# Which Fast Nearest Neighbour Search Algorithm to Use?

Aureo Serrano, Luisa Micó, and Jose Oncina

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante
E-03080 Alicante, Spain
{aserrano,mico,oncina}@dlsi.ua.es

**Abstract.** Choosing which fast Nearest Neighbour search algorithm to use depends on the task we face. Usually kd-tree search algorithm is selected when the similarity function is the Euclidean or the Manhattan distances. Generic fast search algorithms (algorithms that works with any distance function) are only used when there is not specific fast search algorithms for the involved distance function.

In this work we show that in real data problems generic search algorithms (*i.e.* MDF-tree) can be faster that specific ones (*i.e.* kd-tree).

## 1 Introduction

One of the most useful and simplest similarity search techniques for Pattern Recognition is the Nearest Neighbour (NN) rule. Given an object database and a query object the goal is to find the object in the database that minimizes a similarity measure (usually called *distance*) with respect to the query object. NN rule can be trivially implemented comparing the query with all the objects in the database. However, when the databases are large or the distance function is very time consuming, this approach becomes very inefficient.

A classical method to speed up the search is to build up, in preprocess time, a data structure (*index*), usually a tree, to assist the search algorithm. The use of the index leads to significant reductions in time by means of some branch and bound techniques.

The several ways of building these indexes have resulted in many search algorithms. According to the information the search algorithm can access we distinguish two types of algorithms:

- *generic* algorithms: this type of algorithms does not have access to the way the objects are represented and only relies on some properties of the distance (*i.e.* the triangular inequality) to speed up the search.
- *specialized* algorithms: they have access to the way the objects are represented and then, they only work with some type of distances. Very often the objects are represented as vectors of real numbers and the distance function is the Euclidean distance ($L_2$) or the Manhattan distance ($L_1$). Some algorithms (*vector-based* algorithms) can access to the coordinates of the vectors and then, they are allowed to use further speed up techniques.

J.M. Sanches, L. Micó, and J.S. Cardoso (Eds.): IbPRIA 2013, LNCS 7887, pp. 567–574, 2013.

Choosing which algorithm to use depends on the problem we face. If the database is small the obvious choice is the brute force search. If the database is large and there is a specialized search algorithm for the distance function, the choice is to use this search algorithm (*i.e.* kd-trees [1][2][3], R-trees [4], quadtrees [5], and more recently X-trees [6]). Otherwise use a generic algorithm (*i.e.* MDF-tree [7][8], vp-trees [9], GNAT [10], M-tree [11]).

In this paper we show that this way of proceeding is not always the optimal. We are going to compare the speed of three search algorithms: the brute force, a representative of vector-based search algorithms (kd-tree) and a representative of generic search algorithms (MDF-tree algorithm) on artificial and real data problems. The speed is measured directly in execution time (seconds) instead of on the number of distance computations as is customary in this field. This measure has the drawback of depending on the architecture of the machine that is used (*i.e.* large caches benefit the brute force algorithm) and the concrete implementation of the algorithm. The interest of such measure is that it takes into account the time expended in auxiliary tasks.

The choice of the kd-tree [1] as representative of the vector-based search algorithms is because it is widely used and there exists very efficient implementations of it [12]. The choice of the MDF-tree algorithm [8] is because it is simple enough and previous papers showed that it is very efficient.

In this paper we show that although in the artificial databases the kd-tree shows to be faster than the MDF-tree this is not always the case in real world problems.

## 2 Algorithms

Let we describe shortly the three algorithms.

### 2.1 Brute Force

Brute Force (BF) is the trivial method to search a query in a database. It completely traverses the database comparing each object with the query and getting the closest one. This method has the advantage of not requiring any preprocessing.

### 2.2 kd-Tree

The kd-tree [1][2] is a generalization of the simple binary tree used for sorting and searching. This algorithm is specially designed to take advantage of the coordinate information that is available in vector spaces, this information is used for partitioning the search space. For each non-leaf node, an object of the database (*pivot*) and a split coordinate is chosen. The space is partitioned according to the value of the splitting coordinate of the pivot (*splitting value*). All the objects with a value in the splitting coordinate smaller than the splitting value go to the left subtree, the objects with a value in the splitting coordinate

bigger than the splitting value go to the right subtree, and the pivot is stored as the root.

For the experiments in this paper we have used the ANN Library [12], that implements this algorithm.

### 2.3 MDF-Tree

The MDF-tree [8], a particularization of the mb-tree [13], is a generic algorithm that selects some objects of the data base (*pivots*) to build a recursive partitioning of the space. This partition is represented in the index as a tree. The search algorithm uses this tree in order to prune the exploration in some branches of the tree.

Neither, the index build algorithm nor the search algorithm need to know the internal structure of the objects in the database, the information of the topology of the space is obtained through the distance function only. Then, this algorithm is suitable for any distance function. In order to have meaningful prunes it is also required that the distance fulfils the triangular inequality.

There are several ways to partition the space (build the tree). In its basic form, the algorithm begins by choosing randomly an object in the data set as representative of the root of the tree ($m_\ell$). Then, the algorithm selects the most distant object of the actual representative ($m_r$). The actual representative ($m_\ell$) will become the representative of the left child and its farthest object ($m_r$) will become the representative of the right child. Each object in the database will be distributed depending on which representative is the closest to it. This procedure is repeated recursively while there are still objects in every node. In each internal node, the representative and the distance to the farthest object in the node (radius of the node) is stored for bounding purposes during the search. Each leaf node contains a single object, which is in turn the representative. In [14] was shown that using the set median of the database objects instead of a random object as root of the tree leads to additional distance computation reductions. This is the initialization used in this paper.

During the search, some rules are applied to avoid the exploration of some branches of the tree. In this work the following rules area used: FNR rule and SBR rule from the MDF-tree (see [8]), and the elimination rule from the Generalize Hyperplane tree (see [15]).

## 3 Experiments

In this section we are going to asses the runtime performance of the fast NN algorithms using artificial and real data.

Artificial databases are composed by objects obtained from a uniform distribution in the unit hypercube. Real data was obtained from several databases from the Metric Spaces Library([16]) and the UCI Machine Learning Repository([17]). Table 1 shows the main characteristics of the datasets used in this section.

**Table 1.** Main characteristics of the databases used in the experiments

| Metric Spaces Library | | | | | |
|---|---|---|---|---|---|
| **Database** | **Dimension** | **Size** | **Database** | **Dimension** | **Size** |
| COLORS | 112 | 112682 | NASA | 20 | 410150 |
| **UCI Machine Learning Repository** | | | | | |
| **Database** | **Dimension** | **Size** | **Database** | **Dimension** | **Size** |
| Statlog | 9 | 43501 | YearPredictionMSD | 12 | 515346 |
| Corel | 32 | 68041 | MiniBooNE | 50 | 130065 |
| Pamap2 | 52 | 376418 | CT Slices | 384 | 53502 |

In order to be able to compare with kd-tree algorithm, only $L_1$ and $L_2$ distances (Manhattan and Euclidean distances) was used.

The experiments measure the total time, in seconds, that algorithms take to perform 10000 searches with different database configurations. The time shown in the figures and tables are the average of five repetitions of each experiment.

### 3.1 Experiments with Artificial Data

In the first experiment a training set of 20000 objects for several dimensions were generated.

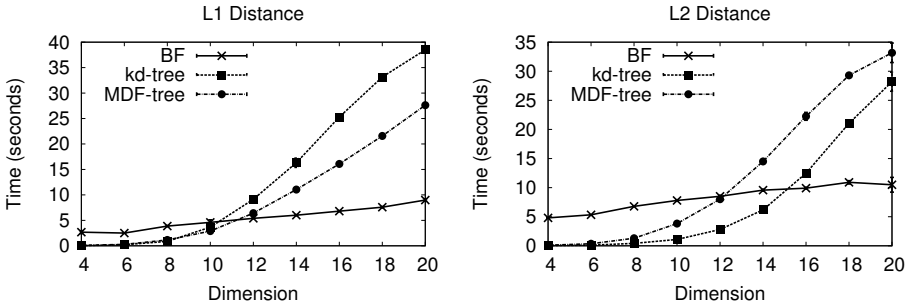


**Fig. 1.** Total time in seconds for 10000 searches, with a database of samples uniformly distributed in the unit hypercube. Training of 20000 objects and dimension variable.

Fig. 1 shows that the runtime of tree-based algorithms (kd-tree and MDF-tree search algorithms) increases dramatically with dimensionality. As expected, the time of the brute force algorithm grows linearly with the dimension. The advantages of brute force algorithm (BF) in this case (simplicity, sequentially access to the memory, efficient use of the cache memory), makes that, despite that $2 \cdot 10^8$ distances were computed, the runtime does not increase significantly.

In a second set of experiments we have analysed the influence of the size of training databases on runtime. For this experiment we have used objects randomly drawn from the uniform distribution in the unit hypercube with dimension 10 and 20.

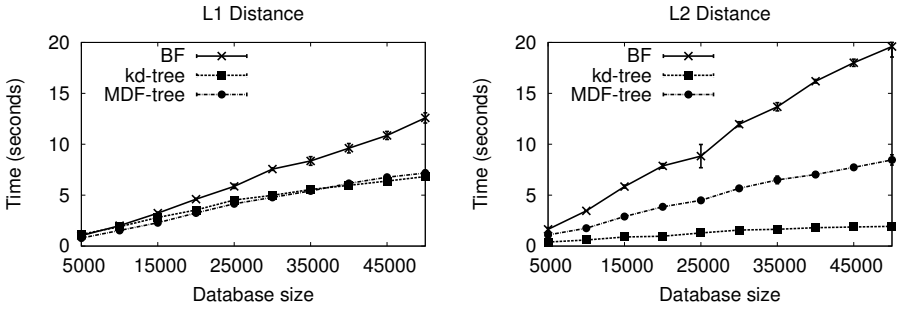


**Fig. 2.** Total time in seconds for 10000 searches, with a database of samples uniformly distributed in the unit hypercube. Dimension 10 with variable size of the training set.

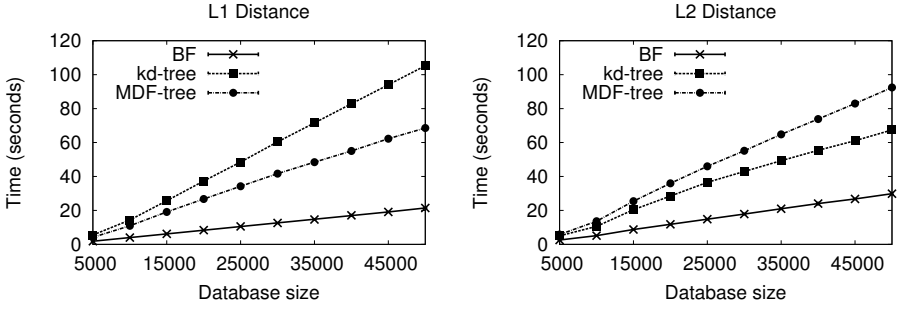


**Fig. 3.** Total time in seconds for 10000 searches, with a training set of samples uniformly distributed in the unit hypercube. Dimension 20 with variable size of the training set.

It is well known that in high dimensional spaces the pruning used in tree-based algorithms decreases quickly with the dimensionality. Then, not surprisingly, experiments (fig. 2 and fig. 3) shows that brute force algorithms outperforms tree-based in such spaces.

Experiments also show that kd-tree runtime increases when using $L_1$ instead of $L_2$ (perhaps due to the reduced efficiency of partial distance pruning with $L_1$ distance). But at the same time, $L_1$ is much faster to compute than $L_2$. That causes a time improvement of MDF-tree algorithm (note that MDF-tree only relies in distance computations to pruning).

### 3.2 Experiments with Real Data

We selected two repositories with real databases where the data has a vector representation.

From the "Metric Space Library" repository we selected the COLORS and the NASA database.

In these experiments (fig. 4 and 5), due to the very high dimension of the data, brute force performs in the same way as with artificial data. However, tree-based algorithms behaves nicely, due to the low intrinsic dimensionality[18] of the data.

With respect to the tree-based algorithms, the behaviour is similar to the observed in artificial data, MDF-tree and kd-tree has an opposite behaviour: kd-tree worsens when $L_1$ is used while MDF-tree improves. Note that with NASA database MDF-tree beats kd-tree with both distances.

To confirm these results, the experiments were repeated using several databases from the UCI Machine Learning Repository ([17]).

The results of the experiments with UCI databases show several trends: when $L_1$ distance is used (Table 2), MDF-tree gives the best results. With $L_2$ distance (Table 3), in almost all the cases, the kd-tree performs best, but there are some clear exceptions, as with the CT Slices database when the MDF-tree consumes up to 65% less time than the kd-tree.

As we can see from these results, when we use real data databases, the type of distance used can be a determining factor when choosing which algorithm to use, but not the dimension of the elements of the database.

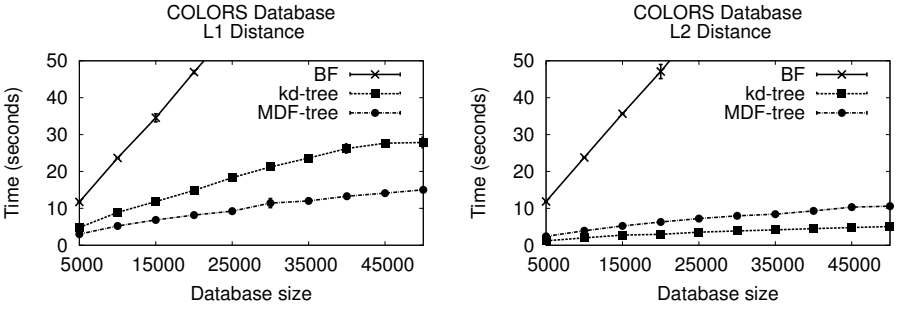


**Fig. 4.** Total time in seconds of 10000 searches using the COLORS database

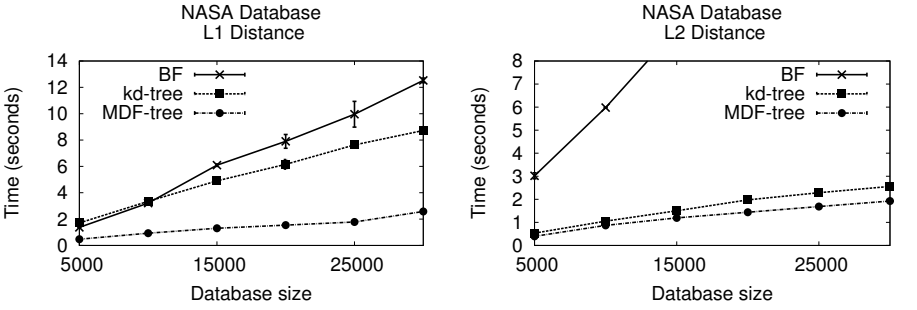


**Fig. 5.** Total time in seconds of 10000 searches using the NASA database

**Table 2.** Total time in seconds for 10000 searches with $L_1$ distance, using several UCI databases

| Training | | 5000 | | | 15000 | | | 30000 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Algorithm / DB/Dim | | BF | kd-tree | MDF | BF | kd-tree | MDF | BF | kd-tree | MDF |
| Statlog(Shuttle) | 9 | 1.6 | 0.08 | ***0.03*** | 3.56 | 0.08 | ***0.05*** | 8.83 | ***0.11*** | 0.16 |
| YearPredictionMSD | 12 | 0.97 | 0.84 | ***0.40*** | 3.02 | 2.16 | ***1.27*** | 6.56 | 4.48 | ***2.98*** |
| Corel | 32 | 2.13 | 1.07 | ***0.51*** | 9.02 | 4.08 | ***2.00*** | 18.97 | 6.92 | ***3.65*** |
| MiniBooNE | 50 | 4.50 | 0.72 | ***0.55*** | 14.38 | 1.64 | ***1.51*** | 28.93 | 2.46 | ***2.44*** |
| Pamap2 | 52 | 4.67 | 2.39 | ***0.65*** | 14.99 | 6.31 | ***1.43*** | 30.04 | 9.65 | ***1.85*** |
| CT Slices | 384 | 29.73 | 16.46 | ***2.23*** | 107.47 | 41.46 | ***2.19*** | 214.79 | 62.75 | ***2.18*** |

**Table 3.** Total time in seconds for 10000 searches with $L_2$ distance, using several UCI databases

| Training | | 5000 | | | 15000 | | | 30000 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Algorithm / DB/Dim | | BF | kd-tree | MDF | BF | kd-tree | MDF | BF | kd-tree | MDF |
| Statlog(Shuttle) | 9 | 1.62 | 0.06 | ***0.05*** | 4.99 | ***0.06*** | 0.08 | 11.09 | ***0.11*** | 0.19 |
| YearPredictionMSD | 12 | 1.57 | ***0.34*** | 0.37 | 4.71 | ***0.73*** | 1.20 | 10.10 | ***1.41*** | 2.44 |
| Corel | 32 | 3.74 | ***0.33*** | 0.66 | 9.79 | ***0.57*** | 1.75 | 19.37 | ***0.92*** | 2.79 |
| MiniBooNE | 50 | 4.34 | ***0.23*** | 0.37 | 14.03 | ***0.44*** | 1.05 | 35.57 | ***0.73*** | 1.74 |
| Pamap2 | 52 | 4.57 | 0.71 | ***0.53*** | 14.59 | 1.77 | ***1.14*** | 37.09 | 2.92 | ***1.63*** |
| CT Slices | 384 | 38.21 | 16.89 | ***9.93*** | 121.89 | 25.24 | ***10.2*** | 228.95 | 34.69 | ***8.30*** |

## 4 Conclusions and Future Works

In this work we have shown that generic search algorithms should not be *a priori* discarded with the argument that an specialized search algorithm can be used. Our experiments show that, in real data problems, MDF-tree can beat kd-tree algorithm.

The reason of this behaviour may lie in the internal dimensional structure of the real data. Probably the kd-tree results can be improved using complex dimensionality reduction techniques. Note that such techniques usualy modify the geometry of the space and the nearest neighbour can change.

Another way to improve the kd-tree is by allowing the partitions being no parallels to the axis. These techniques involve rotations and projections of the space in real time that can waste the improvements obtained by optimising the partitions.

Despite of these arguments, should be deeply explored in future works.

**Acknowledgements.** The authors thank the Spanish CICyT for partial support of this work through project TIN2009-14205-C04-C1 and la Consellería de Educación de la Comunidad Valenciana through project PROMETEO/2012/01.

## References


1. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM 18(9), 509–517 (1975)
2. Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Trans. Math. Softw. 3(3), 209–226 (1977)
3. Bentley, J.L.: Multidimensional binary search trees in database applications. IEEE Trans. Softw. Eng. 5(4), 333–340 (1979)
4. Guttman, A.: R-trees: a dynamic index structure for spatial searching. SIGMOD Rec. 14(2), 47–57 (1984)
5. Samet, H.: The quadtree and related hierarchical data structures. ACM Comput. Surv. 16(2), 187–260 (1984)
6. Berchtold, S., Keim, D.A., Kriegel, H.P.: Readings in multimedia computing and networking, pp. 451–462. Morgan Kaufmann Publishers Inc., San Francisco (2001)
7. Micó, L., Oncina, J., Carrasco, R.: A fast branch and bound nearest neighbor classifier in metric spaces. Pattern Recognition Letters 17, 731–773 (1996)
8. Gómez-Ballester, E., Micó, L., Oncina, J.: Some approaches to improve tree-based nearest neighbour search algorithms. Pattern Recognition 39(2), 171–179 (2006)
9. Yianilos, P.: 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 (1993)
10. Brin, S.: Near neighbor search in large metric spaces. In: Proceedings of the 21st International Conference on Very Large Data Bases, pp. 574–584 (1995)
11. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Proceedings of the 23rd International Conference on VLDB, Athens, Greece, pp. 426–435. Morgan Kaufmann Publishers (1997)
12. Mount, D.M., Arya, S.: Ann: A library for approximate nearest neighbor searching (2010)
13. Noltemeier, H., Verbarg, K., Zirkelbach, C.: Monotonous bisector* trees – a tool for efficient partitioning of complex scenes of geometric objects. In: Monien, B., Ottmann, T. (eds.) Data Structures and Efficient Algorithms. LNCS, vol. 594, pp. 186–203. Springer, Heidelberg (1992)
14. Serrano, A., Micó, L., Oncina, J.: Impact of the initialization in tree-based fast similarity search techniques. In: Pelillo, M., Hancock, E.R. (eds.) SIMBAD 2011. LNCS, vol. 7005, pp. 163–176. Springer, Heidelberg (2011)
15. Uhlmann, J.K.: Satisfying general proximity/similarity queries with metric trees. Inf. Process. Lett. 40(4), 175–179 (1991)
16. Figueroa, K., Navarro, G., Chávez, E.: Metric spaces library (2007), `http://www.sisap.org/Metric_Space_Library.html`
17. Frank, A., Asuncion, A.: UCI machine learning repository (2010)
18. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquin, J.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (2001)