

Masaryk University
Faculty of Informatics



Indexing Structures for Similarity Search in Metric Spaces

Ph.D. Thesis Proposal

Vlastislav Dohnal

Supervisor: doc. Ing. Pavel Zezula, CSc.

In Brno, September 16, 2002

Supervisor

Contents

1	Introduction	3
2	Metric Space Searching	4
2.1	Motivation	4
2.2	Metric Spaces	5
2.3	Distance Functions	5
2.4	Similarity Queries	6
3	Indexing techniques	7
3.1	Discrete Distance Functions	7
3.1.1	Burkhard-Keller Tree	7
3.1.2	Fixed Queries Tree	8
3.1.3	Fixed Queries Array	9
3.2	Continuous Distance Functions	10
3.2.1	Vantage Point Tree	10
3.2.2	Multi Vantage Point Tree	10
3.2.3	Excluded Middle Vantage Point Forest	11
3.2.4	Generalized Hyperplane Tree	11
3.2.5	Geometric Near-neighbor Access Tree	12
3.2.6	M-Tree	12
3.3	Choosing References	13
4	Thesis Plan	13
4.1	Critical view	14
4.2	Objectives	15
4.3	The Structure Outline	15
4.4	Future Plans	17
	Bibliography	17
A	Current Results of My Study	20
B	Summary / Souhrn	21

1. Introduction

Searching has always been one of the most prominent data processing operations because of its useful purpose of delivering required information efficiently. However, exact match retrieval, typical for traditional databases, is not sufficient or feasible for data types of present digital age, that is image databases, text documents, audio and video collections, DNA or protein sequences, to name a few. What seems to be more useful, if not necessary, is to base the search paradigm on a form of *proximity*, *similarity*, or *dissimilarity* of query and data objects. Roughly speaking, objects that are near to a given query object form the query response set. In this place, the notion of mathematical *metric space* [20] provides a useful abstraction for *nearness*.

Similarity search is needed as a fundamental computational task in a variety of application areas, including multimedia information retrieval, data mining, pattern recognition, machine learning, computer vision, genome databases, data compression, and statistical data analysis. This problem, originally mostly studied within the theoretical area of computational geometry, is recently attracting more and more attention in the database community, because of the increasingly growing needs to deal with large volumes of data.

An index structure supporting similarity retrieval should be able to execute similarity queries without any limitations, for example performing range queries with any search radius. However, a significant group of applications needs retrieval of objects which are in a very near vicinity of the query object – e.g., copy (replica) detection, cluster analysis, DNA mutations, corrupted signals after their transmission over noisy channels, or typing (spelling) errors. Thus in some cases, special attention may be warranted.

The development of Internet services often requires an integration of heterogeneous sources of data. Such sources are typically unstructured whereas the intended services often require structured data. The main challenge is to provide consistent and error-free data, which implies the *data cleaning*, typically implemented by a sort of *similarity join*. In order to perform such tasks, similarity rules are specified to decide whether specific pieces of data may actually be the same things or not. However, when the database is large, the data cleaning can take a long time, so the processing time (or the performance) is the most critical factor that can only be reduced by means of convenient similarity search indexes.

Several storage structures, such as [25, 6, 14, 5], have been designed to support efficient similarity search execution over large collections of metric data where only a distance measure for pairs of objects is possible to quantify. However, the performance is still not satisfactory and further investigation is needed. Though all of the indexes are trees, many tree branches have to be traversed to solve a query, because similarity queries are basically range queries that typically cover data from several tree nodes or some other content-specific partitions. In principle, if many data partitions need to be accessed for a query region, they are not *contrasted* enough and such partitioning is not useful from the search

point of view.

This thesis concentrates on investigation of new strategies which support efficient similarity search executions. The rest of this proposal is organized as follows. Section 2 embraces background and definitions of similarity searching in metric spaces. In Section 3, a survey of existing index structures for metric spaces is contained. The plan of thesis is outlined in Section 4. In addition, current results of my study are summarized in Appendix A.

2. Metric Space Searching

This section introduces the problem of indexing of metric spaces and provides an essential background and definitions of metric spaces and similarity queries.

2.1 Motivation

The classical approach to the search problem is based on exact match paradigm. That is, a search query is given and a record (number or string) which is *exactly equal* to the search query is retrieved. Traditional databases are built on the concept of exact match searching. However, more sophisticated searches such as range queries rely on the full linear order of searched domain.

New data types, images, audio records, video sequences, fingerprints to name a few, cannot be meaningfully queried in the classical sense. There are no applications which are interested in searching an audio clip exactly the same as a given query. Furthermore, these data types cannot be linearly ordered, thus the classical range search cannot be applied. Practically all multimedia applications are interested in objects similar to a given query.

Imagine a collection of images. Interesting queries are of the type "find an image of a lion with a savanna background". The classical scheme can be applied if the collection has been tagged, it means that every image has assigned a list of text descriptions of objects contained in it. Unfortunately, such a tagging is not possible to make automatically due to immature state of objects' recognition in real world scenes regardless of issues of describing some image elements, for example textures, in natural language. Another possible query can specify a part of an image and wants to retrieve all images that contains this image fraction.

The problem of correcting misspelled words in written text is rather old, and the experience reveals that 80% of these errors are corrected allowing just one insertion, deletion, or transposition. But the problem is not only grammatical, because an incorrect word that is entered in the database cannot be retrieved anymore on the exact match bases. According to [21], text typically contain about 2% of typing and spelling errors. Moreover, text collections digitalized via optical character recognition (OCR) contain a non negligible percentage of errors (7 – 16%). Such numbers are even higher for text obtained through

conversion from a voice.

These short examples share the same concept of similarity searching, that means the problem of finding objects which are close to a given query object.

2.2 Metric Spaces

A convenient way to assess similarity between two objects is to apply metric functions to decide the closeness of objects as a distance, that is the dis-similarity of objects. A *metric space* $\mathcal{M} = (\mathcal{D}, d)$ is defined by a domain of objects (elements, points) \mathcal{D} and a total (distance) function d .

Distance functions, used in metric spaces, must satisfy the following properties:

- $\forall x, y \in \mathcal{D}, d(x, y) \geq 0$ (positiveness),
- $\forall x, y \in \mathcal{D}, d(x, y) = 0 \Leftrightarrow x = y$ (identity),
- $\forall x, y \in \mathcal{D}, d(x, y) = d(y, x)$ (symmetry)
- $\forall x, y, z \in \mathcal{D}, d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality)

We also assume that the maximum distance never exceeds d^+ , thus we consider a *bounded metric space*.

2.3 Distance Functions

Distance functions (metrics) are usually specified by an expert, however, their definitions are not restricted to any type of queries that can be asked. In the following, we provide some examples of similarity measures in more detail.

Similarity of images can be measured in many ways, such a quantification can be based on a shape extraction, textures, color histograms, image patterns. The distance between histograms can be measured by a *Minkowski* metric function, so-called L_n distance function, defined as

$$L_n[(x_1, \dots, x_k), (y_1, \dots, y_k)] = \sqrt[n]{\sum_{i=1}^k |x_i - y_i|^n}.$$

L_1 metric is called the *Manhattan distance*, L_2 denotes well-known the *Euclidean distance* and $L_\infty = \max_{i=1}^k |x_i - y_i|$ is named the *maximum distance*. The usage of L_n family of metric functions to compute distance between histograms does not introduce any correlations among individual colors. Therefore, more complex measures such as a quadratic form must be applied [18]. However, because color histograms are typically high-dimensional vectors (256 or 64 distinct colors) the distance measure is computationally expensive.

Another example of distance, that can be applied on histograms, is the *Hausdorff metric* [19]. This metric measures a distance between two sets A, B and it is defined as follows:

$$\begin{aligned}d_p(x, B) &= \inf\{d(x, y), y \in B\} \\d_p(A, y) &= \inf\{d(x, y), x \in A\} \\d_s(A, B) &= \sup\{d_p(x, B), x \in A\} \\d_s(B, A) &= \sup\{d_p(A, y), y \in B\}.\end{aligned}$$

Then the Hausdorff distance is:

$$d(A, B) = \max\{d_s(A, B), d_s(B, A)\}.$$

Apparently, the Hausdorff distance is a very time-consuming operation since its time complexity is $\mathcal{O}(n^2)$.

Text strings are usually compared by the *edit distance*, so-called the *Levenshtein distance* [22]. This measure computes the minimal number of insertions, deletions, and replacements of one character needed to transform one string into another. This distance function can be also applied on other granularities, such as words, paragraphs, or whole documents. All known algorithms implementing the edit distance have the time complexity $\mathcal{O}(n^2)$. Thus, the evaluation of edit distance is a time-consuming operation. The nice survey [23] presents numerous solutions to this problem.

There are many other similarity measures which can be used on various data types. However, computationally expensive distance functions are not exceptions. Generally, metric space indexing structures consider a metric function as a high CPU consuming operation. Therefore, their aim is to minimize the number of evaluations of a distance function.

2.4 Similarity Queries

In general, the problem of indexing metric spaces can be defined as follows.

Problem 1 *Let \mathcal{D} be a set, d a distance function defined on \mathcal{D} , and $\mathcal{M} = (\mathcal{D}, d)$ a metric space. Given a set $X \subseteq \mathcal{D}$, preprocess or structure the elements of X so that similarity queries can be answered efficiently.*

A similarity query returns objects which are close to a given query object. We typically distinguish three fundamental types of similarity queries:

- **Range Query** (q, r) – retrieves all objects which are within the distance of r to the query object q , that is $\{o \in X, d(q, o) \leq r\}$,
- **k -Nearest Neighbor Query** $NN_k(q)$ – retrieves the k closest objects to the query object q , that is a set $\{R \subseteq X, |R| = k$ and $\forall x \in R, y \in X - R, d(q, x) \leq d(q, y)\}$,

- **Join Query** JQ_t – retrieves all pairs of objects $(x, y) \in X \times X$ such that the distance between x and y is less than or equal to a threshold t , that is $\{(x, y) \in X \times X, d(x, y) \leq t\}$.

Notice that a query object q does not necessary have to belong to an indexed set of objects X , i.e. $q \notin X$. However, the query object must be included in the domain of metric space \mathcal{D} , that is $q \in \mathcal{D}$.

Other similarity queries can be defined, nevertheless, they are often combinations of these three basic types. For example, a meaningful query is the nearest neighbor query NN_k where the maximum distance r to the furthest nearest neighbor is specified. Similarly, the join query can be bounded to the number of retrieved pairs.

3. Indexing techniques

In this section, we provide a short survey on existing indexes for metric spaces. The more complete and more detailed survey is available in [12]. Presented index structures are divided into two categories with respect to the type of distance function they support:

- **discrete** distance function,
- **continuous** distance function.

Indexing techniques of metric spaces assume that a discrete distance function is a function which returns only a small set of values, while a continuous distance function is a function which delivers an infinite set of possible values or a very large set of alternatives. These assumptions are far away from strict mathematical definitions, nevertheless, they are sufficient for the case of our usage. A typical example of the discrete distance function is the *edit distance*, while *Minkowski distance* measures are typical representatives of the other type.

The remaining part of this section is organized into three subsections. The first part presents several index structures for discrete distance functions while several techniques for continuous functions are described in the second subsection. Finally, the last part concerns the problem of selecting convenient pivots.

3.1 Discrete Distance Functions

In this section, structures which take the advantage of discrete distance functions, i.e. small sets of values, are introduced.

3.1.1 Burkhard-Keller Tree

Probably the first solution to support searching in metric spaces is presented in [8]. A **BKT** tree (Burkhard-Keller Tree) is proposed and it is defined as

follows: an arbitrary object $p \in X$ is selected as the root of tree. For each distance $i > 0$, X_i is defined as the set of all objects with the distance of i to the root p , $X_i = \{o \in X, d(o, p) = i\}$. A child node of the root p is built for every non-empty set X_i . All child nodes are recursively re-partitioned until no new child node can be created. Objects assigned as roots of sub-trees, saved in internal nodes, are called *pivots*.

The range search (q, r) starts at the root p of the tree and enters all child nodes i such that

$$d(q, p) - r \leq i \leq d(q, p) + r \quad (3.1)$$

and proceed down recursively. If $d(q, p) \leq r$ holds the object p is reported. Notice that the inequality 3.1 cuts out some branches of the tree.

Pruning some branches of the tree is allowed by validity of the triangle inequality. Assume a pruned branch and an object o stored in this branch, the distance between o and its parent p is equal to i , that is $d(p, o) = i$. The condition $|d(p, q) - i| > r$ holds (from the pruning criterion 3.1), because this branch has already been excluded. From the triangle inequality $d(p, q) \leq d(p, o) + d(o, q)$ we get $d(o, q) \geq d(p, q) - d(p, o) > r$.

Figure 3.1 shows an example where the BKT tree is constructed from objects of the space which is illustrated in the left side of figure. Objects p , o_1 , and o_4 are selected as roots of sub-trees, so-called pivots. The range query is also presented and specified by the object q and radius $r = 2$. The search algorithm evaluating the query $(q, 2)$ discards some branches, the accessed branches are emphasized.

Obviously, if the search radius of range query grows, the number of accessed subtrees (branches) increases. This leads to higher search costs which are measured in the number of distance computations needed to answer a query. The increasing number of distance computations can be attributed to the fact that different pivots are selected at the same level of tree. The following structure, FQT trees, reduces such the costs.

The BKT trees are linear in space ($\mathcal{O}(n)$). The search time complexity is $\mathcal{O}(n^\alpha)$ where $0 < \alpha < 1$ and depends on the search radius and on the structure of indexed space [12].

3.1.2 Fixed Queries Tree

The **FQT** (Fixed Queries Tree), originally presented in [2], is an extension of BKT tree. Fixed Queries Trees use the same pivot at the same level, that is in contrast to BKT trees where several pivots per level are used. Objects of a given metric space are stored in leaf nodes only. The range search algorithm is the same as for the BKT. The advantage of this structure is following, if more than one subtree must be accessed only one distance computation between the query object and the pivot per level is computed because all nodes at the same level share the same pivot. The experiments, presented in [2], reveal that FQTs perform less distance computations than BKTs.

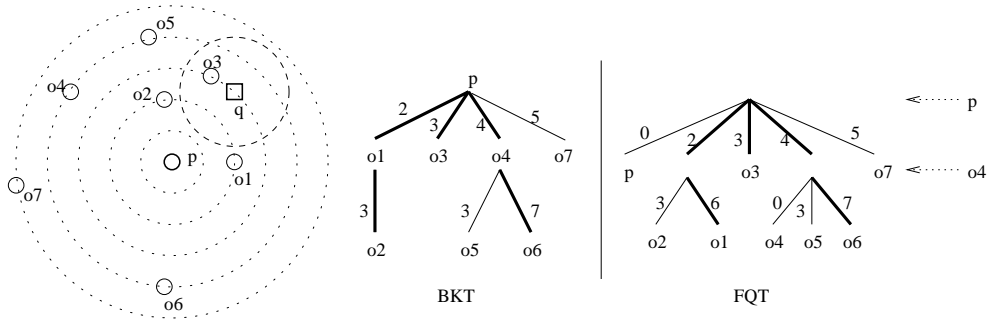


Figure 3.1: On the left, partitioning of the space by pivot p . On the right, corresponding BKT and FQT trees.

An example of FQT tree is demonstrated in Figure 3.1. The tree is built from objects depicted on the left, where objects p and o_4 are pivots of corresponding levels. Notice that all objects are stored in leaves, including objects selected as pivots. The range query $(q, 2)$ is defined and accessed branches of the tree are highlighted.

The space complexity is superlinear, because the objects selected as pivots are duplicated, and is upper-bounded by $\mathcal{O}(n \log n)$. The time complexity is $\mathcal{O}(n^\alpha)$ where $0 < \alpha < 1$ depending on the query radius and an indexed space.

In [2, 3], the authors propose a variant of FQT trees, called **FHFQT** (Fixed-Height Fixed Queries Tree). This structure has all leaf nodes at the same level, that is, leaves are at the same depth h . This makes some branches of the tree deeper than necessary, however, this makes sense because any additional distance computations are not needed since they were already computed at upper levels.

3.1.3 Fixed Queries Array

The Fixed Queries Array (**FQA**) is presented in [11, 10]. Generally, the FQA is not a tree structure, however, its organization comes out from a construction of a tree. The FHFQT tree [3] with fixed height h is built on a given set of objects. The leaf nodes of the FHFQT are concatenated to an array and the result is the FQA. For every member of the array, h numbers are computed. These numbers represent the path from the root to the leaf. Indeed, these h numbers are distances to h pivots used in the FHFQT tree. Each of these h distances is coded in b bits and concatenated together to form one hb -bit number where the most significant bits correspond to higher levels of the tree. The FQA is a sorted array of these hb -bit numbers.

FQAs are able to use more pivots at the same space than FHFQT trees, this improves the efficiency. The authors of [11] show that the FQA outperforms the FHFQT.

3.2 Continuous Distance Functions

The usage of discrete distance functions is restrictive on the application domain. The discrete case is, in fact, a specialization of the continuous case. This section concentrates on index structures supporting continuous distance functions.

Structures designed for usage of discrete distance functions cannot be applied on the continuous case without any changes. In a metric space with a continuous distance function, there is usually very small probability that a pair of objects will have the same distance. Therefore, the usage of tree structures designed for discrete functions results in a flat tree of height of 2. In fact, this is the same as the sequential scan.

3.2.1 Vantage Point Tree

The *Metric Tree*, published in [24], is a tree structure designed for continuous distance functions. More complete work [25] using the same idea presents the **VPT** (Vantage-Point Tree). The tree is built recursively in the following way: at random an object p (pivot) is picked and promoted to the root. The *median* of all distances to other objects is computed ($M = \text{median}\{d(p, o), o \in X\}$) and all objects are partitioned by the median into the left and the right subtree, $\{o \in X, d(o, p) \leq M\}$ and $\{o \in X, d(o, p) > M\}$ respectively. Subtrees are re-partitioned until they are empty. This procedure produces a balanced tree.

A range query (q, r) execution traverses the tree from the root to leaves and computes distances $d(p, q)$ between the query object q and the node p . All objects p that satisfies $d(p, q) \leq r$ are reported to output. Some tree branches can be discarded using this following criterion: if $d(p, q) - r \leq M$ holds, the left subtree is accessed, and vice versa, if $d(p, q) + r > M$ holds, the right subtree is entered. Notice that both subtrees can be visited.

If the search algorithm has to enter both subtrees, no objects are excluded from the search procedure and its performance deteriorates. This issue is handled by extending the VPT tree to an m -ary tree. This variant uses m -thresholds (percentiles) instead of one median to partition the data set into m spherical cuts, where $m > 2$. These modified trees are called *Multi-Way Vantage Point Trees* (mw-VPTs) and presented in [4, 5].

Unfortunately, the experiments reveal that the performance of mw-VPT trees is not very satisfactory because the spherical cuts become thin for more percentiles and again more tree branches have to be examined during the search.

The VPT trees take $\mathcal{O}(n)$ space. The time complexity for very small range query radii is $\mathcal{O}(\log n)$.

Extensions of algorithms for similarity queries are presented in [13].

3.2.2 Multi Vantage Point Tree

In [4, 5], the **MVPT** (Multi Vantage-Point Tree) is presented. MVPTs are defined very similarly as VPTs but they use more pivots in internal nodes. An internal node splits a given set into 2^m partitions instead of 2 partitions, where m

is the number of pivots in the internal node usually greater than 2. An internal node of MVPT which employs 2 pivots and partitions a set into 4 partitions can be viewed as a VPT tree consisting of 2 levels, however, using the same pivot at the second level. While the VPT stores objects in all nodes (internal nodes and leaves), the MVPT stores objects in leaf nodes only. Moreover, leaf nodes of the MVPT are buckets where more objects can be stored.

A new idea is being introduced. In the construction time, many distance computations between objects and pivots are evaluated. These distances are saved in leaves. During the evaluation of search algorithm, these precomputed distances are used for excluding objects without actually computing distances to the query object. If the search algorithm enters a leaf node where more objects are stored, every object o has to be checked on the condition $d(o, q) \leq r$. Actually, before computing the distance $d(o, q)$ the *exclusion condition* $|d(o, p_i) - d(q, p_i)| > r$ is checked for each saved distance, i.e. for each pivot p_i , and if it holds for any pivot the object o is excluded. This concept of using precomputed distances in search algorithms is called *pivot-based strategy*.

Experiments evaluated in [5] show that MVPTs outperform VPTs, because the pivot-based strategy saves many distance computations.

3.2.3 Excluded Middle Vantage Point Forest

The **VPF** (Excluded Middle Vantage Point Forest), presented in [26], is a generalization of vantage point trees. It introduces a notion of *excluding middle distances* to pivots, that is, a pivot p divides objects into two sets and an exclusion set as follows: objects which satisfy $d(p, o) \leq M - \rho$ form the first set (i.e. the left subtree), objects satisfying $d(p, o) > M + \rho$ are grouped in the second set (the right subtree), any other objects are excluded from the partitioning process and form an exclusion set, where M is the median of distances from the pivot p to all objects. The left and right subtree are recursively re-partitioned and excluded objects are accommodated in the exclusion set until subtrees are empty. The second tree is created from the exclusion set of the first tree. This process is repeated and the forest of vantage point trees is obtained.

This idea of excluding the middle gives the *worst-case* sublinear time of range searches with radii upto ρ . The expected search time is $\mathcal{O}(n^{1-\alpha} \log n)$ where $0 < \alpha < 1$ depends on ρ . Since objects are stored in both internal nodes and leaf nodes the space is linear.

3.2.4 Generalized Hyperplane Tree

The previous approaches employ one pivot and a threshold to divide a set into two partitions – the inner and outer partition. Following structures uses different technique to partition metric data sets. This technique is based on clustering, that is, a partition is built around a pivot and accommodates close objects to the pivot. Notice the relationship between this idea and the *Voronoi-like* partitioning [1].

The **GHT** (Generalized Hyperplane Tree) is proposed in [24]. The tree is built recursively as follows: two objects p_1, p_2 are selected and promoted to *pivots*. Objects closer to p_1 forms the left subtree while objects closer to p_2 forms the right subtree. At the search time, the hyperplane between two pivots p_1, p_2 are used as the pruning criterion. In more detail, the left subtree is entered if $d(q, p_1) - r < d(q, p_2) + r$ holds and the right subtree is visited if $d(q, p_1) + r \geq d(q, p_2) - r$ is satisfied. Once again, it is possible to access both subtrees. The authors of [24] claim that GHTs could work better than VPTs in high dimensional vector spaces.

The search algorithm, traversing the tree, enters an internal node and have to decide whether the left or right subtree or both subtrees must be visited or not. This decision is based on distances to both pivots used in the internal node and on application of the pruning criterion. This process is repeated in child nodes. The idea of reusing one of the parent pivots can save one distance computation in a child node since this distance had to be evaluated in the parent node. Trees adopting the idea are presented in [7].

The GHT trees are linear in space. The time complexity of range search algorithm was not analyzed by the authors.

3.2.5 Geometric Near-neighbor Access Tree

The **GNAT** (Geometric Near-neighbor Access Tree) is an extended version of the GHT, presented in [6]. The generalized hyperplane partitioning principle, which is used by GHT, is modified as follows: m pivots (p_1, \dots, p_m) are selected and corresponding m partitions (X_1, \dots, X_m) are defined, $X_i = \{o \in X, d(p_i, o) < d(p_j, o), \forall j \neq i\}$. Each partition X_i is re-partitioned until it is empty. At construction time, the tree stores an $\mathcal{O}(m^2)$ size table in each internal node. This table contains minimum and maximum distances from each pivot to each partition, that is $range_{ij} = [\min_{o \in X_j}(p_i, o), \max_{o \in X_j}(p_i, o)]$ for every pivot p_i and partition X_j . Thus, the space complexity is $\mathcal{O}(nm^2)$.

The search algorithm utilizes these tables for pruning branches of the tree. If the interval $[d(p_i, q) - r, d(p_i, q) + r]$ does not intersect $range_{ij}$ for any pivot p_j , then the partition X_j can be discarded. This rule is applied on the range query (q, r) until no partition can be excluded. Thereafter, the search algorithm enters all non-discarded subtrees. Experiments reveals the fact that GHT is worse than the VPT, while GNAT of arities between 50 and 100 outperforms them. Authors also mention that arities of subtrees could vary from level to level of the tree, but give no clear criteria of doing that.

3.2.6 M-Tree

The **MT** (M-Tree) data index [14, 27] provides dynamic capabilities and good I/O performance in addition to few distance computations. The MT stores objects in leaves, while internal nodes are used for partitioning only. The partitioning principle of the MT is similar to the GNAT's strategy. An internal node contains several pivots and their distances to a corresponding parent pivot.

Pivots divide an indexed set in the following way: an object is assigned to a partition of the closest pivot. The authors of [14] claims that this strategy leads to unbalanced splits, i.e. different cardinalities of partitions. However, this is the most effective variant. Each pivot also stores the covering radius of its subtree.

At search time, the query is compared against all pivots of the internal node and the algorithm enters recursively into all subtrees that cannot be discarded using the *covering radius criterion*. This criterion is based on application of the triangle inequality and employs covering radii and stored distances to parent pivots. The formal specification of this criterion is in [14].

The MT provides dynamic capabilities, that is, insertions and deletions. Insertions into the MT are handled in the similar way as for the B-Tree, that is, the tree grows from leaves to the root. Applying such the insertion strategy assures the balanced tree regardless of how many insertions have been made. Specifically, a new object is inserted in the most suitable leaf, it means that the tree is descended from the root along the subtree for which no enlargement or the least enlargement of its covering radius is needed. If the leaf is full the split algorithm is triggered. The problem of node splitting is discussed in [14] and several algorithms are defined.

3.3 Choosing References

The problem of choosing reference objects (pivots) is important for any search technique in general metric spaces, because all such structures need, directly or indirectly, some "anchors" for partitioning and search pruning, see Section 3. It is well known that the way in which pivots are selected can affect the performance of search algorithms. This has been recognized and demonstrated by several researchers, e.g. [25] or [5], but specific suggestions for choosing good reference objects are rather vague.

Due to the problem complexity, pivots are often chosen at random, or as space outliers, i.e. having large distances to other objects in the metric space. Obviously, independent outliers are good pivots, but any set of outliers does not ensure that the set is good. Qualitatively, the current experience can be summarized as:

- good pivots are *far away* from the rest of objects of the metric space,
- good pivots are *far away* from each other.

Recently, the problem was systematically studied in [9], and several strategies for selecting pivots have been proposed and tested.

4. Thesis Plan

The subject of indexing multimedia data can be divided into two different areas. The first area concerns the problem of finding a suitable metric (distance func-

tion) for the data domain. This issue is briefly discussed in Subsection 2.3. The second area is involved in the design of efficient indexing techniques for fast similarity searches, see Section 3. This thesis is concentrated on the development of indexing structures.

4.1 Critical view

The growing need to deal with large, possibly distributed, archives requires an indexing support to speedup retrieval. Most of existing techniques are not primarily designed to support a disk memory storage and they operate in the main memory only. Therefore they are not able to deal with large collections of data because the main memory is several degrees of magnitude smaller in capacity than a disk memory storage.

The development of index structures which support a disk storage introduces a new performance issue. Two types of possible applications can be specified: in the first one, the similarity of text strings of one hundred characters is computed by using the *edit distance*, which has a quadratic computational complexity; in the second application, one hundred dimensional vectors of 4-byte real numbers are compared by the *inner product*, with a linear computational complexity. Since the text objects are four times shorter than the vectors and the distance computation on the vectors is one hundred times faster than the edit distance, the minimization of I/O accesses for the vectors is much more important than for the text strings. On the contrary, the CPU costs are more significant for computing the distance of text strings than for the distance of vectors. In general, indexes based on distances and using a disk storage should minimize both of the cost components. Besides, we would mention that most of existing indexing techniques optimize only the CPU costs.

The next issue is involved in dynamic capabilities. Existing index designs suffer from being intrinsically *static*, which limits their applicability in dynamic environments that are permanently a subject of change as a function of time. In such environments, updates (insertions or deletions) are inevitable and the costs of update are becoming a matter of concern.

Most of index structures presented in Section 3 do not support dynamic capabilities by their design. It is certainly feasible to insert or delete objects into them but it will lead to unbalanced trees. Hence the search costs increase and the performance of similarity searches deteriorates. The unbalanced tree can be re-built, of course, but it takes enormous time, thus it is unfeasible. Moreover, some trees do not provide delete capability directly because some stored objects are promoted to pivots and used for partitioning of indexed space, thus deletion of them is impossible.

4.2 Objectives

The aim of this work is to point out weaknesses and merits of existing approaches for indexing metric spaces and to design a new metric space index structure which eliminates weak properties. Therefore, a proposed technique should satisfy the following characteristics:

- Dynamic Capabilities – the proposed index structure must support insertions and deletions of objects and minimize the costs to insert or delete an object. Minimal costs are needed in dynamic environments where insertions and deletions are performed permanently.
- Disk storage – support of a disk memory storage is the first prerequisite to deal with large volumes of data.
- Scalability – if data stored in an index structure doubles, search algorithms of the index structure should stay efficient, that is, the search costs should increase linearly at maximum.
- I/O and CPU costs – it is very important to optimize both these costs because different distance measures with completely different CPU requirements can be applied. This issue is discussed in the previous subsection.
- Search algorithms – the basic types of similarity queries, such as range, nearest-neighbor, and join queries, must be supported.
- Parallelism – the structure and its algorithms must not be restricted to one CPU or one computer, that is, the structure must be easily modifiable to a parallel and distributed environment.

The proposed structure will be evaluated and its properties will be verified on different data sets, both synthetic and real world data sets with different distance distributions, to demonstrate its wide range of applicability. The structure will be also compared with other index structures, for example M-Tree [14].

4.3 The Structure Outline

We propose a multi-level metric structure, consisting of the *search-separable* buckets at each level. The structure supports easy insertion and bounded search costs because at most one bucket needs to be accessed at each level for range queries up to a predefined value of search radius ρ . At the same time, the applied *pivot-based strategy*, which uses precomputed distances, significantly reduces the number of distance computations in accessed buckets. In the following, we provide a brief overview of this structure.

The partitioning principles of the proposed structure are based on a multiple definition of a mapping function, called the ρ -split function, as illustrated in Figure 4.1a. This function uses one pivot x_v and the *medium distance* d_m to

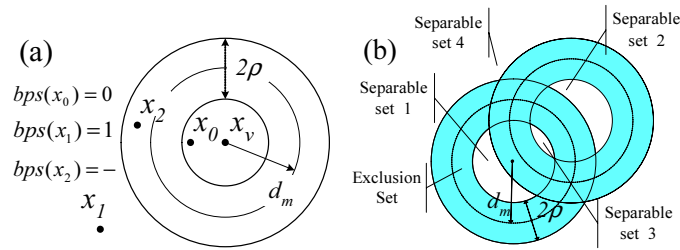


Figure 4.1: The *bps* split function (a) and the combination of two *bps* functions (b).

partition a data set into three subsets. The result of the following *bps* function gives a unique identification of the set to which the object x belongs:

$$bps(x) = \begin{cases} 0 & \text{if } d(x, x_v) \leq d_m - \rho \\ 1 & \text{if } d(x, x_v) > d_m + \rho \\ - & \text{otherwise} \end{cases}$$

The subset of objects characterized by the symbol '-' is called the *exclusion set*, while the subsets of objects characterized by the symbols 0 and 1 are the *separable sets*, because any range query with radius not larger than ρ cannot find qualifying objects in both the subsets.

More separable sets can be obtained as a combination of *bps* functions, where the resulting exclusion set is the union of the exclusion sets of the original split functions. Furthermore, the new separable sets are obtained as the intersection of all possible pairs of separable sets of the original functions. Figure 4.1b gives an illustration of this idea for the case of two split functions. The separable sets and the exclusion set form the separable buckets and the exclusion bucket of one level of the structure, respectively.

Naturally, the more separable buckets we have, the larger the exclusion bucket is. For large exclusion bucket, the structure allows an additional level of splitting by applying a new set of split functions on the exclusion bucket of the previous level. The exclusion bucket of the last level forms the exclusion bucket of the whole structure. The ρ -split functions of individual levels should be different but they must use the same ρ . Moreover, by using different number of split functions (generally decreasing with the level), the structure can have different number of buckets at individual levels. In order to deal with overflow problems and growing files, buckets are implemented as *elastic buckets* and consist of the necessary number of fixed-size blocks (pages) – basic disk access units.

Due to the mathematical properties of the split functions precisely defined in [16], the range queries up to a radius ρ are solved by accessing at most one bucket per level, plus the exclusion bucket of the whole structure. This can intuitively be comprehended by the fact that an arbitrary object belonging to a separable bucket is at distance at least 2ρ from any object of other separable

bucket of the same level. With additional computational effort, the structure executes range queries of radii greater than ρ . The structure is able to support other similarity queries, such as nearest-neighbor queries and similarity joins.

4.4 Future Plans

The proposed structure has already been designed and its working prototype has been implemented. Algorithms for range search and nearest-neighbor search queries were implemented. In Autumn Term 2002, several join algorithms will be outlined and implemented. Their properties will be evaluated and compared. In Spring Term 2003, we will concentrate on developing methodologies that would support an optimal design of the proposed structure for specific applications, that is, specifying variable parameters of the proposed index structure. The proposed structure and its properties will be described in the Ph.D. thesis.

Bibliography

- [1] F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3), 1991.
- [2] R. Baeza-Yates, W. Cunto, U. Manber, S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, p. 198-212, 1994.
- [3] R. Baeza-Yates. Searching: an algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, vol. 37, p. 331-359. Marcel Dekker Inc., 1997.
- [4] T. Bozkaya, M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. ACM SIGMOD International Conference on Management of Data*, SIGMOD Record 26(2), p. 357-368, 1997.
- [5] T. Bozkaya, M. Ozsoyoglu. Indexing Large Metric Spaces for Similarity Search Queries. *ACM TODS*, 24(3):361-404, 1999.
- [6] S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference on Very Large Databases (VLDB'95)*, p. 574-584, 1995.
- [7] E. Bugnion, S. Fhei, T. Roos, P. Widmayer, F. Widmer. A spatial index for approximate multiple string matching. In *Proc. 1st South American Workshop on String Processing (WSP'93)*, p. 43-53, 1993.
- [8] W. Burkhard, R. Keller. Some approaches to best-matching file searching. *Comm. of the ACM*, 16(4):230-236, 1973.

- [9] B. Bustos, G. Navarro, E. Chavez. Pivot Selection Techniques for Proximity Searching in Metric Spaces. *Proceedings of the XXI Conference of the Chielan Computer Science Society (SCCC01)*, IEEE CS Press, p. 33-40, 2001.
- [10] E. Chavez, J. L. Marroquin, G. Navarro. Overcoming the curse of dimensionality. In *European Workshop on Content-Based Multimedia Indexing (CBMI'99)*, p. 57-64, 1999.
- [11] E. Chavez, J. L. Marroquin, G. Navarro. Fixed Queries Array: A Fast and Economical Data Structure for Proximity Searching. *Multimedia Tools and Applications (MTAP)*, 14(2):113-135, Kluwer, 2001.
- [12] E. Chavez, G. Navarro, R. A. Baeza-Yates, J. L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3): 273-321, 2001.
- [13] T. Chiueh. Content-Based Image Indexing. *Proceedings of the 20th VLDB Conference*, p. 582-593, 1994.
- [14] P. Ciaccia, M. Patella, P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. 23rd Conference on Very Large Databases (VLDB'97)*, p. 426-435, 1997.
- [15] V. Dohnal, C. Gennaro, P. Savino, P. Zezula. Separable Splits in Metric Data Sets. *Proceedings of 9-th Italian Symposium on Advanced Database Systems*, Venice, Italy, June 2001, p. 45-62, LCM Selecta Group - Milano.
- [16] V. Dohnal, C. Gennaro, P. Savino, P. Zezula. D-Index: Distance Searching Index for Metric Data Sets. To appear in *Multimedia Tools and Applications*, Kluwer, 2002.
- [17] V. Dohnal, C. Gennaro, P. Zezula. A Metric Index for Approximate Text Management. To appear in *Proceedings of IASTED International Conference on Information Systems and Databases (ISDB'02)*, Tokyo, Japan, September 2002.
- [18] J. L. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, W. Niblack. Efficient Color Histogram Indexing for Quadratic Form Distance Functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17(7): 729-736 (1995).
- [19] D. P. Huttenlocher, G. A. Klanderman, W. J. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(3):850-863, September 1993.
- [20] J. L. Kelly. *General Topology*, D. Van Nostrand, New York, 1955.
- [21] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377-439, 1992.

- [22] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8-17, 1965.
- [23] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 2001, 33(1):31-88.
- [24] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175-179, 1991.
- [25] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, p. 311-321, 1993.
- [26] P. N. Yianilos. Excluded Middle Vantage Point Forests for Nearest Neighbor Search. In *DIMACS Implementation Challenge, ALNEX'99*, Baltimore, MD, 1999.
- [27] P. Zezula, P. Ciaccia, F. Rabitti. M-tree: A dynamic index for similarity queries in multimedia databases. *TR 7, HERMES ESPRIT LTR Project*, 1996. Available at URL <http://www.ced.tuc.gr/hermes/>.

A. Current Results of My Study

Last two years of my study at Faculty of Informatics, Masaryk University, Brno, I was interested in the area of indexing techniques of multimedia data. The first year, I concentrated on becoming familiar with the basic principles and existing structures for indexing metric spaces. I focused on principles of partitioning of metric spaces into separable subsets. Several principles were proposed and compared in [15]. The results have also been presented on the poster session of Seminar on Informatics.

During the last year, the design of the new metric index structure (D-Index) was developed and the first prototype was implemented. This prototype was tested on synthetic data sets to validate its properties. In Autumn Term 2001, the range search and nearest-neighbor search algorithms were implemented. Properties of these algorithms and the whole structure were tested on real world data sets and the results are summarized and presented in [16]. Principles of the D-Index were also presented as a poster in Seminar on Informatics.

In Spring Term 2002, the principle of storing objects of the D-Index has been revised and evaluated. A preliminary version of algorithm for similarity join operation was designed and implemented to the D-Index. The similarity join was evaluated on large text string collections with *edit distance* metric function and the results are presented in [17]. Currently, we are working on more sophisticated algorithm for similarity joins. In this term, the paper concerning basic principles and algorithms for searching in metric spaces was prepared for presentation on Seminar on Informatics.

During my study, I cooperate with my supervisor Pavel Zezula and my colleagues Claudio Gennaro and Pasquale Savino from ISTI-CNR, Pisa, Italy. I am also supervising a master thesis by Milan Pindryč.

B. Summary / Souhrn

Summary: The problem of searching the elements of a set which are close to a given query element under some similarity criterion has a vast number of applications in many branches of computer science, from pattern recognition to textual and multimedia information retrieval. We are interested in the general case where the similarity criterion defines a *metric space*. As the complexity and volume of modern data grows, there is an increasing need of index structures which support disk memory storages. Unfortunately, many existing index structures operate in the main memory only thus they are not able to deal with large, possibly distributed, collections of data. Existing index structures suffer from being intrinsically static, which limits their applicability in dynamic environments that are permanently a subject of change. In such environments, updates (insertions or deletions) are inevitable and the costs of update are becoming a matter of concern. This work is aimed at the development of a novel similarity search index structure that combines the clustering technique and the pivot-based strategy to speed up execution of similarity search queries. The proposed structure supports easy insertions/deletions and bounded search costs. It is suitable for distributed and parallel implementations.

Souhrn: Problém vyhledávání prvků množiny, které jsou vzhledem k jistému podobnostnímu kritériu blízké danému dotazovému prvku, se vyskytuje v různých odvětvích počítačové vědy (computer science), od aplikací rozpoznávání vzorů až po vyhledávání v textových a multimediálních datech. V obecném případě je vhodnou abstrakcí podobnostního kritéria *metrický prostor*. S rostoucí složitostí a objemem moderních dat se zvyšuje potřeba indexovacích struktur, které umožňují použití diskových úložišť. Bohužel mnoho existujících indexovacích struktur podporuje použití pouze operační paměti, tedy není schopno pracovat s velkými, někdy i distribuovanými, databázemi. Současné indexovací struktury byly navrženy jako statické, což omezuje jejich použití v dynamických prostředích. Aktualizace dat jsou v těchto prostředích nevyhnutelné a tím se náklady na ukládání nebo mazání stávají významnými a je nutné je minimalizovat. Tato práce se zaměřuje na vytvoření nové indexovací struktury pro podobnostní hledání, která kombinuje techniku shlukování (clustering) a pivoťovací strategii (pivot-based strategy) s cílem zrychlit provádění podobnostních dotazů. Navrhovaná struktura má nízké náklady na ukládání/mazání dat a shora omezuje cenu vyhledávání, je schopna pracovat v distribuovaných a paralelních prostředích.