Multimed Tools Appl
DOI 10.1007/s11042-011-0731-3

Adapting metric indexes for searching in multi-metric spaces

Benjamin Bustos · Sebastian Kreft · Tomáš Skopal

Abstract An important research issue in multimedia databases is the *retrieval of similar objects*. For most applications in multimedia databases, an exact search is not meaningful. Thus, much effort has been devoted to develop efficient and effective similarity search techniques. A recent approach that has been shown to improve the effectiveness of similarity search in multimedia databases resorts to the usage of combinations of metrics (i.e., a search on a multi-metric space). In this approach, the desirable contribution (weight) of each metric is chosen at query time. It follows that standard metric indexes cannot be directly used to improve the efficiency of dynamically weighted queries, because they assume that there is only one fixed distance function at indexing and query time. This paper presents a methodology for adapting metric indexes to multi-metric indexes, that is, to support similarity queries with dynamic combinations of metric functions. The adapted indexes are built with

This paper is partially funded by FONDECYT (Chile) Project 11070037 (B. Bustos and S. Kreft), CONICYT Master's Scholarship (S. Kreft) and by Czech Science Foundation Project 201/09/0683 (T. Skopal).

B. Bustos (✉) · S. Kreft
Department of Computer Science, University of Chile,
Av. Blanco Encalada 2120, Santiago, Chile
e-mail: bebustos@dcc.uchile.cl

S. Kreft
e-mail: skreft@dcc.uchile.cl

T. Skopal
Department of Software Engineering, Charles University in Prague,
Malostranské nám. 25, 118 00, Prague, Czech Republic
e-mail: skopal@ksi.mff.cuni.cz

Published online: 19 January 2011

a single distance function and store partial distances to estimate the dynamically weighed distances. We present two novel indexes for multimetric space indexing, which are the result of the application of the proposed methodology.

Keywords Information storage and retrieval · Content analysis and indexing methods

1 Introduction

Similarity search in multimedia database systems is becoming increasingly important. This is due to a rapidly growing amount of available multimedia data like images, audio files, video clips, 3D objects, time series, and text documents. As we see progress in the fields of acquisition, storage, and dissemination of various multimedia formats, the application of effective and efficient database management systems becomes indispensable to handle these formats. Application domains for multimedia databases include molecular biology, medicine, geographical information systems, Computer Aided Design/Computer Aided Manufacturing (CAD/CAM), virtual reality, and many others.

a) In medicine, the detection of similar organ deformations can be used for diagnostic purposes [17].
b) Biometric devices (e.g., fingerprint scanners) read a physical characteristic from an individual and then search in a database to verify if the individual is registered or not. The search cannot be exact, as the probability that two fingerprint scans, even from the same person, are exactly equal (bit-to-bit) is very low.
c) A 3D object database can be used to support CAD tools. For example, standard parts in a manufacturing company can be modeled as 3D objects. When a new product is designed, it can be composed of many small parts that fit together to form the product. If some of these parts are similar to one of the standard parts already designed, then the possible replacement of the original part with the standard part can lead to a reduction of production costs.
d) In text databases, a typical query consists of a set of keywords or a whole document. The search system looks in the database for documents that are relevant to the given keywords or that are similar to the query document. A certain tolerance on the search may be allowed in case, e.g., that some of the given keywords were mistyped or an optical character recognition (OCR) system was used to scan the documents (thus they may contain some misspelled words).
e) In bioinformatics, the protein classification or prediction requires complex similarity measuring that mimics the functional similarity of proteins [16, 19].

Many of these applications have in common that the objects of the database are modeled in a *metric space* [10, 20]. That is, it is possible to define a positive real-valued function δ among the objects, called *metric distance*, that satisfies the properties of *strict positiveness* ($\delta(x, y) \geq 0$ and $\delta(x, y) = 0 \Leftrightarrow x = y$), *symmetry* ($\delta(x, y) = \delta(y, x)$), and the *triangle inequality* ($\delta(x, z) \leq \delta(x, y) + \delta(y, z)$). The main motivation for using metric spaces is the fact that they are easily indexable by metric

access methods [10]. An important particular case of metric spaces are *vector spaces*, where the objects are tuples of D real values, i.e., they are vectors in $\mathbb{R}^D$.

A recent proposal to improve the effectiveness (i.e., the quality of the retrieved answer) of similarity search resorts to the use of *combinations of metrics* [6, 7]. Instead of using a single metric to compare two objects, the search system uses a *linear combination of metrics* to compute the (dis)similarity between two objects. The weights of the linear combination can be either *static* (they are a parameter of the search system) or *dynamic* (they are selected at query time). A static combination of metrics has the problem that usually not all metrics are well-suited for performing similarity search with all query objects. Moreover, a bad-suited metric may "spoil" the final result of the query. Thus, to further improve the effectiveness of the search system, *methods for dynamic combinations of metrics* have been proposed, where the query processor weighs the contribution of each metric *depending on the query object* (i.e., higher weights are assigned to the "good" metrics for that query object, and lower weights are assigned to the "bad metrics", according to some quality criteria). This means that, instead of a single metric, the system uses a *dynamic metric function* or *multimetric*. Thus, in *multi-metric spaces* a different metric function is used to perform each similarity query.

This paper presents a methodology to adapt metric indexes for supporting multi-metric similarity queries. We first describe the proposed methodology, explaining how this methodology can be used to adapt two different metric indexes, GNAT and List of Clusters. Next, we explain how previously proposed multi-metric indexes, like the M^3-tree, fit into this methodology. Finally, we show experimentally that the efficiency of the adapted indexes is very close to the lower bound of each method (the optimal achievable efficiency regarding to each index structure), which corresponds to the efficiency achieved when having one metric index per query object. Note that having one index per query is not a practical solution, since the construction cost of each index would be more costly than processing the query with a sequential scan. Also note that in this paper we only deal with the efficiency issues of similarity search in multi-metric spaces. For a discussion on the effectiveness of the multi-metric approach see Bustos et al. [6, 7].

2 Similarity search in metric and multi-metric spaces

Let $(\mathbb{X}, \delta)$ be a metric space and let $\mathbb{U} \subseteq \mathbb{X}$ be a set of objects (i.e., an instance of a database). There are two typical similarity queries in metric spaces:

- *Range query*. A range query (q, r), $q \in \mathbb{X}$, $r \in \mathbb{R}^+$, reports all database objects that are within a tolerance distance r to q, that is $(q, r) = \{u \in \mathbb{U},\ \delta(u, q) \leq r\}$. (q, r) is called the *query ball*.
- *k nearest neighbors query (k-NN)*. It reports the k objects from $\mathbb{U}$ closest to q. That is, it returns the set $\mathbb{C} \subseteq \mathbb{U}$ such that $|\mathbb{C}| = k$ and $\forall x \in \mathbb{C},\ y \in \mathbb{U} - \mathbb{C}$, $\delta(x, q) \leq \delta(y, q)$.

Usually, a single metric function is used to compute the similarity between two objects in the metric space. However, a recent trend to improve the effectiveness of the similarity search resorts to use *several metric functions*. The (dis)similarity function is computed as a linear combination of some selected metrics.

Definition 1 Multi-metric space

Let $\mathcal{M} = \{(\mathbb{X}_i, \delta_i),\ 1 \leq i \leq n\}$ be a set of metric spaces. We define the corresponding *Multi-metric space* as the pair $\left(\prod_{i=1}^{n} \mathbb{X}_i, \Delta_{\mathbb{W}}\right)$, where $\Delta_{\mathbb{W}}$ is a linear multimetric distance, which means

$$\Delta_{\mathbb{W}}(x, y) = \sum_{i=1}^{n} w_i \delta_i(x_i, y_i) \tag{1}$$

In the above definition, the vector of weights $\mathbb{W} = \langle w_i \rangle$ is not fixed, and it is a parameter of Δ. When $\forall i\, w_i \in [0, 1] \wedge \exists i\, w_i > 0$, $\Delta_{\mathbb{W}}$ is also a metric.

If the weights of the combination are fixed, the multi-metric space becomes an ordinary metric space and one could use any standard metric index structure. In our framework, however, the weights are *dynamic*—computed at query time—and therefore the metric distance function is dynamic and depends on the query objects. This has been shown to provide better effectiveness results [6, 7]. Thus, the problem is to develop a metric index structure that returns the correct answer to the similarity query, even if the *query distance function* is not the same as the distance function used to build the index (*index distance function*). The naive (but optimal in terms of search cost) solution would be to have an index structure for each "fixed multi-metric", but this would not be practical because it would imply to build an index for each query object.

Note that for the particular case of vector spaces, Spatial Access Methods (SAMs) can be used to index the data collection (good surveys on this topic are Berchtold et al. [1] and Samet [18]). However, the focus of this paper is the case of combining general metric spaces, thus we will only present examples of metric access methods adapted for multi-metric spaces.

In the following, we describe a few metric access methods (MAMs). It is worth noticing that the methods here explained are the relevant ones to our work and that this section is by no means a survey on MAM indexing techniques. We refer the reader to Chávez et al. [10], Samet [18], or Zezula et al. [20] for excellent surveys on this topic.

2.1 GNAT

Geometric Nearest-Neighbor Access Tree or *GNAT* [2] tries to represent the "intrinsic geometry" of the space using a hierarchic structure based on Dirichlet domains (or Voronoi-like partitioning).

Construction Given ℓ objects $p_1, \ldots, p_\ell$ (the *split points*) from the dataset ($\mathbb{U}$), the Dirichlet domain (Voronoi zone) corresponding to p_i is composed of all the objects that are closer to p_i than any other p_j. In the root of GNAT, the dataset is partitioned in the different Dirichlet domains corresponding to each split point. Then, the maximum and minimum distance from each split point to each zone is computed. Figure 2 shows the split points p and s, the Dirichlet domain D_s corresponding to s and the minimum ($\min_\delta(p, D_s)$) and maximum ($\max_\delta(p, D_s)$) distances of the range defined by p and D_s. Finally, each domain is recursively partitioned in the same way. The partition of the dataset into Dirichlet domains is used just for constructing the index, and not for querying it. Figure 1 shows the pseudocode of this construction algorithm.

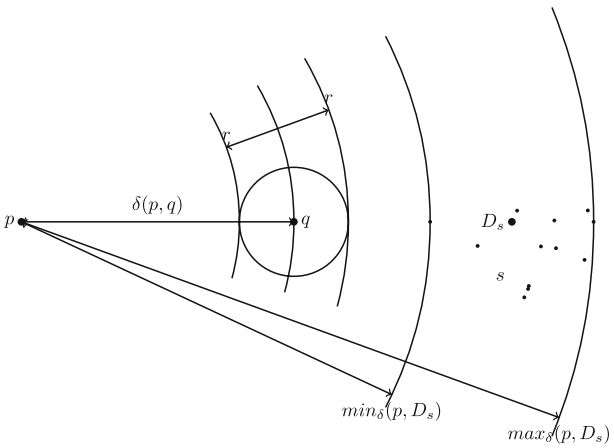
Fig. 1 GNAT build algorithm

```
1  function Build(𝕌)
2  let P = p_1, ..., p_ℓ, set of ℓ split points
3  for all x ∈ 𝕌 − P do
4      associate x with closest split point
5  let D_{p_i} = associated set of p_i
6  for all (p_i, p_j) ∈ P × P do
7      range(p_i, D_{p_j}) = [min_δ(p_i, D_{p_j}), max_δ(p_i, D_{p_j})]
8  for all p_i ∈ P do
9      Build(D_{p_i})
10 end function
```

Range query The range query algorithm (see Fig. 3) computes the distances from the query object to each one of the split points. With these distances, the search algorithm verifies whether the query ball overlaps a zone. If not, the corresponding branch of the tree is pruned. The pruning is performed in line 8 of Fig. 3, where the condition holds because of the triangular inequality (see Figs. 2 and 3).

k-nearest neighbors To perform a k-NN search, we developed an algorithm using the technique presented by Hjaltason and Samet [15]. As far as we know, this algorithm has not been presented before in the literature. The algorithm needs an estimation of the distance from the query object to each of the Voronoi zones. To get this estimation, we use the ranges computed for each pair of split points. Indeed, to estimate the distance between the zone defined by the *split point* p and the query object q we use the ranges of each *split point* to D_p. Given another *split point* s, q may be located: (1) inside zone D_p, in which case the distance is 0; (2) outside the ring defined by s and D_p; (3) between s and D_p. Figure 4 shows Cases (2) and (3). A lower bound distance for Case (2) is $\delta(s, q) - \max_\delta(s, D_p)$, and for Case (3) is $\min_\delta(s, D_p) - \delta(s, q)$. Finally, the estimation of the distance is set as the maximum of all the computed estimations with each of the split points. Figure 5 shows the proposed k-NN algorithm.

Fig. 2 Branch pruning in GNAT using ranges



```
1  function Search(N, q, r)
2  let P = split points(N)
3  for all p ∈ P do
4     compute δ(p, q)
5     if δ(p, q) ≤ r then
6        add p to query result
7     for all s ∈ P do
8        if [δ(q, p) − r, δ(q, p) + r] ∩ range(p, D_s) = ∅ then
9           remove s from P
10 for all p ∈ P do
11    Search(D_p, q, r)
12 end function
```

Fig. 3 GNAT range query algorithm

2.2 List of clusters

List of clusters [9] is a metric index based on compact partitions. It shows good performance in spaces with high dimensionality, and it is also well suited for secondary memory.

Construction To build the index, the algorithm selects an object c from the dataset and a radius r_c. Then, it groups all objects that are within distance r_c from c are in I_c, and the remaining objects in E_c. The construction process is continued recursively in E_c until all objects are indexed. Figure 6 shows the pseudocode of this algorithm.

Figure 7 shows an example of how the structure is organized in $\mathbb{R}^2$ and how the structure can be seen as a list. This figure also shows a fundamental characteristic of List of Clusters: an object belongs necessarily to the first partition (or bucket) that can hold it. That is, if two or more partitions overlap, the objects in the intersection will belong to the partition that was created first. For example, in Fig. 7 u belongs to c_1, even though it is also located in the zone of c_2.

Range query Figure 8 depicts the range query algorithm. The algorithm computes the distance from the query object to the center c of the first cluster, and c is added

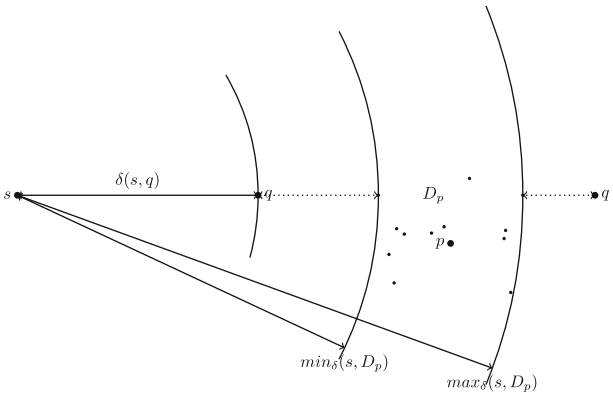


Fig. 4 Distance from an object to a zone in GNAT

```
1  function Knn(N, q, k)
2  let Q = priorityQueue()
3  enqueue((N, 0), Q)
4  while Q ≠ ∅ do
5     let object = dequeue(Q)
6     if object is spatial object then
7        report object
8        if size(query result) = k then
9           return
10    else
11       let P = split points(object)
12       for all p ∈ P do
13          compute δ(p, q)
14          enqueue((p, δ(p, q)), Q)
15          let dist = max_{s∈P}{δ(s, q) − max_δ(s, D_p), min_δ(s, D_p) − δ(s, q), 0}
16          enqueue((D_p, dist),Q)
17 end function
```

Fig. 5 GNAT k-NN algorithm

Fig. 6 List of clusters build algorithm

```
1  function Build(𝕌)
2  if 𝕌 = ∅ then
3     return empty list
4  Select c ∈ 𝕌
5  Select radius r_c
6  I_c ← {u ∈ 𝕌 − {c}, δ(c, u) ≤ r_c}
7  E_c ← 𝕌 − I_c
8  return (c, r_c, I_c) :Build(E_c)
9  end function
```

Fig. 7 Example of list of cluster in $\mathbb{R}^2$

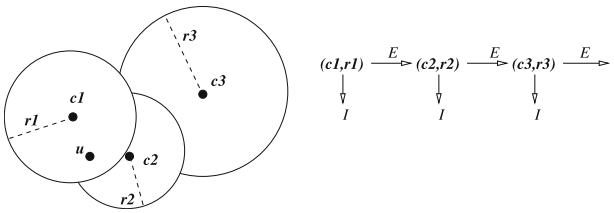


Fig. 8 List of clusters range query algorithm

```
1  function Search(L, q, r)
2  let (c, r_c, I) : E = L
3  compute δ(c, q)
4  if δ(c, q) ≤ r then
5     add c to query result
6  if δ(c, q) ≤ r_c + r then
7     search exhaustively in I
8  if δ(c, q) > r_c − r then
9     Search(E, q, r)
10 end function
```

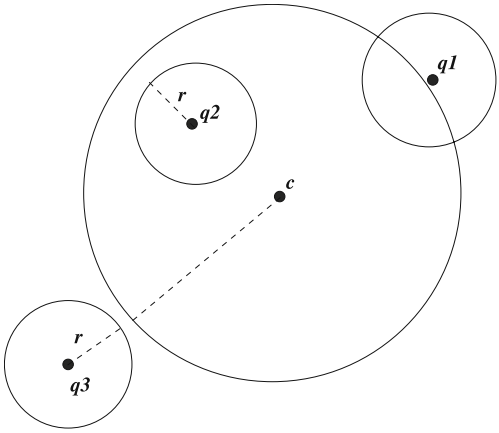


Fig. 9 List of clusters range query search cases

to the result if it intersects the query ball. Then, three different cases may occur, which are presented in Fig. 9. The first case (q_1) is when the query ball intersects the bucket I_c. In this case, the algorithms needs to search the whole bucket. The second case (q_2) is when the query ball is completely contained in I_c. In this case, the search can be pruned after searching I_c. The last case (q_3) is when the query ball does not intersect the bucket, thus I_c can be pruned from the search. In the first and last cases the search must continue recursively in E_c.

k-nearest neighbors The idea behind this algorithm[1] is to first check the buckets with higher possibilities to hold the nearest neighbors. This is done by keeping a list with the k nearest objects found so far. This allows the algorithm to skip some of the buckets while searching those that intersects with the query object. The algorithm is presented in Fig. 10. Function *searchBucketKnn* searches exhaustively a bucket, updating the query result list with the nearest objects found and storing at most k objects in the list.

2.3 Pivot-based indexing

There are many similarity search indexes based on *pivots* [10], which are selected objects from the dataset. Here, we describe the canonical index structure based on pivots and the algorithm for performing range queries using this index.

Given a range query (q, r) and a set of k pivots $\mathbb{P} = \{p_1, \ldots, p_k\}$, $p_i \in \mathbb{U}$, by the triangle inequality it follows that $\delta(p_i, x) \leq \delta(p_i, q) + \delta(q, x)$, and also that $\delta(p_i, q) \leq \delta(p_i, x) + \delta(x, q)$ for any $x \in \mathbb{X}$. From both inequalities, it follows that a lower bound on $\delta(q, x)$ is $\delta(q, x) \geq |\delta(p_i, x) - \delta(p_i, q)|$. The objects $u \in \mathbb{U}$ of interest are those that satisfy $\delta(q, u) \leq r$, so all the objects that satisfy the *exclusion condition* (2) can be discarded, without actually evaluating $\delta(q, u)$.

$$|\delta(p_i, u) - \delta(p_i, q)| > r \text{ for some pivot } p_i. \tag{2}$$

[1]This algorithm was taken from the SISAP library http://www.sisap.org

```
1  function Knn(L, q, k)
2  let (c, r_c, I) : E = L
3  compute δ(c, q)
4  if δ(c, q) ≤ r_c then
5     searchBucketKnn(I, q, k)
6     if size(query result) < k or δ(c, q) + max_δ(query result) ≥ r_c then
7        Knn(E, q, k)
8  else
9     Knn(E, q, k)
10    if size(query result) < k or δ(c, q) − max_δ(query result) ≤ r_c then
11       searchBucketKnn(I, q, k)
12 end function
```

Fig. 10 List of clusters k-NN algorithm

The canonical pivot-based index consists of the kn precomputed distances $\delta(p_i, u)$ between every pivot and every object of the database. Therefore, at query time it is only necessary to compute the k distances between the pivots and the query q, $\delta(p_i, q)$, in order to apply the exclusion condition (2). The list of candidate objects $\{u_1, \ldots, u_m\} \subseteq \mathbb{U}$ that cannot be discarded with the exclusion condition (2) must be directly checked against the query object.

The way how pivots are selected affects the efficiency of the search algorithms [4].

2.4 M-Tree

The *M-tree* [12] is a dynamic (meaning easily updatable) index structure that provides good performance in secondary memory. The M-tree is a hierarchical index, where some of the data objects are selected as centers (local pivots) of regions and the rest of the objects are assigned to suitable regions. This builds up a balanced and compact hierarchy of data regions. Each region (branch of the tree) is indexed recursively. The data are stored in the leaves of the M-tree, where each leaf contains *ground entries* ($grnd(x),\ x \in \mathbb{U}$). The internal nodes store *routing entries* ($rout(y),\ y \in \mathbb{U}$).

Starting at the root level, a new object x is recursively inserted into the best subtree $T(y)$, which is defined as the one where the *covering radius* r_y must increase the least to cover the new object. In case of ties, the subtree whose center is closest to x is selected. The insertion algorithm proceeds recursively until a leaf is reached and x is inserted into that leaf, storing at each level the distance to the routing object of its parent node (so-called *to-parent distance*). Node overflows are managed in a similar way as in the B-tree. If an insertion produces an overflow, two objects from the node are selected as new centers, the node is split, and the two new centers are promoted to the parent node. If the parent node overflows, the same split procedure is applied. If the root overflows, it is split and a new root is created. Thus, the M-tree is a balanced tree (see Fig. 11).

To correctly bound the data in the respective subtree $T(R)$, the routing entry $rout(R)$ must satisfy the *nesting condition*: $\forall O_i \in T(R), r_R \geq \delta(R, O_i)$.

Range queries are implemented by traversing the tree, starting from the root. The nodes which parent region (described by the routing entry) is overlapped by the query ball are accessed (this requires a distance computation). As each node in

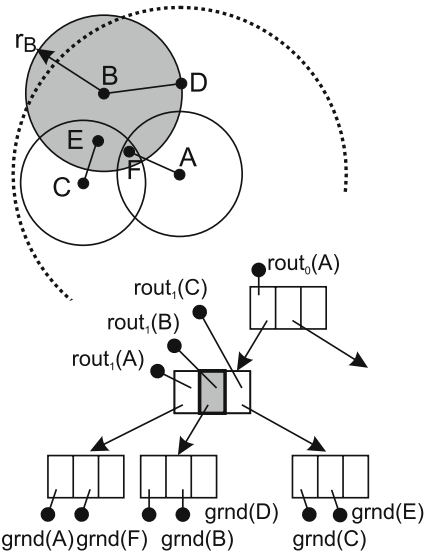


Fig. 11 Example of an M-tree

the tree (except for the root) contains the distances from the routing/ground entries to the center of its parent node (the to-parent distances), some of the non-relevant branches can be further filtered out, without the need of a distance computation, thus avoiding the "more expensive" basic overlap check.

3 Proposed methodology

In this section, we propose a standard methodology for adapting metric indexes to multi-metric indexes. The main idea of our proposed methodology is to build the multi-metric index with a fixed multimetric, and then estimate the distances when the weights are defined at query time. When estimating the distances, we need also to assure that any index-specific invariant is preserved when changing the weights. To estimate the distances, we categorize them in three different types and show how to bound them. We also show how to change the conditionals and statements used in query algorithms to ensure that they always return the correct answer.

3.1 Index-specific invariant

An *index-specific invariant* is any qualitative property of the indexing model which must be preserved under all circumstances, and that is later relevant for the correctness of the query results. For example, the nesting condition defined for M-tree routing entries or the property of List of Clusters (by which each object belong to the first cluster that can hold it) are such an invariant. If the index-specific invariant is just a definition of a distance stored by the structure, we will call it *implicit*, because they are implicitly adapted when changing to multimetrics. Thus, the first presented example will be an implicit-index-specific invariant and the second one an index-specific invariant.

Note that a property that balances the index or improves compactness/efficiency is not considered as an index-specific invariant. The invariant of GNAT by which each object is associated with the closest split point, is an example of an invariant that is not an index-specific invariant.

3.2 Distance bounds

The indexing structures for metric spaces, either pivot based or compact partitioning algorithms [10], store distances between objects. These distances are used later to test if an object belongs to the query ball.

The stored distances can be classified into the following types (further in text denoted as Type 1, 2, 3):

1. Distance between two objects, $d = \delta(x, y)\ x, y \in \mathbb{U}$
2. Maximal distance from an object to a set, $d = \max_{x \in C \subseteq \mathbb{U}} \delta(x, y)\ \ y \in \mathbb{U}$
3. Minimal distance from an object to a set, $d = \min_{x \in C \subseteq \mathbb{U}} \delta(x, y)\ \ y \in \mathbb{U}$

It is important to note that Type 1 is a particular case of both Types 2 and 3.

When working with multi-metric spaces, the distance function $\Delta_{\mathbb{W}}$ is not known when building the index; it is just known at query time—at the moment when the weights $\mathbb{W}$ are defined. For that reason, a multi-metric index must estimate the distances. If the distance is of Type 1, it can be computed exactly. To do this, it is necessary to store the components of the distance between each pair of objects, which later are weighed and summed into $\Delta_{\mathbb{W}}$. If the distance is of Type 2 or 3, it cannot be computed exactly. Later, we will show two lemmas that will allow us to estimate the distances.

Let $\mathbb{X}$ be a multi-metric space, $C \subseteq \mathbb{X}$ a set, $y \in \mathbb{X}$ an object of the space, and let

$$r^{\mathbb{W}}_{\max} = \max_{x \in C} \Delta_{\mathbb{W}}(x, y) = \max_{x \in C} \sum w_i \delta_i(x_i, y_i) \tag{3}$$

$$r^{\mathbb{W}}_{\min} = \min_{x \in C} \Delta_{\mathbb{W}}(x, y) = \min_{x \in C} \sum w_i \delta_i(x_i, y_i) \tag{4}$$

where $\mathbb{W}$ is a vector of weights, whose values are in the range [0.0, 1.0] and at least one weight is not zero. From now on we are using also the notation $\mathbb{W} = 1.0$, which means that all weights are equal to 1.0. Also, the subscript or upperscript lb stands for *lower bound* and ub for *upper bound*.

In the following, we propose two complementary constructions of lower and upper bounds.

Lemma 1 *Bound based on weights*

$$r^{\mathbb{W}}_{\max} \leq r^{\mathbb{W}}_{ub\,1} = \max w_i\ r^{1.0}_{\max} \tag{5}$$

$$r^{\mathbb{W}}_{\min} \geq r^{\mathbb{W}}_{lb\,1} = \min w_i\ r^{1.0}_{\min} \tag{6}$$

Proof

$$\begin{aligned} r^{\mathbb{W}}_{\max} = \max_{x \in C} \sum w_i \delta_i(x_i, y_i) &\leq \max_{x \in C} \sum \max w_i \ \delta_i(x_i, y_i) \\ &= \max w_i \ \max_{x \in C} \sum \delta_i(x_i, y_i) \\ &= \max w_i \ r^{1.0}_{\max} \end{aligned}$$

$$\begin{aligned} r^{\mathbb{W}}_{\min} = \min_{x \in C} \sum w_i \delta_i(x_i, y_i) &\geq \min_{x \in C} \sum \min w_i \ \delta_i(x_i, y_i) \\ &= \min w_i \ \min_{x \in C} \sum \delta_i(x_i, y_i) \\ &= \min w_i \ r^{1.0}_{\min} \end{aligned}$$

□

Lemma 2 *Bound based on distance components*

$$r^{\mathbb{W}}_{\max} \leq r^{\mathbb{W}}_{ub2} = \sum w_i \max_{x \in C} \delta_i(x_i, y_i) \tag{7}$$

$$r^{\mathbb{W}}_{\min} \geq r^{\mathbb{W}}_{lb2} = \sum w_i \min_{x \in C} \delta_i(x_i, y_i) \tag{8}$$

Proof

$$\begin{aligned} r^{\mathbb{W}}_{\max} = \max_{z \in C} \sum w_i \delta_i(z_i, y_i) &\leq \max_{z \in C} \sum w_i \max_{x \in C} \delta_i(x_i, y_i) \\ &= \sum w_i \max_{x \in C} \delta_i(x_i, y_i) \end{aligned}$$

$$\begin{aligned} r^{\mathbb{W}}_{\min} = \min_{z \in C} \sum w_i \delta_i(z_i, y_i) &\geq \min_{z \in C} \sum w_i \min_{x \in C} \delta_i(x_i, y_i) \\ &= \sum w_i \min_{x \in C} \delta_i(x_i, y_i) \end{aligned}$$

□

In the above proof $\max_{z \in C} \sum w_i \max_{x \in C} \delta_i(x_i, y_i) = \sum w_i \max_{x \in C} \delta_i(x_i, y_i)$, because the term inside the sum is constant respect to z.

Corollary 1 *Using Lemmas* 1 *and* 2 *we have*

$$r^{\mathbb{W}}_{\max} \leq r^{\mathbb{W}}_{ub} = \min\left(r^{\mathbb{W}}_{ub1}, r^{\mathbb{W}}_{ub2}\right) \tag{9}$$

$$r^{\mathbb{W}}_{\min} \geq r^{\mathbb{W}}_{lb} = \max\left(r^{\mathbb{W}}_{lb1}, r^{\mathbb{W}}_{lb2}\right) \tag{10}$$

In conclusion, to estimate a distance of Type 2 or 3 it is necessary to store the value of the distance for the metric $\Delta_{1.0}$ and also the maximum or minimum components of the distance. Finally, we note that Lemma 1 has been already considered in Ciaccia and Patella [11] for answering queries with user-defined distance functions. While this approach may be used to answer queries in multi-metric spaces, it requires the definition of lower and upper bounding distances of the query distances. Our approach does not require to define such bounding distances (see Section 3.3), as it relies on preserving the index-specific invariants of the original metric index.

3.3 Adaptation process

The steps needed to convert a metric index into a multi-metric index are the following:

1) Identify which types of distances are involved in the index.
2) Analyze if structure-specific index invariants are preserved in case the weights are changed.
   At this step we do not deal with implicit-index-specific invariants because those are implicitly adapted in Step 4. At this point, one has to check if, given an index built with a fixed multimetric (i.e., weights are fixed), the index preserves the specific invariants of the structure when changing the weights. This step is crucial to ensure that the query results are correct, because if the index-specific invariant does not hold, the result could contain irrelevant objects or lack some relevant ones.
3) Modify the construction of the structure.
   The index is built with the multimetric $\Delta_{1.0}$, and each time a distance is stored we also store the distance by components. At this step, it could be necessary to store new distances to preserve the structure invariants. See Section 4.2.1 to see an example of how a new distance is added to an index-specific invariant to preserve it when changing weights.
4) Modify query algorithms using bounds. Here the conditions and statements of the query algorithms of the metric index are adapted. The only conditions and statements that must be changed are those which use distances.

   – Type 1 distance: the distance is computed exactly, preserving the condition or the statement.
   – Type 2 distance: in this case, the way how a condition changes depends on the action taken when the condition is true.
     There are two basic actions that can be done. The first is to keep searching in the structure, and the second is to prune the search. In the first case, we need to ensure that all the relevant "branches" are going to be visited, thus we must assure that every time the condition is held with the new weights, it is also held with the estimated distance (i.e., $cond(d) \Rightarrow cond(d_{ub})$). In the second case, we need to guarantee that the search is not being ended without reason, thus we must have that every time the condition is held with the estimated distance, it is also held for the exact distance (i.e., $cond(d_{ub}) \Rightarrow cond(d)$).
     Generally, distances of Type 2 with a search action are of the form $d \geq a$, and those of Type 2 with a prune action are of the form $d \leq a$. These conditions satisfy the properties stated above to ensure the correctness of the result. This is true because $d \geq a \Rightarrow d_{ub} \geq a$ (search) and $d_{ub} \leq a \Rightarrow d \leq a$ (prune). If these are conditions of the algorithm, one can use directly the bounds obtained in the Corollary 1. That is, the search condition is changed to $d_{ub} \geq a$ and the pruning condition to $d_{ub} \leq a$. When the conditions in the algorithm are not like the ones stated above one should find a custom way to adapt these particular conditions. However, in all the four methods we studied the conditions are like those previously explained, thus they can be adapted using this method.

Statements generally use distances to estimate other distances (see Section 2.1 for an example). In this case, one must check if the distance being estimated is a lower or upper bound, and then verify if the bounds of the involved distances (lower bound for a Type 2 distance) maintain the bound being estimated (see Section 4.1.3). As a matter of fact, we only found statements in the k-NN algorithms of GNAT and M-Tree, and those were estimating distances.

– Type 3 distance: it can be handled similarly as the case for distances of Type 2.

4 Multi-metric indexes

In this section, we describe how we adapted the metric indexes GNAT and List of Clusters, and how the previously proposed multi-metric indexes [3, 5] can be seen as a direct application of our proposed methodology.

4.1 Multi-metric GNAT (MMGNAT)

The distances stored by GNAT are the ranges of each split point to the zones. The range is of the form $[\min_d(p, D_q), \max_d(p, D_q)]$, where the minimum distance is of Type 2 and the maximum distance of Type 3. It is important to realize that in the construction of GNAT each point is paired with the closest split point, but this condition is never used in the search algorithms. For this reason, GNAT has no index-specific invariant, thus it can be modified directly to index multi-metric spaces.

4.1.1 Construction

Because the structure has no index-specific invariants, the new construction algorithm is quite similar to the original one. The construction is done with the multimetric $\Delta_{1.0}$. Each time a maximum distance of a range is stored, the maximum distance of each component to the zone is also stored. The same is done for the minimum distance of the range. This is because the involved distances are of Type 2 and 3.

4.1.2 Range query

It can be seen in Fig. 3 that the only condition that depends on the stored distances is the one in line 8. This condition is equivalent to:

if $\max_d(p, D_q) < dist(x, p) - r$ **or** $\min_d(p, D_q) > dist(x, p) + r$ **then**
  remove q from P

In the condition shown above, the maximum distances are involved in conditions of type $d < a$, the minimum distances are involved in conditions of type $d > a$, and the action performed corresponds to a prune. Thus, we can replace directly the distances by the bounds, according to the reasoning explained in Section 3.3. Figure 12 shows the algorithm. In line 8, we use Corollary 1 to estimate the maximum and minimum distance of the range.

```
1  function Search(N, q, r)
2  let P = split points(N)
3  for all p ∈ P do
4     compute Δ_W(p, q)
5     if Δ_W(p, q) ≤ r then
6        add p to query result
7        for all s ∈ P do
8           let range_e(p, D_s) = estimate range(p, D_s)
9           if [Δ_W(q, p) − r, Δ_W(q, p) + r] ∩ range_e(p, D_s) = ∅ then
10             remove s from P
11 for all p ∈ P do
12    Search(D_p, q, r)
13 end function
```

Fig. 12 MMGNAT range query algorithm

4.1.3 k-nearest neighbors

The stored distances are only used to estimate the distance from the query object to a zone (see line 15 of Fig. 5). If the distances of a range are replaced by the bounds, we only get an estimation of the real range. Thus, the estimation of the distance from the query object to a zone may produce a different ordering, but this can only affect the efficiency of the pruning, not the effectiveness of the search. The resulting algorithm is shown in Fig. 13.

4.2 Multi-metric list of clusters (MMLCluster)

List of Clusters stores the radius of each cluster. This corresponds to a distance of Type 3, because this distance is computed as the maximum distance from the center of the cluster to each object of the cluster.

4.2.1 Index-specific invariant

The invariant of List of Clusters ensures that if an object could be in two different clusters, it will belong necessarily to the one that was created first. This allows the index to prune earlier the range query search, decreasing the number of distance computations. If the structure is built with a fixed multimetric, let say $\Delta_{1.0}$, and then the weights are changed, the invariant (with the new weights) will not necessarily hold in the previously built structure, because the objects may now belong to a new cluster. Figure 14 shows how an object could belong to a different cluster after changing weights. In the example, the multi-metric space combines two vector spaces of dimension 1. The left figure shows the cluster defined by q_1 when the multimetric is $\Delta_{1.0}$ (note that q_4 does not belong to the cluster defined by q_1), and the right figure shows the new cluster when the multimetric is changed to $\Delta_{\{0.0,1.0\}}$ (now q_4 do belong to the cluster defined by q_1).

In the original List of Clusters, the pruning is made every time the query ball is completely contained in any cluster. This pruning can be made because the invariant assures that all the objects that remain in the list cannot belong to the current cluster. To maintain this invariant when the weights are changed, we reinterpret the way

```
1  function Knn(N, q, k)
2  let Q = priorityQueue()
3  enqueue((N, 0), Q)
4  while Q ≠ ∅ do
5     let object = dequeue(Q)
6     if object is spatial object then
7        report object
8        if size(query result) = k then
9           return
10    else
11       let P = split points(object)
12       for all p ∈ P do
13          compute Δ_W(p, q)
14          enqueue((p, Δ_W(p, q)), Q)
15       for all p ∈ P do
16          let dist = 0
17          for all s ∈ P do
18             let range_e(p, D_q) = estimate range(p, D_q)
19             if min_lb(s, D_p) − Δ_W(s, q) > dist then
20                dist = min_lb(s, D_p) − Δ_W(s, q)
21             if Δ_W(s, q) − max_ub(s, D_p) > dist then
22                dist = Δ_W(s, q) − max_ub(s, D_p)
23          enqueue((D_p, dist), Q)
24 end function
```

Fig. 13 MMGNAT k-NN algorithm

the pruning is performed. If we introduce a new radius r_{external}, where all objects remaining in the list are at distance $d \geq r_{\text{external}}$ from the center of the cluster, the pruning can be stated in a way that the invariant is preserved when changing the weights. Now, the pruning will be made every time the query ball is completely contained in the ball defined by the center of the cluster and r_{external}. This way of pruning does not contradict with the original invariant, because when the weights are fixed the radius of the cluster is lower or equal to r_{external}. The added distance is of Type 2, so we can get a lower bound of it. Thus, we can keep the invariant when changing the weights. This radius will depend on the weights being used and it is given by the following formula:

$$r^{\mathbb{W}}_{\text{external}} = \min_{x \in \mathbb{U}-C} \Delta_{\mathbb{W}}(c, x) \tag{11}$$

where c is the center of the last created cluster, the one for which we are computing r_{external}, and C is the set of all objects added to any cluster. It is important to realize

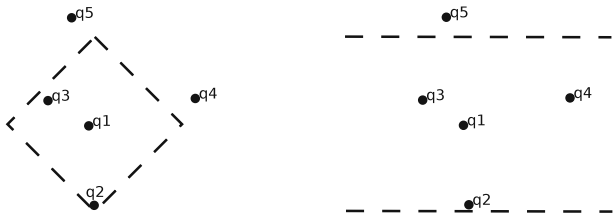


Fig. 14 Objects belong to different clusters when changing weights

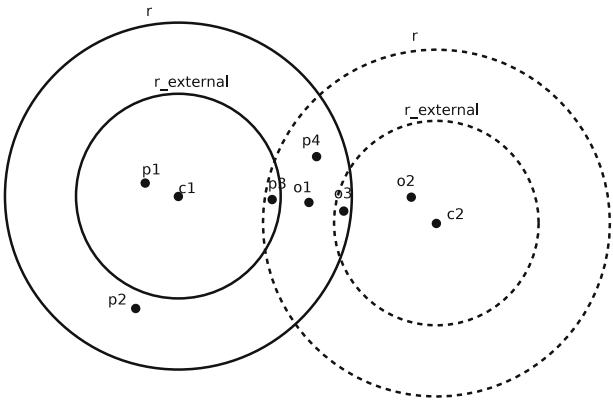


Fig. 15 New distances in MMLCluster

that C is defined when building the index with weights $\Delta_{1.0}$ and that it does not change when the weights are changed.

Figure 15 depicts the structure of the Multi-Metric List of Clusters. The objects o_1 to o_4 belong to the cluster defined by c_1 and the objects o_5 to o_7 to the cluster defined by c_2. Notice that objects o_5 and o_7 are in cluster c_2 even though its distance to c_1 is lower than r. This is because its distance is greater than r_{external}, so the invariant does not ensure that they belong to the first created cluster.

Figure 16 presents the criterion for searching or pruning a range query search. For the query ball q_1, it is not necessary to search in the current cluster. For q_2, as the query ball is completely contained in the ball defined by r_{external}, the search ends after searching in the cluster, and for q_3 and q_4 it is necessary to search the cluster and keep searching in the rest of the list. It is important to notice that for q_4 in the original List of Clusters the search would end because the query ball is completely contained in the cluster, but the new invariant does not allow to prune the search at this point.

Lemma 3 *The new invariant is preserved when changing weights.*

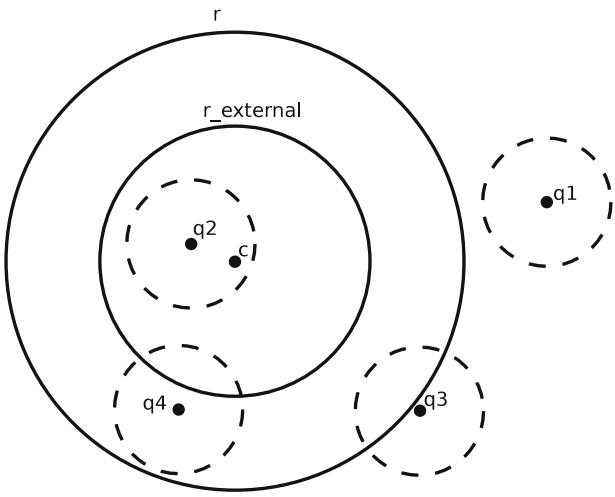


Fig. 16 Range query in MMLCluster

Let C be the set of all objects already added to any previously created cluster and c the last cluster created, then the invariant is stated as follows

$$p \notin C \Rightarrow \Delta_{\mathbb{W}}(p, c) \geq r^{\mathbb{W}}_{\text{external}} \tag{12}$$

that means that every object not yet added to the cluster is at distance greater or equal than $r^{\mathbb{W}}_{\text{external}}$ *from c.*

Proof This invariant holds trivially for all sets of weights, because by definition, $r^{\mathbb{W}}_{\text{external}} = \min_{x \in \mathbb{U}-C} \Delta_{\mathbb{W}}(c, x)$ and as $p \notin C$, then $r^{\mathbb{W}}_{\text{external}} \leq \Delta_{\mathbb{W}}(c, p)$. □

4.2.2 Construction

To build the new index, the space is indexed using the multimetric $\Delta_{1.0}$. Every time a cluster is created, we store the radius and the maximum distance by component from the center to each object in the bucket. We need also to store r_{external} and the minimum distance by component from the center to each object not yet added to the list.

4.2.3 Range query

The range query search should use now the new invariant and distances. That is, the conditions involved are of the form $r_c \geq a$ and $r_{\text{external}} \leq a$ (cf. Section 3.3, Step 4). Also, both conditions have an action of keep searching, so we can directly replace these distances by their bounds: lower bound for r_{external} and upper bound for r_c. The new range query search algorithm is depicted in Fig. 17.

4.2.4 k-nearest neighbors

The conditions in lines 6 and 10 in the *k*-NN algorithm of List of Clusters (see Fig. 10) are the same conditions found in lines 6 and 8 of the range query algorithm of List of Clusters (see Fig. 8), thus the transformations are the same as the ones made for Range Query search. The *k*-NN algorithm is shown in Fig. 18.

4.3 M^3-Tree

The M^3-Tree [3] results of applying the proposed methodology to the M-Tree. Two distances are involved in the M-Tree: the to-parent distance, which is a distance of Type 1; and the maximum distance from a node to each one of its descendants, which

```
1  function Search(L, q, r)
2  let (c, r_c, r_external, I) : E = L
3  compute Δ_W(c, q)
4  let r_c^ub = estimate r_c
5  let r_e^lb = estimate r_external
6  if Δ_W(c, q) ≤ r then
7     add c to result list
8  if Δ_W(c, q) ≤ r_c^ub + r then
9     search exhaustively in I
10 if Δ_W(c, q) > r_e^lb − r then
11    Search(E, q, r)
12 end function
```

Fig. 17 MMLCluster range query algorithm

```
1  function Knn(L, q, k)
2  let (c, r_c, r_external, I) : E = L
3  compute Δ_W(c, q)
4  if Δ_W(c, q) ≤ r_c then
5     searchBucketKnn(I, q, k)
6     let r_e^lb = estimate r_external
7     if size(query result) < k or Δ_W(c, q) + max_ΔW(query result) ≥ r_e^lb then
8        Knn(E, q, k)
9  else
10    Knn(E, q, k)
11    let r_c^ub = estimate cluster.r_c
12    if size(query result) < k or Δ_W(c, q) − max_ΔW(query result) ≤ r_c^ub then
13       searchBucketKnn(I, q, k)
14 end function
```

Fig. 18 MMLclusters *k*-NN algorithm

is of Type 2. When creating the M^3-tree, the space is indexed with the multimetric $\Delta_{1.0}$, storing distances and also the components for each distance. No further changes are needed because there are only implicit index-specific invariants.

Adapting the search algorithms is similar to the cases of GNAT and List of Clusters. Figure 19 depicts the range query algorithm for the adapted M-tree. As the to-parent distance is of Type 1, it can be computed exactly using the previously stored components (see lines 6 and 14) and the conditions need no further changes.

```
1  function Search(N, q, r)
2  // if N is root then Δ_x(R, P) = Δ_x(P, q) = 0
3  let P be the parent routing object of N
4  if N is not a leaf then
5     for all rout(R) in N do
6        compute Δ_W(R, P) with previously computed components
         (weigh components)
7        let r_e(R) be the estimated covering radius of R
8        if |Δ_W(P, q) − Δ_W(R, P)| ≤ r_e(R) + r then
9           compute Δ_W(R, q)
10          if Δ_W(R, q) ≤ r + r_u then
11             Search(ptr(T(R)), q, r)
12 else
13    for all grnd(R) in N do
14       compute Δ_W(R, P) with previously computed components
         (weigh components)
15       if |Δ_W(P, q) − Δ_W(R, P)| ≤ r then
16          compute Δ_W(R, q)
17          if Δ_W(R, q) ≤ r then
18             add R to the query result
19 end function
```

Fig. 19 Adapted M-tree range query algorithm

The covering radius must be estimated, as it is of Type 2. When N is not a leaf, the condition is of the form $d \geq a$ with a search action (cf. Section 3.3, Step 4), so we can directly replace it by its upper bound (see line 7). When N is a leaf, the covering radius has no role, so the condition remains the same.

The only difference between the original implementation of M^3-Tree and the directly adapted M-tree is the fact that, instead of storing all the components of the distances, M^3-tree stores an estimation of each component (i.e, it uses fewer bits). This decreases the amount of memory needed to store the structure and also changes the range query algorithm, because it cannot compute exactly the distance $\Delta_{\mathbb{W}}(R, P)$. Thus, it has to use lower and upper bounds of this distance to adapt the conditions in line 8 and 15 of Fig. 19.

4.4 Pivot-based index for multimetrics

The pivot-based multi-metric index [5] can also be derived as a direct application of the proposed methodology. Indeed, a pivot-based index stores the distance from each point of the database to each one of the pivots. This means that the distances are of Type 1. Thus, to convert this index into a multi-metric index one it only needs to store the components for all distances.

5 Experimental evaluation

We performed an experimental evaluation of the efficiency of the adapted indexes using three real datasets. The first dataset is a 3D models database collected by the University of Konstanz [8], which has 16 features ranging from 32D to 510D. This database has 1,838 objects, where 1,654 were indexed and the remaining 184 (10%) were used as query objects. The second database is the collection of *Corel image features*, available at the *UCI KDD Archive* [14]. This database has 4 features vectors giving a combined feature vector of 89D. Of the 65,615 objects, 1,312 (2%) were used as query objects and the remaining ones were indexed. The last dataset is also a collection of images; these images were taken from Flickr and processed by the CoPhIR group [13], giving 5 MPEG7 features from 12D to 80D. This database has over 100 million images, but in our tests we used only a subset of it, where 199,000 images were indexed and 1,000 (0.5%) were used as query objects.

Before indexing the dataset, we processed the data in order to have meaningful results. The first step was to reduce the dimensionality of the features using PCA. For the 3D Objects database, we chose six different features (namely: 3DDFT, CPX, GRAY, H3D, SIL, and VOX [8]) and then we reduced each feature to 16D. The second step was to normalize the database, that is, that the maximum distance between any two objects is 1.0. Finally, we extracted from the dataset the query objects.

In the tests, we used for each metric space the metric $\delta = L_1$ (Manhattan distance), defined as $L_1(x, y) = \left(\sum_{1 \leq i \leq D} |x_i - y_i|\right)$. The performed tests were the following:

1. 10-NN queries with weights in range $[w, w + 0.1]$.
2. k-NN queries with weights in range $[0, 0.1]$, ranging k from 1 to 10.
3. k-NN queries with a weight equal to 1.0 and the remaining weights equal to 0.0, ranging k from 1 to 10 .

In Tests 1 and 2, the weights are uniformly distributed in ranges of the form $[w, w + 0.1]$, with $w \in [0, 1]$. We chose such weights because in a k-NN query the result does not change when all the weights are equally scaled. For this reason, range $[w, w + 0.1]$ is equivalent to $[\frac{w}{w+0.1}, 1.0]$. Thus, the "complexity" of the query increases when w decreases. It is important to notice that a query with weights in the range [0.0, 0.1] (i.e., $w = 0$) is equivalent to one with weights in [0.0, 1.0], which is the more general case. Note that Test 3 is the more "pathological" one, as it is the same as selecting only one metric to perform the search.

Additionally, we show the efficiency of the proposed indexes with other kinds of weight distributions. We used a normal distribution centered in 0.5 and standard deviation between 0.05 and 0.40. We also used a Zipf distribution to generate the weights. The first distribution is $(Z(\rho) - 1)/10$ and the second is $1 - (Z(\rho) - 1)/10$, with ρ between 1.5 and 2.5. The first Zipf distribution gives weights clustered close to 0.0, and the second one gives weights clustered close to 1.0. We chose the above mentioned values for parameter of the distribution in order to have most of them in interval [0.0, 1.0]. However, sometimes the weights were outside of the interval. In that case, we set them as the closest extreme (either 0.0 or 1.0).

We tested the proposed adapted indexes (MMGNAT, MMLCluster) and the M^3-tree (note that we used the adapted M-tree index, which should have a better performance than M^3-tree because it stores the real distance components, not a few bit approximation). We also created one specific metric index for each of our query objects (i.e., indexed using ordinary metric $\Delta_{\mathbb{W}}$, with fixed $\mathbb{W}$). These query-dependent indexes served us as the baseline, i.e., they show the optimal query processing regarding each correspondent index. Remember that this is an unpractical solution (to have one index per query), and it is only used to show how far/close are the multi-metric indexes to its optimum performance. These lower-baseline indexes are represented in the graphics as "$Index_{lb}$".

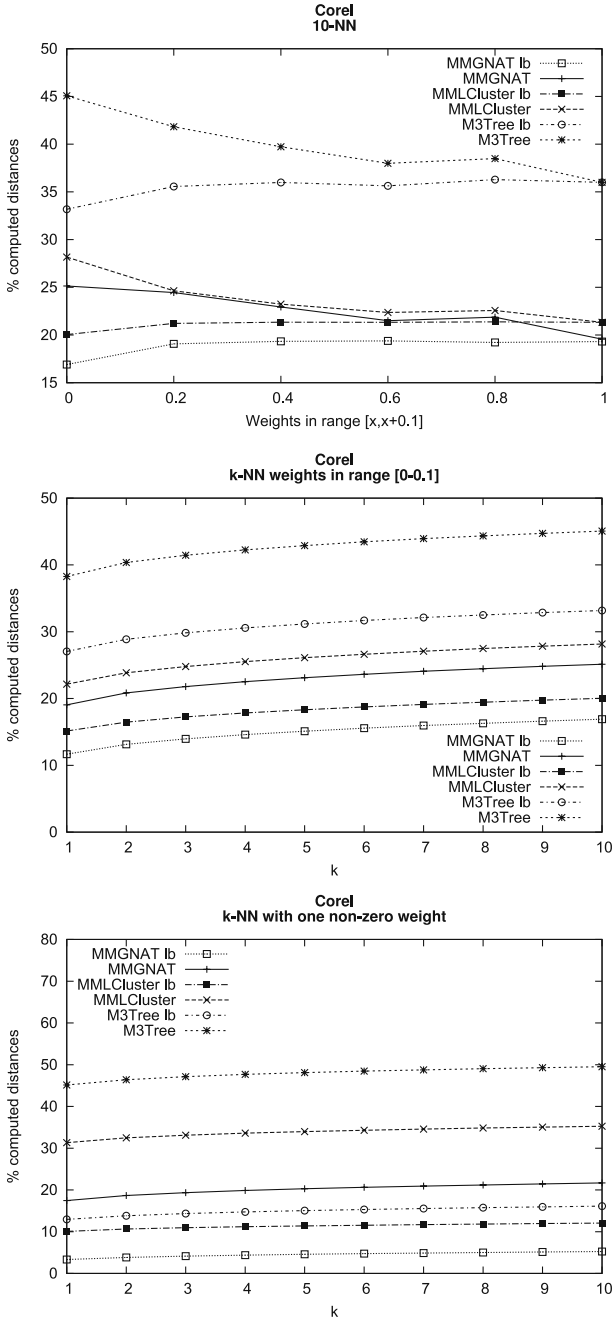
5.1 Experimental results

Figures 20, 21, and 22 show the experimental results of each of the three test for the three datasets, comparing the average number of distances computations required for the respective similarity query. The results clearly show that a single multi-metric index is almost as good as if we have infinitely many metric indexes at our disposal (one metric index built for every possible vector of query weights) if the weights are similar. For the most complex case (weights in the range [0, 0.1]) there is a hit in the performance of the multi-metric index, but they still are able to discard a large part of the dataset while performing similarity queries.

Figure 23 shows the result of the test using the Corel dataset, but now measuring the average time used for the similarity queries. For this experiment, the MML-Clusters is competitive, but note that we are using a metric function that is cheap to compute. If we have used a more expensive distance (e.g., Mahalanobis), the MMGNAT would outperform the other indexes also in time.

The results presented in Figs. 20, 21, 22, 23, 24, and 25 show that the most efficient multi-metric index is MMGNAT. Although the lower bound of MMLCluster is almost as good as the lower bound of MMGNAT, MMLCluster does not behave as well as MMGNAT. This is because MMLCluster stores less information, only two distances for each cluster. As these distances are estimated, it has less chances to

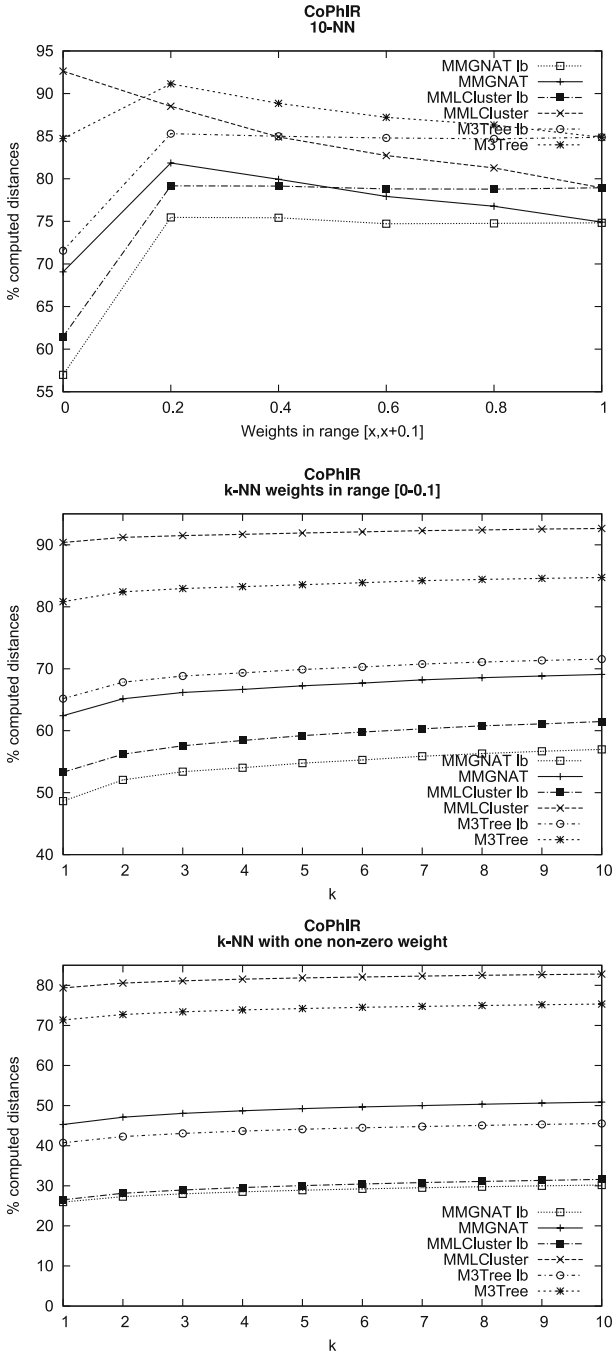
Fig. 20 Corel results



discard objects. Also, Fig. 24 shows that other distribution of weights gives similar results, being MMGNAT consistently the best multi-metric index.

In case all weights are equal to 1.0, the multi-metric index should behave as the original index, because the multi-metric index is built with those weights (see

Fig. 21 CoPhIR results



Section 3.3). However, Fig. 22 shows that MMGNAT compute slightly less distances than GNAT. This may be explained because in the construction of MMGNAT the split points are chosen randomly, and this can affect the efficiency of the index. The

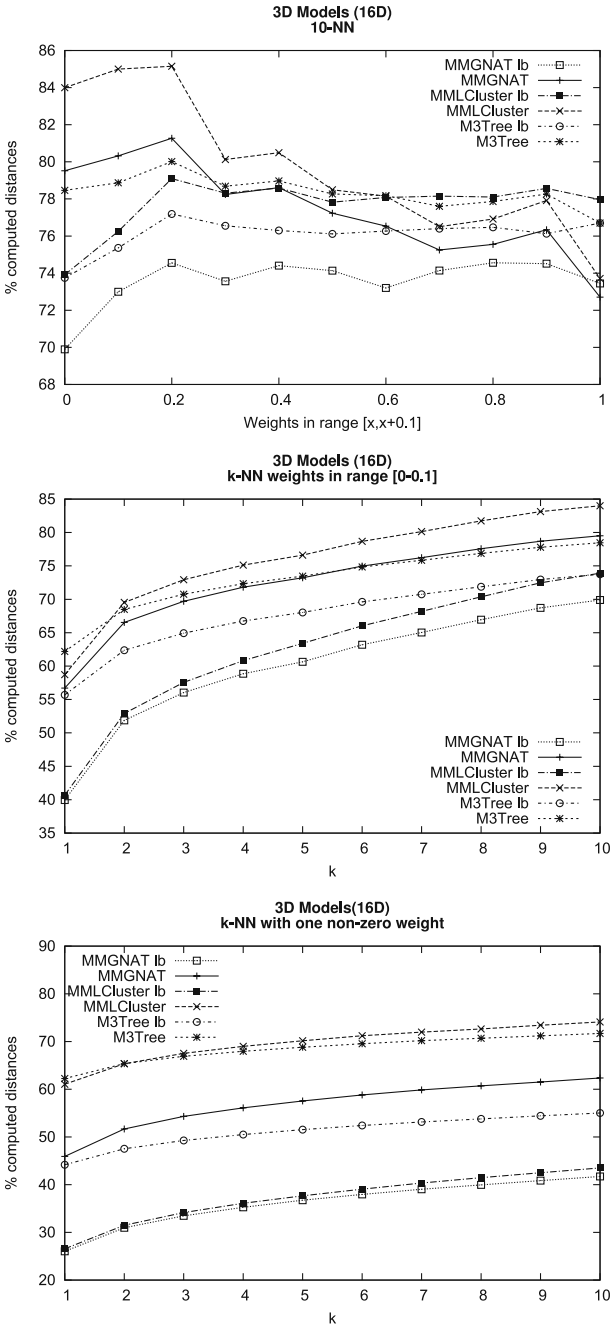


Fig. 22 3D models (16D) results

obtained results also show that in the extreme case where just one weight is non-zero, the performance of the multi-metric indexes is far from their lower bound. In this scenario, it would be probably better to build one index for each metric distance.

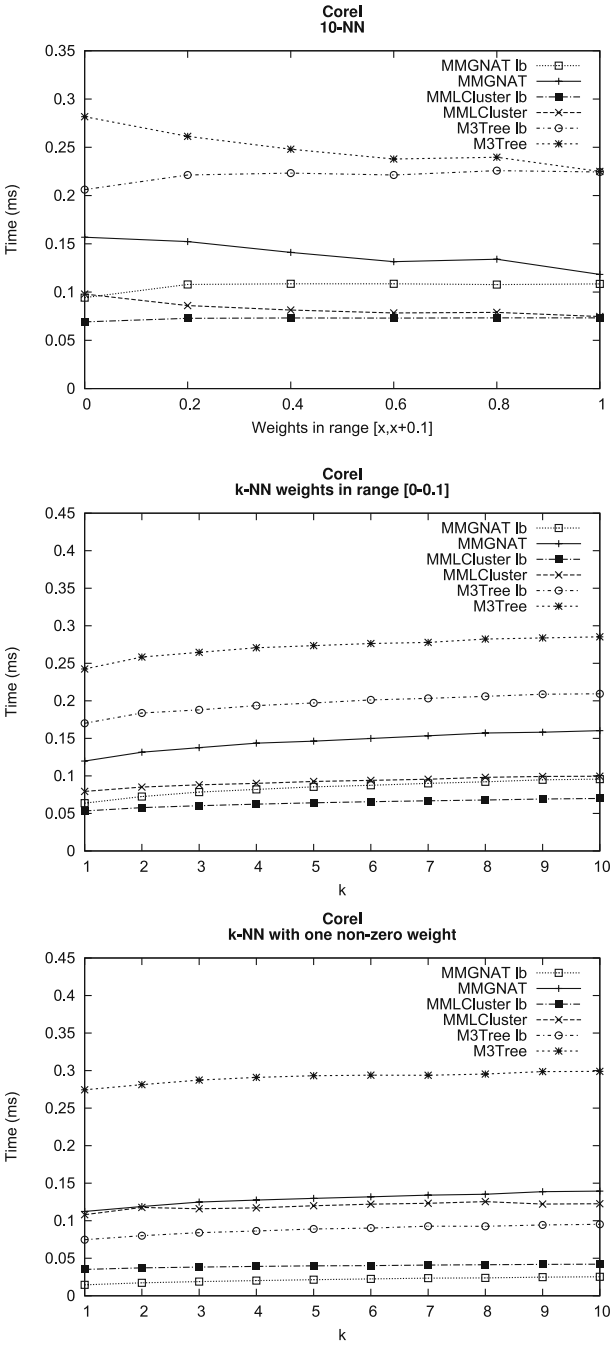


Fig. 23 Corel results (measuring time)

Figure 25 shows that there is an increasing in the number of distances computed when more metrics are combined. This is the expected result, because if more distance functions are combined, it is more difficult to correctly estimate the distance

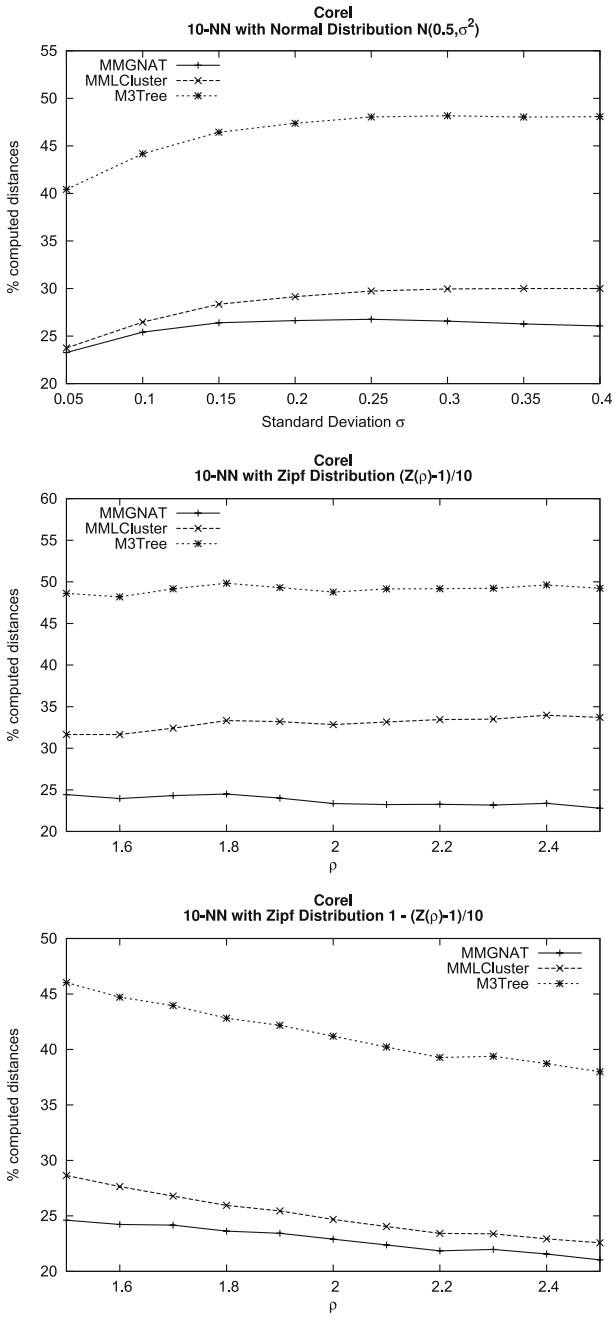


Fig. 24 Corel results with different weight distributions

values of the objects stored in the index structure using Lemmata 1 and 2. Thus, it becomes more difficult to discard objects during the query processing.

Figure 26 shows how the performance of the proposed multi-metric indexes (with the Corel dataset) if only the bound based on weights (Lemma 1) or the bound

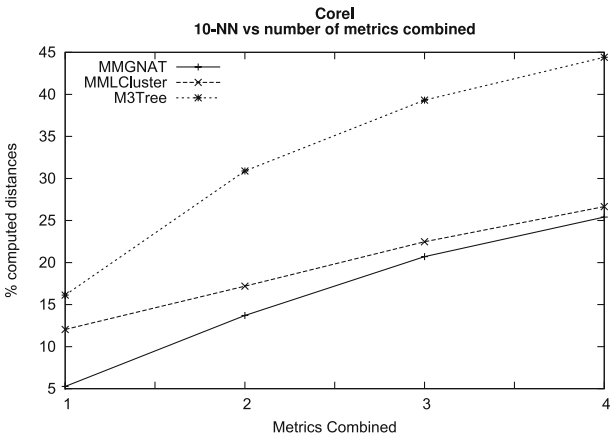


Fig. 25 Corel results with different number of metrics

based on distance components (Lemma 2) is used for estimating distances. The figure shows that for 10-NN queries the only case where it is better to use Lemma 2 is when the weights are in the range [0, 0.1]. However, the other chart shows that Lemma 2

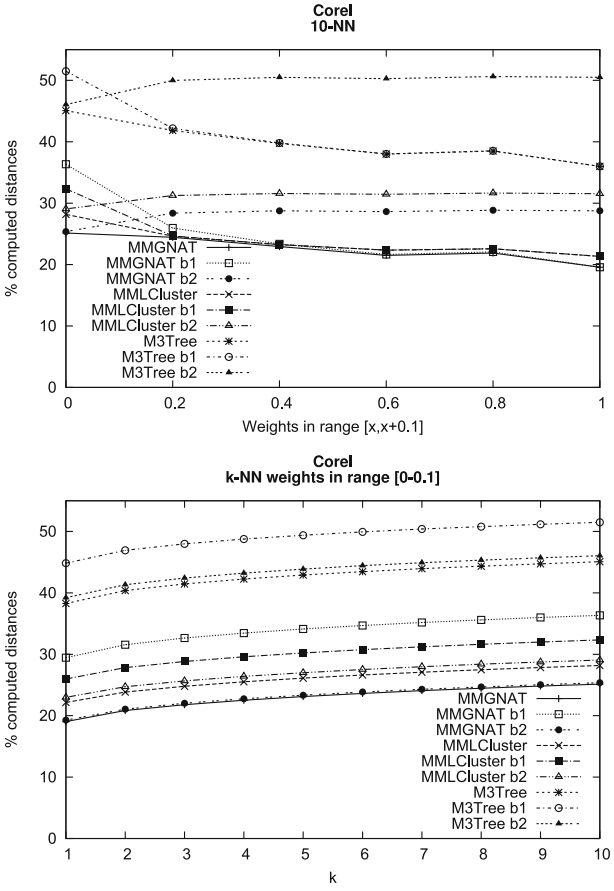


Fig. 26 Comparison of the bounds computed by Lemmas 1 and 2

produces the better results, which is because in this chart the weights were in the range [0, 0.1]. For all other cases, the bound produced by Lemma 1 was better. Thus, given that no bound outperforms the other one in all situation, our recommendation to use the best between both values is the most efficient approach.

In summary, the experimental evaluation shows that the proposed methodology is robust with respect to different weights and data distributions, and even in the complex cases the efficiency of the proposed algorithms is not so far from the lower bound. In our methodology, the distances are estimated according to the data stored by the index. Thus, when the index stores few distances the estimation may be not very tight. This is the reason why MMLCluster did not perform as well as the other tested indexes.

6 Conclusions

In this paper, we presented a methodology to adapt metric indexes to be used as a multi-metric index. The great advantage of this methodology is that it is general and it may be used to adapt any metric index. This gives us the flexibility to choose the most appropriate index depending on data distribution, intrinsic dimensionality, and any other user-defined parameter or requirement.

We showed how the methodology can be applied to two different metric indexes, yielding two novel indexing methods for multi-metric spaces, and how previously proposed multi-metric indexes fit into this approach. Additionally, we proposed a k-NN algorithm for GNAT. Although this algorithm was developed using the technique presented by Hjaltason and Samet, as far as we know, it has not been presented before in the literature.

In the experimental evaluation, the proposed methodology shows good performance. We also showed that MMGNAT outperforms the M^3-tree, thus obtaining better results than the state-of-the-art.

References

1. Böhm C, Berchtold S, Keim DA (2001) Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. ACM Comput Surv 33(3):322–373. doi:http://doi.acm.org/10.1145/502807.502809
2. Brin S (1995) Near neighbor search in large metric spaces. In: Proc. of the 21th international conference on very large data bases (VLDB'95). Morgan Kaufmann, San Mateo, CA, pp 574–584
3. Bustos B, Skopal T (2006) Dynamic similarity search in multi-metric spaces. In: Proc. 8th ACM SIGMM international workshop on multimedia information retrieval (MIR'06). ACM Press, pp 137–146
4. Bustos B, Navarro G, Chávez E (2003) Pivot selection techniques for proximity searching in metric spaces. Pattern Recogn Lett 24(14):2357–2366
5. Bustos B, Keim D, Schreck T (2005) A pivot-based index structure for combination of feature vectors. In: Proc. 20th annual ACM symposium on applied computing, multimedia and visualization track (SAC-MV'05). ACM Press, pp 1180–1184
6. Bustos B, Keim D, Saupe D, Schreck T, Vranić D (2004) Automatic selection and combination of descriptors for effective 3D similarity search. In: Proc. IEEE international workshop on multimedia content-based analysis and retrieval (MCBAR'04). IEEE Computer Society Press, Los Alamitos, CA, pp 514–521

7. Bustos B, Keim D, Saupe D, Schreck T, Vranić D (2004) Using entropy impurity for improved 3D object similarity search. In: Proc. IEEE international conference on multimedia and expo (ICME'04). IEEE, pp 1303–1306
8. Bustos B, Keim D, Saupe D, Schreck T, Vranić D (2006) An experimental effectiveness comparison of methods for 3D similarity search. Int J Digit Libr 6(1):39–54 (Special issue on Multimedia Contents and Management in Digital Libraries)
9. Chávez E, Navarro G (2005) A compact space decomposition for effective metric indexing. Pattern Recogn Lett 26(9):1363–1376
10. Chávez E, Navarro G, Baeza-Yates R, Marroquín JL (2001) Searching in metric spaces. ACM Comput Surv 33(3):273–321. doi:http://doi.acm.org/10.1145/502807.502808
11. Ciaccia P, Patella M (2002) Searching in metric spaces with user-defined and approximate distances. ACM Trans Database Syst 27(4):398–437
12. Ciaccia P, Patella M, Zezula P (1997) M-tree: an efficient access method for similarity search in metric spaces. In: Proc. of the 23rd international conference on very large data bases (VLDB'97). Morgan Kaufmann, San Mateo, CA, pp 426–435
13. Falchi F, Lucchese C, Perego R, Rabitti F (2008) CoPhIR: content-based photo image retrieval. http://cophir.isti.cnr.it/CoPhIR.pdf
14. Hettich S, Bay SD (1999) The UCI KDD archive. http://kdd.ics.uci.edu. University of California, Department of Information and Computer Science, Irvine, CA
15. Hjaltason GR, Samet H (1995) Ranking in spatial databases. In: Proc. of the 4th international symposium on advances in spatial databases (SSD'95). Springer, pp 83–95
16. Hoksza D, Galgonek J (2009) Density-based classification of protein structures using iterative TM-score. In: Computational structure bioinformatics workshop (CSBW'09) (BIBM'09). IEEE
17. Keim D (1999) Efficient geometry-based similarity search of 3D spatial databases. In: Proc. ACM international conference on management of data (SIGMOD'99). ACM Press, pp 419–430
18. Samet H (2005) Foundations of multidimensional and metric data structures (the Morgan Kaufmann series in computer graphics and geometric modeling). Morgan Kaufmann, San Mateo, CA
19. Smith T, Waterman M (1981) Identification of common molecular subsequences. J Mol Biol 147(1):195–197
20. Zezula P, Amato G, Dohnal V, Batko M (2005) Similarity search: the metric space approach (advances in database systems). Springer, New York

Benjamin Bustos is an Assistant Professor at the Department of Computer Science, University of Chile. He is head of the PRISMA Research Group. He leads research projects in the domains of multimedia retrieval, video copy detection, sketch-based image retrieval, and retrieval of handwritten documents. His interests include similarity search, multimedia retrieval, 3D object retrieval, and (non)-metric indexing. He has a doctoral degree in natural sciences from the University of Konstanz, Germany (2006).

Sebastian Kreft is a master student in the Department of Computer Science at the University of Chile. His areas of interest are multimedia information retrieval, pattern recognition and succinct data structures. He has a professional title in Computer Engineering (2009).

Tomáš Skopal is Associate Professor in the Department of Software Engineering at the Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic. His research interests are metric access methods, database index structures, multimedia databases, and similarity modeling. He has published extensively in the area of metric and nonmetric similarity search. He has a doctoral degree in computer science and applied mathematics (2004) from the Technical University of Ostrava, Czech Republic.