Near neighbor search in metric and nonmetric space

Joannès Vermorel

Near neighbor search in metric and nonmetric space

Joannès Vermorel
joannes.vermorel@ens.fr

May 9, 2005

Abstract

We consider the computational problem of finding nearest neighbors in metric and nonmetric spaces. Nonmetric spaces are the generalization of the general metric spaces but without requiring the triangular inequality assumption. Non-metric spaces are often encountered. Many of the similarity measures (between images, proteins, etc) do not verify the triangular inequality. The nonmetric case lead us to introduce a novel approach, to our knowledge, to the near neighbor search problem by introducing the notions of exploration and exploration accuracy profile. Those notions lead us to introduce a new search structure, called *densitree* (contraction of "density tree"),based on classifiers to estimate point densities in nonmetric spaces. Against well established datasets, our preliminary empirical results indicates that contrary to a common belief the triangular inequality (or an equivalent pruning criterion) is not required to perform efficient near-neighbor search. Additionally, the densitrees, although very naively implemented, seems to outperform existing methods both in metric and non-metric situation.

Contents

1 Introduction

A near neighbor search consist of finding points similar to a given query point within a given dataset. Performing fast near neighbor search is a useful for many applications: pattern recognition or statistical learning [11], multimedia information retrieval, protein database search, audio and video compression.

Historically, the methods used to perform search operation was leveraging the *structure* of the data, often numerical or alphabetical. In such a simple case where a total order exist between the data points, near neighbor search are performed in logarithmic time of the size of dataset (see [8]). Traditional databases, structured around the notion of relational algebra, deal with the notion of *exact structured search*. Language like SQL emphasizes the combination of explicit total ordering criterions. This approach has the clear advantage of being computationally very efficient. Nevertheless, the main drawback of the structured search lies in the tight bound of the method itself to a certain type data. This bound between the search method and the data structure raises two issues:

- How do we deal with data that does not fit a relational representation? For example, proteins are encoded sequences of variable length of amino acids.
- How do we deal with data when structured search criterion cannot express the desired similarity? For example, images (of identical dimensions) can be represented as fixed length real vectors, but dimension-wise similarity criterion will lead to a poor notion of similarity.

Those issue have led to a more general and unifying approach of near neighbor search: searching a dataset for the points that are close a to given query point given a similarity function. A vast literature has been devoted to this model (see [5] for a review), in particular when the similarity function verifies the *distance* assumptions such as the triangular inequality (more detail given here below).

2 Neighbor search in metric space

Most of the NNS literature has been devoted to the *metric space* model. Intuitively the dataset contains points draw from a *metric space* if a point-wise measure of similarity d exists and verifies several assumptions. Those assumptions ensure that d has a behavior somewhat similar to the simple geometric euclidian distance.

Definition 1 (Metric function and metric space) *Let E be a set. A function $d : E^2 \rightarrow \mathbb{R}^+$ is said to be a metric over E if d verifies the separation assumption $\forall (x, y) \in E^2, d(x, y) = 0 \Leftrightarrow x = y$, the symmetry assumption $\forall (x, y) \in E^2, d(x, y) = d(y, x)$ and the triangular inequality $\forall (x, y, z) \in E^3, d(x, y) + d(y, z) \leq d(x, z)$. A couple (E, d) is called a metric space if d is a metric over E.*

In the following, E will be the dataspace, X the dataset with a cardinal $n = |X|$ and $d : E^2 \rightarrow \mathbb{R}^+$ a metric.

Given the metric space formalism, the near neighbor search (NNS) literature classically distinguishes two type of range queries. The **range query** is the selection within X of all points included a ball of center q the query point and of a given radius r (eg. the range), $\mathcal{B}_d(q, r) = \{x \in X | d(q, x) \leq r\}$. The **near neighbors query** is the selection within X of the k nearest points of the query point q such as $\mathcal{N}_d(q, k) = \{x_1, \ldots, x_k | \forall x \in X, d(q, x) < d(q, x_k) \Rightarrow \exists j, x = x_j\}$.

The NNS literature is devoted to algorithms performing those two types of queries. In following, when we will speak of computational cost for such algorithms, we will always refer to the number of calls to the current metric [1].

The strongest of the three metric space assumptions, see here above, is the *triangular inequality*. Intuitively the classical NNS algorithms, like [18], makes an intensive use of the triangular inequality in order to achieve a sub-linear computational cost for NNS. Such sub-linear cost is achieved by pruning the search thanks to the triangular inequality. As described in [5], if any of the other assumptions is removed, it is relatively easy to slightly to adapt the classical "metric" algorithms to such cases.

2.1 Metric trees

The *metric tree* [17, 16, 6] is a data structure that supports efficient nearest neighbor search in metric space. Intuitively, the metric tree organizes the space in a hierarchical manner, each node corresponding to a particular hyperplane that split the dataspace in two. This particular structure enable an efficient branch pruning based on the triangular inequality during the computation of a query. A remarkable feature of the metric trees is that they can be implemented into a *dynamic collection* that allows insertion and deletion. Those elements are beyond the scope of this document, please refer to [6] for more details. We will

[1] Other elements could taken into account, like overall computational cost, I/O costs and memory; see [5] for more details

now provide a more precise description of the two main algorithms related to metric trees: building a metric tree, and performing a near neighbor query.

Building a metric tree. The algorithm 1 provides the pseudo-code detail of the metric tree building. Note that an actual implementation should take care of never calling twice the distance function for a given pair of points. This can be easily achieved by caching the distance values when they will be required again later. Nevertheless, for the sake of clarity, in the algorithm 1 pseudo-code, the use of such caches have been made implicit.

Intuitively, the process consists of choosing two pivots for each node and to recursively split the set of points assigning each point to its closest pivot. The arguments of the algorithm are the distance d and the set of points X. The partition requires two *pivots* (refered as l and r in algorithm 1). The lines 1-3 are in fact a linear approximation of $\arg\max_{x,y\in Y}\{d(x,y)\}$ whose exact computation would require $\mathcal{O}(n^2)$. The recursive partition into *left* and *right* of the dataset is actually performed lines 6-14. The left and right *radii* (refered as u and v) are computed within the same loop. The tree returned line 15 consists of two pivots associated with their respective radii and respective subtrees.

Algorithm 1 BUILDMTREE(d, X)

1: $s \leftarrow$ RANDOMPOINTOF(X)
2: $l \leftarrow \arg\max_{x\in X}\{d(x,s)\}$
3: $r \leftarrow \arg\max_{x\in X}\{d(x,l)\}$
4: $L \leftarrow R \leftarrow \emptyset$
5: $u \leftarrow v \leftarrow 0$
6: **for all** $x \in X \setminus \{r,l\}$ **do**
7: **if** $d(l,x) < d(r,x)$ **then**
8: $L \leftarrow L \cup \{x\}$
9: $u \leftarrow \max\{u, d(l,x)\}$
10: **else**
11: $R \leftarrow R \cup \{x\}$
12: $v \leftarrow \max\{v, d(r,x)\}$
13: **end if**
14: **end for**
15: Return $(l, u,$ BUILDMTREE$(d, L), r, v,$ BUILDMTREE$(d, R))$

The algorithm 1 is mostly heuristic. There are no actual guaranties over the computational complexity, that can be $\mathcal{O}(n^2)$, or the tree depth, that can be $\mathcal{O}(n)$. Nevertheless, in practice the depth of tree is $\mathcal{O}(\ln(n))$ (implying a $\mathcal{O}(n\ln(n))$ computational complexity).

Performing a near neighbor query. The algorithm 2 provides the pseudo-code detail of of NNS algorithm. Note that for the sake of clarity, caching distance values is implicit in algorithm 2 (see discussion here above). Intuitively, the search explores first the nearest node (the notion of proximity being defined

as the distance between the query point and the node pivot). Before exploring any node, the triangular inequality is used to check whether this particular node can be pruned.

The arguments of the algorithm are the metric tree (refered as t), the distance function (refered as d), the query point (refered as q) and the number of neighbors (refered as k). The algorithm 2 relies on two heaps: the nodes B remaining to be explored and the closest points T encountered so far. We assume that the heap orders the elements in a increasing order, ie the element being associated with the smallest value being at the root of the heap. Those two heaps are initialized lines 1-3. The number r represents the current radius (the distance between the query point and the k^{th} closest point encountered so far). The lines 5-34 represent the main exploration loop (the exploration goes forth until there is no more node left). Lines 7-9 correspond to the triangular inequality pruning criterion. Lines 10-16 correspond to the case where a point closer that the former k^{th} closest point is encountered (such encounter can lead to a new value for the current radius r). Lines 17-22 corresponds to pseudo-recursive tree exploration. The notation $left(b)$ (respectively $right(b)$) refers to the left branch of the node b (respectively to the right branch of b). Note the node exploration order is defined by the proximity of the nodes to the query point.

The algorithm 2 is an *exact search algorithm*. We will give here a sketch of the proof. First, we notice that radius associated to any node in the algorithm 1 is the maximum distance between the node pivot and any of the point contained in the subtree defined by this node. Then the pruning test $d(b,q) - radius(b) > r$ performed at line 7 in the algorithm 2 simply checks, based on the triangular inequality assumption, whether is possible that the node b actually contains any point closer to q than the closest k^{th} point encountered so far (whose distance is expressed by r).

2.2 Vantage Point trees

The *vantage point tree* [18, 19, 20] is somehow similar to the metric tree and exploit the triangular inequality in a slightly different way.

Intuitively, the vantage point tree organizes the space in a hierarchical manner, each node corresponding to a particular sphere that split the dataspace in two (inside the sphere vs. outside the sphere). So far it seems that all proposed vantage point tree implementation are *static* data collection (build once and then no point can be either added or removed from the collection).

Proceeding like we did for the metric tree, we will now provide a more precise description of the two main algorithms related to vantage point trees: building a metric tree, and performing a near neighbor query.

Building a vantage point tree. The algorithm 3 provides the pseudo-code detail of the vantage point tree building. Intuitively, the process consists of choosing one pivot for each node, and to recursively split in two the set of points, the points having a distance to the pivot less than the median distance

Algorithm 2 SEARCHMTREE(t, d, q, k)

```
1:  B ← EMPTYHEAP()
2:  B.add(root(t), d(q, root(t)))
3:  T ← EMPTYHEAP()
4:  r ← +∞
5:  while B ≠ ∅ do
6:      b ← B.pop()
7:      if d(b, q) − radius(b) > r then
8:          continue
9:      end if
10:     if d(b, q) < r then
11:         T.add(b, −d(q, b))
12:         if T.count() > k then
13:             T.pop()
14:             r ← −T.peekValue()
15:         end if
16:     end if
17:     if left(b) ≠ NULL then
18:         B.add(left(b), d(q, left(b)))
19:     end if
20:     if right(b) ≠ NULL then
21:         B.add(right(b), d(q, right(b)))
22:     end if
23: end while
24: return T
```

being assigned to the inner sphere, the other points being assigned to the outer sphere.

The arguments of the algorithm are the distance d and the set of points X. The vantage point tree process requires only one pivot to build a node partition. The lines 1-2 are the *vantage point* heuristic, the point c is often called the *corner*. Then the median distance μ to the corner is computed line-3.

The recursive partition into the *inner* sphere and the *outer* of the dataset is actually performed lines 6-12. The comparison of distance point to corner and the median distance is used at line 6 as a criterion to distinguish between inner and outer spheres.

Algorithm 3 BUILDVPTREE(d, X)

1: $s \leftarrow$ RANDOMPOINTOF(X)
2: $c \leftarrow \arg\max_{x \in X}\{d(x, s)\}$
3: $\mu \leftarrow$ MEDIANOF$_{x \in X}(d(x, c))$
4: $L \leftarrow \emptyset$
5: $R \rightarrow \emptyset$
6: **for all** $x \in X \setminus \{c\}$ **do**
7: **if** $d(x, c) < \mu$ **then**
8: $L \leftarrow L \cup \{x\}$
9: **else**
10: $R \leftarrow R \cup \{x\}$
11: **end if**
12: **end for**
13: Return $(c, \mu,$ BUILDVPTREE$(d, L),$ BUILDVPTREE$(d, R))$

The algorithm 3 requires a $\mathcal{O}(n \ln(n))$ computational time and (n) memory. Like the metric tree building algorithm, the algorithm 3 is mostly heuristic.

Performing a search query. The algorithm 4 provides the pseudo-code detail of of NNS algorithm. Note that for the sake of clarity, caching distance values is implicit in algorithm 4 (see discussion here above). Intuitively, the process is similar to the metric tree search.

For the sake of simplicity, the algorithm 4 is presented in a recursive manner. The arguments of the algorithm 4 are b the node to be searched, d the distance function, q the query point, k the number of researched neighbors, T the heap of neighbors. Initially, the method SEARCHVPTREE should be called with b the root of the vantage point tree and T an empty heap. The heap T contains k nearest neighbors encountered so far ordered in a decreasing order of their respective distances to the query point (notice the similarity with T the heap in the algorithm 2). The lines 2-7 handle the node corner and possibly add this point in the heap T. The lines 8-18 define the exploration behavior of the algorithm. Depending on the query position, the inner sphere or the outer sphere is explored first. The tests at line 10 and line 15 reflect triangular inequality pruning.

Algorithm 4 SEARCHVPTREE(b, d, q, k, T)

```
1: δ ← d(q, b)
2: if δ < T.peekValue() then
3:    T.add(b, δ)
4:    if T.count() > k then
5:       T.pop()
6:    end if
7: end if
8: if δ < median(b) then
9:    SEARCHVPTREE(inner(b), d, q, k, T)
10:   if δ + T.peekValue() > median(b) then
11:      SEARCHVPTREE(outer(b), d, q, k, T)
12:   end if
13: else
14:   SEARCHVPTREE(outer(b), d, q, k, T)
15:   if δ − T.peekValue() < median(b) then
16:      SEARCHVPTREE(inner(b), d, q, k, T)
17:   end if
18: end if
19: return T
```

The algorithm 4 is an *exact search algorithm*. The proof is quite straightforward and very similar to the proof of the correctness of the algorithm 2.

3 Neighbor search in non-metric space

In the previous section, we have been focusing on the case of NNS within a *metric space*. The metric trees and vantage point trees relies mostly on exploration pruning based on triangular inequality constraints. This principle is the foundation of the algorithm 2 and the algorithm 4. This section focuses on the NNS in the case of *non-metric space*. Formally the three metric assumptions (relative to the distance function) are simply removed. Only remains E the dataspace and $s : E^2 \rightarrow \mathbb{R}^+$ the similarity criterion. The pair (E, s) is refered as a non-metric space.

Let us note that the three metric axioms (separation, symmetry and triangular inequality) are far from having an "equal weight" for the NNS purpose. Indeed, as clearly described in [5], the metric NNS algorithms can easily be adapted if the separation and/or the symmetry axioms are removed. Nevertheless the triangular inequality has a critical importance. Although, the triangular inequality can be slightly relaxed (the detail goes beyond the scope of this document, see also [5] for the detail), it is strongly required by the classical metric NNS algorithms.

Indeed the immediate consequence of the non-metric assumption is that there is no "hard" assumption that could be used to ensure any guaranteed

search pruning. Therefore the only guaranteed algorithm that performs queries in nonmetric space is the naive exhaustive search algorithm. To our knowledge, no general approach have been proposed so far for the non-metric case. Here we introduce first a framework whose purpose is to evaluate the performance of a non-metric NNS algorithm. Then we introduce several methods that can be used in the context of non-metric NNS. Experimental evaluations are proposed for those methods. [2]

3.1 Approximate queries in non-metric space

As discussed here above, the query methods in the non-metric situation are either naive (involving n similarity computations) or approximate. In this section, we will introduce approximation criterions for queries in non-metric spaces.

The dataspace E is associated to a similarity function s. Let $\widetilde{\mathcal{B}}(q,r)$ be the approximate result provided by a densitree to the range query $\mathcal{B}(q,r)$, idem $\widetilde{\mathcal{N}}(q,k)$ being the approximation of $\mathcal{N}(q,k)$. In order to be able to evaluate the approximation quality, we need to introduce some quality criterion.

Most commonly used approximation criterions in the NNS literature are *geometric*. Typically an algorithm based on geometric approximations will ensure that distance from the query point to the returned point is smaller than $1+\epsilon$ times the distance of the query point its nearest neighbor (see [3, 9]). Nevertheless, as pointed out in [4], such geometric approach may suffer from intrinsic instabilities as the dimension increase. Additionally, since the triangular inequality do not hold here, the value of similarity between two points has little interest in itself. Therefore, we will restrict the discussion in this paper to rank-based approximation criterion.

The most simple criterion is simply the ratio of correct items in the approximate result. We call this criterion the **validity**[3]. Formally the validity could be defined as

$$v(\widetilde{\mathcal{N}}(q,k)) = \frac{|\widetilde{\mathcal{N}}(q,k) \bigcap \mathcal{N}(q,k)|}{|\mathcal{N}(q,k)|} = \frac{1}{k} \left| \widetilde{\mathcal{N}}(q,k) \bigcap \mathcal{N}(q,k) \right|$$

The validity value is in the interval $[0,1]$, *zero* being a poor answer, *one* being the exact answer. The validity of the range queries $v(\widetilde{\mathcal{B}}(q,r))$ could be similarly defined. Although being very simple, the validity is not entirely satisfying as a quality criterion. Indeed the validity is not sensitive to the quality of the incorrect items. Intuitively, returning very close but incorrect neighbors will not lead to a higher validity than returning very distant neighbors. In order to overcome this drawback of the validity, let us introduce a rank based criterion.

[2] Problem: it seems that the non-metric case has been studied once in Zhang 2002, see [21]. But the paper is poorly written, quotes 30 years old papers (all modern references are missing), is purely empirical, and seem to have a very narrow application spectrum (1 specific metric function). Additionally the proposed algorithm is not even applied to a non-metric case since, the function is almost a metric (only the separation assumption is not verified and it is easy to adapt any classical algorithms to handle that, see [5]).

[3] NOTE: never seen this criterion anywhere in the near neighbor literature.

Let $r_q(x)$ be the **rank** of the item $x \in X$ according to the query q, defined as $r_q(x) = |\{y \in X, s(q,y) < s(q,x)\}|$. Then the **shift**[4] could be defined[5] as

$$f(\widetilde{\mathcal{N}}(q,k)) = \frac{\sum_{x \in \widetilde{\mathcal{N}}(q,k)} r_q(x)}{\sum_{x \in \mathcal{N}(q,k)} r_q(x)} = \frac{2}{k(k+1)} \sum_{x \in \widetilde{\mathcal{N}}(q,k)} r_q(x)$$

The shift value is in the interval $[1, \infty)$, *one* being the exact answer. Note that the shift if sensitive to the overall quality of the answer.

3.2 Graphical exploration models

In this section, we introduce a general framework for describing non-metric NNS algorithms. The purpose of this framework is to outline the key aspects of non-metric NNS.

Intuitively, any non-metric NNS algorithm $\mathcal{A}$ lies on a particular data structure that contains X the dataset. When performing a query with $\mathcal{A}$, the dataset X can be divided in two parts: the set of explored points (the similarity between any of those points and the query has been computed) and the set of unexplored points. Based on the information relative to the set of explored points, the algorithm $\mathcal{A}$ decides which point will be explored next. The data structure $\mathcal{G}$ usually restraints the set of candidate points. In other word, all unexplored points are not necessary candidates to be explored next. Notice that, here, the term "exploration" refers to process of choosing the next point to be compared to the query point based on the information previously acquired.

More formally, let us introduce an oriented graph $\mathcal{G} = (V, \mathcal{E})$, a mapping $\preceq_{\mathcal{K}}$ and a root vertex $r \in V$ as a general representation of $\mathcal{A}$. The set V (respectively $\mathcal{E} \subset V \times V$) represents the vertices (respectively the edges) of the graph $\mathcal{G}$. We have adopted the notation V for the vertices to be more consistent with the literature, but please note that here $V = X$. For any subset $C \subset V$, we note $\mathcal{C}$ as the "neighbors" of C, formally defined by

$$\mathcal{N}(C) = \{y \in V | (x,y) \in \mathcal{E} \text{ and } x \in C\}$$

Let $\mathcal{K}$ represents a set of pairs $\{x, s\}$ where $x \in X$ is a point and $s \in \mathbb{R}$ is a similarity value. The set $\mathcal{K}$ is an abstraction of the set of explored points. The mapping $\preceq_{\mathcal{K}}$: $\mathcal{K} \mapsto \preceq$ associates a relation order $\preceq$ to a set of explored points $\mathcal{K}$. The vertex r represents the "root" of the algorithm $\mathcal{A}$, i.e. the point that will be compared first to the query. Using those notations, left us now provide the detail of the general representation of the algorithm $\mathcal{A}$ with the following pseudo-code.

The algorithm 5 is quite simple. It begins with the exploration of the root at line 1. The main loop (lines 2-6) consists of finding the set C of candidate

[4]NOTE: never seen this criterion anywhere in the near neighbor literature. May have already a usual (but different?) name somewhere else.

[5]For the sake of simplicity, the equality is based on the assumption that the dataset could be ordered, e.g $d(q,x_1) < d(q,x_2) < \cdots < d(q,x_n)$

Algorithm 5 NonMetricQuery(q)

1: $\mathcal{K} \leftarrow \{r, s(q,r)\}$
2: **while** SomeTerminationConditions **do**
3: $\quad C \leftarrow \mathcal{N}(\mathcal{K}) \setminus \mathcal{K}$
4: $\quad v \leftarrow \min_{\preceq_\mathcal{K}} C$
5: $\quad \mathcal{K} \leftarrow \mathcal{K} \cup \{v, s(q,v)\}$
6: **end while**
7: Return $\mathcal{K}$

vertices at line 3, of choosing within C the vertex v to be explored next at line 4 and finally to explore v at line 5. The algorithm returns directly $\mathcal{K}$ the set of explored vertices. Note that the constraints induced by the model underlying the algorithm 5 are very reasonable ones. This model implies that no similarities are computed except the ones between the query point and the dataset points (forbidding the computation of the similarity between two points of the dataset for ex.). This model also implies that each allowed similarity computed once at most.

The purpose of the generic non-metric NNSalgorithm representation presented here above is not to lead to practical efficient implementations. The purpose of this framework is to help us to outline the following aspects of non-metric NNS.

- Heuristic stopping criterion.
- Accurate exploration is the key of non-metric NNS.
- No difference between range query and neighbor query.

As discussed here above, the only exacts non-metric NNS algorithm is actually the exhaustive search through the whole dataset. Therefore, an heuristic stopping criterion (as expressed line 2 of the algorithm 5) is required. Since any NNS algorithm $\mathcal{A}$ requires to keep $\mathcal{K}$ the set of already encountered vertices (or at least a subset of $\mathcal{K}$ associated to the closest points), an algorithm $\mathcal{A}$ can be interrupted at any time, the set $\mathcal{K}$ being returned. Intuitively the more exploration is allowed, the more accurate will be the results extracted from $\mathcal{K}$.

This initial remark leads us to the second aspect: an accurate exploration is the key of non-metric NNS. Indeed, since any algorithm $\mathcal{A}$ will be stopped before having explored the whole dataset, it is critical, for the accuracy of the results, to explore as fast as possible the nearest points of the query. Therefore the main issue in non-metric NNSis not to design elaborate pruning criterions (as opposed to the metric NNS case), but is to design a mapping $\preceq_\mathcal{K}$ that will lead the exploration as fast as possible to the nearest neighbors of the query.

Finally, the non-metric NNS is conceptually more simple than the metric NNS. Indeed, the lack of "hard constraints" like triangular inequality implies that there is no practical difference between a range query and a near-neighbor query. It is known for the metric case, that there are tight relationships between

the two query types. General methods are available to convert efficiently a range query into near-neighbor query and vice-versa (see [5] for the detail). But in the non-metric situation, the relationships are even stronger. The process described in algorithm 5 remains the same independently of the query type; only the query point matters. In both cases, the result will be a subset[6] of the returned set $\mathcal{K}$. In the following, we will restrict our focus to the near-neighbor queries.

3.3 Query accuracy profile

This section introduce the notion of *query accuracy profile* that is an extension of the "accuracy" criterion introduced here above. First, let us motivate the need for the introduction of an additional criterion. The accuracy indicates the ratio of correct items returned by a certain algorithm $\mathcal{A}$ for a given query. The algorithm 5 provides a new insight to the notion of accuracy. Indeed it become clear that the accuracy value will be bound to the termination condition (see line 2) of the algorithm $\mathcal{A}$. Nevertheless, as discussed previously, the termination condition is a simple tradeoff, whereas the exploration is the critical issue of non-metric NNS. Those elements call for *a criterion that could be used to evaluate the quality of exploration rather than the termination condition.*

Intuitively the query accuracy profile is a mapping $p : \#call \mapsto accuracy$. The accuracy profile $p(k)$ is equal to the accuracy of the best subset of $\mathcal{K}$ (taking the notations of the algorithm 5) after the k^{th} call the similarity function (the similarity function is called iteratively at line 5 in algorithm 5) Formally, if $\mathcal{N}$ is the set result from an exact computation of the query and $\mathcal{K}$ the set of encountered points after the k^{th} call to the similarity function, then

$$p(k) = \frac{|\mathcal{K} \cap \mathcal{N}|}{|\mathcal{N}|}$$

The query accuracy profiles are the core of the empirical results present in section 5. Note that a *query shift profile* can be defined in a similar fashion.

We will now discuss how query accuracy profiles can be computed in practice. The main issue is the fact that the profile computation actually requires the exact query result. Therefore, the most simple solution consists of a two-passes method: first compute the query result using the naive NNS method (exhaustive exploration); second, perform the exploration and record at each similarity function call the reached accuracy. When correctly implemented, this method is $\mathcal{O}(n)$ which is the best asymptotical bound that can be achieved here (recall the fact that the *exact* query result is required for the profile computation). Nevertheless, this method can be easily be improved into a single-pass method. The detail is given in the algorithm 6 which is a slightly modified version of the algorithm 5.

Note that the termination condition of the algorithm 6 is equivalent to "until there are no more unexplored vertices". Additionally, a list $\mathcal{L}$ gathers the tuples

[6] In practice, a heap will probably be chosen to represent $\mathcal{K}$

Algorithm 6 ProfileQuery(q)

1: $\mathcal{K} \leftarrow \{r, s(q,r)\}$
2: $i \leftarrow 0$
3: $\mathcal{L} \leftarrow \{r, s(q,r), i\}$
4: **while** $\mathcal{N}(\mathcal{K}) \setminus \mathcal{K} \neq \emptyset$ **do**
5: $\quad C \leftarrow \mathcal{N}(\mathcal{K}) \setminus \mathcal{K}$
6: $\quad v \leftarrow \min_{\preceq_{\mathcal{K}}} C$
7: $\quad \mathcal{K} \leftarrow \mathcal{K} \cup \{v, s(q,v)\}$
8: $\quad i \leftarrow i + 1$
9: $\quad \mathcal{L} \leftarrow \{v, s(q,v), i\}$
10: **end while**
11: Return ExtractProfile($\mathcal{L}$)

$(v, s(q,v), i)$ added at each iteration (where i counts the number of previous calls to the similarity function). The profile is extracted as post-query process by calling ExtractProfile whose detail is not given here because being quite straightforward.

3.4 Exploration stopping criterion

The algorithm 5 refers to an abstract SomeTerminationConditions criterion to stop the exploration and returns $\mathcal{K}$. In this section, we provide a simple criterion and discuss how such a criterion could be used in practice.

Intuitively, for a given exploration method, there are two opposing factor for the client of the NNS algorithm:

- The computational cost of the query.
- The accuracy of the tradeoff.

Intuitively, the most flexible solution is simply to provide the tradeoff curve

$$u : accuracy \mapsto \#call$$

to the client and let him chose the desired level of accuracy. Actually, we have already introduced, although from a different perspective, such a tradeoff curve in the previous section as the concept of "query profile".

A practical way of constructing the tradeoff curse u is to take a random sample of points from the dataset, to run the algorithm 6 with those points as arguments, and to compute the average query profile. From such average query profile, the reverse function (providing the number of similarity calls based on a chosen accuracy) can easily be computed.

4 Densitrees

The previous section was providing general considerations on the non-metric NNS algorithms. This section introduces a class of particular data structures

that we call *densitrees*.

4.1 Definition of densitrees

Intuitively, densitrees are a class of decorated trees that hold the points of the dataset in a way similar to the metric tree or vantage point tree. The critical difference lies in the nature of tree decoration; instead of having one or several real values reflecting some bounds on the triangular inequality attached to every tree node, each densitree node is associated to a particular classifier called here a *density estimator*. We speak of the densitrees (plural) rather than densitree (singular) because in practice many variations based on the chosen density estimators types are possible within the densitrees framework.

A density estimator provides estimates of the number of points contained within a ball (specified by its center and radius). When the query is performed, the densitree exploration, which starts at the root, follows greedily the paths of greatest densities. The insights offered by the algorithm 5 motivates the idea of exploring the nearest points as fast as possible because it will lead to a better accuracy for any given query termination condition.

Formally, nodes and leaves in the densitree will be treated indifferently. A densitree node is a tuple defined by $\alpha = (x, \widehat{\phi}, \alpha_L, \alpha_R)$ where $x \in X$ is point, $\widehat{\phi}$ is a density estimator, α_L and α_R are respectively left and right nodes (that may be null). By convention, the root node of the densitree is refered as α_0.

A density estimator $\widehat{\phi}$ is a mapping $\widehat{\phi} : E \times \mathbb{R} \rightarrow [0; 1]$. In order to clarify the notion of density estimator, let us first introduce the notion of density function. The (exact) density function for a dataset X is defined by

$$\phi_X(q, r) = \frac{1}{|X|} |X \bigcap \mathcal{B}(q, r)|$$

The function ϕ_X associates the ratio of the dataset contained in the specified ball. Since E always represents the dataset, in the following, the density function will be refered as ϕ for the sake of simplicity. From an exploration perspective, the density function ϕ is obviously the optimal function to guide the exploration. Nevertheless the use of the density function does not lead to any computational improvement over the naive exhaustive search, therefore density estimators (that can be viewed as efficient approximation of the density function) are be used instead.

4.2 Queries in densitrees

For the sake of clarity, we will begin our densitree discussion by introducing the densitree query algorithms. Based on the generic model introduced with the algorithm 5, the algorithm 7, detailed below, provides some insight on the behavior of the densitrees.

Let us give some details about the conventions used in the pseudo-code of the algorithm 7. The notation ϕ_α refers to the density estimator associated

Algorithm 7 DensitreeQuery(q)

1: $\mathcal{K} \leftarrow$ EmptyStack()
2: $r \leftarrow$ CurrentRange($\mathcal{K}$)
3: $\mathcal{K}$.Push($\alpha_0, \phi_{\alpha_0}(q, r)$)
4: **while** SomeTerminationConditions **do**
5: $\alpha \leftarrow \mathcal{K}$.Pop()
6: $r \leftarrow$ CurrentRange($\mathcal{K}$)
7: **if** $left(\alpha) \neq null$ **then** $\mathcal{K}$.Push($left(\alpha), \phi_{left(\alpha)}(q, r)$)
8: **if** $right(\alpha) \neq null$ **then** $\mathcal{K}$.Push($right(\alpha), \phi_{right(\alpha)}(q, r)$)
9: **end while**
10: Return $\mathcal{K}$

to the node α. The left branch (respectively the right branch) of the node α is refered by $left(\alpha)$ and $right(\alpha)$. Here, the set of explored points $\mathcal{K}$ has an explicit structure of heap, the points being ordered by decreasing estimated densities. The densities are estimated base on the ball radius provided by the method CurrentRange that takes $\mathcal{K}$ as argument. Like we did for the algorithm 5, the termination criterion at line 4 is left unspecified.

There are several issues related to the function CurrentRange that we will now discuss more extensively. First, the case where the query is a "range query" is quite simple, in such case CurrentRange simply returns the range specified by the query. The difficulties arise in the case of the near-neighbor query. The usual approach, as taken in the algorithm 2 or algorithm 4 would suggest to return an infinite radius until $|\mathcal{K}| = k$ (k being the number of neighbor specified by the query), and then to return the radius associated the k^{th} nearest neighbor in $\mathcal{K}$. Unfortunately, this approach is not efficient in practical. The reason lies in the fact that $\phi_\alpha(x, \infty)$ is always equal to 1. Thus an infinite radius leads to a random exploration at least for the first elements. Heuristics can be used to overcome this difficulty. Intuitively the idea consists of estimating initially the final range of the query.

An accurate estimation of the final range is, as we have seen, a critical issue to ensure an efficient exploration for a near-neighbor query. A simple solution consists of performing a sample of random queries (just after building the densitree, see next section) for a given number of neighbor and of storing the average achieved final range. This is the empirical estimator that is used in practice in 5. This estimator has the advantage of being computationally very efficient (it does not require any additional call to the similarity function at query-time). Nevertheless, the main draw of this estimator lies in its high (statistical) bias. The use of better estimators, with a more accurate bias vs variance tradeoff handling, may lead to significant over the results proposed in section 5.

4.3 Building the densitrees

As we have seen how to perform a query in a densitree, this section presents how to build a densitree. Building a densitree relies on two elements: (i) a partitioning method that split a set of points in two, (ii) a density estimator learning method (this point is left to the section 4.4). Intuitively, this process is almost identical to the metric tree or vantage point tree building, except that density estimators should be additionally learned in order to decorate the tree. The detail is given in the algorithm 8.

Algorithm 8 BuildDensitree(X)

1: $\phi \leftarrow$ LearnDensity(X)
2: $(X_L, X_R) \leftarrow$ Split(X)
3: $\alpha_L \leftarrow$ BuildDensitree(X_L)
4: $\alpha_R \leftarrow$ BuildDensitree(X_R)
5: **Return** $(\phi, \alpha_L, \alpha_R)$

The algorithm 8 has a simple recursive structure (for the sake of simplicity, the algorithm 8 does not provide the detail of the trivial stopping criterions). The function Split, as the name suggests, splits the dataset into a left and a right part (respectively X_L and X_R following the notations of the pseudo-code). The discussion of the function LearnDensity is left to the next section.

Many heuristics have been proposed in metric NNS literature to find a good pivot (see [5] for a review). A interesting point to note is that the non-metric assumption has little importance here. Indeed most of the tree building method of the metric literature (see algorithm 1 or algorithm 3) does not relies on the metric assumptions. In the section 5, the metric tree method (ie. the algorithm 1) has been used to build the densitree. This adapted method has the advantage of being adapted to the densitree purposes and computationally efficient. The computational cost of this method is $\mathcal{O}(n \ln(n))$ calls the similarity function if we assume a linear learning cost for the density estimators (linearity based on the number of items provided to the function LearnDensity, this assumption will be extensively discussed in the next section).

4.4 Density estimators

As we have seen the purpose of the density estimator is to efficiently approximate the density function. This implies that two main opposing constraints apply on density estimators: first the density estimator must be as accurate as possible, second the density estimator must have a computational cost as low as possible.

Based on the insights provided by algorithm 5, it can be noticed that the absolute value returned by the density estimators no importance only their respective values matters. Therefore, a very simple density estimator $\widehat{\phi} : E \times \mathbb{R} \rightarrow \mathbb{R}$, yet very efficient in practice (see section 5 for more details) is defined by $\widehat{\phi}(x, r) = s(x, q)$ where q is the query point. Intuitively, this density estimator leads to a greedy exploration, exploring first the nearest point.

We will now introduce a more complicated density estimator, that we call patriot(path To root density estimator), whose purpose is to take advantage of the previously computed similarities. Let us begin by an intuitive description of the patriot. The densitree is, as we have seen, a tree decorated by density estimators. Each density estimator is associated to a node. When the density estimator is called, it is natural to compute the similarity value between the query point and the node point and to exploit this information (it is even possible to rely solely on this information, as we have seen here above). Nevertheless, it would be interesting to exploit also the information provided by the previously computed similarities. In particular, all the nodes along the path that leads from the root to the current node have already been compared to the query point. The proposed density estimator includes a sample of points from the subtree that have been "projected" on the dimensions defined by the nodes constituting the path that leads to the root. The density value is estimated within this sample based on an "induced" similarity (a norm L_1 in $\mathbb{R}^k$); the advantage of this induced similarity being that it does not require any additional call the similarity function.

We will now provide a more formal definition of the patriot density estimator $\widehat{\phi}$. Let α be the node of interest. Let $X_\alpha \subset X$ be the subset of points associated to the subtree defined by the node α. Let us consider the list of points $B = (x_1, \ldots, x_d) \in X^d$ that constitutes the path to the root (x_1 would be the root point and x_d the point associated to the node α). Any point $x \in X$, can be "projected" on B with $\bar{x} = (s(x, x_1), \ldots, s(s, x_d))$. Let $\bar{X}_\alpha$ be the point-wise projection of X_α over B. Let $\| \, . \, \|$ be a norm over $\mathbb{R}^d$ (more details will be given later about this norm). Let $\kappa : \mathbb{R} \to \mathbb{R}$ be a real function; κ is the "scaling function" (more details will be given later about this function).

We have now all the elements required to define $\widehat{\phi}$. For a point $q \in E$ and a range $r \in \mathbb{R}$, we have

$$\widehat{\phi}(q, r) = \frac{1}{|X_\alpha|} \left| \mathcal{B}_{\|.\|, X_\alpha}(\bar{q}, \kappa(r)) \right| \tag{1}$$

Let us give a word of explanation about this definition. The notation $\mathcal{B}_{\|.\|, X_\alpha}$ refers the the ball defined over the set X_α based of the norm $\| \, . \, \|$ (used as the similarity function). Intuitively the query point q is simply projected as $\bar{q}$, then the density is evaluated within the projected space which required to scale the range r into $\kappa(r)$ as well. The result is considered as being an estimation of $\widehat{\phi}(q, r)$ itself.

The patriot method raises several practical issues (listed here below) that will be discussed in the following.

- How to choose $\| \, . \, \|$?
- How to choose κ?
- Practical implementation of $\widehat{\phi}$.
- The computational cost to build $\widehat{\phi}$.

- The memory cost to store the $\widehat{\phi}$.
- The computational cost to call $\widehat{\phi}$ during exploration.
- The empirical accuracy of $\widehat{\phi}$ vs other estimators.

The definition of the norm $\| \, . \, \|$ is critical for the accuracy of of patriot estimator $\widehat{\phi}$. A natural idea would be use $\| \, . \, \|_1$ or $\| \, . \, \|_2$ as very classical metrics over $\mathbb{R}^d$. Those metrics have been tried (although the experiments are not listed in section 5) and lead unfortunately to very poor accuracy for $\widehat{\phi}$. Let us provide the insight of this poor behavior. To simplify largely the problem, we can say that the exploration (if not stopped by any termination criterion) has three phases: (i) descending down the densitree to query close neighborhood (ii) visiting the query close neighborhood (iii) visiting the remaining remote neighborhoods. In practice, the average similarity values between the query point and the points explored the phase (i) are much greater than the similarity values between the query point and the points explored during the phase (ii). As we have designed patriot, this phenomenon leads to a considerable overweighting of the points at the top of the densitree (points close to the densitree root).

This issue suggests that more complicated $\| \, . \, \|$ should be considered. A natural solution to overcome the overweighting mentioned here above is to introduce linear coefficients such as

$$\| \, \bar{x} \, \| = \sum_{i=1}^{d} \lambda_i |\bar{x}_i|$$

This option raises an other issue that is the actual choice of the coefficients λ_i. In section 5, we have opted to normalize each dimension dividing by the mean similarity. Thus we have $\mathrm{E}[\lambda_i |\bar{x}_i|] = 1$ for all i. This choice is motivated by the insight provided here above and suppress the overweighting problem. In practice, this option leads to great improvements over the use of uniform norms like $\| \, . \, \|_1$. Nevertheless, this option is certainly not definitive, and we suspect that vast amelioration can be obtained by a better "tuning" of this norm. In particular, empirical observations lead us to suggest that the closer the pivot point is from the query point, the more accurate is the prediction associated to the resulting dimension. Better approaches would certainly take into account such phenomenon. A more general would approach would even suggest that the $\{\lambda_i\}_i$ coefficients should be directly optimized (learned) in order to fit the proposed purposes. Such approach goes beyond the scope of the present document.

The values of the scaling function κ is tightly bound to actual metric $\| \, . \, \|$. Nevertheless we propose a method to estimate the scaling function κ is a fashion that does not depend on the chosen metric $\| \, . \, \|$. The method consists simply in drawing random pair of points (x, y) within X_α and computing the pair of values $(s(x, y), \| \, \bar{x} - \bar{y} \, \|)$. Let $(u_i, v_i)_i$ be a list of such sample pairs ordered

against the similarity values (ie. $i <= j \Rightarrow u_i <= u_j$). Considering the list $(u_i, v_i)_i$, a simple definition for κ would be $\kappa(u) = v_{\arg\max_i \{u_i < u\}}$. Such choice would be motivated by the intuition that choosing the norm value associated to a similar similarity value would be an accurate option. Unfortunately, empirical observations suggest that such an option leads to a very poor overall accuracy for the patriot estimator. Indeed, the use of a single pair does not provide an accurate estimate. The empirical solution to overcome this issue that have been used in the section 5 is to smooth the previously proposed κ function by clustering the list $(u_i, v_i)_i$ (gathering the values based on the u_i similarity, and taking the mean of the u_i and v_i values for each cluster). In the section 5, the number of random pairs was fixed to $|X_\alpha|$ (requiring a linear additional number of metric calls), then the resulting list is clustered into $\sqrt{|X_\alpha|}$ (in order to assure an unbiased yet convergent estimator for κ).

A efficient implementation for $\widehat{\phi}$ is required to ensure low computational and memory cost for the densitrees. The patriot method, described here above, induces an additional NNS problem. Although our overall objective is to perform a non-metric NNS, it can be noticed that the task performed by a patriot, is actually a *metric* NNS. Therefore, most of the solutions proposed in the NNS literature can be used here. In section 5, we have opted for the metric tree algorithm, although other options (possibly approximate) may have been used instead.

The accuracy of the patriot estimator determines the overall exploration quality of the densitree. The patriot estimator aims to a better accuracy than the greedy estimator previously mentioned (explore first the nearest node). Intuitively, the patriot estimator exploits more information than the greedy estimator (several similarity values are used to "score" a node vs a single value). The drawback of this approach is that the patriot estimator is less "robust" (statistical meaning) that the greedy estimator. Empirical observations (yet not included in the section 5) indicates that the patriot estimator is much less accurate than the greedy estimator when the number of points contained in the node subtree is low. Therefore the method used in the section 5 is a mixed method where the comparison operator $\preceq_{\mathcal{K}}$ (using the notation of algorithm 5) compare the densities returned by the patriot estimators if both subtree sizes are greater than an given threshold; if not, the comparison is based on the respective node similarity values to the query point. The threshold used in the experiments in section 5 is $\sqrt{|n|}$ where $n = |X|$ is the number of points in the dataset. This threshold is mainly arbitrary (although it seems to perform reasonably well, see the results of section 5) but it is motivated by the fact that $\sqrt{n} \to \infty$ when $n \to \infty$ thus ensuring the density estimator variance will tend to zero.

5 Experiments

This section provides experimental results based on the datasets already used in [13] for a benchmark of near neighbor search algorithms. All those datasets are available at [12]

- **aerial**: Texture feature data contain 275.000 feature vectors of 60 dimensions representing texture information of aerial photograph [14, 15].
- **corel_hist**: 20.000 histograms with 44 non-zero dimensions of color images taken from the COREL STOCK PHOTO Library (s) [13, 10].
- **corel_uci**: 68.000 histograms with 64 dimensions of color images from the COREL library [13]. This dataset differs significantly from the `Corel_hist` dataset available at the UCI repository [1].
- **disk_trace**: 40.000 content traces of disk-write operations, each being a 1kb block. The traces are generated from a desktop computer running SuSe Linux during daily operation [13].
- **sprot40**: 101601 sequences extracted from the Swiss-Prot database version 40 [2].

The objective of the experiments presented in the following sections is to evaluate the quality of exploration of the various algorithm that have been introduced previously. This objective implicitly states that the main computational cost of a search algorithm is the number of calls to the metric function. It is not entirely true, I/O costs can also be considered [6], but it is commonly accepted that the number of call to the metric is representative of the overall computational cost independently of the specific implementation [5].

We will begin by providing and discussing empirical results about the more classical metric situation (see section 5.1); then we will review the non-metric situation (see the section 5.2).

5.1 Results in metric space

The results presented in this section are gathered in the figures 1 to 15. Those figures are query accuracy profiles (introduced in section 3.3). Although having been introduced under non-metric assumption, the query accuracy profile discussion is straightforward to transpose in the metric case; the only subtlety (in respect to our previous discussion) being that the pruning which occurs in the metric situation restraints the required amount of exploration.

Our experiments involve the two metric NNS algorithms that have been previously described:

- **M-Tree**: the metric tree structure [17, 6]. An efficient implementation can be found at [7].

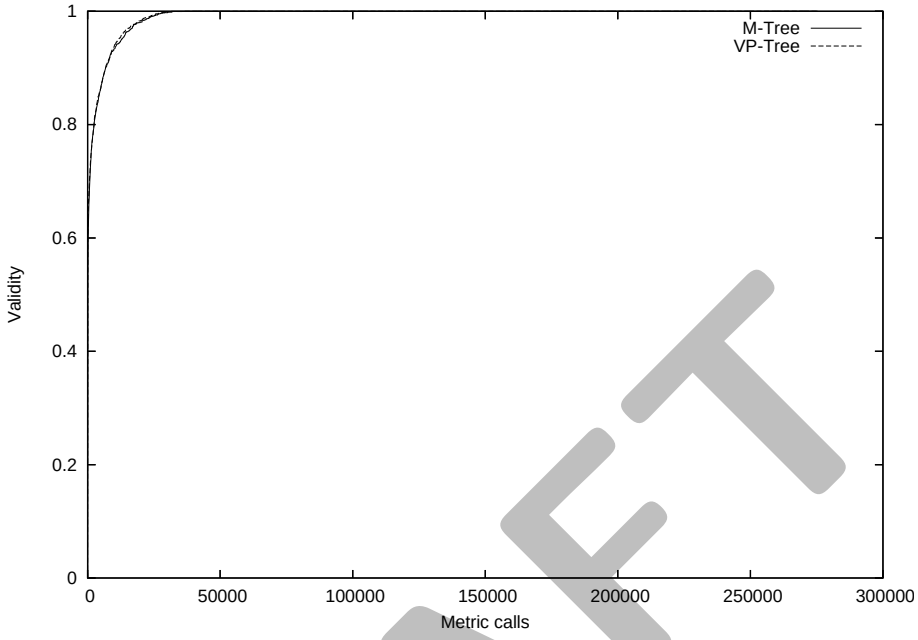

Figure 1: aerial, k = 2

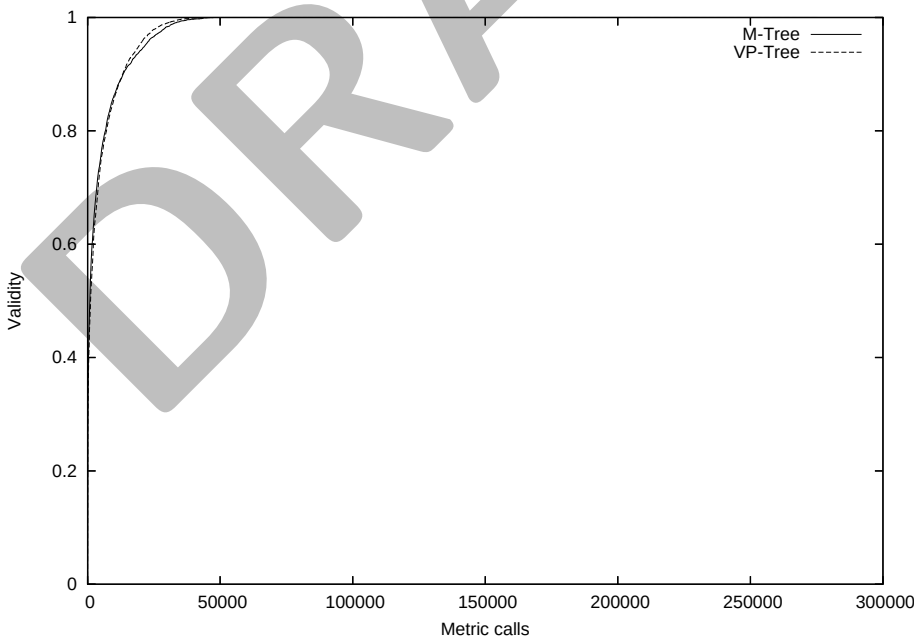

Figure 2: aerial, k = 8

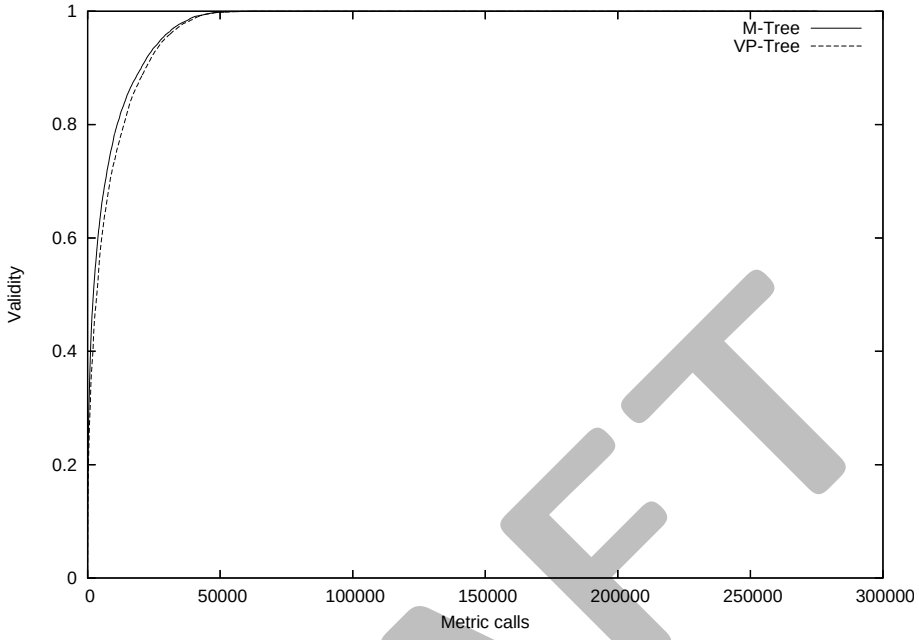

Figure 3: aerial, k = 64

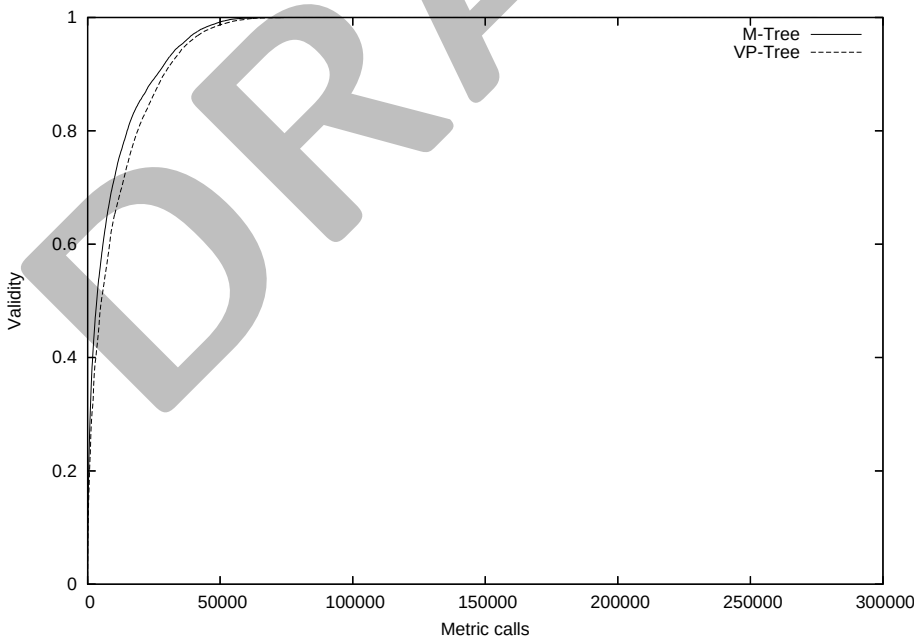

Figure 4: aerial, k = 256

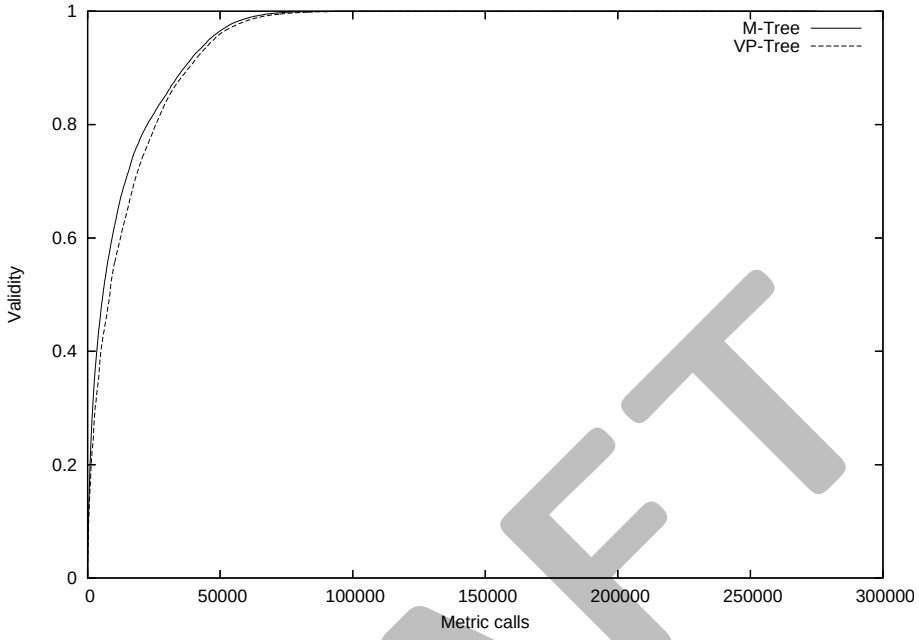

Figure 5: aerial, k = 1024

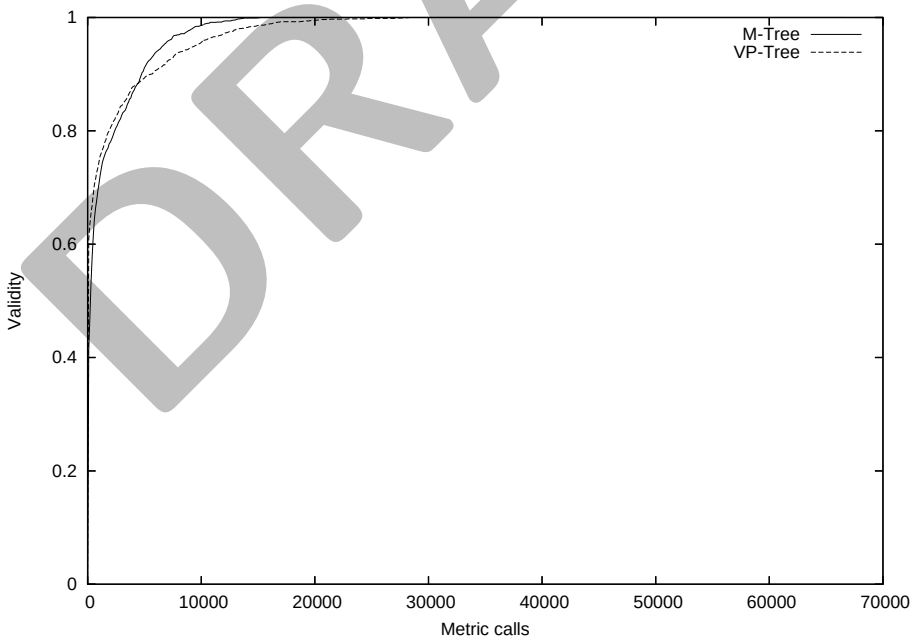

Figure 6: corelUci, k = 2

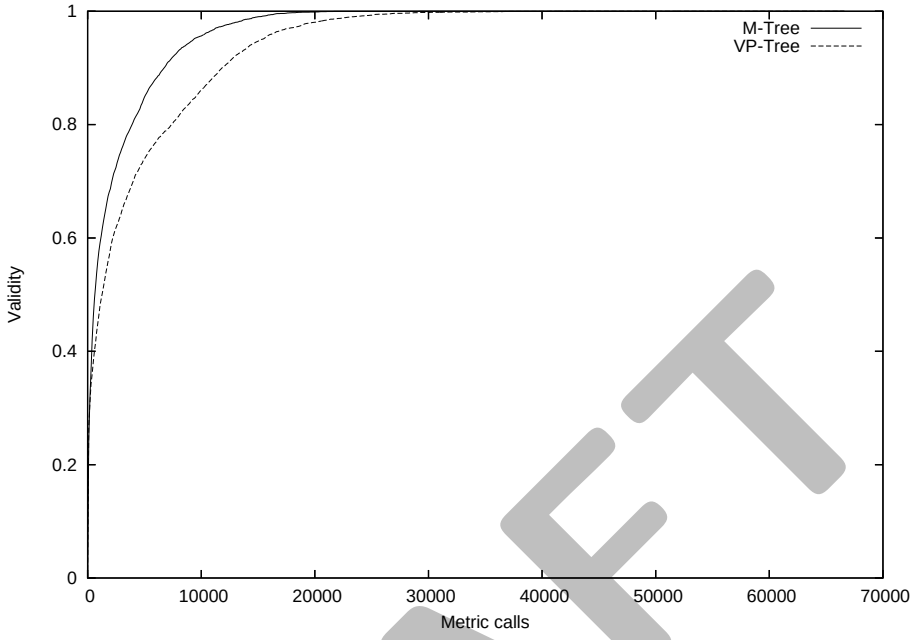

Figure 7: corelUci, k = 8

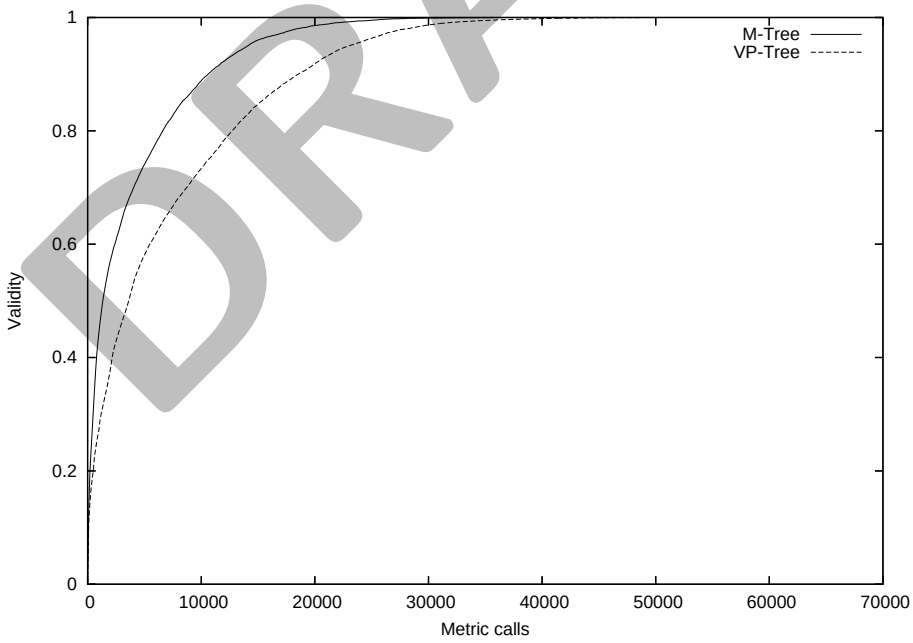

Figure 8: corelUci, k = 64

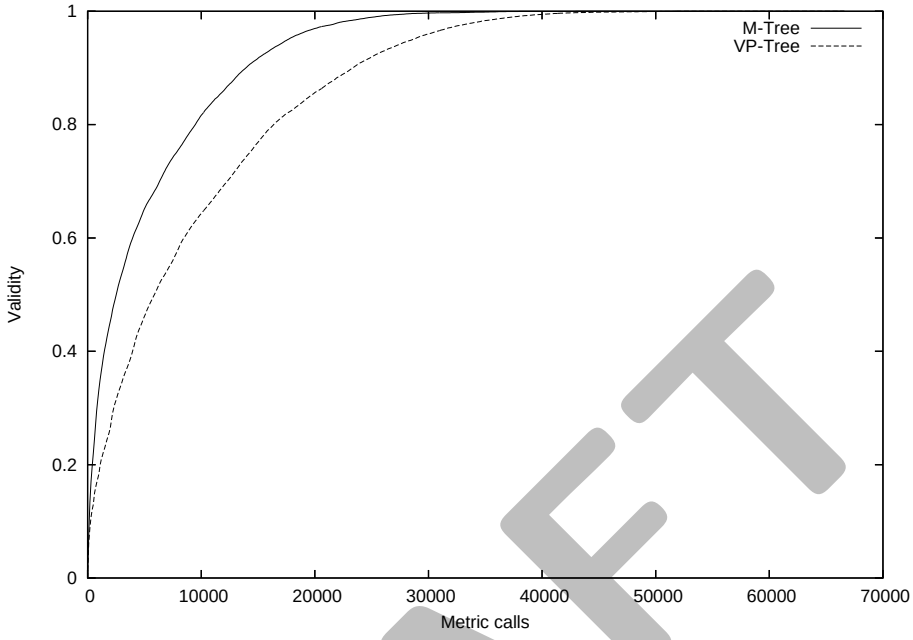

Figure 9: corelUci, k = 256

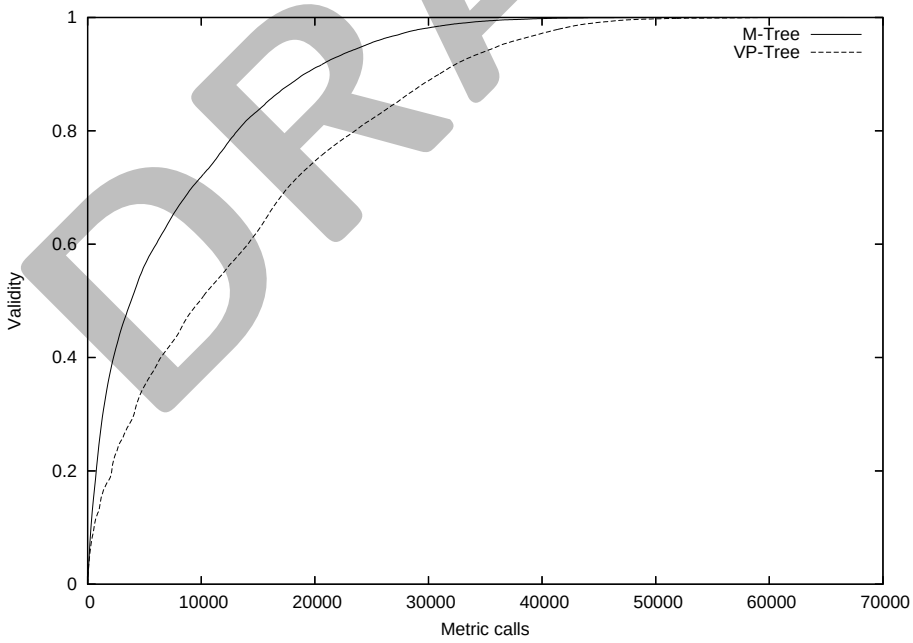

Figure 10: corelUci, k = 1024

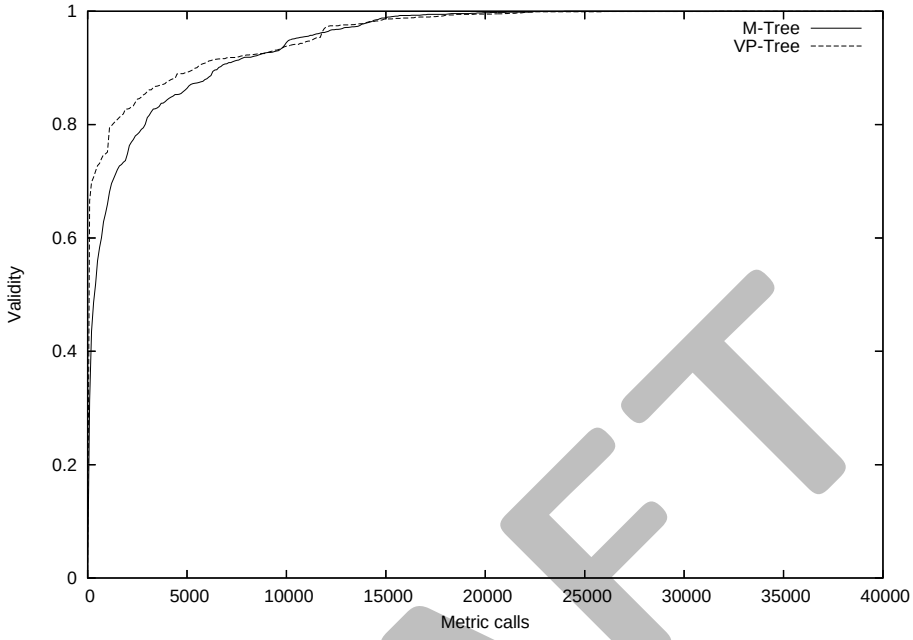

Figure 11: disk, k = 2

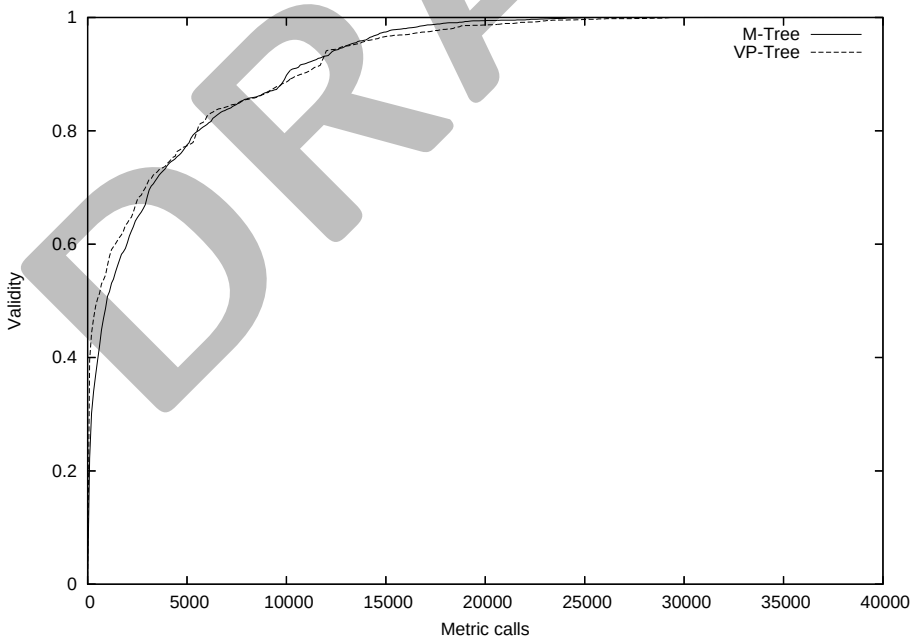

Figure 12: disk, k = 8

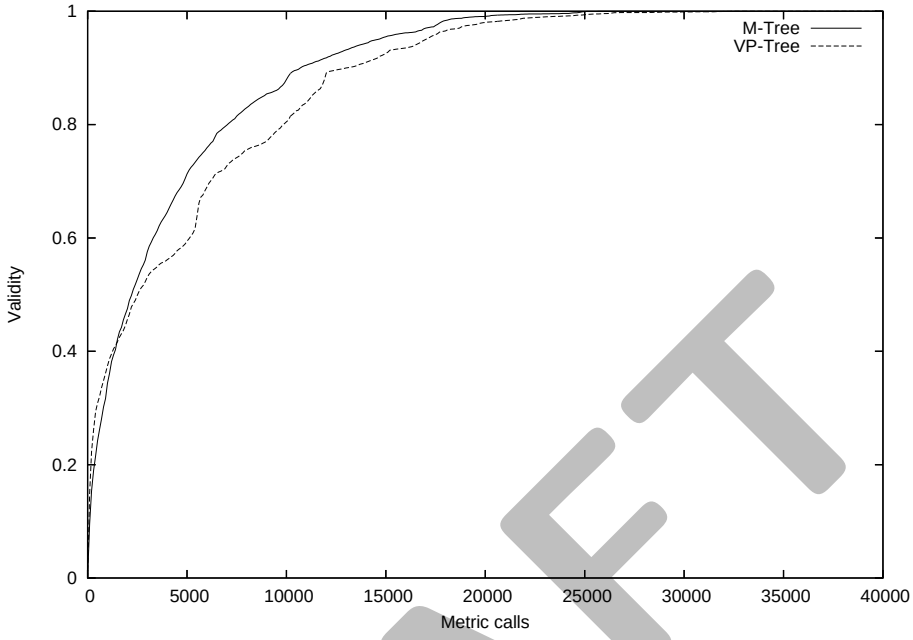

Figure 13: disk, k = 64

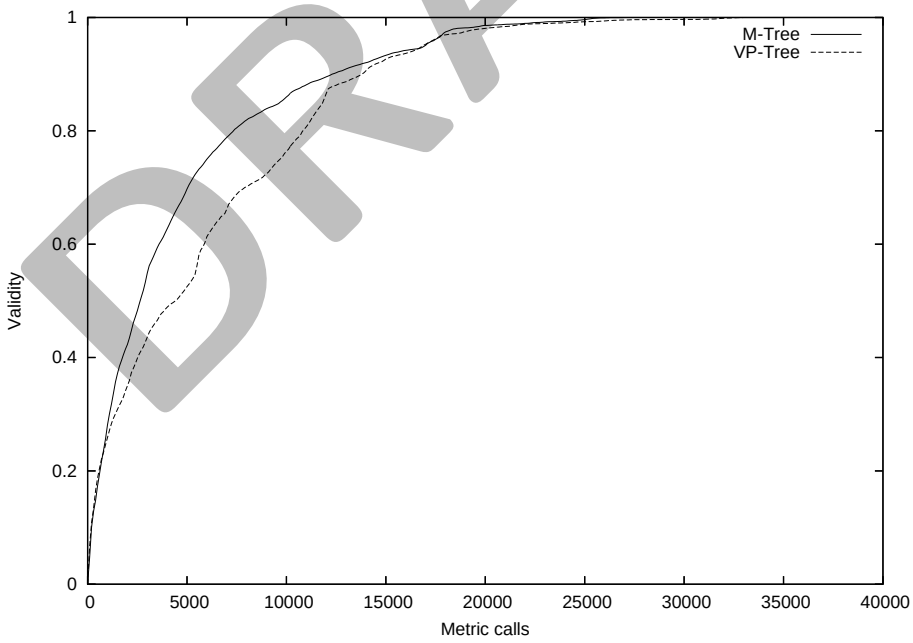

Figure 14: disk, k = 256

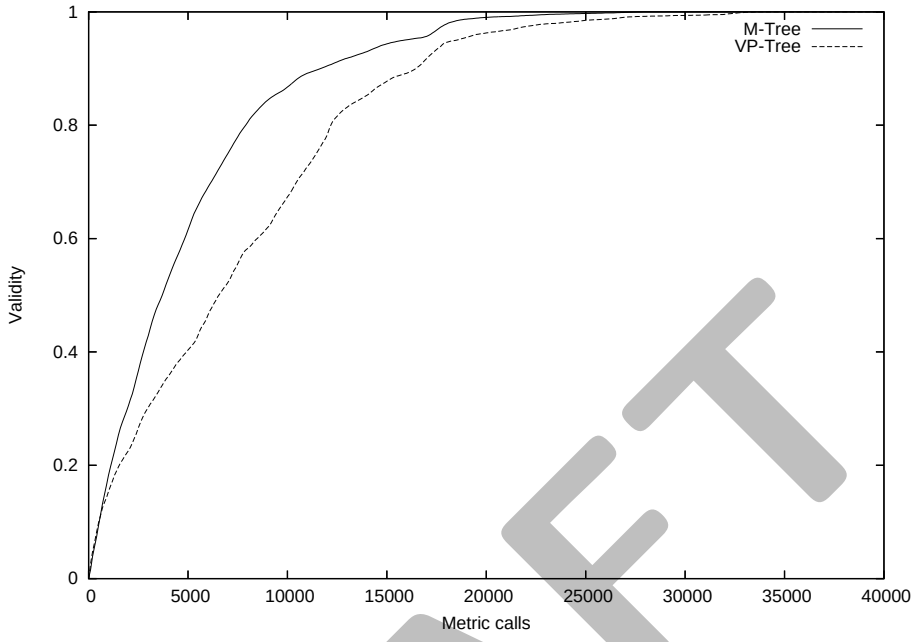

Figure 15: disk, k = 1024

- **VP Tree**: the vantage point tree structure [18, 19, 20].

Those two algorithms have been tried against the dataset **aerial**, **corel_hist**, **corel_uci** and **disk_trace** previously described. Note that the dataset **sprot40** is left apart because of its associated non-metric similarity function. The queries of interest here are near-neighbor search queries with various number of neighbors. Please note that no experiments have been done with a single (nearest) neighbor search. This choice is motivated by the fact that the query points are actually drawn from the dataset and therefore contained in the data structure where the query is actually performed. The empirical results for single (nearest) neighbor are indeed highly biased in favor of the data structures under tests. Nevertheless, the 2-near-neighbor search queries can considered as representative for the behavior of the nearest neighbor search.

Several empirical conclusions can be gathered from those results. First, there is no clear-cut advantage of using metric trees rather than vantage point trees. The results suggest that for the *nearest* neighbor search, the vantage point tree often outperforms the metric tree. In the other hand, the metric tree tends to outperforms the vantage point tree when the queries involve a large number of neighbors. To our knowledge, although those algorithms are very classical, it does not seem that such behavior having been noticed before. We do not have a rigorous answer to provide to explain this behavior, nevertheless

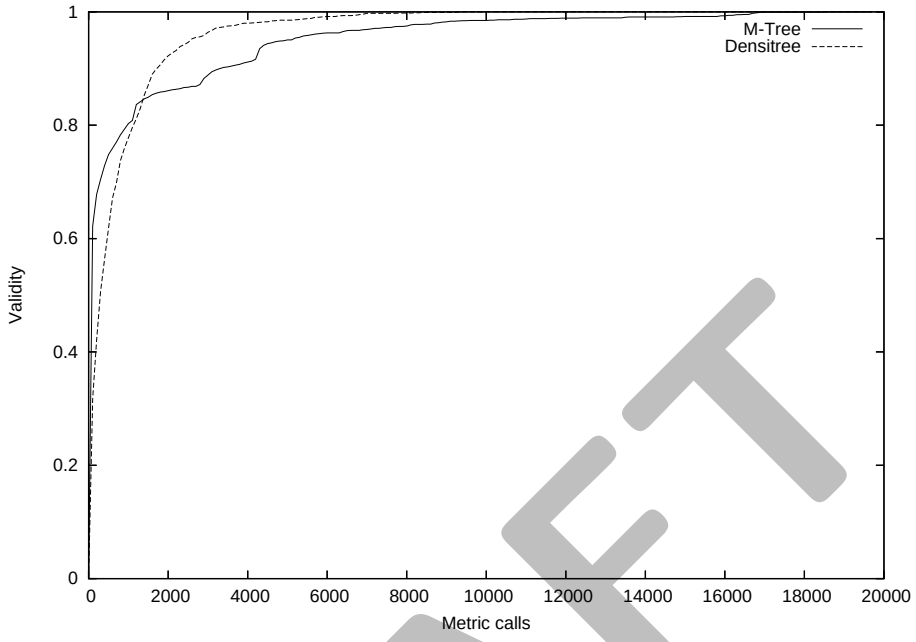

Figure 16: Non-metric, corelHist, k = 2

we will provide here our interpretation based our empirical observations. Let us recall the three exploration phases that have discussed here above: (i) descend (ii) visiting the near-neighbors (iii) visiting the remote neighbors. The insights provided by the those phases help to interpret the behavior discussed here. The main difference between vantage point trees and metric trees lies in the fact that an exploration step requires two similarity function calls for the metric tree vs a single call for the vantage point tree. Our interpretation is that the single call method advantages the vantage point tree (compared to the metric tree) during the phase (i) because it avoids calls that might not be exploited later one. Nevertheless, the price of those "avoided" calls is a less efficient exploration in phase (iii). The difference between a single neighbor search or a numerous neighbors search is the respective weights of the phase (i) and (iii). In a single neighbor search, the phase (i) is the most heavy phase; in a numerous neighbor search, the phase (iii) becomes preponderant. In our view, those considerations provides an explanations of the observed empirical behaviors. Nevertheless more theoretical approaches might bring much more satisfying answers.

5.2 Results in non metric space

The results presented in this section are gathered in the figures 16 to 26. Similarly to the previous section, those figures are query accuracy profiles (introduced in section 3.3).

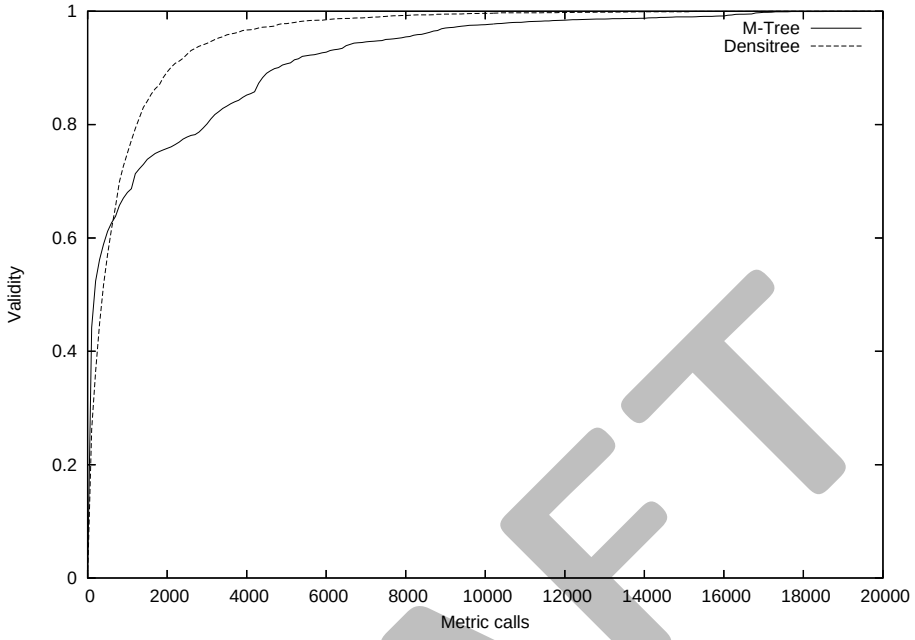

Figure 17: Non-metric, corelHist, k = 8

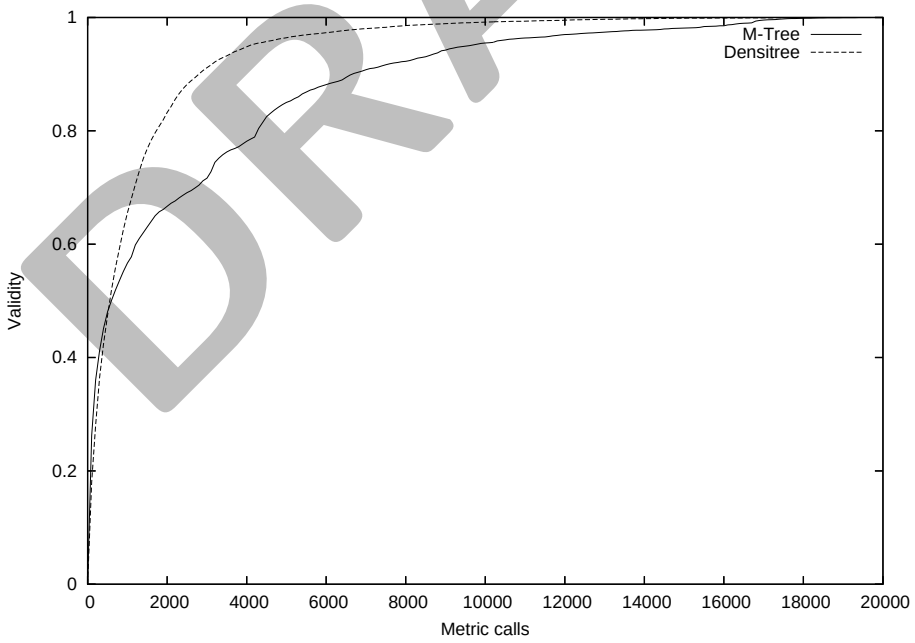

Figure 18: Non-metric, corelHist, k = 64

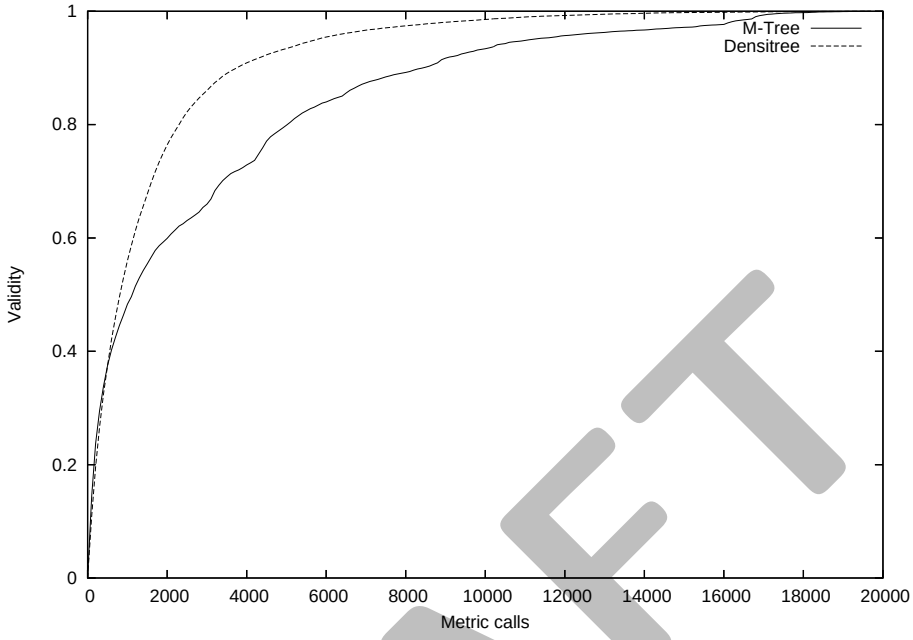

Figure 19: Non-metric, corelHist, k = 256

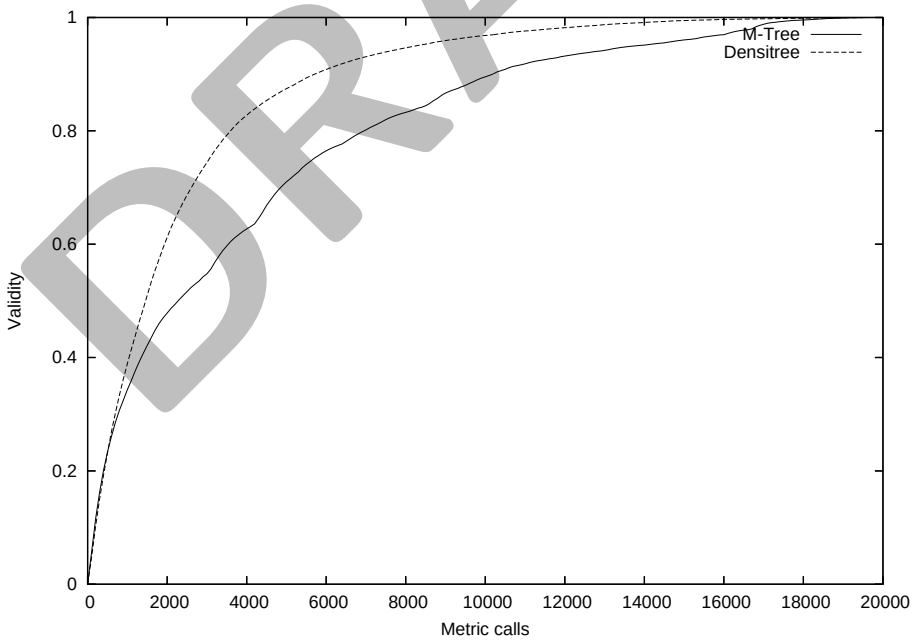

Figure 20: Non-metric, corelHist, k = 1024

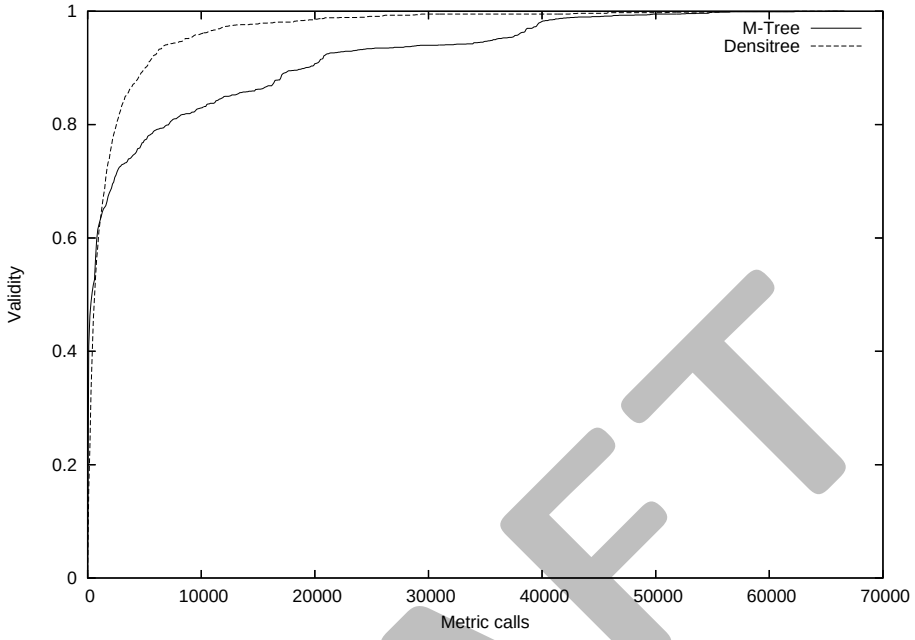

Figure 21: Non-metric, corelUci, k = 2

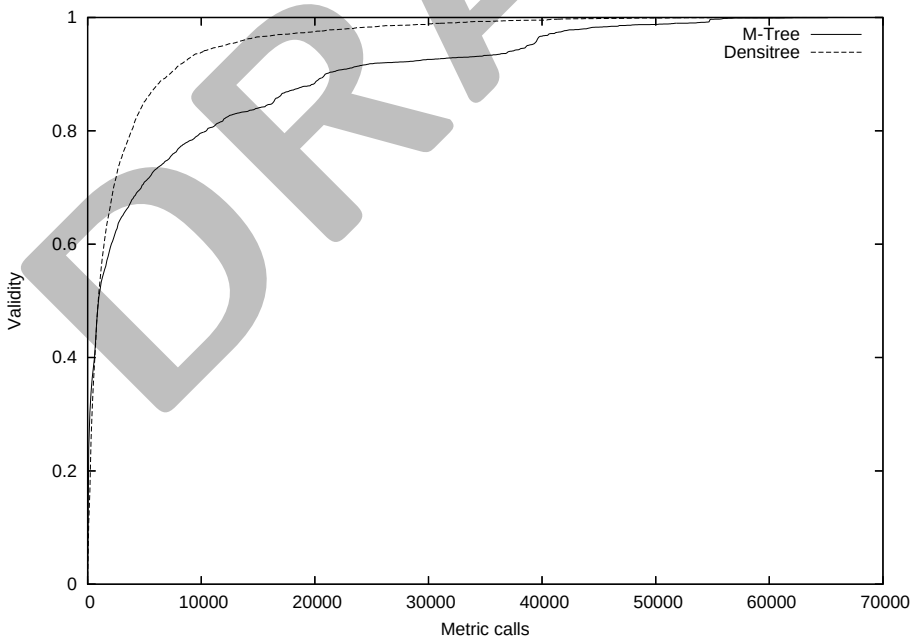

Figure 22: Non-metric, corelUci, k = 8

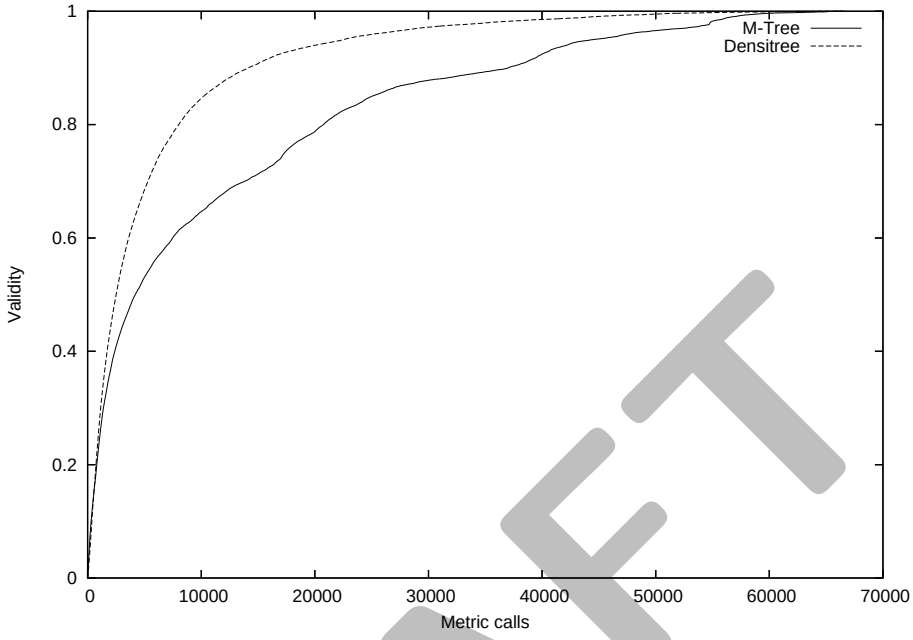

Figure 23: Non-metric, corelUci, k = 1024

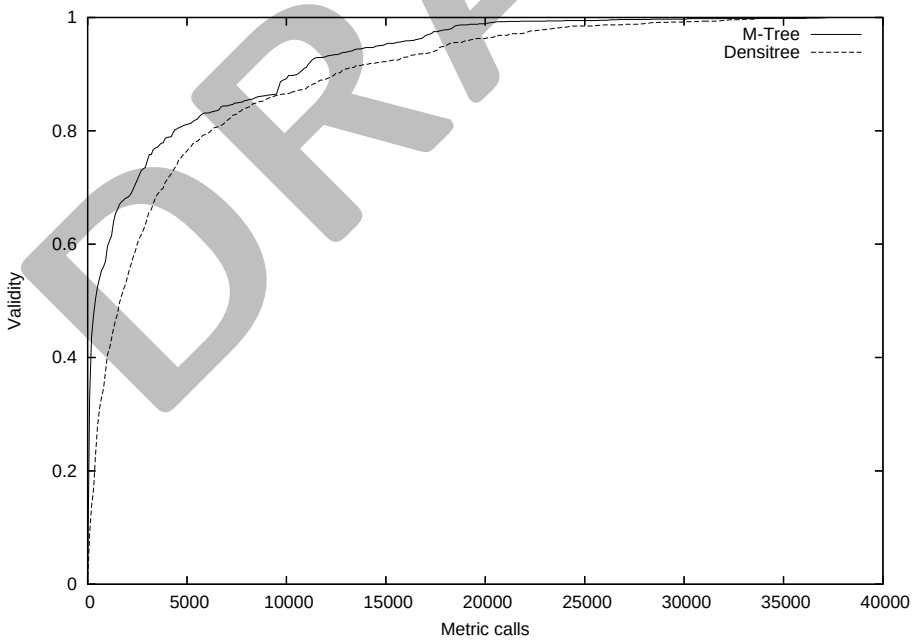

Figure 24: Non-metric, disk, k = 2

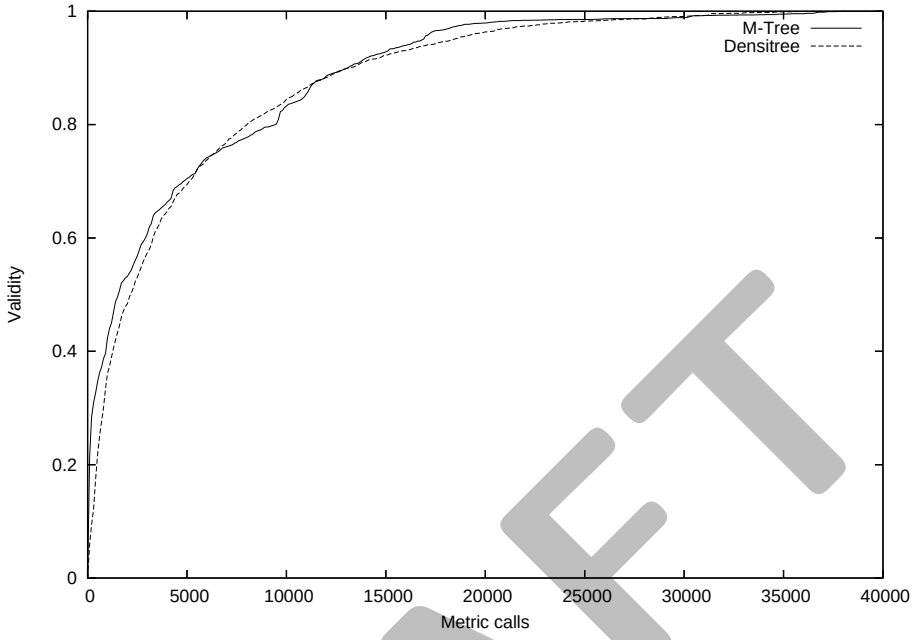

Figure 25: Non-metric, disk, k = 8

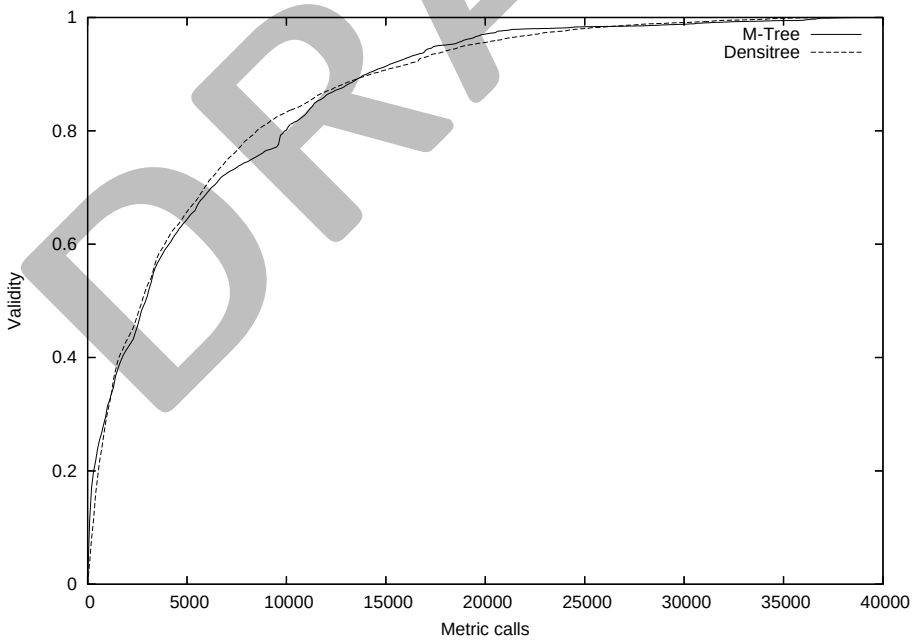

Figure 26: Non-metric, disk, k = 64

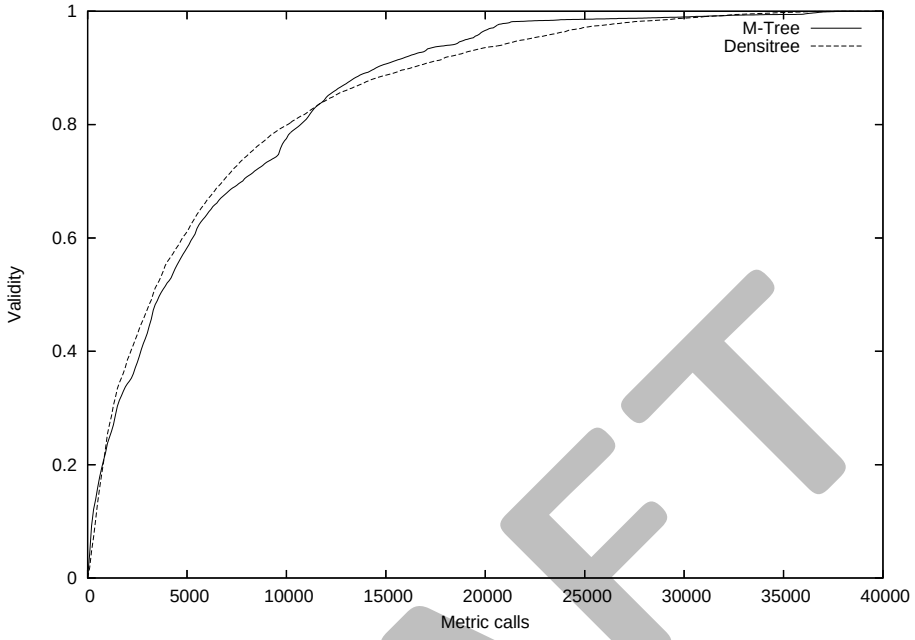

Figure 27: Non-metric, disk, k = 256

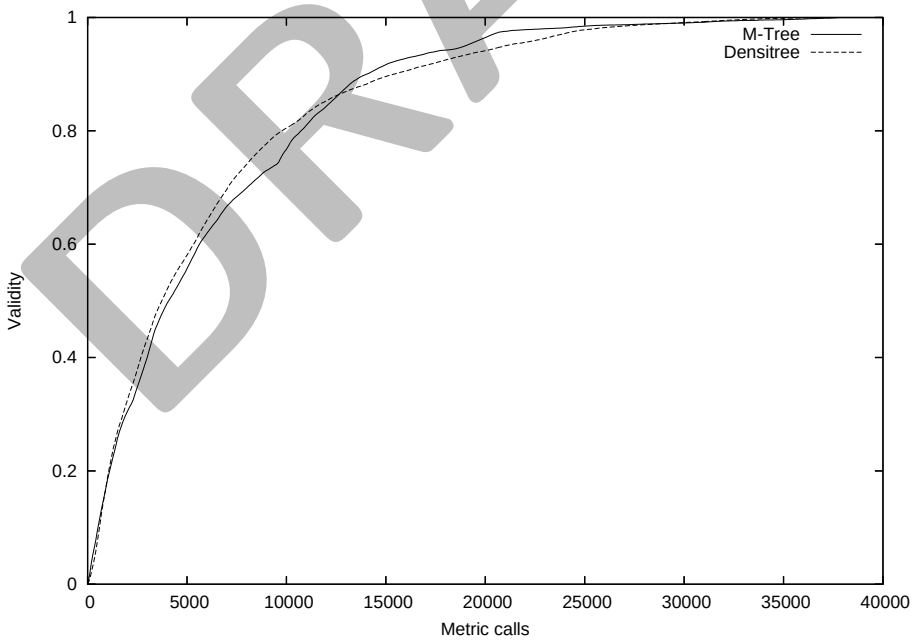

Figure 28: Non-metric, disk, k = 1024

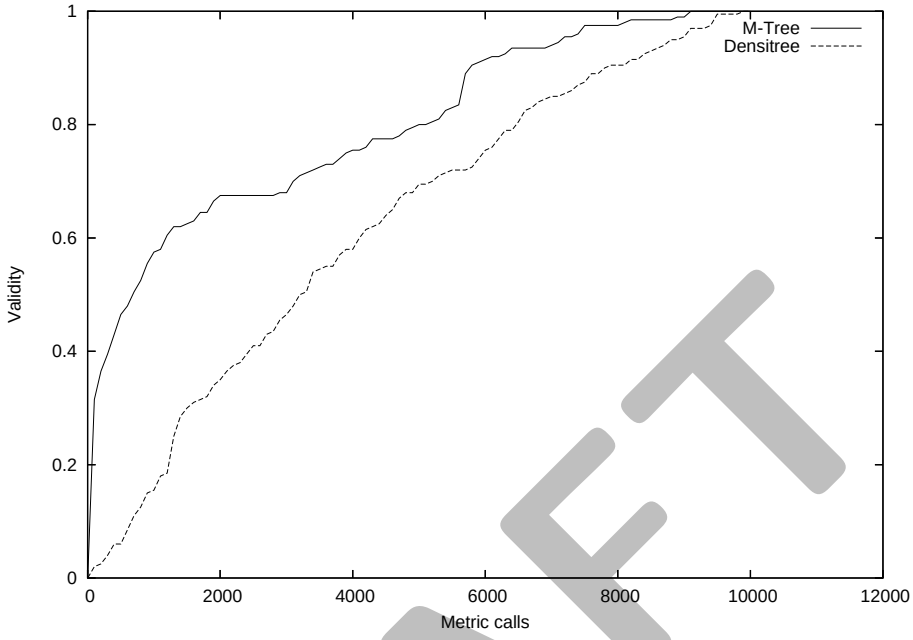

Figure 29: Non-metric, sprot, k = 2

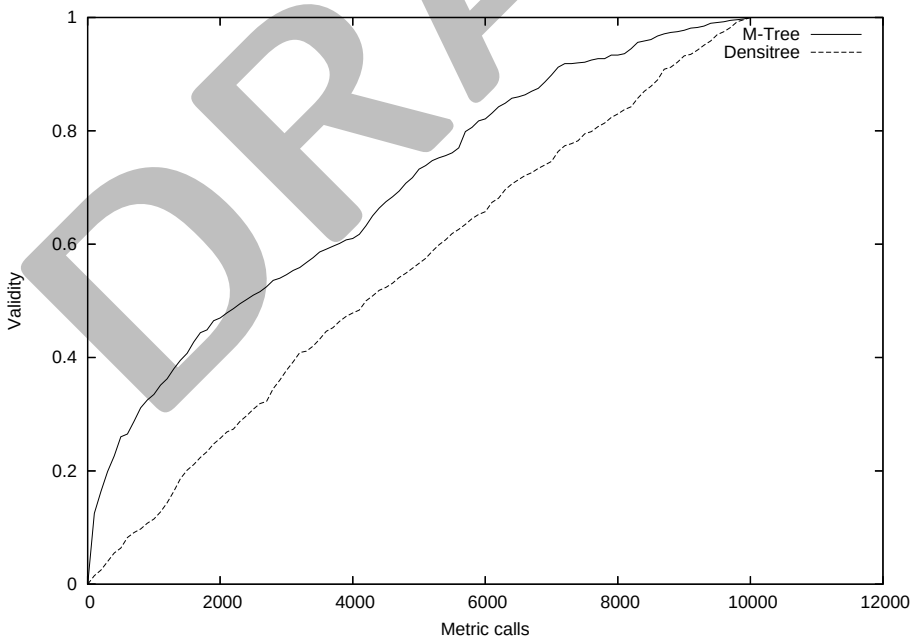

Figure 30: Non-metric, sprot, k = 8

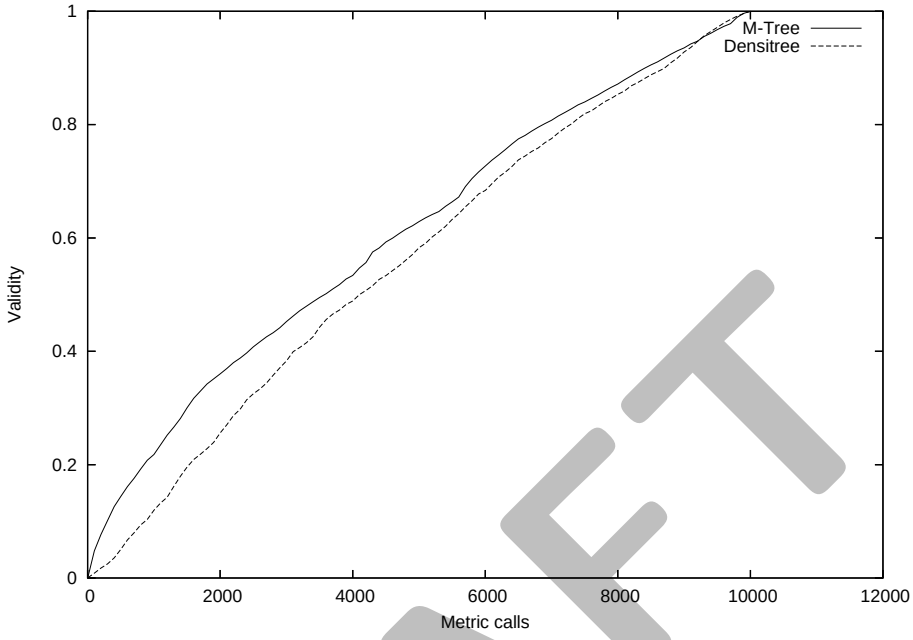

Figure 31: Non-metric, sprot, k = 64

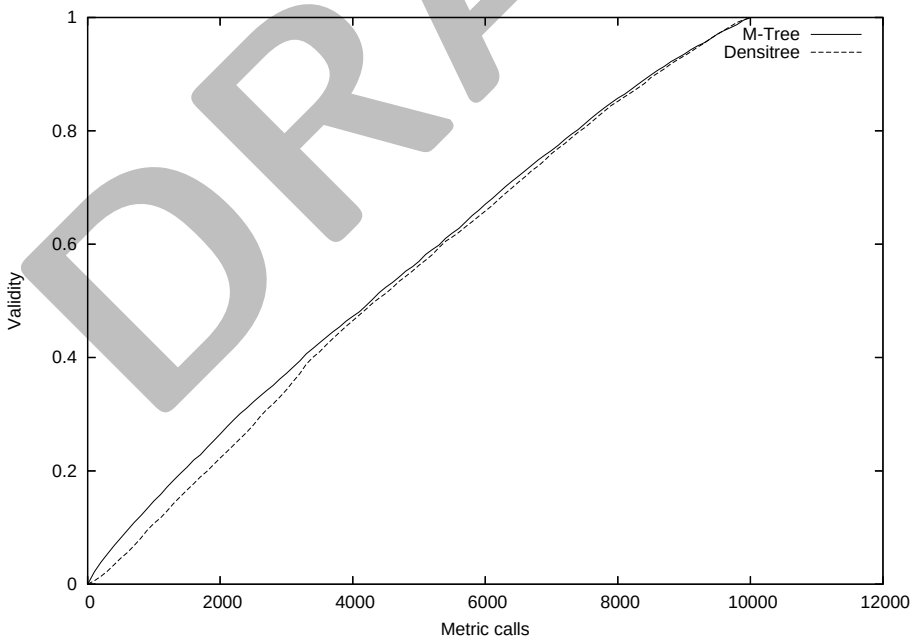

Figure 32: Non-metric, sprot, k = 256

Our experiments involve the two metric NNS algorithms that have been previously described:

- **M-Tree**: the metric tree structure (see references here above).
- **Densitree**: the densitree extensively discussed in section 4.

The metric tree used in this section has been slightly modified to fit the *non-metric* assumption. In particular, the triangular inequality pruning has been removed from the metric tree query algorithm. Consequently, the metric tree exploration relies solely on a greedy nearest-node first approach.

The details of the densitree implementation have been given in section 4, and will not be provided here again. Let us just recall that the densitree exploration is based on particular classifiers that we call "density estimator". Those classifiers aim to exploit several similarity values to score each node.

Several empirical conclusions can be drawn from the experiments of this section. The most important conclusion, is that, contrary to a common belief, the triangular inequality is far from being a necessity to perform an efficient exploration. It is clear, by comparing the d accuracy profiles with or without the triangular inequality, that relying on the triangular inequality improves the algorithm efficiency; nevertheless the improvement is rather limited.

The second empirical conclusion is that the densitree tends to outperforms the metric tree based on the sole query accuracy profile. The behavior is not surprising, because the densitree exploration exploits much more data to decide which will be explored next. Additionally, as discussed in section 4, our current implementation of density estimators relies on a large number of heuristics. Therefore many improvements can be expected from a better "tuning" of the densitree query algorithm. Although not presented here, please not that this behavior would be exactly the same if the triangular inequality had been used. Indeed, the dataset partitioning of the densitree is identical to the metric tree one's. Therefore, the two methods (metric tree and densitree) would be benefit of the triangular inequality pruning in an equal fashion. This implies, in particular, that the densitrees, can actually also lead to any improved NNS even in the more classical metric case.

Nevertheless, it has to be noticed that in practice the CPU costs are not limited to the call the similarity function. Based on a pure CPU benchmark, the densitrees would have been outperformed by the metric tree excepts for the datasets that are associated to an expensive similarity function).

6 Conclusions

After reviewing the most classical NNS algorithms that relies on the metric space assumption, we have introduced the more general problem of NNS in non-metric space. In particular, no triangular inequality assumption is available in non-metric spaces. The analysis of the non-metric NNS problem has lead

us to introduction the notion of query accuracy profile to evaluate in a meaningful way the efficiency of a NNS in non-metric space. Taking the accuracy profile as a criterion to evaluate the NNSalgorithms, we have seen that the exploration criterions become the key issue to design an efficient NNS algorithm in non-metric space. We have proposed as simple greedy non-metric search algorithm by on the very classical metric tree structure. Then a deeper analysis of the exploration problem lead us to introduce a new data structure that we call "densitrees". The densitree exploration exploits more information than the simple greedy exploration. Finally the previously discussed algorithms are empirically compared against classical datasets of the NNS literature. The densitrees seems to outperform previously known methods for metric or non-metric NNS.

References

[1] http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html.

[2] http://www.expasy.org/sprot/.

[3] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[4] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? *Lecture Notes in Computer Science*, 1540:217–235, 1999.

[5] Edgar Chavez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and Jose L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[6] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric space. In *Proc. Of the 23 rd VLDB Conference*, 1997.

[7] Paolo Ciaccia, Fausto Rabitti, Pavel Zezula, and Marco Patella. M-tree project. http://www-db.deis.unibo.it/Mtree/.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[9] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *The VLDB Journal*, pages 518–529, 1999.

[10] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529. Morgan Kaufmann Publishers Inc., 1999.

[11] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2001.

[12] Ting Liu and Andrew W. Moore. Datasets repository. http://www.cs.cmu.edu/ tingliu/dataset/dataset.html.

[13] Ting Liu, Andrew W. Moore, Alexander Gray, and Ke Yang. An investigation of practical approximate nearest neighbor algorithms. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 17*, Cambridge, MA, 2005. MIT Press.

[14] B. S. Manjunath. Airphoto dataset. http://vision.ece.ucsb.edu/download.html.

[15] B. S. Manjunath and W. Y. Ma. Texture features for browsing and retrieval of image data. *IEEE Trans. Pattern Anal. Mach. Intell.*, 18(8):837–842, 1996.

[16] M. Omohundro. Bumptrees for efficient function, constraint, and classification learning.

[17] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. In *Information Processing Letters*, volume 40, pages 175–179, 1991.

[18] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

[19] Peter N. Yianilos. Excluded middle vantage point forests for nearest neighbor search. Technical report, NEC Research Institute, Princeton, NJ, July 1998.

[20] Peter N. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 361–370, 2000.

[21] B. Zhang and S.N. Srihari. A fast algorithm for finding k-nearest neighbors with non-metric dissimilarity. In *FHR02*, pages 13–18, 2002.