Exploring intersection trees for indexing metric spaces

Zineddine KOUAHLA
Laboratoire d'informatique de Nantes Atlantique (LINA, UMR CNRS 6241)
Polytechnic School of the University of Nantes, Nantes, France

Email: name.surname@univ-nantes.fr

***Abstract*—Searching in a dataset for objects that are similar to a given query object is a fundamental problem for several applications that use complex data. The general problem of many similarity measures for complex objects is their computational complexity, which makes them unusable for large databases. Here, we introduce a study of a variant of a metric tree data structure for indexing and querying such data. Both a sequential and a parallel versions are introduced. The efficiency of our proposal is demonstrated through experiments on real-world data, as well as a comparison with existing techniques: MM-tree and Slim-tree.**

I. INTRODUCTION

For several decades, indexing techniques have been developed in order to deal with efficient searches over large collections of data. Especially associative searches, i.e., searches where part of the information to be retrieved is provided, namely a key. This basic problem has been extended over the years in order to retrieve information based on any subset of its contents as well as taking care of imprecisions. When considering indexing data as vectors, homogeneous or inhomogeneous ones, it turned out that search and indexing become more and more difficult, when increasing the dimension of the so-called vectors. This has been named the "dimensionality-curse problem." The reader can find several surveys to present and compare existing multidimensional indexing techniques [1], [2], [3]. However, the objects to be indexed are often more complex than mere vectors (e.g., graph matching in ontologies) or cannot be simply and meaningfully concatenated in order to give a larger vector (e.g., colour histograms and keywords for describing images in a multimedia database). Hence, the focus of indexing has partly moved from multidimensional spaces to metric spaces, i.e., from exploiting the data representation itself to working on the similarities that can be computed between objects. Inherently, the difficulties of multidimensional spaces remain, in a generalised version, whereas new difficulties arise due to the lack of information on the objects.

Let us note that any multidimensional space is also a metric space. Generally, it suffices to use one of the norms as a distance, e.g., the Manhattan or the Euclidean distances. This can become an advantage, a given distance revealing the intrinsic dimensionality of the objects rather than each and every dimension of the vectors. Conversely, it is possible to use a few objects as pivots and to compute the distance of each object to each pivot. The concatenation of the distances to each pivot is a real-based vector, e.g., FastMap [4]. So, there exists a lot of commonalities between the two families of indexing structures.

This paper introduces the αIM-tree technique inspired by the MM-tree. It is organised as follows: Section II introduces kNN queries and reviews a short taxonomy of indexing technique in metric spaces. Then, Section III introduces our proposal, the αIM-tree, overviews its main characteristics and the corresponding algorithms. Section IV discusses the experimental results. Section V concludes the paper and introduces research directions.

II. INDEXING IN METRIC SPACES

Metric spaces are becoming more and more useful, for several applications need to compare objects based on a similarity between them that is formalised as a mathematical distance, e.g., multimedia objects. Let us focus the basic search query and introduce some indexing technique from the literature.

A. Similarity Queries in Metric Spaces

There exist numerous measures of similarity applicable to various kinds of "objects", e.g., Minkowski distances, among which the best known are the Manhattan distance and the Euclidean distance, that can be applied to any vector-like data such as colour histograms in multimedia databases. Formally, a metric space is defined for a family of elements that are comparable through a given distance.

Definition 1 (Metric space): Let $\mathcal{O}$ be a set of elements. Let $d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+$ be a distance function, which verifies:

1) non-negativity: $\forall (x, y) \in \mathcal{O}^2, d(x, y) \geq 0$;
2) reflexivity: $\forall x \in \mathcal{O}, d(x, x) = 0$;
3) symmetry: $\forall (x, y) \in \mathcal{O}^2, d(x, y) = d(y, x)$;
4) triangle inequality: $\forall (x, y, z) \in \mathcal{O}^3, d(x, y) + d(y, z) \leq d(x, z)$.

The concept of metric space is rather simple and leads to a limited number of possibilities for querying an actual database of such elements. These are called similarity queries and several variants exist. We consider *k* nearest neighbour (kNN) searches, i.e., searching for the k closest objects with respect to a query object q and a given distance d.

Definition 2 (kNN query): Let $(\mathcal{O}, d)$ be a metric space. Let $q \in \mathcal{O}$ be a query point and $k \in \mathbb{N}$ be the expected number of answers. Then $(\mathcal{O}, d, q, k)$ defines a kNN query, the value of which is $S \subseteq \mathcal{O}$ such that $|S| = k$ (unless $|\mathcal{O}| < k$) and $\forall (s, o) \in S \times \mathcal{O}, d(q, s) \leq d(q, o)$.

B. Background

Metric spaces introduce the notion of topological ball, which is close to a broad match. It allows to distinguish between inside and external objects.

Definition 3 ((Closed) Ball): Let $(\mathcal{O}, d)$ be a metric space. Let $p \in \mathcal{O}$ be a pivot object and $r \in \mathbb{R}^+$ be a radius. Then $(\mathcal{O}, d, p, r)$ defines a (closed) ball, which can partition inner objects from outer objects:

$$I(\mathcal{O}, d, p, r) = \{o \in \mathcal{O} : d(p, o) \leq r\} ;$$
$$O(\mathcal{O}, d, p, r) = \{o \in \mathcal{O} : d(p, o) > r\}.$$

Another useful partitioning concept is the one of generalised "hyper-plane."

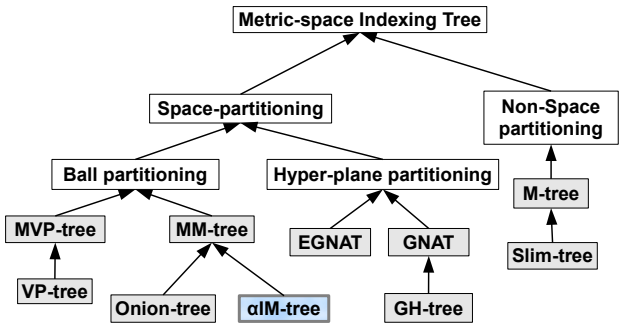

Fig. 1. A simplified taxonomy of indexing techniques in metric spaces

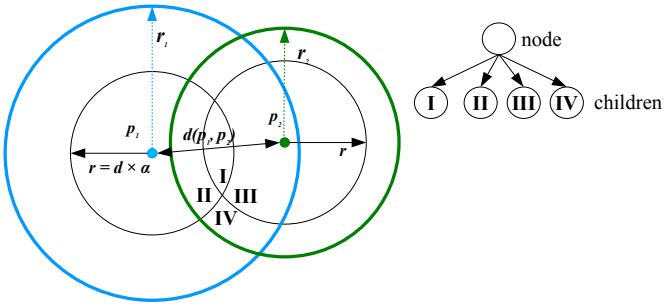

Fig. 2. The partitioning principle in an αIM-tree and the corresponding tree structure

Definition 4 (Generalised hyper-plane): Let $(\mathcal{O}, d)$ be a metric space. Let $(p_1, p_2) \in \mathcal{O}^2$ be two pivots, with $d(p_1, p_2) > 0$. Then $(\mathcal{O}, d, p_1, p_2)$ defines a generalised hyper-plane:

$$H(\mathcal{O}, d, p_1, p_2) = \{o \in \mathcal{O} : d(p_1, o) = d(p_2, o)\}$$

which can partition "left-hand" objects from "right-hand" objects:

$$L(\mathcal{O}, d, p_1, p_2) = \{o \in \mathcal{O} : d(p_1, o) \leq d(p_2, o)\} ;$$
$$R(\mathcal{O}, d, p_1, p_2) = \{o \in \mathcal{O} : d(p_1, o) > d(p_2, oP)\}.$$

Based on these two partitioning techniques, we present a general summary of some indexing techniques in metric spaces. Figure 1 shows a simple taxonomy of some techniques of indexing in metric spaces. There are two main cases:

1) The first one is based on the partitioning of the space. Two sub-approaches are included in this first class:
   a) One of them uses ball partitioning, like VP-tree [5], mVP-tree [6], etc.
   b) The other approach uses hyper-plane partitioning such as GH-tree [7], GNAT [8], etc.
2) The second class that does enforce a partitioning (non-space-partitioning). There, we find M-tree, Slim-tree, etc.

In the second class, the M-tree [9] builds a balanced index, allows incremental updates, and it performs reasonably well in high dimension. Unfortunately it suffers from the problem of overlapping that increases the number of distance calculations to answer a query. There is an optimised version of it: the Slim-tree [10]. It mainly reorganises the M-tree index in order to reduce overlaps. The used algorithm, called slim-down, has shown good performances on the research algorithm and has reduced its processing time. Its defect is the need for reinserting objects.

The first class, the one base on space-partitioning of the space, either with balls or with hyper-planes, is richer.

The GH-tree is a type of index based on the partitioning of hyper-planes. It has proven its efficiency in some dimensions but it is still inefficient in large dimensions. The principle of this technique is the recursive partitioning of space into two regions. We choose each time two pivots, and each one is associated to the nearest objects. The problem of this technique is that intersections between regions are influenced on the search algorithm which made the GH-trees less effective in large spaces.

The VP-tree is a type of index based on a partitioning by a ball. The VP-tree building process is based on finding the median element of a set of objects. The mVP-tree is an enary generalisation of the VP-tree. The mVP-tree nodes are divided into quantiles. In fact, the operations (insertion and search) on this type of index are very similar to VP-trees. Often, it behaves better; but there is not enough differences to investigate further.

In recent years, a new technique has emerged, the MM-tree [11], which also uses the principle of partitioning by balls, but it is based on the exploitation of regions obtaines from the *intersection* between two balls (See Figure 2). The general idea of this structure is to select two pivots from the set of objects, in a random way, then to partition the space into four *disjoint* regions using these two balls, their intersection, their respective differences, and the external objects. The partitioning is done in a recursive way. In order to improve the balancing of the tree, a semi-balancing algorithm is applied near to the leaves, which reorganises the objects in order to gain of level.

An extension of this technique was developed: the Onion-tree [12]. Its aim is to divide the region IV to create successive expansions that improve the search algorithm, because the region IV is particularly vast. The goal is to have an index less deep and wider to go faster to the right answers to a query. In our opinion, the problem is not totally solved because the construction phase is always slow due to the reinsertion of objects.

On the one hand, we believe that parallelism is unavoidable for dealing with large datasets that present inherent "dimensionality" difficulties. Some directions have already been investigated. On the other hand, we think that the important open issue is finding the trade-off between a maximally parallel kNN query algorithm – that scans all the data – and the parallel gathering of (statistical) information about the unknown radius – that can limit the number of data accesses. We also argue that some replication of the data would be beneficial both to a sequential version and also to a parallel one, something that, to the best of our knowledge, has not been exploited in the literature.

III. αIM-TREE

In this section, we introduce a new quadrary tree, called αIM-tree (α Intersection Metric tree), as an indexing technique in metric spaces. It is a memory-based Metric Access Method (MAM) that divides recursively the dataset into four *disjoint* regions by selecting two pivots.

Figure 2 illustrates informally the way the tree is built at a given step in the refining process of recursively splitting the dataset.

Two pivots are chosen as being the farthest from each other in a given data subset. A radius is computed so as to create an intersection between the two balls centred on these pivots but without including them. Therefore, the radius varies between the pivots inter-distance (excluded) and the pivots mid-point (excluded). The computation depends on the α parameter introduced and discussed below.

Thanks to these two inner balls, we can divide the space into four disjoint regions, namely (I) their intersection, (II) and (III) their respective differences, and (IV) the complement to their union. Notice that we adds two "outer" balls defined by the radii, r_1 and r_2, given by the farthest points to p_1 and p_2 respectively; sometimes, these points are located inside the balls, sometimes outside. They help to reduce faster the query radius r_q.

A. Definition

Let us introduce formally the αIM-tree.

Definition 5 (αIM-tree): Let $M = (\mathcal{O}, d)$ be a metric space. Let $\frac{1}{2} < \alpha < 1$ be a given parameter to be studied later.

We define $\mathcal{N}_{M,\alpha}$, or $\mathcal{N}$ for short, as the nodes of a so-called αIM-tree.

A leaf node consists merely of a subset of the indexed objects:

$$L = \langle E \rangle$$

with $E \subseteq \mathcal{O}$.

An inner node is a nonuple:

$$N = \langle p_1, p_2, r, r_1, r_2, N_1, N_2, N_3, N_4 \rangle$$

where:

- $(p_1, p_2) \in \mathcal{O}^2$ are two distinct *pivots*, i.e., with $d(p_1, p_2) > 0$;
- $r = d(p_1, p_2) \times \alpha$ defines two balls, $B_1(p_1, r)$ and $B_2(p_2, r)$, centred on p_1 and p_2 respectively and having a common radius value, large enough for the two balls to have a non-empty intersection;
- $(r_1, r_2) \in \mathbb{R}^{+2}$ are the distances to the farthest object in the sub-tree rooted at that node N with respect to p_1 and p_2 respectively, i.e., $r_i = \max\{d(p_i, o), \forall o \in N\}$ for $i = 1, 2$ where the set notation $o \in N$ is abusely used for the union of the leaf extensions that are rooted at N;
- $(N_1, N_2, N_3, N_4) \in \mathcal{N}^4$ are four sub-trees such that:
  - $N_1 = \{o \in N : d(p_1, o) < r \wedge d(p_2, o) < r\}$;
  - $N_2 = \{o \in N : d(p_1, o) < r \wedge d(p_2, o) \geq r\}$;
  - $N_3 = \{o \in N : d(p_1, o) \leq r \wedge d(p_2, o) < r\}$;
  - $N_4 = \{o \in N : d(p_1, o) \geq r \wedge d(p_2, o) \geq r\}$;

  with the same informal set notation for the extension of an inner node.

B. Incremental construction

Building a αIM-tree is realised incrementally. The insertion is done in a top-down way. Algorithm 1 describes formally the incremental insertion process.

Initially, a tree is empty, i.e., it is a leaf with an empty set of objects.

The first insertions in a leaf make it only grow until a maximum number of elements is attained. This is the $c_{\max}$ parameter in Algorithm 1. Due to time complexity considerations, its value cannot be larger than $\sqrt{n}$ where $n = |E|$ is the cardinal of the whole population of objects to be inserted in the tree.

When the cardinal limit is reached, a leaf is replaced by an inner node and four new leaves are obtained by splitting the former set of objects into four subsets according to the conditions given in Definition 5.

In order to split the object set, two *distinct* pivots have to be chosen. The selection of these pivots plays an important role in our proposal along with the $c_{\max}$ parameter. The goal is to balance, as much as possible, the tree. In Algorithm 1, we decided to choose two objects as far as possible from each other. The larger the cardinal limit, the more representative of the whole collection the objects should be, and the more actually distant should the pivots be with respect to the whole collection E.

Inserting a new object in an inner node amounts to selecting the subtree that has to contain it with respect to the conditions given in Definition 5 and applying the insertion recursively. Also, r_1 and r_2 may be increased. Let us note that, at each inner node, only two distances are calculated in order to insert a new object. Besides, the tree tends to be rather balanced, hence inserting a new object is a logarithmic operation, in amortised cost.

C. Similarity queries

Next, let us describe the algorithms associated to αIM-trees for answering kNN queries. We introduce a standard sequential algorithm along with a parallel version and a companion algorithm to this parallel version.

1) Parallel version of the kNN search: Let us start with the parallel version, which is simpler. Algorithm 2 works as follows. Leaf nodes contain a subset of the indexed data. In order to find the k nearest neighbours with respect to a given leaf, it is sufficient to sort them in increasing distances to the query object. Then, we return *at most* the k^{th} first sorted elements. Notice that a full sort is not necessary; there exists a variant, say "k-sort", in $O(n.\log_2 k)$ rather than $O(n.\log_2 n)$. Here, we have $1 \leq n \leq \sqrt{|E|}$.

In inner nodes, the principle is to start, in parallel, a kNN search in all the *candidate* children. Being a candidate child depends on the intersection between the query ball $B(q, r_q)$ and the topology of the child. The four regions are distinct in shape (See Figure 2), therefore we have to provide four distinct conditions, namely C_1 to C_4 in Algorithm 2.

Notice that in the recursive calls, the upper bound, r_q, which is initially set to $+\infty$ by default, can be (hopefully) decreased. This, again, depends on the considered child, as indicated by the evaluations of r_{q1} to r_{q4}.

The results from zero to all of the four children are merged and at most k of them returned. Notice, again, that this step does not really require a sort, but only a sequence of merges. In both cases, the complexity is "constant", i.e., in $O(4.k)$ – rather than $O(4.k.\log_2 k)$.

In the worst case, the overall computation can be seen as the partial answers going up from all the leaves to the root of the tree; this is a *full* parallel search. In the best case, only the leaves that *could* contain at least one candidate answer take part in the ascending computation; this is the *perfect* parallel search. In both cases, the overall average time complexity can be expected and estimated to be in:

$$\begin{array}{l} O\left(\frac{1}{2}\sqrt{n}.\log_2 k + \log_4 \frac{n}{\sqrt{n}}.4.k\right) = \\ O\left(\frac{1}{2}\sqrt{n}.\log_2 k + \frac{1}{2}\log_4 n.4.k\right) = \\ O(\sqrt{n} + \log_4 n) = \\ O(\sqrt{n}) \end{array}$$

with $n = |E|$ and $c_{\max} = \sqrt{n}$, where the first term corresponds to the parallel computations on the leaves and the second term to the parallel computations on the way up. However, this result holds under the hypotheses that the tree is quasi balanced, that the distribution of elements in the leaves is uniform, and that the distribution of the elements among the various children is also uniform. In that case, it is quite an improvement over a mere "k-sort" in $O(n.\log_2 k)$, though it requires $O(\sqrt{n})$ processors, in the worst case, to be executed at full speed.

Relying on the decreases expressed by the computations of r_{q1} to r_{q2} in Algorithm 2 is quite insufficient. It is necessary to estimate an

Algorithm 1 Incremental insertion in a αIM-tree

Insert $\left(o \in \mathcal{O}, N \in \mathcal{P}(\mathcal{N}), d \in (\mathcal{O} \times \mathcal{O} \to \mathbb{R}^+), c_{\max} \in \mathbb{N}^*, \alpha \in \left]\frac{1}{2}, 1\right[\right) \in \mathcal{N}$
with:

- $E' = E \cup \{o\}$;
- $(p_1, p_2) = \text{argmax}_{(o_1,o_2)\in E'^2}\{d(o_1, o_2) : d(o_1, o_2) > 0\}$;
- $r = d(p_1, p_2) \times \alpha$;
- $E'_1 = \{o \in E' : d(p_1, o) \leq r \wedge d(p_2, o) \leq r\}$;
- $E'_2 = \{o \in E' : d(p_1, o) < r \wedge d(p_2, o) \geq r\}$;
- $E'_3 = \{o \in E' : d(p_1, o) \geq r \wedge d(p_2, o) < r\}$;
- $E'_4 = \{o \in E' : d(p_1, o) \geq r \wedge d(p_2, o) \geq r\}$;

$$\triangleq \begin{cases} \langle E' \rangle & \text{if } N = \langle E \rangle \wedge |E| < c_{\max} \\ \left\langle \begin{array}{l} p_1, p_2, r, r_1, r_2, \\ \langle E'_1 \rangle, \langle E'_2 \rangle, \langle E'_3 \rangle, \langle E'_4 \rangle \end{array} \right\rangle & \text{if } N = \langle E \rangle \wedge |E| = c_{\max} \\ \left\langle \begin{array}{l} p_1, p_2, r, \max\{d(o, p_1), r_1\}, \max\{d(o, p_2), r_2\}, \\ \text{Insert}(o, N_1, d, c_{\max}, \alpha), N_2, N_3, N_4 \end{array} \right\rangle & \text{if } \begin{array}{l} N = \langle p_1, p_2, r, r_1, r_2, N_1, N_2, N_3, N_4 \rangle \wedge \\ d(p_1, o) \leq r \wedge d(p_2, o) \leq r \end{array} \\ \left\langle \begin{array}{l} p_1, p_2, r, \max\{d(o, p_1), r_1\}, \max\{d(o, p_2), r_2\}, \\ N_1, \text{Insert}(o, N_2, d, c_{\max}, \alpha), N_3, N_4 \end{array} \right\rangle & \text{if } \begin{array}{l} N = \langle \ldots \rangle \wedge \\ d(p_1, o) < r \wedge d(p_2, o) \geq r \end{array} \\ \left\langle \begin{array}{l} p_1, p_2, r, \max\{d(o, p_1), r_1\}, \max\{d(o, p_2), r_2\}, \\ N_1, N_2, \text{Insert}(o, N_3, d, c_{\max}, \alpha), N_4 \end{array} \right\rangle & \text{if } \begin{array}{l} N = \langle \ldots \rangle \wedge \\ d(p_1, o) \geq r \wedge d(p_2, o) < r \end{array} \\ \left\langle \begin{array}{l} p_1, p_2, r, \max\{d(o, p_1), r_1\}, \max\{d(o, p_2), r_2\}, \\ N_1, N_2, N_3, \text{Insert}(o, N_4, d, c_{\max}, \alpha) \end{array} \right\rangle & \text{if } \begin{array}{l} N = \langle \ldots \rangle \wedge \\ d(p_1, o) \geq r \wedge d(p_2, o) \geq r \end{array} \end{cases}$$

Algorithm 2 Parallel version of the kNN search

kNN $(N \in \mathcal{N}, q \in \mathcal{O}, k \in \mathbb{N}^*, r_q \in \mathbb{R}^+ = +\infty) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$
with:

- $d_1 = d(q, p_1)$;
- $d_2 = d(q, p_2)$;
- a specific condition for each branch:
  - $C_1 = B(q, r_q) \cap B(p_1, r) \neq \emptyset$, for the intersection;
  - $C_2 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \wedge B(q, r_q) \cap B(p_2, r) \neq \emptyset$, for the partial ball centred on p_1;
  - $C_3 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \wedge B(q, r_q) \cap B(p_2, r) \neq \emptyset$, for the partial ball centred on p_2;
  - $C_4 = B(q, r_q) \cap B(p_1, r_1) \neq \emptyset \vee B(q, r_q) \cap B(p_2, r_2) \neq \emptyset$, for the remaining space;
- a specific upper-bound decrease for each branch:
  - $r_{q1} = \min\{r_q, \min\{d_1 + r_1, d_2 + r_2\}\}$;
  - $r_{q2} = \min\{r_q, d_1 + r_1\}$;
  - $r_{q3} = \min\{r_q, d_2 + r_2\}$;
  - $r_{q4} = \min\{r_q, \max\{d_1 + r_1, d_2 + r_2\}\}$.

$$\triangleq \begin{cases} k\text{-sort}\{(d(x, q), x) : x \in E\} & \text{if } N = \langle E \rangle \\ k\text{-merge}\{\text{kNN}(N_i, q, k, r_{qi}) : C_i, \forall 1 \leq i \leq 4\} & \text{if } N = \langle p_1, p_2, r, r_1, r_2, N_1, N_2, N_3, N_4 \rangle \end{cases}$$

upper-bound to the forthcoming k^{th} distance. In this way, the first call on the root node could be initialised with a much more satisfactory value than $+\infty$, the best estimation leading to a perfect search.

This is the role of a companion algorithm to Algorithm 2. It finds a minimal upper-bound by scanning a limited number, β, of *inner* nodes. (Leaf nodes are too costly.) Due to lack of space, we do not detail this simpler algorithm. Besides, it works in a similar way to the sequential version of a kNN search, that is introduced in the following section.

2) Sequential version of the kNN search: Contrary to its parallel counterpart, a sequential search can benefit from information gathered in previous branches. More specifically, a *fastly* decreasing upper-bound is obtained by the incrementally constructed answer. Conversely, in the parallel version, the parallel computations cannot exchange such an information.

Algorithm 3 is a "standard" algorithm, i.e., a kNN search adapted to our proposal. It runs a kind of "branch-and-bound" algorithm where the upper bound r_q is the monotonically decreasing (future) range of the k^{th} answer. Therefore, Algorithm 3 accepts an A parameter, i.e., a solution "so-far."

The tree is traversed in pre-order. When arriving at a leaf node, the difference with the parallel version is that the currently known sub-solution is merged with the local sub-solution.

When arriving on an inner node, the difference is that the calls are not made in parallel but in sequence. In that way, the sub-solution from a previous call is transmitted to the next call, hence improving the knowledge of the next branch on the query upper-bound.

IV. Experiments and Comparison

In order to show the efficiency of our approach, we run some experiments. Firstly, we chose a dataset and the accompanying queries, then run various variants of the algorithms, and finally

Algorithm 3 Sequential version of the kNN search

KNN $(N \in \mathcal{N}, q \in \mathcal{O}, k \in \mathbb{N}^*, r_q \in \mathbb{R}^+ = +\infty, A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$
with:

- $A = A_0 = ((d_1, o_1), (d_2, o_2), \ldots, (d_{k'}, o_{k'}))$;
- $d_1 = d(q, p_1)$;
- $d_2 = d(q, p_2)$;
- $C_1, \ldots, C_4$ as of Algorithm 2;
- a specific upper-bound decrease for each branch:
  - $A_0 = A$;
  - $C_0 = \text{true}$;
  - $r_{q_0} = \min\{r_q, d_{k'}\}$ if $k' = k, r_q$ otherwise;
  - $A_i = \text{kNN}(N_i, q, k, r_{q_{i-1}}, A_{i-1})$ if C_i, A_{i-1} otherwise
  - $r_{q_i} = \min\{r_{q_{i-1}}, d_k\}$ if $|A_{i-1}| = k \wedge A_{i-1} = ((d_1, o_1), \ldots, (d_k, o_k)), r_{q_{i-1}}$ otherwise.

$$\begin{cases} k\text{-sort}(A \cup \{(d(x,q), x) : x \in E\}) & \text{if } N = \langle E \rangle \\ k\text{-merge}\{A_i, \forall 0 \leq i \leq 4\} & \text{if } N = \langle p_1, p_2, r, r_1, r_2, N_1, N_2, N_3, N_4 \rangle \end{cases}$$

evaluate some measures on the index structure as well as the kNN searches.

A. Indexed collections and queries made

We ran preliminary experiments on a few datasets. Here, we only report representative results on a multimedia dataset.

Effectively, multimedia descriptors are a good example of complex objects. We use a subset of the the MPEG-7 Dominant Color Descriptor (DCD) from the COPhIR dataset (Content-based Photo Image Retrieval).[1] The selected sample consists of 10,000 descriptors, each being a vector of 64 dimensions.

To run the search algorithms, we used 100 different descriptors as queries and averaged the results.

B. Algorithm variants

Firstly, we varied the building algorithm by using different values of α, namely 0.52, 0.55, 0.6, and 0.69. The $c_{\max}$ parameter was set to the square root of the size of the collection, i.e., $\sqrt{10,000} = 100$.

Then, we ran 20-NN search, i.e., $(k = 20)$, using four algorithmic variants:

- Parallel-full. A parallel kNN search as of Algorithm 2 algorithm (with r_q initialised with $+\infty$ and k to 20).
- Sequential. The sequential counterpart kNN search as of Algorithm 3.
- Perfect. A very special search algorithm, for we start our search with r_q initialised to the exact distance to the k^{th} object! So this is an ideal case that allows a base comparison with the other approaches.
- Parallel-bounded. The last variant is the parallel search algorithm where r_q is initialised with an *estimation* of the distance to the k^{th} neighbour, as provided by the companion algorithm. In the experiments, we varied the β parameter of this companion algorithm from 10, to 20, to 50, and to 150 (as the number of inner nodes investigated before running the parallel algorithm).

C. Measures on the index structure

The quality of the index structure is very important for the search algorithms. Contrary to the MM-tree, our proposal is a bucketed metric-tree. Therefore, the semi-balancing algorithm of the MM-tree is replaced by our leaf splitting. The question to answer is that of the balance of the resulting tree.

[1]It is available on demand at the address: http://cophir.isti.cnr.it.

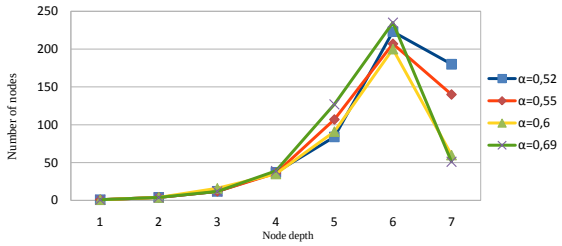

Fig. 3. Index structure with different values of α

Figure 3 shows the structure of the index for different values of α. We see that the chosen value has not a very high impact on the structure of the tree; it always remains balanced. Effectively, with a collection of 10,000 descriptors and leaves with a maximum of $\sqrt{10,000} = 100$ objects, the minimal depth of (i) a complete, (ii) perfectly balanced, and (iii) quadrary tree would be $\log_4 \sqrt{10,000} = 3.3$. Since we cannot reasonably expect each and every leaf to contain exactly 100 objects, nor can we imagine that the balance is perfect, not even that no children is empty, then the observed distribution of Figure 3 is very satisfactory; the actual depth is no more than twice the unreachable perfect case. Besides, we observed that the average number of objects per leaf is 26; therefore, the strictly minimal depth cannot be less than 4.3 rather than 3.3. Finally, notice that the value 0.69 for α almost reduces the expected depth by one with respect to 0.52.

This value of α plays a key role in the distribution of objects in the index. By varying it a lot on different datasets, we could have a hint on a good choice for this value. It seems that a perfect distribution of one third of the population for each of the regions I to III is the best choice; region IV should remain as empty as possible. So the goal seems to obtain a ternary tree, region IV being used mostly for "outliers."

D. Measures on searches

We recall that we ran a hundred different queries, each time looking for the 20 nearest neighbours, and averaged the results. We measured (i) the average percentage of visited leaves, (ii) the average percentage of visited inner nodes, (iii) the average number of distances calculated, and (iv) the average number of visited objects (out of 10,000). The statistics are shown in Table I.

Algorithm	Leaves (%)	Inner Nodes (%)	#distances	#objects
parallel-full	100	100	10800	10,000
sequential	15.65	9.36	7,254	2,951
parallel-bounded	18.1	14.35	8,654	3,521
perfect	2.42	5.22	1,054	960

TABLE I
PERFORMANCE STATISTICS OF FOUR KNN SEARCH ALGORITHMS ON A αIM-TREE

β	Leaves (%)	Inner Nodes (%)	#distances	#objects
10	46.80	57.99	10,231	4,787
20	44.24	55.34	9,963	4,512
50	42.62	53.70	9,098	4,345
150	18.1	14.35	8,654	3,521

TABLE II
PERFORMANCE STATISTICS OF THE PARALLEL-BOUNDED VARIANT FOR DIFFERENT VALUES OF β

Without any surprise, the parallel-full version is the most expensive and the perfect version is the most efficient. But we can see that the parallel-bounded version is quite competitive with respect to the standard sequential approach. However, this has to be taken with a grain of salt; it stands only in its *best* estimation of the upper bound by the companion algorithm.

We varied the value of β, the number of internal nodes traversed, before starting the search parallel algorithm. We see very clearly in Table II that whenever the value of β increases the performance of the search algorithm also increases.

However, we can also see that the improvement is not important for the first values of β. A significant number of inner nodes has to be visited in order to obtain a useful estimation of the upper bound (150 out of $\pm$400). Therefore, there is a trade-off to find between a better (sequential) estimation of the initial query radius and running the faster (potentially parallel) algorithm on a looser estimation.

Figure 4 shows the evolution of the value of r_q in the search algorithms. In the perfect case, the bound is known before hand, so it does not evolve during a search. Next, it is visible that in the case of the sequential approach the curve decreases very rapidly; this is due to the fact that entering leaf nodes provides intermediate results with very relevant approximations. Finally, with no surprise, the parallel approach does not benefit from this information, so the curve goes down more slowly since the algorithm can only take advantage of the distances computed along the visited ancestor inner nodes.

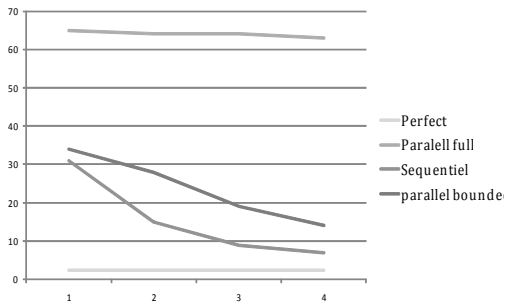

Fig. 4. Evolution of r_q in the search

V. CONCLUSION

In this paper, we have extended the known hierarchy of indexing methods in metric spaces with a variant of the ball-partitioning family, inspired by the recent MM-tree technique.

Our technique can be seen as a parametrisation of the MM-tree. An intermediate result, which is to be further investigated, is that the tree should be ternary rather than quadrary, the *ad hoc* α parameter being replaced by a three-third distribution of objects into regions I to III.

Next, we also investigated the usefulness of a parallel search. Although this variant cannot collect as many information as its sequential counterpart, we showed that the parallel version can be turned competitive thanks to a companion algorithm that estimates an upper-bound of the answer set maximal distance. Therefore, an actual parallelisation is certainly beneficial and is part of future work.

REFERENCES

[1] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Computing Surveys*, vol. 30, no. 2, pp. 170–231, 1998.

[2] C. Böhm, S. Berchtold, and D. A. Keim, "Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases," *ACM Computing Surveys*, vol. 33, no. 3, pp. 322–373, Sep. 2001.

[3] H. Samet, *Foundations of Multidimensional And Metric Data Structures*. Morgan-Kaufmann, Sep. 2006, 993 p.

[4] C. Faloutsos and K.-I. Lin, "Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia data," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, San Jose, CA, 1995, pp. 163–174.

[5] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," *proceedings of the 4th Annual In ACM-SIAM Symposium on Discrete Algorithms*, pp. 311–321, 1993.

[6] T. Bozkaya and M. Özsoyoglu, "Indexing large metric spaces for similarity search queries," *ACM Transactions on Database Systems*, vol. 24, pp. 361–404, Sep. 1999.

[7] J. K. Ulhmann, "Satisfying general proximity/similarity queries with metric trees," *Information Processing Letters*, vol. 40, pp. 175–179, 1991.

[8] S. Brin, "Near neighbor search in large metric spaces," *Proceedings VLDB Conference Switzerland, 1995*, pp. 574–584, 1995.

[9] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," *Proceedings of the 23rd VLDB International Conference*, pp. 426–435, 1997.

[10] C. Traina Jr, A. Traina, B. Seeger, and C. Faloutsos, "Slim-trees: High performance metric trees minimizing overlap between nodes," *International Conference on Extending Database Technology (EDBT)*, 2000.

[11] I. R. V. Pola, C. Traina, Jr, and A. J. M. Traina, "The mm-tree: A memory-based metric tree without overlap between nodes," *ADBIS 2007*, vol. LNCS 4690, pp. 157–171, 2007.

[12] C. C. M. Carélo, I. R. V. Pola, R. R. Ciferri, A. J. M. Traina, C. T. Jr., and C. D. de Aguiar Ciferri, "Slicing the metric space to provide quick indexing of complex data in the main memory," *Inf. Syst*, vol. 36, pp. 79–98, 2011.