $T$: MDF-tree
         $x$: object to be inserted
output: $T$: tree
begin
    if $d(M_T,x) > r_T$ then      // $r_T$ changes
        // Rebuild the branch
        return build($M_T$,$T \cup \{x\} - \{M_T\}$);
    end
    if isEmpty($T_L$) then       // Is a leaf
        // Build the leave
        return build($M_T$,$\{x\}$);
    end
    // Otherwise, follow the search
    $d_\ell = d(M_{T_L},x)$;
    $d_r = d(M_{T_r},x)$;
    if $d_\ell < d_r$ then  // follow left branch
        $T_L$ = insert($T_L$,$x$)
    else
        $T_R$ = insert($T_R$,$x$)
    end
    return $T$;
end
```

Algorithm 3: Non restructuring insertion.

```
function insert($T$,$x$)
input  : $T$: MDF-tree
         $x$: object to be inserted
output: $T$: tree
begin
    if $d(M_T,x) > r_T$ then      // $r_T$ changes
        ;                          // Update $r_T$
        $r_T = d(M_T,x)$ ;
    end
    if isEmpty($T_L$) then       // Is a leaf
        // insert as a new leaf
        return build($M_T$,$\{x\}$) ;
    end
    // Otherwise, follow the search
    $d_\ell = d(M_{T_L},x)$;
    $d_r = d(M_{T_r},x)$;
    if $d_\ell < d_r$ then  // follow left branch
        $T_L$ = insert($T_L$,$x$)
    else
        $T_R$ = insert($T_R$,$x$)
    end
    return $T$;
end
```

that grows with $\log n$.

This procedure avoids to rebuild the subtree but provokes that the resulting tree is no longer MDF: it can exists sibling nodes such that the right representative is not longer the farthest point of the left one. This leads to a degradation on the performance that will be observed in the experiments section.

3 EXPERIMENTS

Three sets of databases (synthetic and real data) were used in our experiments:

- Synthetic prototype sets, generated from uniform distributions in the unit hypercube, with a dimension of 15. The Euclidean distance was used as dissimilarity measure.
- Two string databases:
  - A database of 69 069 words of an English dictionary. The words were randomly chosen from the entire dictionary.
  - A database of 61 293 strings representing contour chains (Freeman, 1970) of the handwritten digits in NIST database.

  In both cases, the edit distance (Levenshtein, 1965)(Wagner and Fischer, 1974) was used as dissimilarity measure.

3.1 Index Depth Experiments

The first set of experiments was devoted to study the rise of the depth as the trees are degraded by an increasing number of insertions using the non restructuring method. In order to study that three MDF indexes representing 5000, 10 000 and 15 000 points databases with 15 dimensional points uniformly distributed in the unit hypercube were chosen (other dimensions where checked with similar results). Over these indexes, 5000 insertion (using the non restructuring method) were performed. In steps of 500 insertions, the depth of the tree was measured. To have a reference, the same experiments were repeated but using the restructuring method (note that there is not degradation when this method is used.) The results are showed in Fig. 1.

In (Serrano et al., 2011), it was noted that MDF trees tend to be very unbalanced. Then, the quick growth of the deep in the restructuring method is quite natural. Although this behaviour may seem a weakness is in fact a strength. In the same work, it was shown that the unbalance gives a flexibility that can

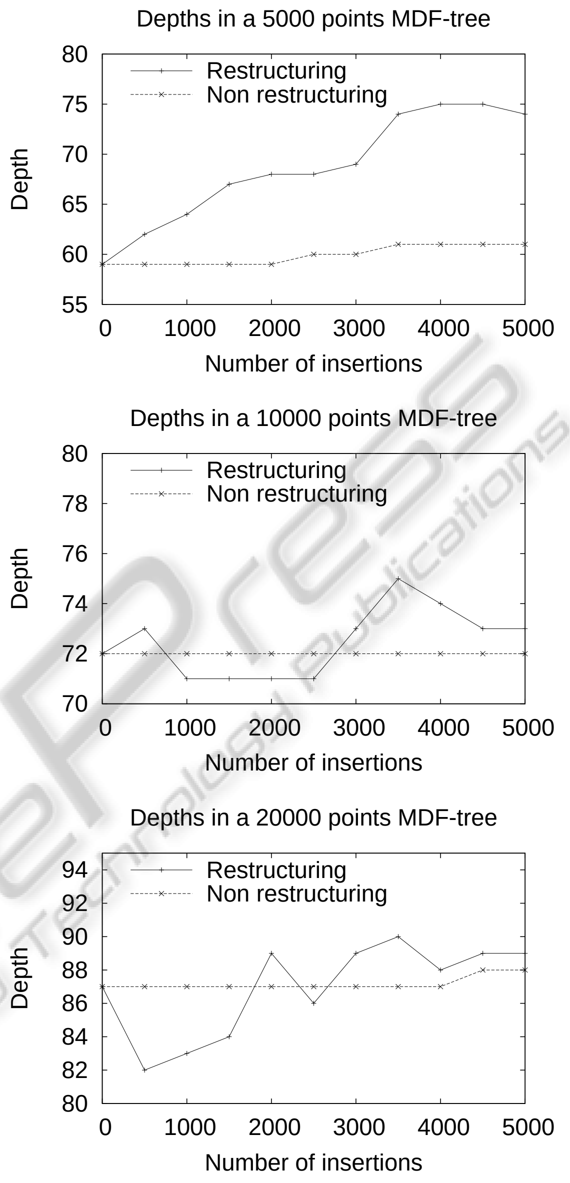

Figure 1: Depths of the trees after the insertions, with a dataset of samples uniformly distributed points in the unit hypercube of dimension 15.

be exploited to reduce the expected number of distance computations. Then, deeper MDF trees does not imply less efficient indexes.

On the other hand the insertions using the non-restructuring method are made in the leaves. Then, as the MDF trees are very unbalanced, insertions in the deeper branch are very unlikely.

The experiments were repeated with the *English* (Fig. 2) and *NIST* (Fig. 3) databases observing a similar behaviour.

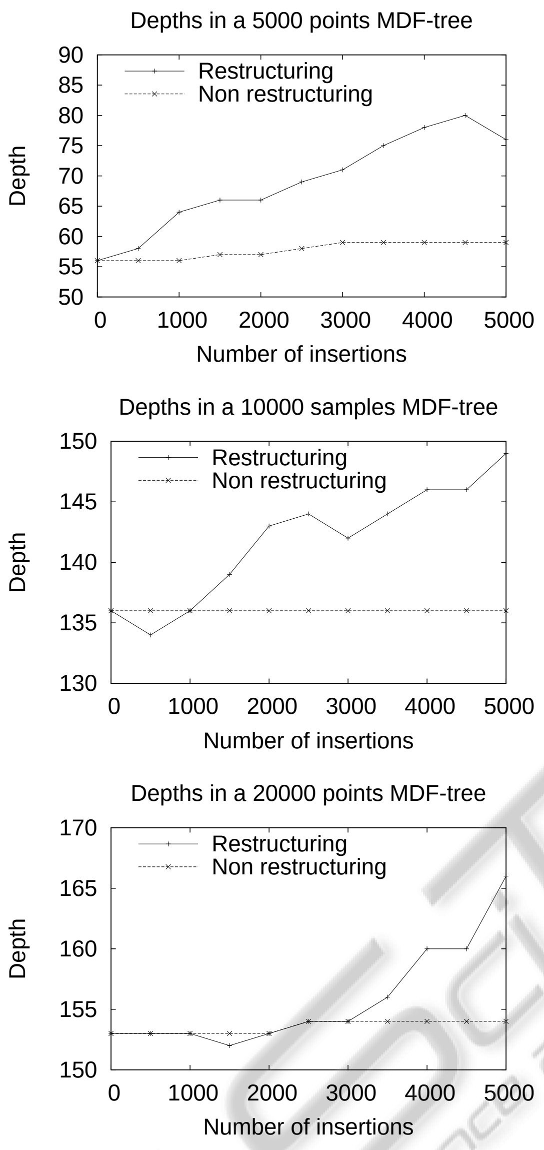

Figure 2: Depths of the trees after the insertions, with a dataset of samples from the English dictionary.

3.2 Insertion Speed up Experiments

In these experiments the expected number of distance computations provoked by an insertion using both techniques are compared.

In order to do that, a series of MDF trees, for increasing database sizes (from 250 to 10 000 in steps of 250) were built. In each database, a new point was inserted (with both methods) and the involved distance computations were counted. The experiment was repeated for 10 000 series of databases.

The experiments were also repeated for *English* and *NIST* databases.

It can be observed (Fig. 4) that the non-restructuring method is much faster that the restructuring one.

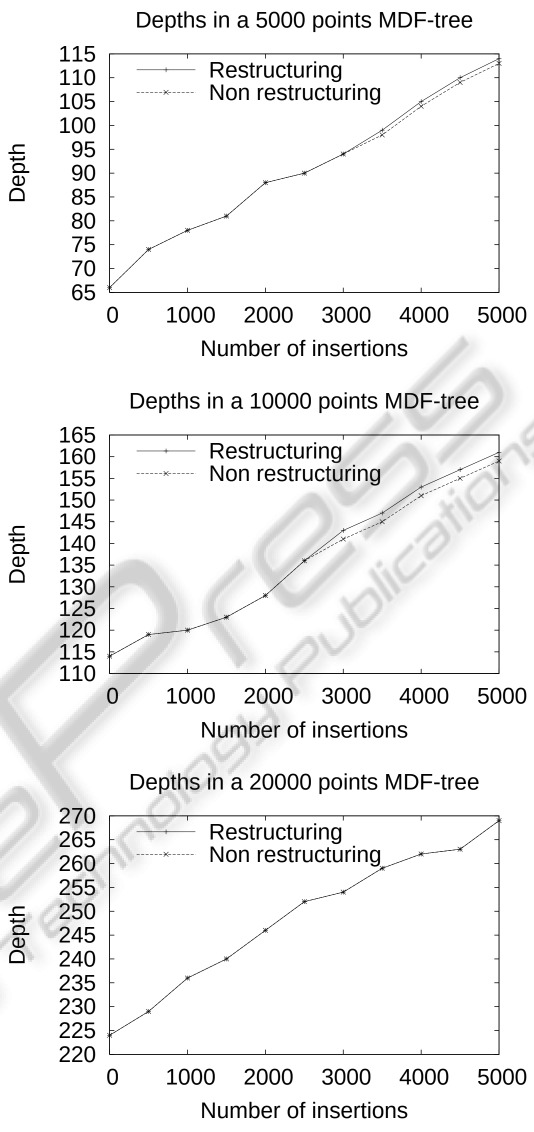

Figure 3: Depths of the trees after the insertions, with a dataset of samples from the NIST database.

This result was expected since the number of distance computations (in the average case) for the restructuring method is bounded by $O(\log^2 n)$ and, as the non-restructuring method is essence a tree descent, the number of distance computations is bounded (in the worst case) by $O(\log n)$.

3.3 Performance Degradation Experiments

These experiments are devoted to study the penalty to be paid, in search time, due to the degradation of the

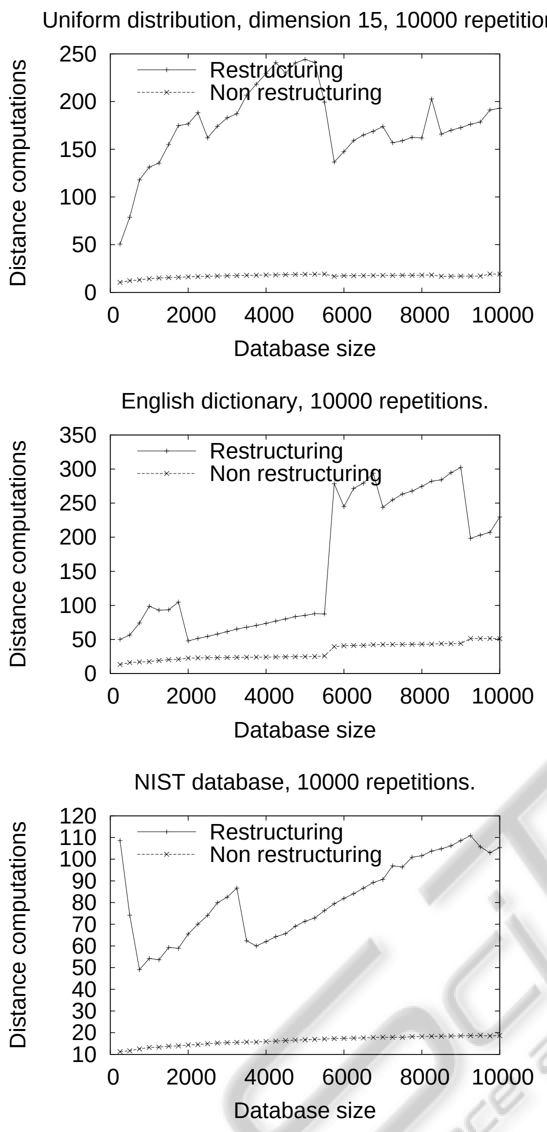

Figure 4: Average number of the distance computations caused by an insertion, for increasing size database sets and for synthetic and real data.

MDF trees.

In this set of experiments it was used a similar setting to the used in section 3.1. Three MDF trees with 5000, 10000 and 15000 points were built. In each tree an increasing number of points (from 0 to 5000) are inserted using the non restructuring method. Each time 500 insertions was done a search of 5000 independent test points was made. In each search, the number of distances were counted and the average for the 5000 is displayed. The same experiment was repeated using the restructuring insertion method for control.

It can be seen (Fig. 5) that the difference increases as the tree.

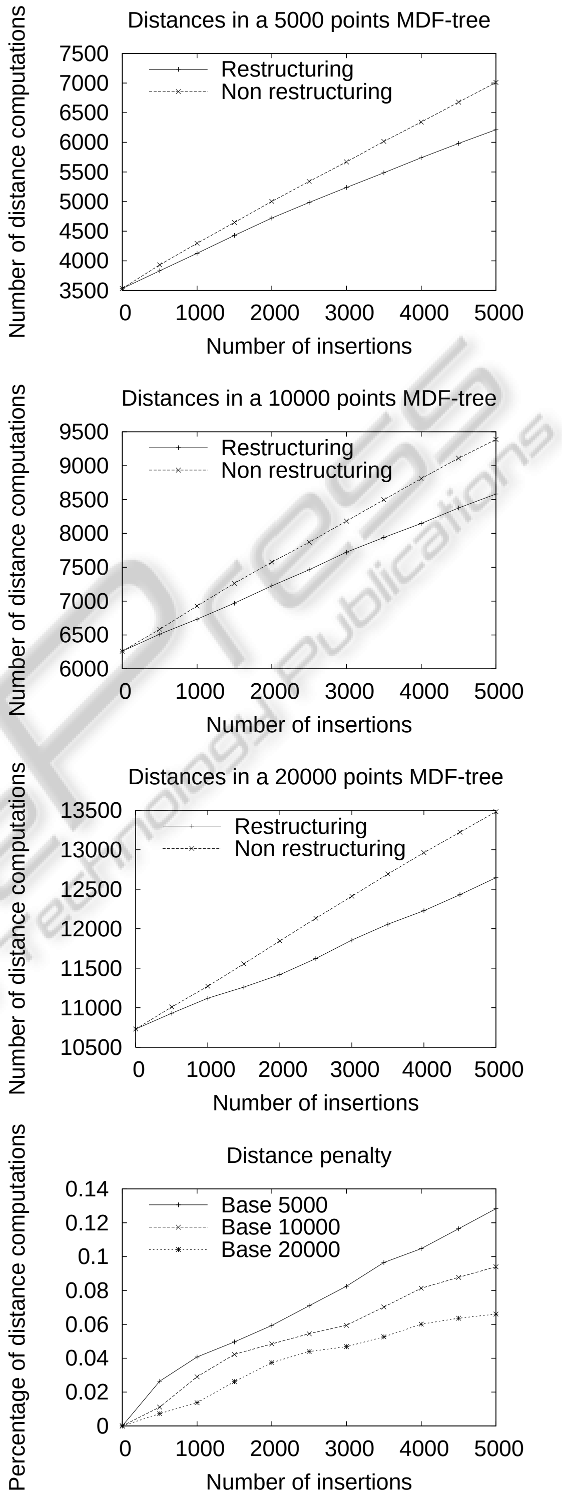

Figure 5: Average number of distance computations after the insertions, with a dataset of samples uniformly distributed points in the unit hypercube of dimension 15.

The experiments were repeated with *English* (Fig. 6) and *NIST* (Fig. 7) databases.

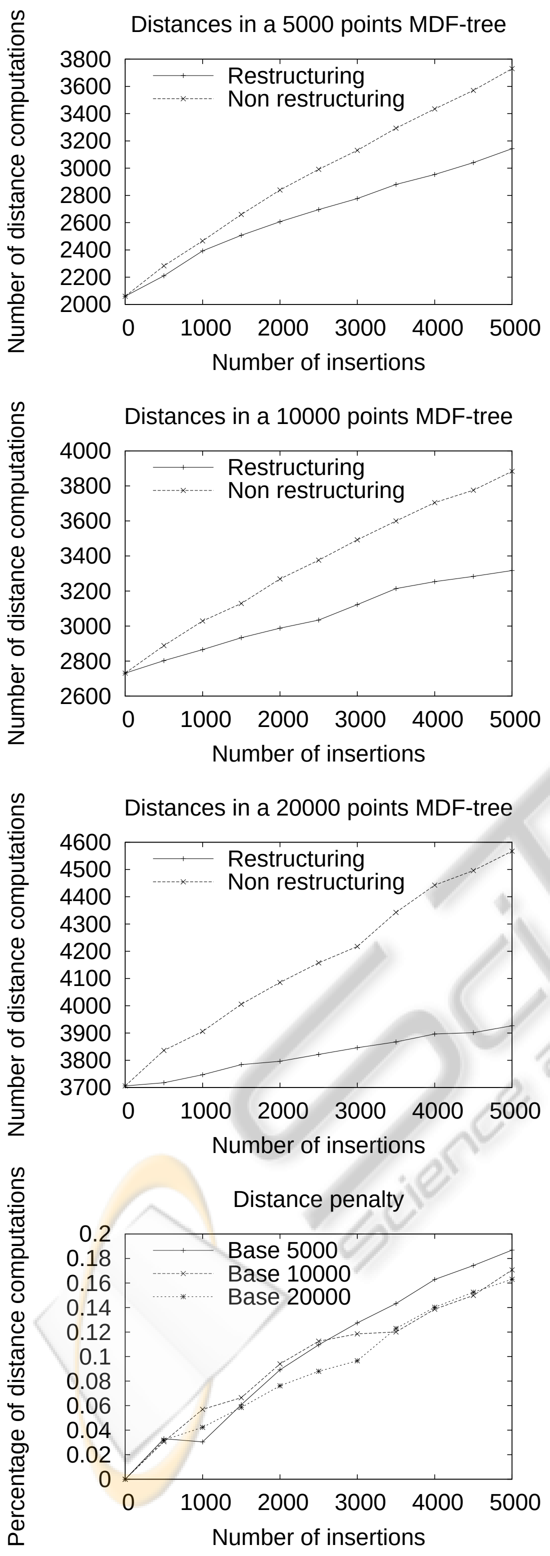

Figure 6: Average number of distance computations after the insertions, with a dataset of samples from the English dictionary.

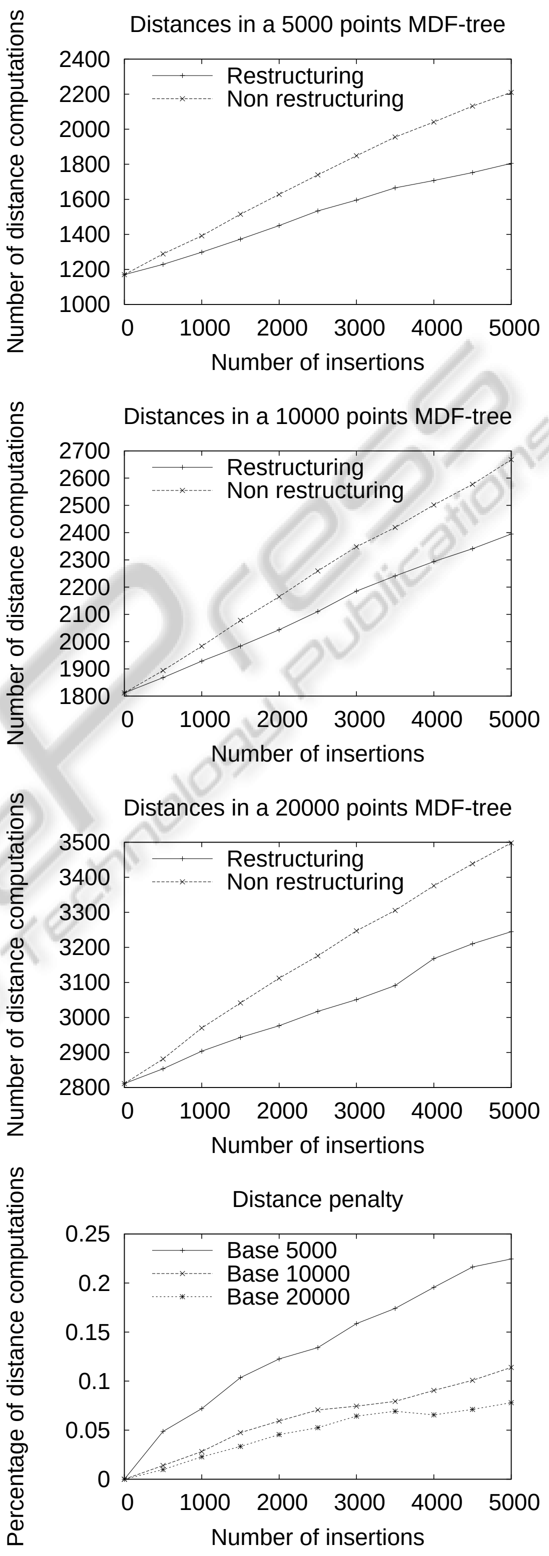

Figure 7: Average number of distance computations after the insertions, with a dataset of samples from the NIST database.

4 CONCLUSIONS

In this work we propose an insertion algorithm for MDF trees. This algorithm focus on reducing the time complexity when inserting at expenses of search time complexity.

We have compared this insertion with the proposed in (Micó and Oncina, 2009) that follows a very similar strategy but focusing on preserving the MDF structure and then, its efficiency when searching.

We have found that a big speed up can be obtained at the expense of worsening search times when the amount of insertions is moderate.

The study suggests that, when the number of insertions exceeds a threshold, a restructuring of the index should be performed to recuperate the MDF structure. This issue will be addressed in future works.

ACKNOWLEDGEMENTS

The authors thank the Spanish CICyT for partial support of this work through projects TIN2009-14205-C04-C1, the IST Programme of the European Community, under the PASCAL Network of Excellence, (IST–2006-216886), and the program CONSOLIDER INGENIO 2010 (CSD2007-00018).

REFERENCES

Brin, S. (1995). Near neighbor search in large metric spaces. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 574–584.

Carrasco, R. C. and Forcada, M. L. (1995). A note on the nagendraprasad-wanggupta thinning algorithm. *Pattern Recognition Letters*, 16:539–541.

Cha, S. H. and Srihari, S. N. (2002). On measuring the distance between histograms. *Pattern Recognition*, 35:1355–1370.

Freeman, H. (1970). Boundary encoding and processing. *Academic Press*, Picture Processing and Psychopictorics:241–266.

Gómez-Ballester, E., Micó, L., and Oncina, J. (2006). Some approaches to improve tree-based nearest neighbour search algorithms. *Pattern Recognition*, 39(2):171–179.

Levenshtein, V. I. (1965). Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk*, (163 (4)):845–848.

Lundsteen, C., Philip, J., and Granum, E. (1980). Quantitative analysis of 6985 digitized trypsin g-banded human metaphase chromosomes. *Clin Genet*, 18(5):355–370.

Micó, L. and Oncina, J. (2009). Experimental analysis of insertion costs in a naïve dynamic mdf-tree. In Araújo, H., Mendonça, A. M., Pinho, A. J., and Torres, M. I., editors, *IbPRIA*, volume 5524 of *Lecture Notes in Computer Science*, pages 402–408. Springer.

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

Micó, L., Oncina, J., and 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.

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

Serrano, A., Micó, L., and Oncina, J. (2011). Impact of the initialization in tree-based fast similarity search techniques. In Pelillo, M. and Hancock, E. R., editors, *SIMBAD*, volume 7005 of *Lecture Notes in Computer Science*, pages 163–176. Springer.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM*, 21 (1):168–173.

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*, pages 311–321.