Searching in Metric Spaces *

Edgar Chávez† Gonzalo Navarro‡ Ricardo Baeza-Yates‡ José L. Marroquín§

Abstract

The problem of searching the elements of a set which are close to a given query 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 rather general case where the similarity criterion defines a *metric space*, instead of more restricted cases of vector spaces. A large number of solutions have been proposed in different areas, in many cases without cross-knowledge. Because of this, the same ideas have been reinvented several times, and very different presentations have been given for the same approaches. We present a unified view of all the known proposals to organize metric spaces, so as to be able to understand them under a common framework. Most approaches turn out to be variations on a few different concepts. We organize those works in a taxonomy which allows us to devise new algorithms from combinations of concepts which were not noticed before because of the lack of communication between different communities. Some of the new techniques that appear as combinations are shown to be very competitive. We present experiments validating our results and comparing the existing approaches, so as to determine the best existing solutions for this problem. We finish with recommendations for practitioners and open questions for future development.

1 Introduction

Searching is a fundamental problem in computer science, present in virtually every computer application. Simple applications pose simple search problems, while a more complex application will require, in general, a more sophisticated form of searching.

The search operation has been traditionally applied to "structured data", i.e. numerical or alphabetical information which is searched for exactly. That is, a search query is given and the number or string which is *exactly equal* to the search query is retrieved. Traditional databases are built around the concept of exact searching: the database is divided into *records*, each record having a fully comparable *key*. Queries to the database return all the records whose keys match the search key. More sophisticated searches such as range queries on numerical keys or prefix searching on alphabetical keys still rely on the concept that two keys are or are not equal, or that there is a total linear order on the keys. Even in recent years, when databases have included the ability to

*This project has been partially supported by CYTED VII.13 AMYRI Project.

†Escuela de Ciencias Físico-Matemáticas, Universidad Michoacana. Edificio "B", Ciudad Universitaria, Morelia, Mich. México 58000. elchavez@zeus.ccu.umich.mx.

‡Depto. de Ciencias de la Computación, Universidad de Chile. Blanco Encalada 2120, Santiago, Chile. {gnavarro,rbaeza}@dcc.uchile.cl.

§Centro de Investigación en Matemáticas (CIMAT). Callejón de Jalisco S/N, Valenciana, Guanajuato, Gto. México 36000. jlm@fractal.cimat.mx.

store new data types such as images, the search has still been done on a predetermined number of keys of numerical or alphabetical types.

With the evolution of information and communication technologies, unstructured repositories of information have emerged. Not only new data types such as free text, images, audio and video have to be queried, but also it is not possible anymore to structure the information in keys and records. Such structuring is very difficult (either manually or computationally) and restricts beforehand the types of queries that can be posed later. Even when a classical structuring is possible, new applications such as data mining require to access the database by any field, not only those marked as "keys". Hence, new models for searching in unstructured repositories are needed.

The above scenarios require more general search algorithms and models than those classically used for simple data. A unifying concept is that of "similarity searching" or "proximity searching", i.e. searching for database elements which are similar or close to a given query element[1]. Similarity is modeled with a distance function that satisfies the triangular inequality, and the set of objects is called a *metric space*. Since the problem has appeared in many diverse areas, solutions have appeared in many unrelated fields, such as statistics, artificial intelligence, databases, computational biology, pattern recognition and data mining, to name a few. Since the current solutions come from so diverse fields, it is not surprising that the same solutions have been reinvented many times, that obvious combinations of solutions have not been noticed, and that no thorough analytical or experimental comparisons have been done. More importantly, there have been no attempts to conceptually unify all those solutions.

In many applications, the problem in general metric spaces is translated to a vector space (i.e. objects are represented as k-dimensional points with some geometric interpretation of similarity). This is because the concept of similarity searching appeared in vector spaces in the first place. This is a natural extension of the problem of searching the closest point in the plane. In this framework optimal algorithms (on the database size) exist in both average and worst case [10] for closest point search. Search algorithms for vector spaces are called *spatial access methods* (SAM). Among the most popular are kd-trees [8, 9], R-trees [36] and the more recent X-trees [11] (see [57, 35] for good surveys). Unfortunately the existing algorithms are very sensitive to the vector space dimension. Closest point search algorithms have an exponential dependency on the dimension of the space (this is called the "curse of dimensionality"). In practical terms, it is considered that the problem becomes intractable in more than 20 dimensions. For this reason, several authors have proposed the use of *distance based* indexing techniques, which use *only* the distance between points and avoid any reference to coordinates, in an attempt to avoid the dimensionality curse. This may be particularly effective when an intrinsically low dimensional set is embedded into an artificially high dimensional space (e.g. a plane in a three dimensional space). Hence, one resorts to general metric spaces not only when the problem has not a coordinate structure, but also when the number of such coordinates is very high.

With regard to general metric spaces, the problem has been tackled from a variety of viewpoints. The general goal is to build a data structure (or *index*) to reduce the number of distance evaluations at query time, since the distance is assumed to be expensive to compute. Some important advances have been done, mainly in the database community where a number of distance-based indices and

[1] The term "approximate searching" is also used, but it is misleading and we use it here only when referring to approximation algorithms.

data structures have been developed and studied from different perspectives. In this particular case, people is concerned additionally with the I/O problem, e.g. efficient disk arrangements to access a (perhaps huge) database [26, 47].

The main goal of this work is to present a unifying framework to describe and analyze all the existing solutions to this problem. We show that all the existing indexing algorithms for proximity searching consist in building a set of equivalence classes, discarding some classes, and searching exhaustively the rest. As a consequence of the analysis we are able to build a taxonomy on the existing algorithms for proximity search, to classify them according to their essential features, and to analyze their effectiveness. We are able to identify essentially similar approaches, to point out combinations of ideas which have not previously been noticed, and to identify the main open problems in this area. We also present experimental results that help to validate our assertions, to answer some open questions, and to determine the best existing choices for practitioners. As a byproduct of this work, some unnoticed combinations of previous ideas are shown to be very competitive.

We remark that we are concerned with the *essential* features of the search algorithms for general metric spaces. That is, we try to extract the basic features from the wealth of existing solutions, so as to be able to categorize and analyze them under a common framework. We focus mainly on the number of distance evaluations needed to execute *range queries* (i.e. with fixed tolerance radius), which are the most basic ones. However, we also pay some attention to the total CPU time, as well as the time and space cost to build the indices. There are many other features which we are not considering in order to keep our scope reasonably bounded, and which deserve a separate study, such as

- dynamic capabilities of the indices, i.e. how to add and delete elements once the index is built, since most indices can be naturally updated and considering this adds extra complications;
- I/O performance of the indices, since we aim at the essential features and this would introduce an extra level of complication;
- vector spaces, as there exist already good surveys on this particular case [57, 35];
- closest point search, as the algorithms are built systematically over range queries (we explain how, however);
- sub-queries (i.e. searching a small element inside a larger element) since the solutions are basically the same after a domain-dependent transformation is done.

This paper is organized as follows. In Section 2 we present a number of applications that motivate proximity searching. In Section 3 we explain some basic concepts. In Section 4 we survey all the search algorithms we are aware of. In Section 5 we present our unifying model. In Section 6 we present a taxonomy on the current solutions based on the unifying model and find new combinations not previously noticed. In Section 7 we present experiments validating our ideas and comparing the techniques. Finally, in Section 8 we present our conclusions, give advices for practitioners, and point out the main open problems in this area.

2 Motivating Applications

We present now a sample of applications where the concept of proximity searching appears. Since we have not presented a formal model yet, we do not try to explain the connections between the different applications. We rather delay this discussion to Section 3.

2.1 Query by Content in Structured Databases

In general, the query posed to a database presents a *piece* of a record of information, and it needs to retrieve the *entire* record. In the classical approach, the piece presented is fixed (the *key*). Moreover, it is not allowed to search with an incomplete or an erroneous key. On the other hand, in the more general approach required nowadays the concept of searching with a key is generalized to searching with an arbitrary subset of the record, allowing or not errors.

This type of searching have deserved a number of names, for example *range query*, *query by content* or *proximity searching*. It is of use in data mining (where the interesting parts of the record cannot be predetermined), when the information is not precise, when we are looking for a range of values, when the search key may have errors (e.g. a misspelled word), etc.

A general solution to the problem of range queries by any record field is the *grid file* [46]. The domain of the database is seen as a hyper-rectangle of k dimensions (one per record field), where each dimension has an ordering according to the domain of the field (numerical or alphabetical). Each record present in the database is considered as a point inside the hyper-rectangle. A query specifies a sub-rectangle (i.e. a range along each dimension), and all the points inside the specified query are retrieved. This does not address the problem of searching on non-traditional data types, nor allowing errors that cannot be recovered with a range query. However, it converts the original search problem to a problem of obtaining, in a given space, all the points "close" to a given query point. Grid files are essentially a disk organization technique to efficiently retrieve range queries in secondary memory.

2.2 Query by Content in Multimedia Objects

New data types such as images, fingerprints, audio and video (called "multimedia" data types) cannot be meaningfully queried in the classical sense. Not only they cannot be ordered, but they cannot even be compared for equality. No application will be interested in searching an audio segment exactly equal to a given one. The probability that two different images are pixel-wise equal is negligible unless they are digital copies of the same source. In multimedia applications, all the queries ask for objects similar to a given one. Some example applications are face recognition, fingerprint matching, voice recognition, and in general multimedia databases [1].

Think for example in a repository of images. Interesting queries are of the type "*find an image of a lyon with a savanna background*". If the repository is tagged, and each tag contains a full description of what is inside the image, then our example query can be solved with a classical scheme. Unfortunately, such a classification cannot be done automatically with the available image processing technology. Object recognition in real world scenes is still in an immature state to perform such complex tasks. Moreover, we cannot predict all the possible queries that will be posed so as to tag the image for every possible query. An alternative to automatic classification

consists in considering the query as an *example* image, so that the system searches all the images similar to the query. This can be built inside a more complex feedback system where the user approves or rejects the images found, and a new query is submitted with the approved images. It is also possible that the query is just part of an image and the system has to retrieve the whole image.

These approaches are based on the definition of a similarity function among objects. Those functions are provided by an expert, but they pose no assumptions on the type of queries that can be answered. In many cases, the distance is obtained via a set of k "features" which are extracted from the object (e.g. in an image a useful feature is the average color). Then each object is represented as its k features, i.e. a point in a k-dimensional space, and we are again in a case of range queries on vector spaces.

There is a growing community of scientists deeply involved with the development of such similarity measures [20, 12, 13].

2.3 Text Retrieval

Although not considered a multimedia data type, unstructured text retrieval poses similar problems as multimedia retrieval. This is because textual documents are in general not structured to easily provide the desired information. Text documents may be searched for strings that are present or not, but in many cases they are searched for semantic concepts of interest. For instance, an ideal scenario would allow to search a text dictionary for a concept such as *"to free from obligation"*, retrieving the word *"redeem"*. This search problem cannot be properly stated with classical tools.

A large community of researchers has been working on this problem from a long time ago [48, 34, 7]. A number of measures of similarity have emerged. The problem is basically solved by retrieving documents similar to a given query. The user can even present a document as a query, so that the system finds similar documents. Some similarity approaches are based on mapping a document to a vector of real values, so that each dimension is a vocabulary word and the relevance of the word to the document (computed using some formula) is the coordinate of the document along that dimension. Similarity functions are then defined in that space. Notice however that the dimensionality of the space is very high (thousands of dimensions).

Another problem related to text retrieval is spelling. Since huge text databases with low quality control are emerging (e.g. the Web), and typing, spelling or OCR (optical character recognition) errors are commonplace in the text and the query, we have that documents which contain a misspelled word are no longer retrievable by a correctly written query. Models of similarity among words exist (variants of the "edit distance" [49]) which capture very well those kind of errors. In this case, we give a word and want to retrieve all the words close to it. Another related application is spelling checkers, where we look for close variants of the misspelled word.

In particular, OCR can be done using a low-level-classifier, so that misspelled words can be corrected using the edit distance to find promising alternatives to replace incorrect words.

2.4 Computational Biology

ADN and protein sequences are the basic object of study in molecular biology. As they can be modeled as texts, we have the problem of finding a given sequence of characters inside a longer

sequence. However, an exact match is unlikely to occur, and computational biologists are more interested in finding parts of a longer sequence which are similar to a given short sequence. The fact that the search is not exact is due to minor differences in the genetic streams that describe beings of the same or closely related species. The measure of similarity used is related to the probability of mutations such as reversals of pieces of the sequences and other rearrangements [56, 49].

Other related problems are to build phylogenetic trees (a tree sketching the evolutionary path of the species), to search patterns for which only some properties are known, and others.

2.5 Pattern Recognition and Function Approximation

A simplified definition of pattern recognition is the construction of a function approximator. In this formulation of the problem one has a finite sample of the data, and each data sample is labeled as belonging to a certain class. When a fresh data sample is provided, the system is required to label this new sample with one of the known data labels. In other words, the classifier can be thought of as a function defined from the object (data) space to the set of labels. In this sense all the classifiers are considered function approximators.

If the objects are m-dimensional vectors of real numbers then a natural choice is neural nets and fuzzy function approximators. Another popular universal function approximator, the k-nearest neighbor classifier, consists in finding the k objects nearest to the unlabeled sample, and assigning to this sample the label having majority among the k nearest objects. Opposed to neural nets and fuzzy classifiers, the k-nearest neighbor rule has zero training time, but if no indexing algorithm is used it has linear complexity [31].

Other applications of this universal function approximator are density estimation [30] and reinforcement learning [52]. In general, any problem where we want to infer a function based on a finite set of samples is a potential application.

2.6 Audio and Video Compression

Audio and video transmission over a narrow-band channel is an important problem, for example in Internet-based audio and video conferencing. A frame (a static picture in a video, or a fragment of the audio) can be thought of as formed by a number of (possibly overlapped) subframes (16×16 subimages in a video, for example). In a very succinct description, the problem can be solved by sending the first frame as-is and for the next frames sending only the subframes having a significative difference from the previously sent subframes. This description encompasses the MPEG standard.

The algorithms use in fact a subframe buffer. Each time a frame is about to be sent it is searched (with a tolerance) in the subframe buffer and if it is *not* found then the entire subframe is added to the buffer. If the subframe is found then only the index of the similar frame found is sent. This implies, naturally, that a fast similarity search algorithm has to be incorporated to the server to maintain a minimum of frames-per-second rate.

3 Basic Concepts

All the applications presented in the previous section share a common framework, which is in essence to find close objects, under some suitable similarity function, among a finite set of elements. In

this section we present the formal model comprising all the above cases.

3.1 Metric Spaces

We introduce now the basic notation for the problem of satisfying proximity queries and for the model used to group and analyze the existing algorithms.

The set $\mathbb{X}$ will denote the universe of *fair* or *valid* objects. A finite subset of them, $\mathbb{U}$, of size $n = |\mathbb{U}|$, is the set of objects where we search. $\mathbb{U}$ will be called the *dictionary*, *database* or simply our set of *objects* or *elements*. The function

$$d \ : \ \mathbb{X} \times \mathbb{X} \longrightarrow \mathbb{R}$$

will denote a measure of "distance" between objects (i.e. the smaller the distance, the closer or more similar are the objects). Distance functions have the following properties:

$(p1)$ $\forall x, y \in \mathbb{X},\ d(x,y) \geq 0$ positiveness,

$(p2)$ $\forall x, y \in \mathbb{X},\ d(x,y) = d(y,x)$ symmetry,

$(p3)$ $\forall x \in \mathbb{X},\ d(x,x) = 0$ reflexivity,

and in most cases

$(p4)$ $\forall x, y \in \mathbb{X},\ x \neq y \ \Rightarrow \ d(x,y) > 0$ strict positiveness.

The similarity properties enumerated above only ensure a consistent definition of the function, and cannot be used to save comparisons in a proximity query. If d is indeed a *distance*, i.e. if it satisfies

$(p5)$ $\forall x, y, z \in \mathbb{X},\ d(x,y) \ \leq \ d(x,z) + d(z,y)$ triangular inequality,

then the pair $(\mathbb{X}, d)$ is called a *metric space*.

If the distance function does not satisfy the strict positiveness property $(p4)$ then the space is called a *pseudo-metric space*. Although for simplicity we do not consider pseudo-metric spaces in this work, all the presented techniques are easily adapted to them by simply identifying all the objects at distance zero as a single object. This works because, if $(p5)$ holds, one can easily prove that $d(x,y) = 0 \Rightarrow \forall z, d(x,z) = d(y,z)$.

In some cases we may have a *quasi-metric*, where the symmetry property $(p2)$ does not hold. For instance, if the objects are corners in a city and the distance corresponds to how much a car must travel to move from one to the other, then the existence of one-way streets makes the distance asymmetric. There exist techniques to derive a new, symmetric, distance function from an asymmetric one, such as $d'(x,y) = d(x,y) + d(y,x)$. However, to be able to bound the search radius of a query when using the symmetric function we need specific knowledge of the domain.

Finally, we can relax the triangular inequality $(p5)$ to $d(x,y) \leq \alpha d(x,z) + \beta d(z,y) + \delta$, and after some scaling we can search in this space using the same algorithms designed for metric spaces. If the distance is symmetric we need $\alpha = \beta$ for consistency.

3.2 Proximity Queries

There are basically three types of queries of interest in metric spaces:

(a) ***Range or proximity query*** Retrieve all elements which are within distance r to q. This is, $\{u \in \mathbb{U} \ / \ d(q,u) \leq r\}$. We denote this query by $(q,r)_d$

(b) ***Nearest neighbor query*** Retrieve the closest elements to q in $\mathbb{U}$. This is, $\{u \in \mathbb{U} \ / \ \forall v \in \mathbb{U},\ d(q,u) \leq d(q,v)\}$. In some cases we are satisfied with one such element (in continuous spaces there is normally just one answer already). We can also give a maximum distance r^* such that if the closest element is at distance more than r^* we do not want anyone reported.

(c) ***k-Nearest neighbor query*** Retrieve the k closest elements to q in $\mathbb{U}$. This is, retrieve a set $A \subseteq \mathbb{U}$ such that $|A| = k$ and $\forall u \in A, v \in \mathbb{U} - A, d(q,u) \leq d(q,v)$. Note that in case of ties we are satisfied with any set of k elements satisfying the condition.

The most basic type of query is (a). The left part of Figure 1 illustrates a query on a set of points which will be our running example. We use $\mathbb{R}^2$ as our metric space for clarity.

A *proximity query* will be therefore a pair $(q,r)_d$ with q a novel element in $\mathbb{X}$ and r a real number indicating the *radius* (or tolerance) of the query. The set $\{u \in \mathbb{U},\ d(q,u) \leq r\}$ will be called the *outcome* of the proximity query. The query is indicated as a d-type query since for different distance functions there are different outcomes of the query.

The other two types of queries are normally solved using a variant of the range queries. For instance, the query of type (b) is normally solved as a range query where the radius r is initially infinite, and is reduced as closer and closer elements to the query are found. This is normally coupled with a heuristic that tries to obtain close elements as quickly as possible (as the problem is always easier for smaller radii). Queries of type (c) are normally solved as a variant of type (b), where the k closest elements are kept and the current r is the largest distance from those elements to the query (as farther elements are not of interest).

Another widely used algorithm for queries of type (b) or (c) based on those of type (a) is to search with fixed radii $r = 2^i\epsilon$, starting with $i = 0$ and increasing it until the desired number of elements (or more) lies inside the search radius $r = 2^i\epsilon$. Later, the radius is refined between $r = 2^{i-1}\epsilon$ and $r = 2^i\epsilon$ until the exact number of elements is included.

The total CPU time to evaluate a query can be split as

$$T \quad = \quad \#\text{ of distance evaluations} \ \times \ \text{complexity of } d() \quad + \quad \text{extra CPU time}$$

and we would like to minimize T. In many applications, however, evaluating $d()$ is so costly that the extra CPU time can be neglected. This is the model we use in this paper, and hence the *number* of distance evaluations performed will be the measure of the complexity of the algorithms. We can even allow a linear (but reasonable) amount of CPU work, as long as the number of distance computations is kept low. However, we will pay some marginal attention to the so-called *extra CPU time*.

It is clear that either type of query can be answered by examining the entire dictionary $\mathbb{U}$. In fact if we are not allowed to preprocess the data, i.e. to build an indexing data structure, then this exhaustive examination is the only way to proceed. An *indexing algorithm* is an off-line

procedure to build beforehand a data structure (or *index*) designed to save distance computations when answering proximity queries later. This data structure can be expensive to build, but this will be amortized by saving distance evaluations over many queries to the database. The aim is therefore to design efficient indexing algorithms to reduce the number of distance evaluations. All these structures work on the basis of discarding elements using the triangular inequality (the only property that allows saving distance evaluations).

3.3 Vector Spaces

If the elements of the metric space $(\mathbb{X}, d)$ are indeed tuples of real numbers (actually tuples of any field) then the pair is called a *finite dimensional vector space,* or vector space for short.

A k-dimensional vector space is a particular metric space where the objects are identified with k real-valued coordinates $(x_1, ..., x_k)$. There are a number of options for the distance function to use, but the most widely used is the family of L_s (or Minkowski) distances, defined as

$$L_s((x_1, ..., x_k), (y_1, ..., y_k)) = \left(\sum_{i=1}^{k} |x_i - y_i|^s \right)^{1/s}$$

The right part of Figure 1 illustrates some of these distances. For instance, the L_1 distance accounts for the sum of the differences along the coordinates. It is also called "block" or "Manhattan" distance, since in two dimensions it corresponds to the distance to walk between two points in a city of rectangular blocks. The points at the same distance r to a given point p form a box centered at the point and rotated 45 degrees, the distance between two opposite corners of the box being $2r$.

The L_2 distance is better known as "Euclidean" distance, as it corresponds to our notion of spatial distance. The points at the same distance r to a given point p form a sphere of diameter $2r$ centered at the point.

The other most used member of the family is L_∞, which corresponds to taking the limit of the L_s formula when s goes to infinity. The result is that the distance between two points is the *maximum* difference along a coordinate:

$$L_\infty((x_1, ..., x_k), (y_1, ..., y_k)) = \max_{i=1}^{k} |x_i - y_i|$$

and the points at the same distance r to a given point p form a box of side $2r$ centered at the point. This distance plays a special role in this survey.

4 Overview of Current Solutions

In this section we explain the existing indices to structure metric spaces. Since we have not yet developed the concepts of a unifying perspective, the description will be kept at an intuitive level, without any attempt to analyze why some ideas are better or worse.

We divide the presentation in four parts. The first one deals with data structures for *discrete* distance functions, that is, functions that deliver a small set of values. The second part corresponds to indices for *continuous* distance functions, where the set of alternatives is infinite or very large.

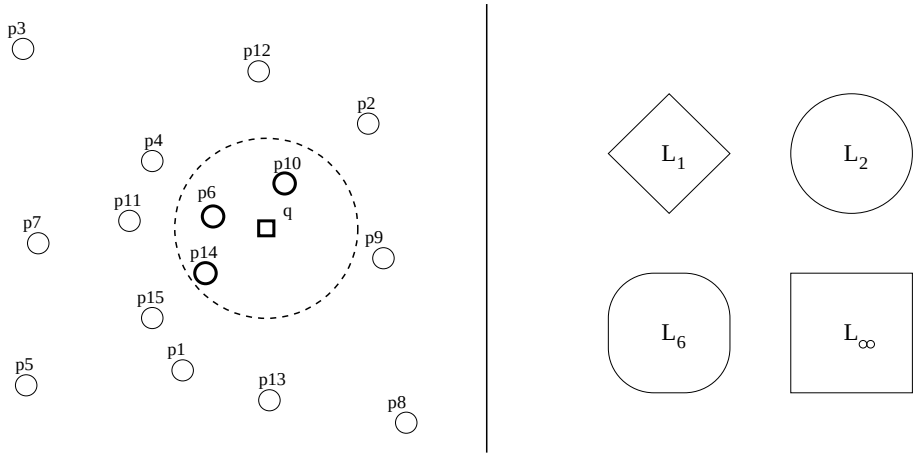

Figure 1: On the left, an example of a range query on a set of points. On the right, the set of points at the same distance to a center point, for different Minkowski distances.

Third, we consider other methods such as clustering and mapping to vector spaces. Finally, we briefly consider approximation algorithms for the problem.

4.1 Discrete Distance Functions

We start by describing data structures that apply to distance functions that return a small set of different values. At the end we show how to cope with the general case.

BKT Probably the first general solution to search in metric spaces was presented in [19]. They propose a tree (thereafter called Burkhard-Keller Tree, or BKT), which is suitable for discrete-valued distance functions. It is defined as follows: an arbitrary element $p \in \mathbb{U}$ is selected as the root of the tree. For each distance $i > 0$, we define $\mathbb{U}_i = \{u \in \mathbb{U}, d(u,p) = i\}$ as the set of all the elements at distance i to the root p. Then, for any nonempty $\mathbb{U}_i$, we build a child of p (labeled i), where we recursively build the BKT for $\mathbb{U}_i$. This process can be repeated until there is only one element to process, or until there are no more than b elements (and we store a *bucket* of size b). All the elements selected as roots of subtrees are called *pivots*.

To answer queries of type (a), where we are given a query q and a distance r, we begin at the root and enter into all children i such that $d(p,q) - r \leq i \leq d(p,q) + r$, and proceed recursively. If we arrive to a leaf (bucket of size one or more) we compare sequentially all its elements. Each time we perform a comparison (against pivots or bucket elements u) where $d(q,u) \leq r$, we report the element u.

The triangular inequality ensures that we cannot miss an answer. All the subtrees not traversed contain elements x which are at distance $d(x,p) = i$ from some node p, where $|d(p,q) - i| > r$. By the triangular inequality, $d(p,q) \leq d(p,x) + d(x,q)$, and therefore $d(x,q) \geq d(p,q) - d(p,x) > r$.

Figure 2 shows an example, where the point p_{11} has been selected as the root. We have built only the first level of the BKT for simplicity. A query q is also shown, and we have emphasized the

branches of the tree that would have to be traversed. In this and all the examples of this section we discretize the distances of our example, so that they return integer values.

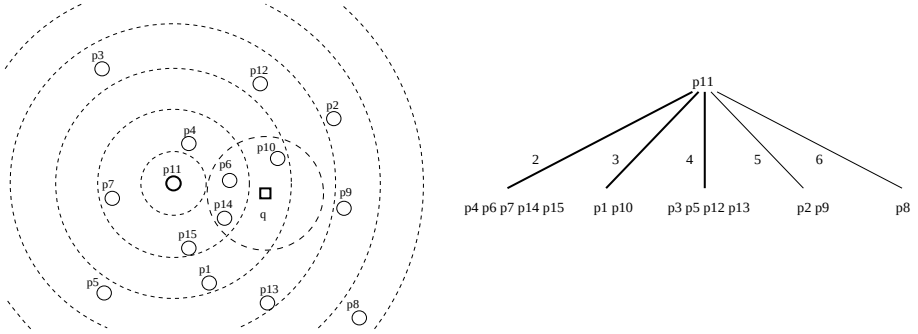

Figure 2: On the left, the division of the space obtained when p_{11} is taken as a pivot. On the right, the first level of a BKT with p_{11} as root. We also show a query q and the branches that it has to traverse. We have discretized the distances so they return integer values.

To answer queries of type (b), we begin at the root and measure $i = d(p, q)$. Now, we consider the edges labeled i, $i-1$, $i+1$, $i-2$, $i+2$, and so on, and proceed recursively in the children[2]. We begin with an estimation of the distance to the closest element, $r^* = \infty$, which is refined for each comparison we perform. Therefore, our exploration ends just after considering the branch $i + r^*$ (r^* is reduced along the process). At the end r^* is the distance to the closest neighbors and we have already seen all them.

Our analytical results on BKTs are extrapolated from those made for FQTs [5], which can be easily adapted to this case. The only difference is that the space overhead of BKTs is $O(n)$ because there is exactly one element of the set per tree node.

FQT A further development over BKTs is the "Fixed-Queries Tree" or FQTs [5]. This tree is basically a BKT where all the pivots stored in the nodes of the same level are the same (and of course do not necessarily belong to the set stored in the subtree). The actual elements are all stored at the leaves. The advantage of such construction is that some comparisons between the query and the nodes are saved along the backtracking that occurs in the tree. If we visit many nodes of the same level, we do not need to perform more than one comparison because all the pivots in that level are the same. This is at the expense of somewhat taller trees. FQTs are shown to be superior to BKTs in [5]. Under some simplifying assumptions (experimentally validated in the paper) they show that FQTs built over n elements are $O(\log n)$ height on average, are built using $O(n \log n)$ distance evaluations, and that the average number of distance computations is $O(n^\alpha)$, where $0 < \alpha < 1$ is a number that depends on the range of the search and on the structure of the space (this analysis is easy to extend to BKTs as well). The space complexity is superlinear since, unlike BKTs, it is not true that a different element is placed at each node of the tree. An upper bound is $O(n \log n)$ since the average height is $O(\log n)$.

[2] This order to traverse the children is just one alternative, other choices may be better. However, our experimental results show that this is the best ordering to index words under the edit distance (see Section 2.3).

FHQT In [5, 4], the authors propose a variant which is called "Fixed-Height FQT" (or FHQT for short), where all the leaves are at the same depth h, regardless of the bucket size. This makes some leaves deeper than necessary, which makes sense because we may have already performed the comparison between the query and the pivot of an intermediate level, therefore eliminating for free the need to consider the leaf. In [6] it is shown that by using $O(\log n)$ pivots, the search takes $O(\log n)$ distance evaluations (although the cost depends exponentially on the search radius r). The extra CPU time, i.e. number of nodes traversed, remains however $O(n^\alpha)$. The space, like FQTs, is somewhere between $O(n)$ and $O(nh)$. In practice the optimal $h = O(\log n)$ cannot be achieved because of space limitations.

FQA In [24], the Fixed Queries Array (FQA) is presented. The FQA, although not properly a tree, is no more than a compact representation of the FHQT. Imagine that an FHQT of fixed height h is built on the set. If we traverse the leaves of the tree left to right and put the elements in an array, the result is the FQA. For each element of the array we compute h numbers representing the branches to take in the tree to reach the element from the root (i.e. the distances to the h pivots). Each of these h numbers is coded in b bits and they are concatenated in a single (long) number so that the higher levels of the tree are the most significant digits.

As a result the FQA is sorted by the resulting hb-bits number, each subtree of the FHQT corresponds to an interval in the FQA, and each movement in the FHQT is simulated with two binary searches in the FQA (at $O(\log n)$ extra CPU cost factor, but no extra distances are computed). There is a similarity between this idea and suffix trees versus suffix arrays [34]. This idea of using less bits to represent the distances appeared also in the context of vector spaces [14].

Using the same memory, the FQA simulation is able to use much more pivots than the original FHQT, which improves the efficiency. The b bits needed by each pivot can be lowered by merging branches of the FHQT, as suggested for FQTs in [5] for the case of distance functions with many different outcomes. This allows to use even more pivots with the same space usage. For reasons that are made clear later, the FQA is also called FMVPA in this work.

Figure 3 shows an arbitrary BKT, FQT, FHQT and FQA built on our set of points. Notice that, while in the BKT there is a different pivot per node, in the others there is a different pivot per level, the same for all the nodes of that level.

Hybrid In [51], the use of more than one element per node of the tree is proposed. Those k elements allow to eliminate more elements per level at the cost of doing more distance evaluations. The same effect would be obtained if we had a mixture between BKTs and FQTs, so that for k levels we had fixed keys per level, and then we allowed a different key per node of the level $k+1$, continuing the process recursively on each subtree of the level $k+1$. The authors conjecture that the pivots should be selected to be outside the clusters.

Adapting to continuous functions If we have a continuous distance or if it gives too many different values, it is not possible to have a children of the root for any such value. If we did that, the tree would degenerate into a flat tree of height 2, and the search algorithm would be almost like sequential searching for the BKT and FQT. FHQTs and FQAs do not degenerate in this sense, but they loose they sublinear extra CPU time.

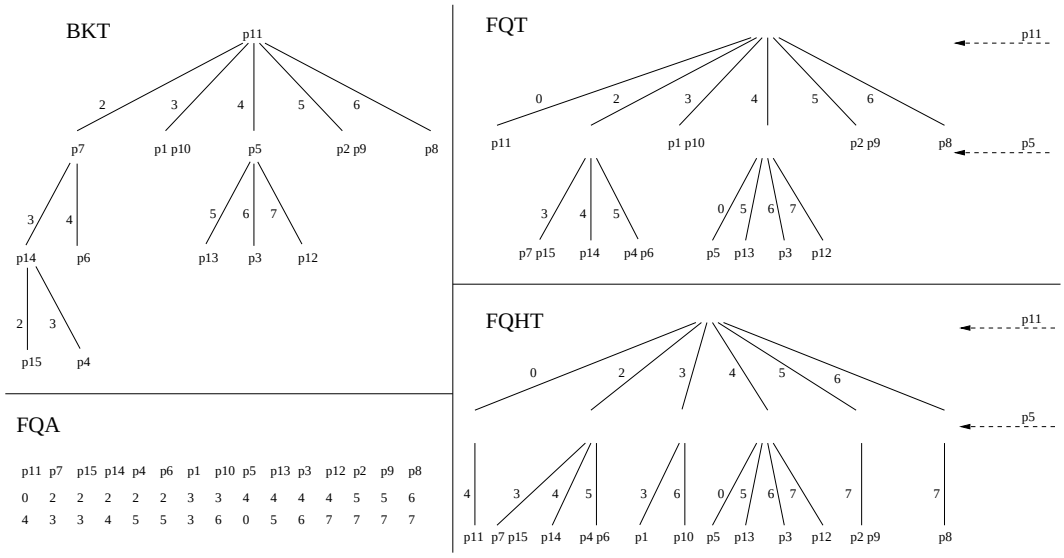

Figure 3: Example BKT, FQT, FHQT and FQA for our set of points. We use $b = 2$ for the BKT and FQT, and $h = 2$ for FHQT and FQA.

In [5] the authors mention that the structures can be adapted to a continuous distance by assigning a range of distances to each branch of the tree. However, they do not specify how to do this. Some approaches explicitly defined for continuous functions are explained later (VPTs and others), which assign the ranges trying to leave the same number of elements at each partition.

4.2 Continuous Distance Functions

We present now the data structures designed for the continuous case. They can be used also for discrete spaces with virtually no modifications.

VPT The first approach designed for continuous distance functions is called "Metric Trees" in [54]. A more complete work on the same idea [59] calls them "Vantage-Point Trees" or VPTs. They build a binary tree recursively, taking any element as the root p and taking the *median* of the set of all distances, $M = median\{d(p,u) \ / \ u \in \mathbb{U}\}$. Those elements u such that $d(p,u) \leq M$ are inserted into the left subtree, while those such that $d(p,u) > M$ are inserted into the right subtree. The VPT takes $O(n)$ space and is built in $O(n \log n)$ worst case time, since it is balanced. To solve a query of type (a) in this tree, we measure $d = d(q,p)$. If $d - r \leq M$ we enter into the left subtree, and if $d + r > M$ we enter into the right subtree (notice that we can enter into both subtrees). We report every element considered which is close enough to the query. See Figure 4.

Queries of type (b) can be solved by refining an estimation of the largest distance as before and exploring subtrees in any heuristically promising ordering. One is proposed in [53].

The query complexity is argued to be $O(\log n)$ in [59], but as they point out, this is true only for very small search radii, too small to be an interesting case.

In trees for discrete distance functions, the exact distance between an element in the leaves and

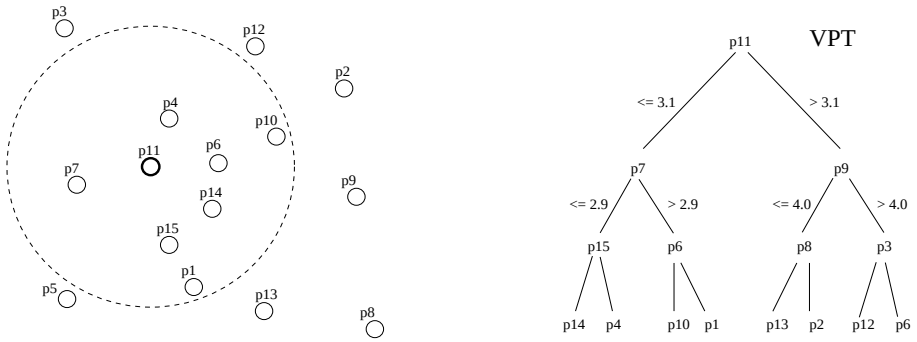

Figure 4: Example VPT with root p_{11}. We plot the radius M used for the root. For the first levels we show explicitly the radii used in the tree.

any pivot in the path to the root can be inferred. However, here we only know that the distance is larger or smaller than M. Unlike the discrete case, it is possible that we arrive to an element in a leaf which we do not need to compare, but the tree has not enough information to discover that. Some of those exact distances lost can be stored explicitly, as proposed in [59], to prune more elements before checking them. Finally, the author of [59] considers the problem of pivot selection and argue that it is better to take elements far away from the set.

A similar structure, easier to update, is presented in [26]. This structure, the M-tree, is designed for secondary memory and allows overlaps in the areas covered (i.e. a point may belong to more than one partition). This idea is also present in R-trees [36] for vector spaces.

MVPT The VPT can be extended to m-ary trees by using the $m-1$ uniform percentiles instead of just the median. This is suggested in [16, 15]. In [15], the "Multi-Vantage-Point Tree" (MVPT) is presented. They propose the use of many elements in a single node, much as in [51]. It can be seen that the space is $O(n)$, since each internal node needs to store the m percentiles but the leaves do not. The construction time is $O(n \log n)$ if we search the m percentiles hierarchically at $O(n \log m)$ instead of $O(mn)$ cost. The authors show experimentally that the idea of m-ary trees slightly improves over VPTs (and not in all cases), while a larger improvement is obtained by using many pivots per node. The analysis of query time for VPTs can be extrapolated to MVPTs in a straightforward way.

VPF Another generalization of the VPT is given by the VPF (shorthand for Excluded Middle Vantage Point Forest) [60]. This algorithm is designed for radii limited nearest neighbor search (a query of type (b) with a maximum radius r^*). The method consists in excluding, at each level, the elements at intermediate distances to their pivot (this is the most populated part of the set): if r_0 and r_n stand for the closest and farthest elements to the pivot p, the elements $u \in \mathbb{U}$ such that $d(p, r_0) + \delta \leq d(p, u) \leq d(p, r_n) - \delta$ are excluded from the tree. A second tree is built with the excluded "middle part" of the first tree, and so on to obtain a forest. With this idea they eliminate the backtracking when searching with a radius $r^* \leq (r_n - r_0 - 2\delta)/2$, and in return they have to search all the trees of the forest. The VPF, of $O(n)$ size, is built using $O(n^{2-\rho})$ time and answers

queries in $O(n^{1-\rho} \log n)$ distance evaluations, where $0 < \rho < 1$ depends on r^*. Unfortunately, to achieve $\rho > 0$, r^* has to be too small to be of use in high dimensions.

GHT Another proposal of [54] is the "Generalized-Hyperplane Tree" (GHT). This is a binary tree built recursively as follows. At each node, two pivots p_1 and p_2 are selected. The elements closer to p_1 than to p_2 go into the left subtree and those closer to p_2 into the right subtree. To answer queries of type (a) in this tree, we evaluate $r_1 = d(q, p_1)$ and $r_2 = d(q, p_2)$, and enter into the left subtree if $r_1 - r < r_2 + r$ and into the right subtree if $r_2 - r \leq r_1 + r$. Again, it is possible to enter into both subtrees. The reporting is done as always, as well as the extension to handle queries of type (b). In [54] it is argued that this could work better than VPTs in high dimensions. To avoid performing two distance evaluations at each node, it is proposed in [18] to reuse one of the pivots of the previous level. Figure 5 illustrates the first step of the tree construction.

No analysis is given in [54], but we obtain it by specializing the more general GNATs.

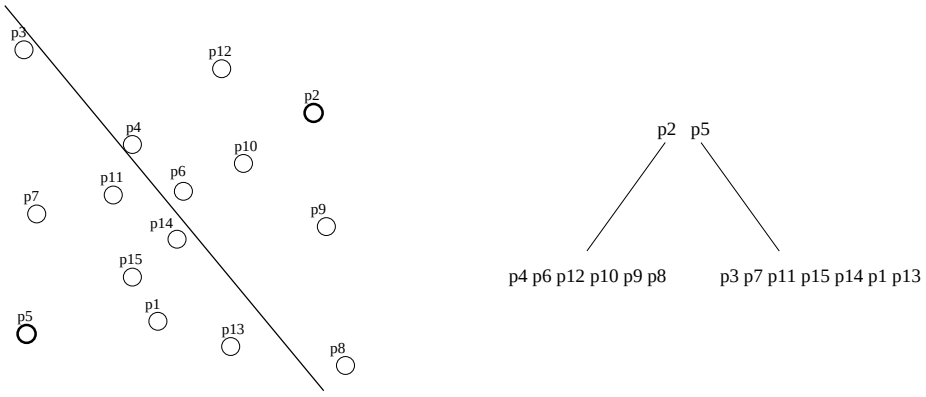

Figure 5: Example of the first level of a GHT.

GNAT The GHT is extended in [16] to an m-ary tree, called GNAT (Geometric Near-neighbor Access Tree), keeping the same essential idea. We select, for the first level, m pivots $p_1 ... p_m$, and define $\mathbb{U}_i = \{u \in \mathbb{U}, d(p_i, u) < d(p_j, u), \forall j \neq i\}$. That is, $\mathbb{U}_i$ are the elements closer to p_i than to any other p_j. From the root, m children numbered $i = 1..m$ are built, each one recursively with a GNAT for $\mathbb{U}_i$. They also add information at each subtree about the maximum distance between p_i and an element in $\mathbb{U}_i$, to increase pruning. Finally, they give some criteria to select the p_i elements far enough. Figure 6 shows a simple example of the first level of a GNAT. Notice the relationship between this idea and a Voronoi-like partition of a vector space [3].

The authors use a gross analysis to show that the tree takes $O(nm^2)$ space and is built in close to $O(nm \log n)$ time. Experimental results show that the GHT is worse than the VPT, which is only beaten with GNATs of arities between 50 and 100. Finally, they mention that the arities of the subtrees could depend on their depth in the tree, but give no clear criteria to do this. A very similar structure is later analyzed in [28] under some assumptions on the volume of the data set, and it is shown that the query time is $O(\text{polylog } n)$, where the degree of the polynomial depends in a complex way on the structure of the space.

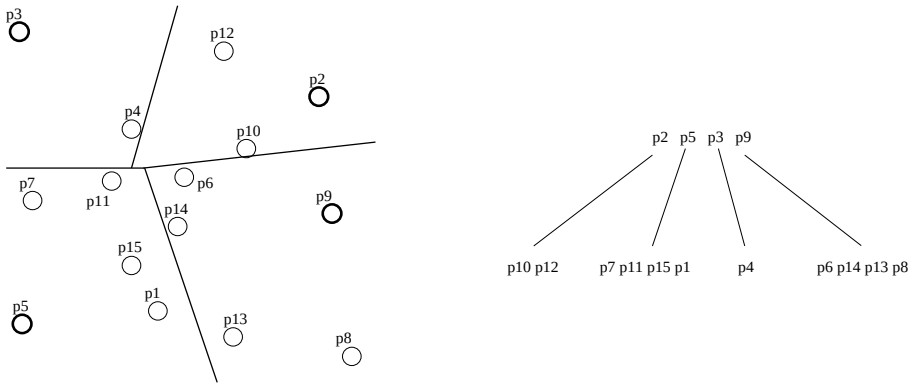

Figure 6: Example of the first level of a GNAT with $m = 4$.

AESA An algorithm which is close to many of the presented ideas but performs surprisingly better by an order of magnitude is [55] (called AESA, for "Approximating Eliminating Search Algorithm"). The structure is simply a matrix with the $n(n-1)/2$ precomputed distances among the elements of $\mathbb{U}$. At search time, they select an element $p \in \mathbb{U}$ at random and measure $r_p = d(p,q)$, eliminating all elements u of $\mathbb{U}$ which do not satisfy $r_p - r \leq d(u,p) \leq r_p + r$. Notice that all the $d(u,p)$ distances are precomputed, so only $d(p,q)$ has been calculated at search time. This process of taking a random pivot among the (not yet eliminated) elements of $\mathbb{U}$ and eliminating more elements from $\mathbb{U}$ is repeated until few enough elements remain in the set. These are compared against q directly. Figure 7 shows an example with a first pivot p_{11}.

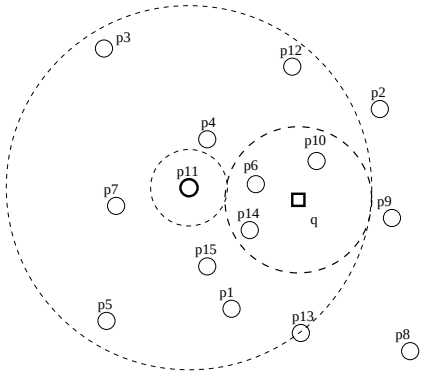

Figure 7: Example of the first iteration of AESA. The points between both rings centered at p_{11} qualify for the next iteration.

Although this idea seems very similar to FQTs, there are three key differences. The first one, only noticeable in continuous spaces, is that there are no predefined "rings" so that all the intersected rings qualify (recall Figure 2). Instead, only the minimal necessary area of the rings qualifies. The second difference is that the second element to compare against q is selected from the qualifying set, instead of from the whole set as in FQTs. Finally, the algorithm determines on the fly whether to take more pivots, while FQTs must precompute that decision (i.e. bucket size).

The problem with the algorithm [55] is that it needs $O(n^2)$ space and construction time. This is unacceptably high for all but very small databases. In this sense the approach is close to [50], although in this latter case they may take less distances and bound the unknown ones. AESA is experimentally shown to have $O(1)$ query time.

LAESA and variants In a newer version of AESA, called LAESA (for Linear AESA) [42], they propose to use k fixed pivots, so that the space and construction time is $O(kn)$. In this case, the only difference with an FHQT is that fixed rings are not used, but the exact set of elements in the range is retrieved. FHQT uses fixed rings to implement a search algorithm, while in this case no search algorithm is given. In LAESA, the elements are simply linearly traversed, and those that cannot be eliminated after considering the k pivots are directly compared against the query.

A search algorithm is presented later in [41], which builds a GHT-like structure using the same pivots. The algorithm is argued to be sublinear in CPU time. Alternative search structures to reduce CPU time not loosing information on distances are presented in [45, 23], where the distances to each pivot are sorted separately so that the relevant range $[d(q,p)-r, d(q,p)+r]$ can be binary searched[3]. Extra pointers are added to be able to trace an element across the different orderings for each pivot (this needs more space, however).

SAT The algorithm SAT ("Spatial Approximation Tree") [44] does not use pivots to split the set of candidate objects, but rather relies on "spatial" approximation. An element p is selected as the root of a tree, and it is connected to the elements $u \in \mathbb{U}$ such that u is closer to p than to any other element connected to p (note that the definition is self-referential). All the elements connected to p are recursively the roots of subtrees, whose elements are those closer to that root than to any other root connected to p.

This allows to search elements with radius zero by simply moving from the root to its "neighbor" (i.e. connected element) which is closest to the query q. If a radius r is allowed, then we consider that an unknown element of the set is searched with tolerance r, i.e. we search as before and consider that any distance measure may have an "error" of at most r. Therefore, we may have to enter into many branches of the tree (not only the closest one), since the measuring "error" could make that a different neighbor is the closest one. The tree is built in $O(n \log n / \log\log n)$ time, takes $O(n)$ space and inspects $n^{1-\Theta(1/\log\log n)}$ elements. Figure 8 shows an example and the search path for a query.

4.3 Other Techniques

Mapping An interesting and natural reduction of the proximity search problem consists in a mapping Φ from the original metric space into a vector space. In this way, each element of the original metric space will be represented as a point in the target vector space. The two spaces will be related by two distances: the original one $d(x,y)$ and the distance in the projected space $d'(\Phi(x),\Phi(y))$. If the mapping is *contractive*, i.e. $d'(\Phi(x),\Phi(y)) \leq d(x,y)$ for any pair of elements, then one can search queries of type (a) in the projected space with the same radius. The outcome of the query in the projected space is a candidate list, which is later refined with the original

[3] Although in [45] they consider only vector spaces, the same technique can be used here.

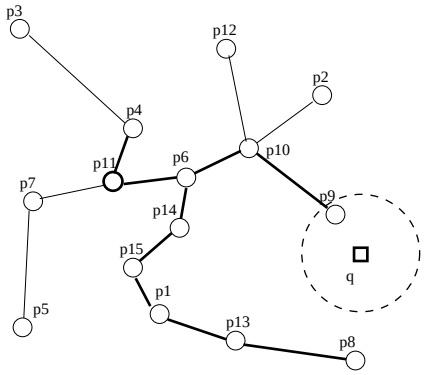

Figure 8: Example of a SAT and the traversal towards a query q, starting at p_{11}.

distance to obtain the actual outcome of the query. If, on the other hand, the mapping is *proximity preserving*, i.e. $d(x,y) \leq d(x,z) \Rightarrow d'(\Phi(x),\Phi(y)) \leq d'(\Phi(x),\Phi(z))$, then queries of type (b) and (c) can be directly performed in the projected space. Indeed, most current algorithms for queries of type (b) and (c) are based in type (a), and with some care they can be done in the projected space if the mapping is contractive, even if it is not proximity preserving.

In a theoretical paper [33] it is proven that there exists an algorithm using a constant (in n) number of distance calculations for nearest neighbor search. The idea suggested in [33] is basically to use the distances to k fixed pivots as the coordinates to project the metric space into $\mathbb{R}^k$. The projected distance function is L_∞ (later we explain this more in depth). They show that if the proper pivots are selected, it is possible to build an algorithm which is $k + O(1)$ search time. However, the proof is not constructive and does not give any clue to select the "proper pivots". The k value is independent on n and related to the "intrinsic dimension" of the data (a concept that we explain later).

With this mapping in mind many algorithms (indeed most of them as we see later) can be considered as a mapping of certain type. Figure 9 shows an example with only two coordinates. Notice that some points originally quite far away are mapped to the same cell, so the mapping does not preserve proximity.

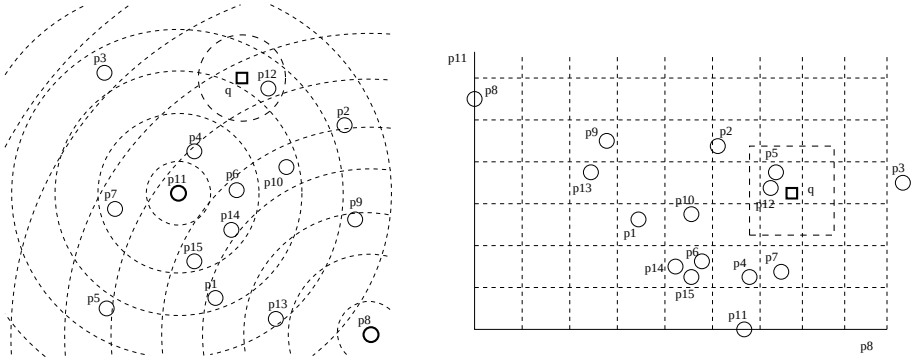

Figure 9: Mapping to a vector space of two coordinates, and how a query is transformed.

A more elaborated version of this idea was introduced under the name *fastmap* [32]. In this case they did the mapping from an n-dimensional into an m-dimensional vector space, for $n > m$. Fastmap is a heuristic to approximate the behavior of more expensive mappings which try to reduce the dimensionality while preserving the original distances as much as possible [29, 37]. Since these methods work only for vector spaces, we do not consider them in depth here (see also [15] for more on this).

This type of mapping is a special case of a general idea in the literature which says that one can find cheaper to compute distances that lower-bound the real one, and use the cheaper distance to filter out most elements (e.g. for images, the average color is cheaper to compute than the differences in the color histograms). While in general this is domain-dependent, mapping onto a vector space can be done without knowledge of the domain. After the mapping is done and we have identified each data element with a point in the projected space, we can use a general purpose spatial access method (SAM) for vector spaces to retrieve the candidate list. The elements found in the projected space must be finally checked using the original distance function.

Therefore, there are two types of distance evaluations: first to obtain the coordinates in the projected space and later to check the final candidates. These are called "internal" and "external" evaluations, respectively, later in this work. Clearly, incrementing internal evaluations improves the quality of the filter and reduces external evaluations, and therefore we seek for a balance.

Notice finally that the search itself in the projected space does not use evaluations of the original distance, and hence it is costless under our complexity measure. Therefore, the use of kd-trees, R-trees or other data structure aims at reducing the extra CPU time.

Clustering Clustering is a very wide area with lots of applications [39]. The general goal is to divide a set in subsets of elements close to each other in the same subset. A few approaches to index metric spaces based on clustering exist.

A technique proposed in [19] is to recursively divide the set $\mathbb{U}$ in compact subsets $\mathbb{U}_i$ and choose a representative p_i for each. They compute numbers $r_i = max\{d(p_i, u) \ / \ u \in \mathbb{U}_i\}$ (which upper bound the "radii" of the subsets). To search for the closest neighbor, the query q is compared against all the p_i and the sets are considered from smallest to largest distance. The r_i are used to determine that there cannot be interesting elements in some sets $\mathbb{U}_i$. They propose a complex "clique" criterion to select the sets and their representatives. The experimental results show that this method is slightly worse than the BKT, and that the algorithm to find the cliques is very slow. They also propose that the elements in a clique could be in turn subdivided into clusters, which is a formulation very similar to (though less complete than) GNATs [16].

4.4 Approximate and Probabilistic Algorithms

For the sake of a complete overview we include a brief description of an important branch of similarity searching, where a relaxation on the query precision is allowed to obtain a speedup in the query time complexity. This is reasonable in some applications because the metric space modelization involves already an approximation to the true answer (recall Section 2), and therefore a second approximation at search time may be acceptable.

Additionally to the query one specifies a precision parameter ε to control how far away (in some sense) we want the outcome of the query from the correct result. A reasonable behavior for

this type of algorithms is to asymptotically approach to the correct answer as ε goes to zero, and complementarily to speed up the algorithm, loosing precision, as ε moves in the opposite direction.

This alternative to *exact similarity searching* is called *approximate similarity searching*, and encompasses approximate and probabilistic algorithms.

Approximation algorithms are considered in depth in [57]. As an example, we mention an approximate algorithm for nearest neighbor search in real vector spaces using any Minkowski metric L_s [2]. They propose a data structure, the BBD-tree, inspired in kd-trees, that can be used to find "$(1+\varepsilon)$ nearest neighbors": instead of finding

$$u \text{ such that } d(u,q) \leq d(v,q) \ \ \forall v \in \mathbb{U}$$

they find an element u^*, an $(1+\varepsilon)$-nearest neighbor, differing from u by a factor of $(1+\varepsilon)$, i.e.

$$u^* \text{ such that } \frac{d(u^*,q)}{1+\varepsilon} \leq d(v,q) \ \ \forall v \in \mathbb{U}$$

The essential idea behind this algorithm is to locate the query q in a cell (each leaf in the tree is associated with a cell in the decomposition). Every point inside the cell is processed to obtain the current nearest neighbor (u). The search stops when no promising cells are encountered, i.e. when the radius of any ball centered at q and intersecting a nonempty cell exceeds the radius $d(q,p)/(1+\varepsilon)$. The query time is $O(\lceil 1+6k/\varepsilon \rceil^k k \log n)$.

An example of a probabilistic algorithm is given in [28]: a data structure somewhat similar to a skip list is presented, with almost linear space and construction time, which can be queried for nearest neighbors in $O(K \log n)$ time. The answers are wrong with probability $O(\log^2(n)/K)$.

4.5 Summary

Table 1 summarizes the complexities of the different approaches. These are obtained or inferred from the source papers, which as explained use different (and incompatible) assumptions and in many cases give just gross analyses or no analysis at all. Keep in mind that the query complexity is always on average, as in the worst case we can be forced to compare all the elements.

5 A Unifying Model

At first sight, the indexing algorithms and data structures seem to emerge from a great diversity, and different approaches are analyzed separately, often under different assumptions. Currently, the only realistic way to compare two different algorithms is to apply them to the same data set.

In this section we make a formal introduction to our unifying model. Our intention is to provide a common framework to analyze all the existing approaches to proximity searching. As a result, we will be able to capture the similarities of apparently different approaches. We will also obtain truly new ways of viewing the problem.

The conclusion of this section can be summarized in Figure 10. All the indexing algorithms partition the set $\mathbb{U}$ into subsets. An index is built which allows to determine a set of candidate sets where the elements relevant for the query can appear. At query time, que index is searched to find the relevant subsets (the cost to do this is called "internal complexity") and those subsets are checked exhaustively (which corresponds to the "external complexity" of the search).

Data Structure	Space Complexity	Construction Complexity	Query Complexity	Extra CPU query time
BKT	n pointers	$O(n \log n)$	$O(n^{\alpha})$	—
FQT	$n..n \log n$ pointers	$O(n \log n)$	$O(n^{\alpha})$	—
FHQT	$n..nh$ pointers	$O(nh)$	$O(\log n)$ if $h = \log n$	$O(n^{\alpha})$
FQA	nhb bits	$O(nh)$	$O(\log n)$ if $h = \log n$	$O(n^{\alpha} \log n)$
VPT	n pointers	$O(n \log n)$	$O(\log n)$ (*)	—
MVPT	n pointers	$O(n \log n)$	$O(\log n)$ (*)	—
VPF	n pointers	$O(n^{2-\alpha})$	$O(n^{1-\alpha} \log n)(*)$	—
GHT	n pointers	$O(n \log n)$	$O(\text{polylog } n)$	—
GNAT	nm^2 distances	$O(nm \log_m n)$	$O(\text{polylog } n)$	—
AESA	n^2 distances	$O(n^2)$	$O(1)$	$O(n)..O(n^2)$
LAESA-like	kn distances	$O(kn)$	$k + O(1)$ if k is large	$O(\log n)..O(kn)$
SAT	n pointers	$O(n \log n / \log \log n)$	$O(n^{1-\Theta(1/\log \log n)})$	—

(*) Only valid when searching with very small radii.

Table 1: Average complexities of the existing approaches, according to the source papers. Time complexity considers only n, not other parameters such as dimension. Space complexity mentions the most expensive storage units used. α is a number between 0 and 1, *different for each structure*, while the other letters are parameters particular of each structure.

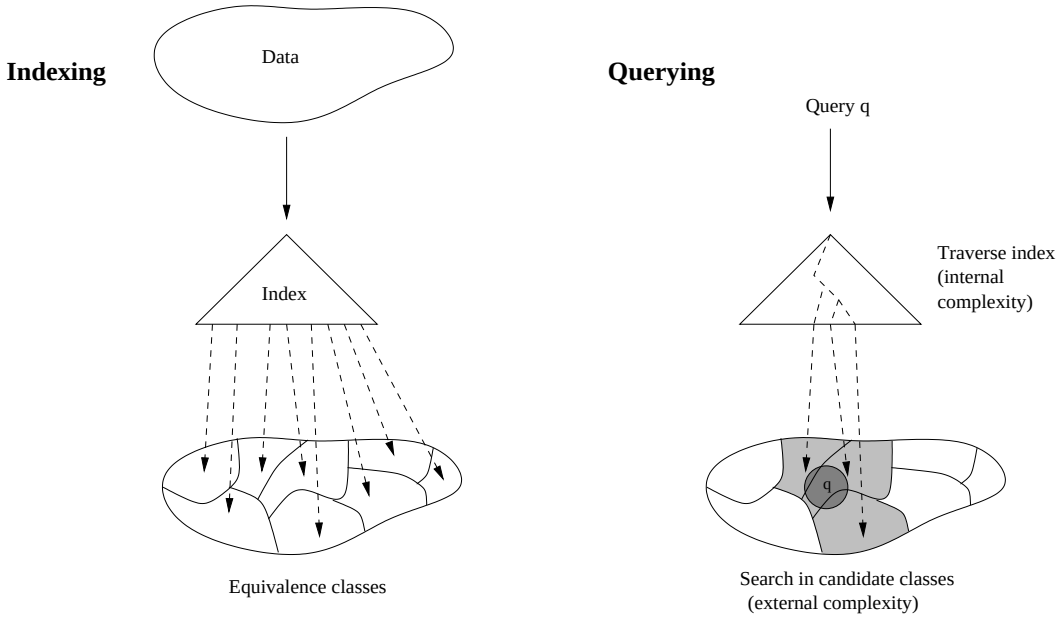

Figure 10: The unified model for indexing and querying metric spaces.

5.1 Equivalence Relations and Cosets

Given a set $\mathbb{X}$, a *partition* $\pi(\mathbb{X}) = \{\pi_1, \pi_2, \cdots\}$ is a subset of the power set $\mathcal{P}(\mathbb{X})$ such that every element of the set belongs exactly to one partition, i.e. $\cup \pi_i = \mathbb{X}$ and $\forall i \neq j,\ \pi_i \cap \pi_j = \emptyset$.

A relation, denoted by $\sim$, is a subset of the cross product $\mathbb{X} \times \mathbb{X}$ (the set of ordered pairs) of $\mathbb{X}$. Two elements x, y are said to be related, denoted by $x \sim y$, if the pair (x, y) is in the subset. A relation $\sim$ is said to be an *equivalence relation* if it satisfies, for all $x, y, z \in \mathbb{X}$, the properties of reflexivity ($x \sim x$), symmetry ($x \sim y \Leftrightarrow y \sim x$) and transitivity ($x \sim y \wedge y \sim z \Rightarrow x \sim z$).

It can be shown that every partition $\pi(\mathbb{X})$ induces an equivalence relation $\sim$ and, conversely, every equivalence relation induces a partition [25]. Two elements are related if they belong to the same partition. Every element π_i of the partition is then called an *equivalence class*. An equivalence class is often named after one of its representatives (any element of π_i can be taken as a representative). An alternative definition of an equivalence class of an element x is the set of all y such that $x \sim y$. We will denote the equivalence class of x as $[x] = \{y : x \sim y\}$.

Given the set $\mathbb{X}$ and an equivalence relation $\sim$, we obtain the quotient $\pi(\mathbb{X}) = \mathbb{X}/\sim$. It indicates the set of equivalence classes or cosets, obtained when applying the equivalence relation to the set $\mathbb{X}$. In the example above, the cosets obtained from the real numbers with the given equivalence relation are the intervals $[i, i+1)$, for all integers i.

The relevance of equivalence classes in this survey comes from the possibility of using them on a metric space so that a new metric space is derived from the quotient set. This new metric space is a coarser version of the original one.

5.2 Indexing and Partitions

The equivalence classes obtained with an equivalence relation of a metric space can be considered themselves as objects in a new metric space, provided we define a distance function on this new metric space.

We introduce a new function $D_0 : \pi(\mathbb{X}) \times \pi(\mathbb{X}) \longrightarrow \mathbb{R}$ now defined in the quotient. Since D_0 is defined between equivalence classes, a natural choice is $D_0([x], [y]) = \inf_{x \in [x], y \in [y]} \{d(x, y)\}$, so that it gives the maximum possible values that keep the mapping *contractive* (i.e. $D_0([x], [y]) \leq d(x, y)$ for any x, y). Unfortunately, D_0 does not satisfy the triangle inequality, just $(p1)$ to $(p3)$, and in most cases $(p4)$ (recall Section 3.1). Hence, D_0 itself is not suitable for indexing purposes.

However, we can use any distance D that satisfies the properties of metric spaces and that lower bounds D_0 (i.e. $D([x], [y]) \leq D_0([x], [y])$). We call such D the *extension* of d. Since D is a distance, $(\mathbb{X}/\sim, D)$ is a metric space and therefore we can make queries in the coset in the same way we have done in the set. We redefine the outcome of a query in the coset as $([q], r)_D = \{u \in \mathbb{U}, D([u], [q]) \leq r\}$ (although formally we should retrieve classes, not elements).

Since the mapping is contractive ($D([x], [y]) \leq d(x, y)$) we can convert one search problem into another, hopefully simpler, search problem. For a given query $(q, r)_d$ we find out which equivalence class the query q belongs to (i.e. $[q]$). Then, using the new distance function D the query is transformed into $([q], r)_D$. As the mapping is contractive, we have $(q, r)_d \subseteq ([q], r)_D$. That is, $([q], r)_D$ is indeed a candidate list, so it is enough to perform an exhaustive search on that candidate list (now using the original distance), to obtain the actual outcome of the query $(q, r)_d$.

Our main thesis is that the above procedure is in fact used in virtually every indexing algorithm. In other words:

> *All the existing indexing algorithms for proximity searching consist in building a set of equivalence classes, discarding some classes, and searching exhaustively the rest.*

As we see shortly, the most important tradeoff when designing the equivalence partition is to balance the cost to find $[q]$ and to check the final candidate list.

In Figure 11 we can see a schematic example of the idea. We divide the space in several regions (equivalence classes). The objects inside each region are indistinguishable. We can consider them as elements in a new metric space. To find the answer, instead of exhaustively examining the entire dictionary we just examine the classes that contain potentially interesting objects. In other words, if a class can contain an element that should be returned in the outcome of the query, then the class will be examined (see also the rings considered in Figure 2).

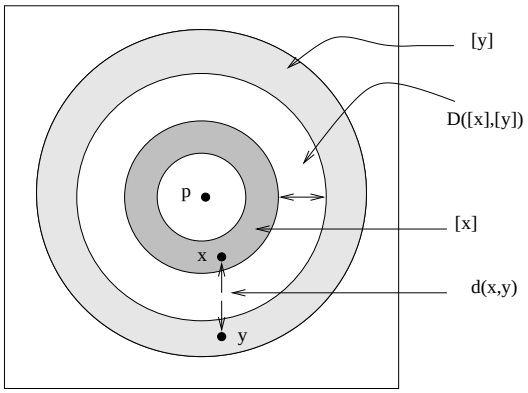

Figure 11: Two points x and y, and their equivalence classes (the shaded rings). D gives the minimal distance among rings, which lower bounds the distance between x and y.

We recall that this property is not enough for any algorithm to find the nearest neighbors to work (since the mapping would have to preserve proximity instead), but most existing algorithms for nearest neighbors are based on type (a) queries, and these algorithms can be applied as well.

Some examples may help to understand the above definitions, for both the concept of equivalence relation and the obtained distance function.

Example 1. The void indexing algorithm, i.e. the brute force method of not indexing and examining every element in the dictionary for each query, creates one equivalence class per object in the set $\mathbb{X}$. In this case, the coset obtained is the same as the original set $\pi(\mathbb{X}) = \mathbb{X}/\sim = \mathbb{X}$ and in this particular case $x \sim y \Leftrightarrow x = y$. Note that in this case $D([x],[y]) = d(x,y)$ for any pair x, y and consequently $(q,r)_d = ([q],r)_D$. In other words the candidate list is actually the outcome of the query. No further effort is done in trimming the candidate list, however all the work have been done in building the candidate list.

Example 2. Another trivial example, situated in the other side of the spectrum, is when all elements in $\mathbb{X}$ are assigned to the same equivalence class. The equivalence relation is defined as

$x \sim y \Leftrightarrow x, y \in \mathbb{X}$. Since there is only one equivalence class, it is true that $[x] = [y]$ for all elements in the set $\mathbb{X}$. In this case we have $\pi(\mathbb{X}) = \mathbb{X}/\sim = \{\cdot\}$, a set with a single element. The distance function is $D([x],[y]) = 0$ for every pair of objects. In this case finding the candidate list is trivial, since it is the dictionary itself, but trimming the list is as difficult as the original problem.

Example 3. A more realistic example, indeed a true indexing algorithm, is when we have an arbitrary reference pivot $p \in \mathbb{X}$ and the equivalence relation is given by $x \sim y \Leftrightarrow d(p,x) = d(p,y)$. In this case $D([x],[y]) = |d(x,p) - d(y,p)|$ is a safe lower bound for the D_0 distance (guaranteed by the triangle inequality). For a query of the form $(q,r)_d$ the candidate list $([q],r)_D$ consists of all elements x such that $D([q],[x]) \leq r$, or which is the same, $|d(q,p) - d(x,p)| \leq r$. Graphically, this distance represents a *ring* centered at p containing a *disk* centered at q and radius r (recall Figures 11 and 7). This is the familiar rule used in many independent algorithms to trim the space.

Example 4. As explained, the similarity search problem was firstly introduced in vector spaces, and the very first family of algorithms used there was based on a coset operation. These algorithms were called *bucketing* methods, and consist in the construction of cells or buckets [10]. Searching for an arbitrary point in $\mathbb{R}^k$ is converted into an exhaustive search in a finite set of cells. The procedure used two steps: (1) first they find which cell the query point belongs to and then they build a set of candidate cells using the query range; (2) this set of candidate cells is inspected exhaustively to find the actual points inside the query range[4]. In this case the equivalence classes are the cells, and the tradeoff: the larger the cells, the cheaper it is to find the appropriate ones, but the more costly is the final exhaustive search.

5.3 Coarsening and Refining a Partition

For a fixed set $\mathbb{X}$, consider two equivalence relations $\sim_1$ and $\sim_2$. We say that $\sim_1$ is a *refinement* of $\sim_2$ if for any pair $x, y \in \mathbb{X}$ such that $x \sim_1 y$ it holds $x \sim_2 y$. Equivalently, a partition $\pi^1(\mathbb{X})$ is a refinement of partition $\pi^2(\mathbb{X})$ if $\pi_i^1 \subset \pi_j^2$ for every partition element π_i^1 of π^1 and some coset π_j^2 of π^2. We may also say that π^2 (equivalently $\sim_2$) is a *coarsening* of π^1 (equivalently $\sim_1$).

Refinement and coarsening are important concepts for the topic we are discussing. They are the very essence of indexing algorithms. The following theorem formalizes our intuitive assertions.

Theorem 1. *If $\sim_1$ is a coarsening of $\sim_2$ then the extended distances D_1 and D_2 have the property $D_1([x],[y]) \leq D_2([x],[y])$.*

Proof. $D_1([x],[y]) = \inf_{x \in [x]_1, y \in [y]_1}\{d(x,y)\} \leq \inf_{x \in [x]_2, y \in [y]_2}\{d(x,y)\} = D_2([x],[y])$, since $[x]_2 \subseteq [x]_1$ and $[y]_2 \subseteq [y]_1$. We are using $[x]_i$ and $[y]_i$ to denote the equivalence class of x and y under equivalence relation $\sim_i$.

An interesting idea arising from the above theorem is to build a hierarchy of coarsening operations. Using this hierarchy we could proceed downwards from a very coarse level building a candidate list of equivalence classes of the next level. This candidate list will be refined using the next distance function and so on until we reach the bottom level.

[4] The algorithm is in fact a little more sophisticated because they try to find the nearest neighbor of a point. However, the version presented here for range queries is in the same spirit as the original one.

5.4 Measures of Efficiency

As sketched previously, most indexing algorithms rely on building an equivalence class. The corresponding search algorithms have two parts:

1. Find the classes that may be relevant for the query.
2. Exhaustively search all the elements of these classes.

The first part involves performing some evaluations of the d distance, as shown in the Example 3 above. It may also involve some extra CPU time (which although not the central point in this paper, must be kept reasonable). The distance evaluations performed in this stage are called *internal*, and their number define the *internal complexity*.

The second part consists of directly comparing the query against the candidate list. These evaluations of d are called *external*. The amount of external evaluations is called *external complexity* and is related to the *discriminative power* of the D distance.

We define the discriminative power as the ratio between the number of objects in the candidate list and the actual outcome of a query q, averaged over all $q \in \mathbb{X}$. Notice that this depends on r. The discriminative power serves as an indicator of the performance or fitness of the equivalence relation (or equivalently, of the distance function D).

In general, it will be more costly to have more discriminative power. The indexing scheme needs to find a balance between the complexity to find the relevant classes and the discriminative power of D.

Examples 1 and 2 can serve as upper and lower bounds of what is done by the actual indexing algorithms. The first algorithm has minimal external complexity, since the distance function D discriminates as much as the original distance function d. However, the internal complexity is maximal, in the sense that finding the relevant classes is as hard as solving the original problem. This case shows maximum discriminative power, as the metric spaces $(\mathbb{X}, d)$ and $(\pi(\mathbb{X}), D)$ are isometric [43]. Example 2 has minimal internal complexity, since it is trivial to compute the relevant equivalent class. However, its external complexity is as high as in the original problem, since all the elements are candidates.

Example 3 is in between for internal and external complexity. The internal complexity is 1 distance evaluation (the distance from q to p), and the external complexity will correspond to the number of elements that lie in the selected ring. We could intersect it with more rings (increasing internal complexity) to reduce the external complexity.

The tradeoff is partially formalized with this theorem.

Theorem 2. *If A_1 and A_2 are indexing algorithms based on equivalence relations $\sim_1$ and $\sim_2$, respectively, and $\sim_1$ is a coarsening of $\sim_2$, then A_1 has higher external complexity than A_2.*

Proof. We have to show that $([q], r)_{D_2} \subseteq ([q], r)_{D_1}$. But this is clear, since $D_1([x], [y]) \leq D_2([x], [y])$ implies $([q], r)_{D_2} = \{y \in \mathbb{U} : D_2([q], [y]) \leq r\} \subseteq \{y \in \mathbb{U} : D_1([q], [y]) \leq r\} = ([q], r)_{D_1}$.

Although having more discriminative power normally costs more internal evaluations, one can make better or worse use of the internal complexity. We elaborate more on this in the next section.

5.4.1 Locality of a Partition

The equivalence classes can be thought of as a set of *non intersecting* cells in the space, where every element inside a given cell belongs to the same equivalence class. However, the mathematical definition of an equivalence class is not confined to a single cell.

A consequence of this is that we need an additional property which will be called *locality*, that stands for how much the equivalence class resembles a cell. A non-local partition stands for cases where the classes are partitioned (see Figure 12) or span a non-compact area in the space.

It is natural to expect better performance, i.e. more discriminative power, from a local partition than from a non-local one. This is because in a non-local partition the candidate list obtained with the distance D will contain elements actually far away from the query.

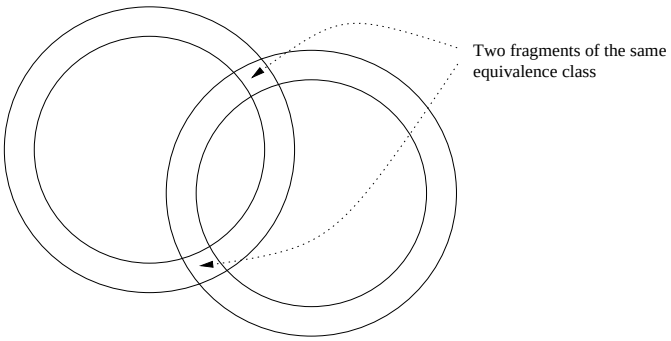

Figure 12: With two rings we define an equivalence based on being at the same distance to both points. The resulting class is partitioned.

Notice that in Figure 12, the fragmentation would disappear if we added a third pivot. In a vector space of k dimensions, it suffices to consider $k+1$ pivots in general position[5] to obtain a local partition. In general metric spaces, we should take enough pivots to obtain local partitions as well.

However, obtaining local partitions is not enough, otherwise the bucketing method for vector spaces [10] explained in Example 4 would have excellent performance. Even with such a local partition and assuming uniformly distributed data, a number of empty cells are verified, whose number grows exponentially with the dimension.

5.5 Intrinsic Dimensionality

As explained, one of the motivations for the development of indexing algorithms for general metric spaces is the existence of the so called *high dimensional*. This is because traditional indexing techniques for vector spaces have an exponential dependency on the dimension of the space. In other words, if a vector space has a large number of coordinates then an indexing algorithm using explicit information on the coordinates (such as the *kd*-tree) will use exponential time (on the dimension) to answer the query. This motivated the research on the so-called *distance-based indexing algorithms*, which do not use explicit information on the coordinates.

[5] That is, not lying in a $(k-1)$-hyperplane.

This works especially well in some vector spaces of apparently high dimension but effective low dimension. Think for example in a set of 5-dimensional points with the 3 last coordinates in zero, or the more sophisticated example where the points actually reside in a 2-dimensional plane immersed in a 5-dimensional space. However, it is also possible that the data has *intrinsic high dimensionality.*

As we show next, the concept of high dimensionality is not exclusive of vector spaces, but it can also be characterized in metric spaces. If a vector space has intrinsically low dimension, then considering it as a metric space may help to take advantage of that fact, while if it has intrinsically high dimension the resulting metric space will be intractable anyway. In the following we explain the effect of high dimensionality in metric spaces, and why it makes the problem intractable [22].

Let us consider the histogram of distances between objects in the dictionary. This information is mentioned in many papers, e.g. [16, 27]. For a fixed dictionary element u consider the histogram of distances $d(u, u_i)$, $u_i \in \mathbb{U}$. It is clear that the number of elements within the range $(u, r)_d$ is proportional to the area under the histogram, from 0 to r inclusive.

Now, if we calculate the histogram considering *all* the possible pairs $d(u_i, u_j)$, the area described above is proportional to the average number of elements expected for a query of range r, provided $q \in \mathbb{U}$.

It is reasonable to assume that the query distributes according to the same law as the dictionary elements. In other words the distribution of distances from the query to the dictionary elements looks like the computed histogram for the data set. For the range search algorithm, if we select a ring centered at a dictionary element u^*, with radii $d(u^*, q) - r$ and $d(u^*, q) + r$ the fraction of dictionary elements captured in the ring is, on the average, approximated by the area under the density function in the interval $[d(u^*, q) - r, d(u^*, q) + r]$. More precisely, if f_d is the density function for the distances between elements in the given data set, then the fraction of elements captured is $n_r = \int_{d(u^*,q)-r}^{d(u^*,q)+r} f_d(x)dx$.

For every pivot-based algorithm the value of the above integral governs the behavior of the algorithm. At each step the number of elements eliminated is proportional to $1 - n_r$. If $n_r = 1$ then no elimination is carried out. We show now that in some cases we can infer that the discriminative power of the algorithm will be very low. Let us assume that the interval where f_d is larger than zero is $[r_a, r_b]$. We note that

1. r_a represents the average distance from an element to its nearest neighbor.
2. r_b represents the average distance from a given element to the object farthest to it.
3. If $r < r_a$ then the expected number of dictionary elements inside a ball centered at the query and with radius r is zero.

These properties are direct consequences of the definition. Now an elementary but interesting property can be stated as a theorem.

Theorem 3. *If $2r_a > r_b$ then $n_r = 1$. In other words, on the average no elements are eliminated.*

Proof. If we choose an arbitrary element u^* as a pivot then $d(x, u^*) \in [r_a, r_b]$. From property 3, we need to perform queries of range $r \geq r_a$ to retrieve some result. The number of qualifying

elements is

$$n_r \;=\; \int_{d(u^*,q)-r_a}^{d(u^*,q)+r_a} f_d(x)dx \;\geq\; \int_{r_a}^{r_b} f_d(x)dx \;=\; 1$$

since the interval $[d(u^*,q)-r_a, d(u^*,q)+r_a]$ clearly contains $[r_a, r_b]$.

The above proposition implies that there are metric spaces in which no eliminations will be carried out (see Figure 13). Moreover, *any* algorithm using the same elimination rule will be limited in the same way. For example, when the metric space has a density function according to this theorem *all* the branches of an FQT will be visited and $O(n)$ distance calculations will be carried out.

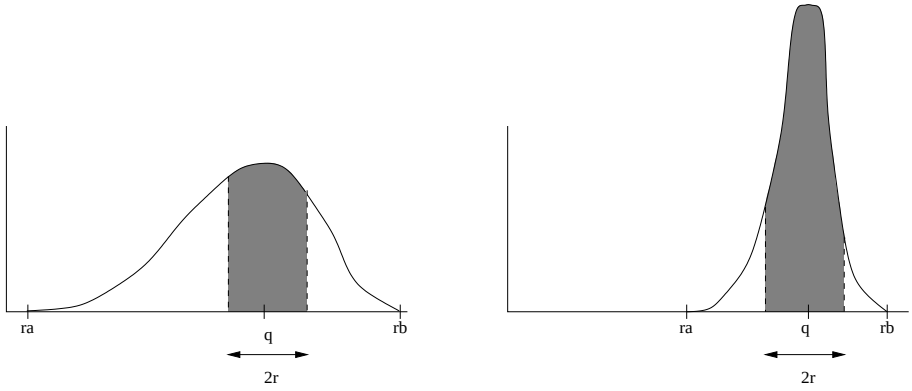

Figure 13: A low-dimensional (left) and high-dimensional (right) histogram of distances, showing that on high dimensions virtually all the elements become candidates for the exhaustive evaluation. Moreover, we should use a larger r in the second plot in order to retrieve some elements.

The condition $2r_a > r_b$ is a gross measurement of the dependence of the dimensionality in the performance of pivot-based algorithms. We have not yet established the relationship between skewed histograms and high dimensionality. In $\mathbb{R}^k$ Euclidean spaces, the distribution of the distance between two random points has larger mean and smaller variance as k grows. Therefore, the phenomenon of a skewed histogram moved to the right (i.e. large r_a and small $r_b - r_a$) is typical of high dimensions. If the objects are intrinsically of high dimension, this behavior is inherited when the space is considered as a general metric space. Following this idea, we can define the intrinsic dimensionality of a general metric space by considering the shape of its histogram. This gives a direct and general explanation to the so-called "curse of dimensionality".

Many authors stress an extreme case which is a good illustration: a distance such that $d(x,x) = 0$ and $d(x,y) = 1$ for all $x \neq y$. Under this distance, we do not obtain any information from a comparison except that the element considered is or is not our query. It is clear that it is not possible to avoid a sequential search in this case. The histogram of this distance is totally concentrated on its maximum value. The subject of intrinsic dimensionality is discussed also in [59].

6 A Taxonomy of Search Algorithms

In this section we apply our unifying model to organize all the known approaches in a taxonomy. This helps to identify the essential features of all the existing techniques, to find possible combinations of algorithms not noticed up to now, and to detect which are the most promising areas for optimization.

We first notice that almost all the indexing algorithms are built on an equivalence relation. The aim is to group the elements of the set in clusters, so that the partitions are as local as possible and have good discriminative power. In practice, it turns out that most algorithms to build the equivalence relations are based on obtaining, for each element, up to k values (also called coordinates), so that the equivalence classes can be considered as points in a k-dimensional vector space. Most of these algorithms obtain the k values as the distance of the object to k different (and hopefully independent) *pivots*. We call them "pivoting" algorithms. The algorithms differ in their method to select the pivots, in when is the selection made, and in how much information on the comparisons is used. We first explain in detail this large class of algorithms and then cover the other ones.

6.1 The Pivot Equivalence Relation

Most of the known algorithms to search in metric spaces are built on this equivalence relation. This is based on considering the distances between an element and a number of preselected "pivots" (i.e. elements of the universe, called also vantage points, keys, queries, etc. in the literature).

The equivalence relation is defined in terms of the distances of the elements to the pivots, so that two elements are equivalent if they are at the same distance to all pivots. If we consider one pivot p, then this equivalence relation is

$$x \sim_p y \iff d(x,p) = d(y,p)$$

The equivalence classes or partition elements correspond to the intuitive notion of the family of sphere shells with center p. Points falling in the same sphere shell are said to be equivalent or indistinguishable from the view point of p.

The above equivalence relation is easily generalized to k pivots or reference points p_i to give

$$x \sim_{\{p_i\}} y \iff \forall i,\ d(x,p_i) = d(y,p_i)$$

and a graphical representation of the partition in the general case corresponds to the intersection of several balls centered at the points p_i (recall Figure 12).

The distance $d(x,y)$ cannot be smaller than $|d(x,p) - d(y,p)|$ for any element p, because of the triangular inequality. Hence $D([x],[y]) = |d(x,p) - d(y,p)|$ is a safe lower bound to the D_0 function corresponding to the class of sphere shells centered in p. This is easy to generalize to k pivots, namely $D([x],[y]) = \max_i\{|d(x,p_i) - d(y,p_i)|\}$. This D distance lower bounds d and hence can be used as our distance in the quotient space.

Alternatively, we can consider the equivalence relation as a projection to the vector space $\mathbb{R}^k$, being k the number of pivots used. The i-th coordinate of an element is the distance of the element to the i-th pivot. Once this is done, we can identify points in $\mathbb{R}^k$ with elements in the

original space with the L_∞ distance. As we have described in Section 5, the indexing algorithm will consist in finding the set of equivalence classes such that they fall inside the radius of the search when using the extension of d in the quotient of the metric space. In this particular case for a query of the form $(q,r)_d$ we have to find the candidate list as the set $([q],r)_D$, i.e. the set of equivalence classes $[y]$ such that $D([q],[y]) \leq r$. In other words, we want the set of objects y such that $\max_i\{|d(q,p_i) - d(y,p_i)|\} \leq r$. This is equivalent to search with the L_∞ distance in the vector space $\mathbb{R}^k$ where the equivalence classes are projected. Figure 14 illustrates this concept (Figure 9 is also useful).

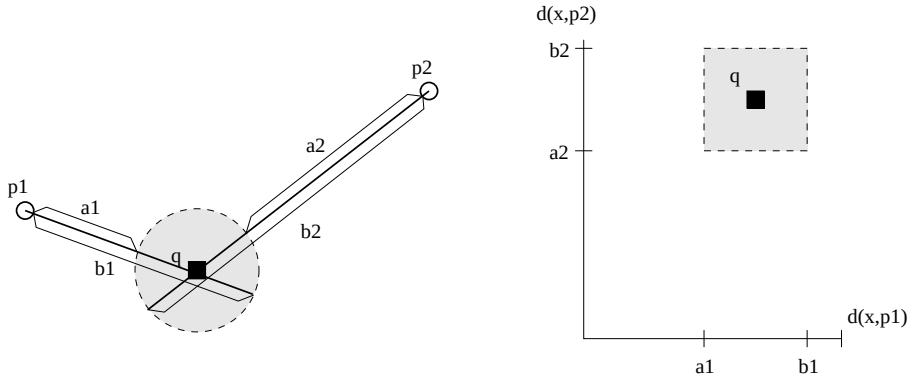

Figure 14: Mapping from a metric space onto a vector space under the L_∞ space, using two pivots.

Yet a third way to see the technique, less formal but perhaps more intuitive, is as follows: to check if an element $u \in \mathbb{U}$ belongs to the query outcome, we try a number of random pivots p_i. If, for any such p_i, we have $|d(q,p_i) - d(u,p_i)| > r$, then by the triangular inequality we know that $d(q,x) > r$ without need to actually evaluate $d(q,x)$. At indexing time we precompute the $d(u,p_i)$ values and at search time we compute the $d(q,p_i)$ values. Only those elements u that cannot be discarded by looking at the pivots are actually checked against q.

6.2 Selecting the Pivots

We have to find the proper balance for the number of pivots, k. If k is too small, then finding the classes will be cheap, but the partition will be very coarse and probably non-local (as explained in previous sections), and we will pay a high cost at the exhaustive search. If k is too large then the final partitions will be small and cheap to traverse, but the cost to compute them will be high. As explained before, the number of pivots needed to obtain a good partition is related to the intrinsic dimensionality of the data set.

In [33], they prove formally that if the dimension is constant, then after properly selecting a constant number k of pivots the exhaustive search costs $O(1)$ (but their theorem does not show how to select the pivots). In AESA [55], they show empirically that $O(1)$ pivots are necessary to achieve this goal, so that they can have $O(1)$ overall search time (recall that their algorithm is impractical). On the other hand, in [6], they show that $O(\log n)$ pivots are necessary to have a final exhaustive cost of $O(\log n)$. This difference is due to different models of the structure of the space (e.g. finiteness in volume) and the statistic behavior of the distance function. The correct

answer probably depends on the particular metric space considered.

A related issue is how to select the pivots. All the current schemes select the pivots at random from the set of objects $\mathbb{U}$. This is done for simplicity, although tackling this problem could yield dramatical improvements. For instance, in [51] it is recommended to select pivots outside the clusters while in [5] they suggest to use one pivot from each cluster. All authors agree in that the pivots should be far apart from each other, which is evident since close pivots will give almost the same information. On the other hand, pivots selected at random are indeed far apart in a high-dimensional space.

The distances histogram gives a formal characterization of good pivots. A good pivot has a flatter histogram, which means that it will discard more elements at query time (note that we could select pivots outside $\mathbb{U}$, although this needs specific knowledge of the application). Unfortunately, this depends on the query q and the search radius r. However, a Monte Carlo algorithm could be used to heuristically evaluate the goodness of the pivots for random queries and a predefined radius r, as done in [21].

The histogram characterization explains a well-known phenomenon: to discriminate among a compact set of candidates, it is a good idea to select a pivot from those same candidates. This makes it more probable to select an element close to them (the ideal would be a centroid). In this case, the distances tend to be smaller and the histogram is not so concentrated in large values. For instance, for LAESA [42] they do not use the pivots in a fixed order, but the next one is that with minimal L_1 distance to the current candidates. We do not try to plot an example, because the effect in two dimensions seems to be the opposite.

6.3 Search Algorithms

Once we have determined the equivalence relation to use (i.e. the k pivots), we preprocess the set by storing, for each element of $\mathbb{U}$, its k coordinates (i.e. distance to the k pivots). This takes $O(kn)$ preprocessing time and space overhead. The "index" can be seen as a table of n rows and k columns as shown in the left part of Figure 15.

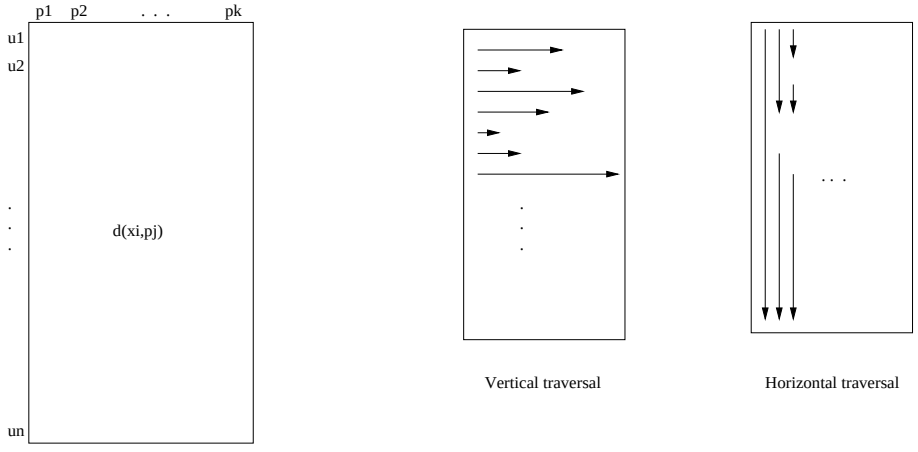

Figure 15: Schematic view of a pivot index, as well as vertical and horizontal traversal.

At query time, we first compare the query q against the k pivots, hence obtaining its k coordinate $(y_1, ..., y_k)$ in the target space. This corresponds to determining which equivalence class the query belongs to, i.e. computing $[q] = (y_1, ..., y_k)$. The cost of this is k evaluations of the distance function d, which corresponds to the internal complexity of the search. We have now to determine, in the target space, which classes may be relevant to the query (i.e. which ones are at distance r or less in the L_∞ metric, which corresponds to the D distance). This does not use further evaluations of d, but it may take extra CPU cost. Finally, the elements belonging to the qualifying classes (i.e. those that cannot be discarded after considering the k pivots) are directly compared against q (external complexity).

The simplest search algorithm proceeds row-wise: consider each element of the set (i.e. each row $(x_1, ..., x_k)$ of the table) and see if the triangular inequality allows to discard that row, i.e. whether $\max_{i=1..k}\{|x_i - y_i|\} > r$. For each row not discarded using this rule, compare the element directly against q. This is equivalent to traversing the quotient space, using D to discard uninteresting classes.

Although this traversal does not perform more evaluations of d than necessary, it is not the best choice. The reasons will be made clear later, as we discover the advantages of alternative approaches. First, notice that the amount of CPU work is $O(kn)$ in the worst case. However, as we abandon a row as soon as we find a difference larger than r along a coordinate, the average case is much closer to $O(n)$ for queries of reasonable selectivity.

The first improvement is to process the set column-wise. That is, we compare the query against the first pivot p_1. Now, we consider the first column of the table and discard all the elements which satisfy $|x_1 - y_1| > r$. Then, we consider the second pivot p_2 and repeat the process only on the elements not discarded up to now. An algorithm implementing this idea is LAESA.

It is not hard to see that the amount of evaluations of d and the total CPU work remains the same as for the row-wise case. However, we can do better now, since each column can be sorted so that the range of qualifying rows can be binary instead of sequentially searched [45, 23]. This is possible because we are interested, at column i, in the values $[y_i - r, y_i + r]$.

This is not the only trick allowed by a column-wise evaluation which cannot be done row-wise. A very important one is that it is not necessary to consider all the k coordinates (recall that we have to perform one evaluation of d to obtain each new query coordinate y_i). As soon as the remaining set of candidates is small enough, we can stop considering the remaining coordinates and directly verify the candidates using the d distance. This point is difficult to estimate beforehand: despite the (few) theoretical results existing [33, 6], one cannot normally understand the application well enough to predict the actual optimal number of pivots (i.e. the point where it is better to switch to exhaustive evaluation).

Another trick that can be used with column-wise evaluation is that the selection of the pivots can be done on the fly instead of fixed as we have presented it. That is, once we have selected the first pivot p_1 and discarded all the uninteresting elements, the second pivot p_2 may depend on which was the result of p_1. However, for each potential pivot we have to store the coordinates of all the elements of the set for this pivot (or at least some, as we see later). That is, we select k potential pivots and precompute the table as before, but we can choose in which order are the pivots used (according to the current state of the search) and where we stop using pivots and compare directly.

An extreme case of this idea is AESA, where $k = n$, i.e. all the elements are potential pivots,

and the new pivot at each iteration is randomly selected among the remaining elements. Despite its practical inapplicability because of its $O(n^2)$ preprocessing time and space overhead (i.e. all the distances among the known elements are precomputed), the algorithm performs a surprisingly low number of distance evaluations, much better than when the pivots are fixed. This shows that it is a good idea to select pivots from the current set of candidates (as discussed in the previous section).

Finally, we notice that instead of a sequential search in the mapped space, we could use an algorithm to search in vector spaces of k dimensions (e.g. kd-trees or R-trees). Depending on their ability to handle larger k values, we could be able to use more pivots without significantly increasing our extra CPU cost. Recall also that, as more pivots are used, the search structures for vector spaces perform worse. This is a very interesting subject which has not been pursued yet, that accounts for balancing between distance evaluations and CPU time.

6.4 Coarsening the Equivalence Relation

The alternative of not considering all the k pivots if the remaining set of candidates is small is an example of coarsening an equivalence relation. That is, if we do not consider a given pivot p, we are merging all the classes that differ only in that coordinate. In this case we prefer to coarsify the pivot equivalence relation because computing it with more precision is worse than checking it as is.

There are many other ways to coarsify the equivalence relation, and we cover them here. However, in these cases the coarsification is not done for the sake of reducing the number of distance evaluations, but to improve space usage and precomputation time, as $O(kn)$ can be prohibitively expensive for some applications. Another reason is that, via coarsening, we obtain search algorithms that are sublinear in their extra CPU time. We consider in this section *range coarsening*, *bucket coarsening* and *adaptive coarsening*. Their ideas are roughly illustrated in Figure 16.

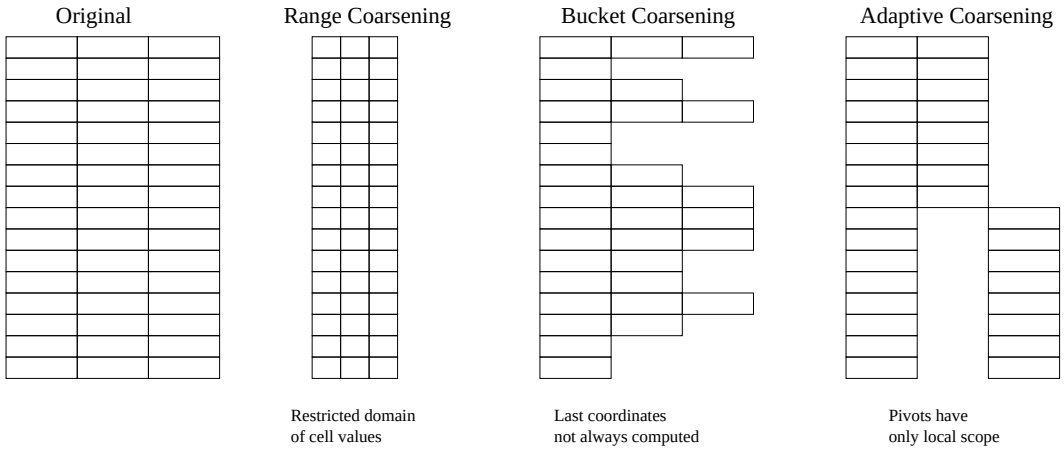

Figure 16: Different coarsification methods. Shorter cells mean smaller domain in cell values.

It must be clear that all these types of coarsenings *reduce* the discriminative power of the resulting equivalence classes, making it necessary to exhaustively consider *more* elements that in the uncoarsened versions of the relations. In the example of the previous section this is amortized by the lower cost to obtain the coarsened equivalence relation. Here we reduce the effectiveness of

the algorithms via coarsening, for the sake of reduced preprocessing time and space overhead.

However, space reduction may have a counterpart in time efficiency. If we use less space, then using the same memory as before we can have more pivots (i.e. larger k). This can result in an overall improvement. The fair way to compare two algorithms is to give them the same space to use.

6.4.1 Range Coarsening

The auxiliary data structures proposed by most authors for continuous distance functions are aimed to reduce the amount of space needed to store the coordinates of the elements in the mapped space, as well as the time to find the relevant classes. The most popular form is to reduce the precision of d. This is written as

$$x \sim_{p,\{r_i\}} y \iff \exists i,\ r_i \le d(x,p) \le r_{i+1} \text{ and } r_i \le d(y,p) \le r_{i+1}$$

with $\{r_i\}$ a partition of the interval $[0,\infty)$. That is, we assign the same equivalence class to elements falling in the same range of distances with respect to the same pivot p. This is obviously a coarsening of the previous relation $\sim_p$.

As in the previous example we can naturally extend our definition to more than one pivot:

$$x \sim_{\{p_j\},\{r_i\}} y \iff \forall p_j,\ \exists i,\ r_i \le d(x,p_j) \le r_{i+1} \text{ and } r_i \le d(y,p_j) \le r_{i+1}$$

Figure 2 exemplifies a pivoting equivalence relation where range coarsening is applied, for one pivot. Points in the same ring are in the same equivalence class, despite that their exact distance to the pivot may be different.

A number of actual algorithms use one or another form of this coarsening technique. VPTs and MVPTs divide the distances in slices so that the same number of elements lie in each slice (note that the slices are different for each pivot). VPTs use two slices and MVPTs use many. Their goal is to obtain balanced trees. BKTs, FQTs and FHQTs, on the other hand, propose range coarsening for continuous distance functions but do not specify how to coarsify.

In this work we consider that the "natural" extension of BKTs, FQTs and FHQTs assigns slices of the same width to each branch: the tree has a uniform arity across all its nodes. At each node, the slice width is recomputed so that using slices of that fixed width the node has the desired arity. This motivates us to define a new data structure

FHQA Is similar to an FQA except because the slices are of fixed width. At each *level* the slice width is recomputed so that a maximum arity is guaranteed.

The difference between FHQTs and FHQAs is not just a matter of storage. In the FHQT each node has a different slice width, while in the FHQA we have a fixed slice width for all the nodes of the same level. This width ensures that the maximum arity in the level does not exceed the desired value, although some nodes may have smaller arity.

The FHQA is not the same FQA of [24], since there they use variable width slices to ensure that the subtrees are balanced in size. This is another form of range coarsening, closer to MVPTs. The original FQA will be called **FMVPA** to distinguish it from the FHQA.

For completeness of the scheme, we define now a new structure which is a combination of FHQTs and MVPTs.

FMVPT The range of values is divided using the $m-1$ uniform percentiles to balance the tree, as in MVPTs. The tree has a fixed height h, as FHQTs. At each node the ranges are recomputed according to the elements lying in that subtree. This differentiates the structure from FQAs, which recompute the slices once per level of the tree according to all the elements of the set. The particular case where $m=2$ will be called **FHVPT**. We could add bucket coarsening to obtain a percentile version of the FQT, but the results are not good.

The combinations we have just created allow us to explain some important concepts. First consider FHQAs and FMVPAs. They are no more than LAESA with different forms of range coarsening (and different algorithms to reduce extra CPU time). They use k fixed pivots and use b bits to represent the coordinates (i.e. the distances from each point to each of the h pivots). So only 2^b different values can be expressed. The two structures differ only in how they coarsify the distances to put them into 2^b ranges. Their total space requirement is kbn bits.

For example, imagine that we select $b=1$, i.e. one bit per coordinate. This needs 1/32 to 1/64 of the space required to store the actual coordinates. The price is that the equivalence relation is coarsened, and therefore less elements will be eliminated using the k pivots. For each pivot we have just two equivalence classes, represented by the bit. The criterion to discard an element is that, along some coordinate, the query plus its tolerance area is totally inside the other class.

However, range coarsening is not just a technique to reduce space. The same space can be used in a different way. In the example above, if we have reduced the space per pivot, say, 32 times, then we may have 32 times more pivots at the same space usage. It is not immediate how much is it convenient to coarsify in order to use more pivots, but it is clear that this technique can improve the overall effectiveness of the algorithm.

Consider the example of a FMVPA with two pivots with one bit per pivot versus one pivot using two bits. In the first case, after two comparisons against the pivots, we have divided the set in four more or less equal classes. In the second case we do the same with just one comparison. Despite that the partitions differ, it is clear that in general using less pivots with more precision should improve the performance. On the other hand, when we have enough bits to discriminate well between all the outcomes of the distance function, adding more bits will not improve the discriminative power, and in return we will have less pivots at the same space usage. This shows that the optimum range coarsening for a fixed space usage is an intermediate value, which unfortunately depends strongly on the specific metric space under consideration.

Another unclear issue is whether fixed slice is better or worse than percentile splitting. On one hand, a balanced data structure has obvious advantages because the internal complexity may be reduced (think on a balanced or unbalanced BKT). Fixed slices produce unbalanced structures since the outer rings have much more elements (since their volume is larger, especially on high dimensions). On the other hand, in high dimensions the outer rings tend to be too narrow if percentile splitting is used (because a small increment in radius gets many new elements inside the ball). If the rings are too narrow, many rings will be frequently included in the radius of interest of the queries (see Figure 17).

An alternative idea (still very preliminary) is shown in [21], where the slices are optimized to minimize the number of branches that must be considered. In this case, each class can be an arbitrary set of slices.

We consider the tree structures now. FHQTs and FMVPTs are almost tree versions of FHQAs

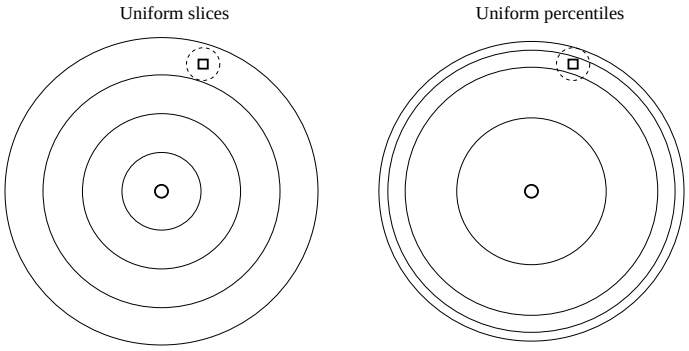

Figure 17: The same query intersects more rings when using uniform percentiles.

and FMVPAs, respectively. They are m-ary trees where all the elements belonging to the same coarsened class are stored in the same subtree. Instead of explicitly storing the m coordinates, the trees store them *implicitly*: the elements are at the leaves, and their path from the root spell out the coarsened coordinate values. This makes the space requirements closer to $O(n)$ in practice, instead of $O(bkn)$ (although the constant is very low for the array versions, which actually take less space). Moreover, the search for the relevant elements can be organized using the tree: if all the interesting elements have their first coordinate in i, then we just traverse the i-th subtree. This reduces the extra CPU time. Indeed, the search algorithm described for FQA in Section 4.1 is borrowed from the tree.

Since under our complexity model there would be no difference between trees and array versions, we have given the trees the ability to define the slices at each node instead of at each level as the array versions. This allows them to adapt better to the data, but the values of the slices used need more space. It is not clear whether it pays off or not to store all these slice values.

A particular case where a tree can be built without coarsening the equivalence relation is that of discrete distance functions, i.e. when the set of values returned by d is finite and reasonably small. In FHQTs the pivots are fixed at each level, and at depth i each element x is stored at subtree $d(x, p_i)$. If compared to simply storing the k discrete values for each element, we can see that an FHQT is a *trie* [40] data structure storing n strings, where each string is the sequence of k "symbols" (coordinates) that identify each class. The leaves are the classes. This allows us to structure the search using the tree, since we have to traverse only the subtrees numbered $d(q, p_i) - r$ to $d(q, p_i) + r$. FQAs can be used on discrete distance functions as well, without any coarsening. They implement a similar search strategy without using a tree.

If the distance is too fine-grained, the root will have near n children and the subtrees will have just 1 children. Hence, the structure will be very similar to a table of k coordinates per element. The FHQT reduces extra CPU time assuming that the distance is not fine grained, otherwise it degenerates to a row-wise linear search. If this is the case, the equivalence relation can be coarsened as in the previous examples to improve extra CPU time at the expense of more distance evaluations. BKTs and FQTs are similar to FHQTs, but they involve other techniques that are explained shortly.

Summarizing, range coarsening can be applied using fixed-width or fixed-percentile slices, and other optimized schemes are possible. They can reduce the space necessary to store the coordinates,

which can allow to use more pivots with the same amount of memory. Therefore, it is not just a technique to reduce space but it can improve the search complexity. Range coarsening can also be used to organize tree-like search schemes which are sublinear in extra CPU time.

6.4.2 Bucket Coarsening

To reduce space requirements in the above trees, we can avoid building subtrees which have few elements. Instead, all their elements are stored in a *bucket*. When the search arrives to a bucket, it has to exhaustively consider all the elements.

This is a form of coarsening, since for the elements in the bucket we do not consider the last pivots, and resembles the previous idea (Section 6.3) of not computing the k pivots. However, in this case the decision is taken off-line, at index construction time, and this allows reducing space by not storing those coordinates. In the previous case the decision was taken at search time. The crucial difference is that if the decision is taken at search time, we can know exactly the total amount of exhaustive work to do by not taking further coordinates. On the other hand, in an off-line decision we can only consider the search along this branch of the tree, while we cannot predict how many branches will be considered at search time.

This idea is used for discrete distance functions in FQTs, which are similar to FHQTs except for the use of buckets. It has been also applied to continuous setups to reduce space requirements further.

6.4.3 Adaptive Coarsening

The last and least obvious form of coarsening is the one we call "adaptive coarsening". In fact, the use of this form of coarsening makes it difficult to notice that many algorithms based on trees are in fact pivoting algorithms.

This coarsening is based on, instead of storing all the coordinates of all the elements, just storing some of them. Hence, comparing the query against some pivots helps to discriminate on some subset of the database only. To use this fact to reduce space, we must determine off-line which elements will store their distance to which pivots. There is a large number of ways to use this idea, but it has been used only in the following way.

In FHVPTs there is a single pivot per level of the tree, as in FHQTs. The left and right subtrees of VPTs, on the other hand, use *different* pivots. That is, if we have to consider both the left and right subtrees (because the radius r does not allow to completely discard one), then comparing the query q against the left pivot will be useful for the left subtree only. There is no information stored about the distances from the left pivot to the elements of the right subtree, and vice-versa. Hence, we have to compare q against both pivots. This continues recursively. The same idea is used for BKTs and MVPTs.

Although at first sight it is clear that we reduce space, this is not the direct way in which the idea is used in those schemes. Instead, they combine it with a huge increase in the number of potential pivots. For each subtree, an element belonging to the subtree is selected as the pivot and deleted from the set. If no bucketing is used, the result is a tree where each element is a node somewhere in the tree and hence a potential pivot. The tree takes $O(n)$ space, which shows that

we can successfully combine a large number of pivots with adaptive coarsening to have low space requirements (n instead of n^2 as in AESA).

The possible advantage (apart from guaranteed linear space and slightly reduced space in practice) of this structure over those that store all the coordinates (as FQTs and FHQTs) is that the pivots are better suited to the searched elements in each subtree, since they are selected from the same subset. This same property is which makes AESA such a good (though impractical) algorithm.

In [51, 15] they propose hybrids (for BKT and VPT, respectively) where a number of fixed pivots are used at each node, and for each resulting class a new set of pivots is selected. Note that, historically, FQTs and FHQTs are an evolution over BKTs.

6.5 Voronoi Equivalence Partitions

A very good algorithm for proximity searching in 2-dimensional Euclidean spaces is obtained with the Voronoi partition [3, 58]. The entire space is divided into n equivalence classes, each one assigned to one dictionary element. The equivalence relation is defined as influence zones: a point $x \in \mathbb{X}$ is assigned to the j-th equivalence class if $u_j \in \mathbb{U}$ is the nearest neighbor of x. It is clear that this partition is as local as possible, and that it has maximum discriminative power since it assigns one dictionary element per equivalence class.

A "Voronoi graph" is defined, where the nodes are the elements of $\mathbb{U}$ and the edges connect points whose influence zones have borders in common. Using this graph, an $O(\log n)$ algorithm exists to find the nearest neighbor of an arbitrary point. Unfortunately, this algorithm does not generalize to more than two dimensions. The Voronoi graph, which has $O(n)$ edges in two dimensions, can have even $O(n^2)$ edges in three and more dimensions.

A simple search algorithm that can be used in more than two dimensions is to start at any point and move to any neighbor[6] closer to the query. When this is not possible anymore we are already in the element closest to the query.

This idea can be generalized to an arbitrary metric space. Unfortunately neither the Voronoi graph nor a non-trivial superset of it can be built using only the distance matrix as input, as proved in [44]. Specific knowledge of the particular metric space is necessary.

The algorithm SAT is a simplification of this idea, where the search can only start at a fixed element of $\mathbb{U}$. The Voronoi partitions are built considering only the elements of $\mathbb{U}$, not $\mathbb{X}$. The result would be a structure able to search only queries of type (b) for $q \in \mathbb{U}$ (quite useless). To search general queries $q \in \mathbb{X}$ of type (a), the algorithm assumes that an *unknown* element $u \in \mathbb{U}$ is searched (type (b) query), from which one only knows that it is at distance at most r to q. This allows to bound the possible distances from u to other elements via comparing them against q instead of u. Since the exact distance is not known, the algorithm has to explore many neighbors which could be the closest to u (some can be discarded thanks to the known bounds).

The other algorithms which rely on a Voronoi-like partition of the space are GHT and GNAT. This time, however, the search for the relevant classes is exhaustive (not as in SAT), and the algorithms resort to a hierarchical partitioning to improve the search time.

[6] In the graph sense, i.e. directly connected to the point by a direct link.

In the GHT, two pivots p_1 and p_2 are selected and the elements closer to p_1 are in one class and those closer to p_2 are in the other. GNAT is a generalization of GHT to m pivots. The actual algorithms are simply a hierarchical partition idea used on this equivalence relation. That is, they divide the space using m pivots and recursively continue inside each partition.

Notice, however, that we have to perform m distance evaluations to discriminate among m classes, which may be expensive. Moreover, what we need is basically to find the pivots close to our query q, i.e. basically our same original problem with less elements. If m is large, this could be done by using another search algorithm (e.g. AESA) to find the relevant classes instead of testing all them. This has not been attempted up to now, and the $O(m^2)$ distances needed by AESA are already stored in the GNAT.

GHTs and GNATs use many of the coarsening techniques presented before. For instance, when we compute the distance to the m pivots, we have mapped the space onto an m-dimensional vector space. However, from that coordinate information we only store, for each element in the set, which is the smallest coordinate (i.e. which is the closest pivot). This is, again, to reduce space[7]. They also use bucket and adaptive coarsening. A GHT or GNAT with fixed pivots per level has not been proposed up to now. Since it fits well in our taxonomy, we define such a structure now.

FGNAT Is a GNAT where there is a selected set of m pivots per level (not per node). Hence, the subsets obtained after the first partition are partitioned in the second level according to a new set of pivots which is the same for all the sets. The case $m = 2$ is called **FGHT**.

The Voronoi partition is an attempt to obtain local classes, more local than those based on pivots. A general technique to do this is to identify clusters of close objects in the set. There exist many clustering algorithms to build equivalence relations. However, most are defined on vector spaces instead of general metric spaces. An exception is [17], which reports very good results. However, it is not clear that good clustering algorithms directly translate into good algorithms for proximity searching. Another clustering algorithm, based on cliques, is presented in [19], but the results are similar to the simpler BKT. This area is largely unexplored, and the developments here could be converted into improved search algorithms.

Given any equivalence relation, the hierarchical partitioning scheme can be applied to obtain a new search algorithm. That is, we first partition the set in very coarse classes, and find the relevant ones. Then, instead of exhaustively searching in the resulting classes, we partition them again into subclasses and search again. When the classes have few enough elements, we search them exhaustively. Of course the partitioning into classes is done beforehand, not at query time.

The algorithm resulting from this hierarchical scheme is qualitatively different from the base partitioning and searching algorithm used. We can also use different search algorithms at each level of the partition, resulting in hybrid algorithms which have not been considered up to now.

As an example, the tree data structures covered previously can be considered as different hierarchical extensions to the simple algorithm that takes only one pivot and groups the elements in classes according to their distance to the pivot. The same can be said of GHTs and GNATs.

[7] It is interesting that for both VPTs and GNATs they suggest to store some of the actual distances between the elements and the pivots to reduce the coarsening.

Range coarsening	With Adaptive Coarsening	Without Adaptive Coarsening
Fixed slices	BKT	FQT, FHQT, *FHQA*
Fixed quantiles	VPT, MVPT, VPF	*FMVPT*, FMVPA
No range coarsening		AESA, LAESA-like
Voronoi-like	GHT, GNAT, SAT	*FGNAT*

Table 2: Taxonomy of the existing algorithms. The methods in italics are combinations that appear naturally as soon as the taxonomy is built. The Voronoi-like coarsening is separated because it is not a type of range coarsening. All adaptive coarsening structures, as well as FQTs, can do bucket coarsening as well.

6.6 Summary

Table 2 summarizes our classification of methods attending to the most important features (e.g. we left aside the algorithm to traverse the set in order to reduce extra CPU time).

7 Experiments

In this section we experimentally test and validate many of the facts and open questions discussed in the paper. We also compare the existing approaches and find their best combinations, obtaining recommendations on what to use depending on the case.

We concentrate only on the number of distance evaluations as our complexity measure. All the data structures have been carefully implemented by us as prototypes, and despite that their CPU time could be reduced by better coding, their number of distance evaluations reported depends only on the algorithm.

In our attempt to be fair with the memory usage of the algorithms and at the same time simplify the implementation, we have not optimized their main memory implementation but have been very careful to count their space usage. For instance, the trees can be stored with no pointers. In an extreme case one could work with those representations (perhaps at extra CPU cost). Careful implementations may obtain better CPU performance and approach the space requirements of our representations. We consider that real numbers take 4 bytes, element identifiers of $\mathbb{U}$ take 4 bytes, and arities and others small numbers take as many bytes as necessary.

We have implemented the data structures and search algorithms in ANSI C, and ran the code on a Dual Pentium II of 333 Mhz, 256 Mb of RAM, under Linux kernel 2.2.1.

In which follows we describe the structures we have implemented, including how we compute their most significant space usage in addition to the n identifiers of the elements of $\mathbb{U}$ that need to be stored.

BKT We use slices of fixed width, which is defined to keep a desired arity m in the tree (m is a free parameter). The bucket size is always 1, which gives the optimal time performance for this data structure. The pivot at the root of each subtree is randomly selected from the elements that lie into that subtree. The space usage is n distances, to store the slice widths used.

FQT The same as BKT, except that the pivots are randomly selected from $\mathbb{U}$. The pivot ids are too few to be significant. The space usage is a distance per node of the tree (for the slice widths). This data structure does not appear in the experiments because it took too much time and space to build. This shows that the extension of fixed slices that we designed does not work well in this case.

FHQT The same as FQT, but this time the height is fixed at h. We have used different h values depending on the case, but the optimal h is unreachable with our machine. The space usage is n distances (to store the slice widths at each node) plus that of FHQAs, which give a more compact implementation.

FHQA Given b bits to represent the coordinates, the ranges are split in 2^b slices. The slice widths are computed once per level to obtain at most 2^b slices in that level. The performance depends also on the parameter h, i.e. number of pivots used. The pivots are randomly chosen from $\mathbb{U}$. The space usage is hbn bits, the rest is not significant.

MVPT We use buckets of size 1 to maximize the performance and leave the arity m as a parameter. The root of each subtree is randomly chosen from the elements that lie into that subtree. VPTs are the particular case $m = 2$. We have not included extensions such as storing some actual distances at the leaves because this is a hybrid with LAESA that would complicate isolating the effectiveness of the ideas. For the same reason we have not extended the structure to more than one pivot per node (a hybrid with FMVPT). The space usage is n distances. Despite that the BKT needs just a distance (slice width) per node and here we need m distances per node, we can charge the cost of the distances to the corresponding edges and then realize that there are $n - 1$ edges.

FMVPT Same as FHQT, except that instead of fixed slices we use fixed percentiles. The space usage is similar to FHQTs.

FMVPA Same as FHQAs, except that fixed percentiles are used instead of fixed slices. The space usage is the same as FHQAs.

GNAT We leave the arity m as a parameter and use buckets of minimal size (also m) to optimize the number of distance evaluations. The arity m is kept constant across the tree. Although the authors suggest to reduce the arity at deeper nodes, they give no clear criterion to modify the arity. The m pivots at each node are randomly selected from the subtree. The authors suggest to take elements far away from each other, but we obtained worse results doing this, and in any case random elements are quite far away in high dimensions. GHT is the particular case $m = 2$. The space requirements are n distances (the maximum radii of the subtrees).

FGHT Same as GHT but the pivots are the same for all nodes at each level. The height h is a parameter. The space usage is hn bits.

LAESA We leave the number k of pivots as a parameter. We use indeed the implementation of [23] (spaghettis), which is much faster in terms of CPU time, but count the space of LAESA which is minimal (kn distances). The reason is that LAESA would perform the same number

of distance evaluations (but our experiments would take many days). In particular, AESA has not been implemented because either a real implementation or a simulation would take weeks in our machine. In the cases where LAESA has been compared against other structures and it was restricted to use the same amount of memory, it should have used just 1 pivot (32 bits). Since it performed poorly and it can be argued that real values could need less bits, we have allowed it to use 4 pivots in this case. In other cases, where it has been allowed to use more memory, we assume that each pivot needs 32 bits to compute the amount of memory used.

SAT There are no tuning parameters in this algorithm. The space is n arities plus n distances (radii of subtrees).

As our metric space, we have used synthetic points in vector spaces with Euclidean (L_2) distance. Our coordinates are randomly chosen in the real interval $[0, 1]$. Despite that the set is actually a vector space, we take it as an abstract metric space and do not make any use of coordinate information. The reason to use this set instead of real-world metric spaces is that with this setup we can control exactly the dimensionality of the space and show how the performance is affected as the dimension varies. If we used abstract metric spaces or real-world data on vector spaces, the intrinsic dimensionality would not be obvious (either because there is no explicit dimension or because real vector data normally has lower dimension than the space it is in). Hence, we would have to guess the intrinsic dimension by considering the histograms or the behavior of the structure and the observations would be complicated. Despite that considering real world data is interesting, using synthetic data where the dimension is clear fits better our purposes of understanding the basic features of the problem with a clean experiment. We use $n = 100,000$ to $n = 500,000$, and up to 20 dimensions (as seen later, no method works well for more than 20 dimensions). Nevertheless, we present at the end an example using a real-world set and a discrete distance function.

We use a search radii that on average retrieves 0.01% of the dictionary. This is a reasonable output size in many applications. However, we include at the end a test with radii retrieving up to 0.1% of the set.

We have repeated 1000 times the experiments. Since, however, most indexing algorithms involve selecting pivots at random and the final performance strongly depends on that selection, we have built 10 different indices on the same data and run 100 random queries on these indices. Points and queries are generated at random.

In which follows, we present our experimental results. The sequence of experiments corresponds to the order in which the main concepts are introduced and the open questions appear.

7.1 Optimal Number of Pivots

The first important concept in the model of k fixed pivots is that by incrementing their number the internal complexity increases and the external complexity decreases, and therefore an optimal number of pivots exists. We would like to (1) show that there exists an optimal k^*, (2) show how the optimal k^* grows quickly with the dimension of the space and that it is $O(1)$ or $O(\log n)$ with respect to the set size (recall that there exist different analysis concluding either choice), and (3) show that the cost using the optimal k^* is $O(k^*)$.

Unfortunately, we need too much space to obtain serious figures of (2) and (3). Storing 4-byte coordinates for $n = 100,000$ needs 100 Mb for $k = 256$ pivots, which is smaller than the optimum for more than 12 dimensions. So we content ourselves by showing that for 8 dimensions there exists an optimum near $k^* = 110$. Figure 18 shows internal, external and total distance evaluations in 8 dimensions, using a LAESA-like algorithm with up to 256 pivots. Recall that the algorithm simply compares the query against the k pivots (internal complexity) and then compares the query against the elements that cannot be discarded using the triangular inequality with the k distances computed (external complexity).

Despite that with 256 pivots we reach an optimum only until 8 dimensions, it seems clear that the optimal k^* grows much faster than the number of dimensions. The optimal k is 15, 30 and 110 for 4, 6, and 8 dimensions, respectively. This matches with many theoretical predictions that say that k^* is exponentially depending on the dimension, and therefore even if we were able to have k^* pivots the cost would grow exponentially with the dimension (just to compare the query against the pivots). This can be improved with a more careful selection of pivots, a problem quite poorly understood today. Figure 18 shows also the case of higher dimensions, where the optimum is not achieved with 256 pivots.

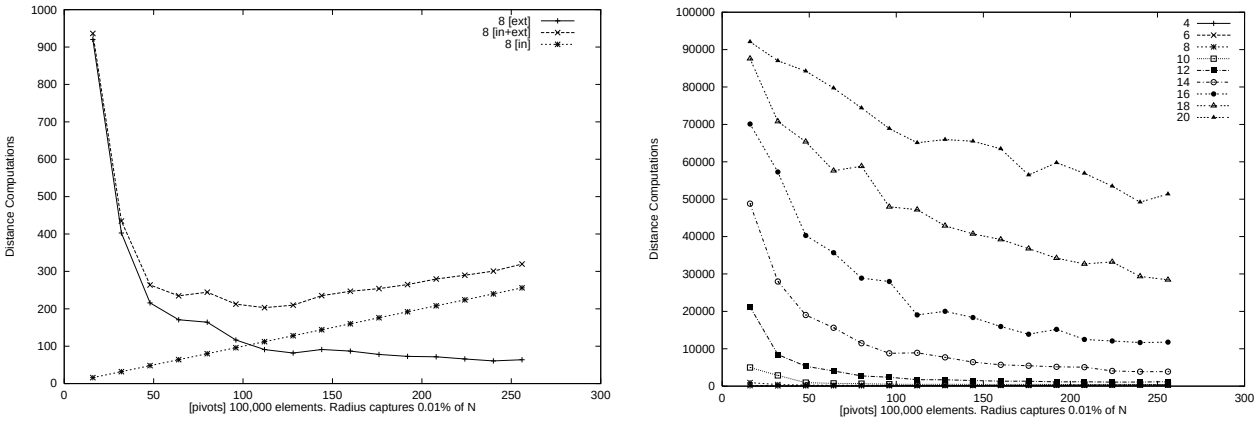

Figure 18: On the left, internal, external and overall distance evaluations in 8 dimensions, using different number of pivots. On the right, overall distance evaluations for different number of dimensions.

7.2 Amount of Range Coarsening

We have presented many choices related to range coarsening. A first question is how much range coarsening is desirable. It is clear that, if the number of pivots is fixed, range coarsening reduces memory usage but degrades the complexity. As explained, however, range coarsening can be used to improve the overall effectiveness of the algorithm if the memory is fixed, because if one needs less space to store the distances then more pivots can be added. The appropriate tradeoff between number and precision of the pivots is not obvious.

Figure 19 shows the effect of trading number for precision in the pivots, so that the total number of bits used is the same. The top four figures correspond to the four forms of range coarsening

(FHQT, FHQA, FMVPT, FMVPA). FHQTs and FMVPTs need more space because they have to store n extra distances. We use 16 bits per element in the trees and 32 bits per element in the arrays. This still favors the trees (which are using 48 bits per element when one counts the n distances), but even in this case they are inferior, as seen later. The results show that the optimal arity is always an intermediate value. The reason can be found in Section 6.4.1.

The two bottom plots of Figure 19 show the effect of range coarsening combined with adaptive coarsening: BKTs and MVPTs are studied. The optimal type and amount of range coarsening is different if we have adaptive coarsening, since the height is not fixed. Larger arity implies better discrimination, but also less pivots in the path to a leaf. Another way to see it is that the result of a comparison is useful for more elements if the arity is reduced. Interestingly, all achieve their best performance when their arity is minimal (2).

7.3 Type of Coarsening

Now that the optimum amount of range coarsening for each structure has been identified, we proceed to compare the different types of range and adaptive coarsening. From now on, each structure uses its optimal amount of range coarsening.

The two first plots of Figure 20 compare the type of range coarsening used in the structures that have or do not have adaptive coarsening, separately. The goal is to determine which type of adaptive coarsening is better: with fixed slices or with fixed percentiles.

The plots make clear that the arrays work better than the trees (even when they are using less memory), which shows that it is not a good idea to pay so much memory to ensure a better partitioning at each node. Finally, FMVPAs work better than FHQAs on low dimensions, and the situation is reversed for high dimensions, where the phenomenon discussed in Section 6.4.1 (especially in Figure 17) becomes more noticeable. Balancing the tree as is done in VPTs and MVPTs has impact only in low and medium dimensions, but in high dimensions the price of having to traverse more children at search time shows up. Adaptive coarsening techniques show the same effect, but in that case BKTs (fixed slices) are superior to MVPTs (percentile slices) for all dimensions.

The four last plots of Figure 20 aim at determining which is better between having or not having adaptive coarsening. BKTs and FHQAs are compared among them; and MVPTs and FMVPAs are compared among them. In both groups we have added LAESA (no range coarsening). The tree versions (FHQTs and FMVPTs) were excluded because we know already that they make worse use of the memory than their array versions.

Since BKTs and MVPTs use fixed amount of memory and FHQAs, FMVPAs and LAESA can use more and more memory by incrementing h (or k), two questions are of interest: (1) which is better if they are allowed to use the same amount of memory? (2) how much memory needs the second kind of algorithms to beat the algorithms of the first kind?

The two middle plots answer question (1), and the last two, question (2). The ranges used are the best for each structure. Recall that LAESA uses 4 pivots for the case (1), despite it should use just one. As it can be seen, the structures that use adaptive coarsening improve over those that do not, if all use the same amount of memory. On the other hand, these last structures beat the first ones if more memory is used: LAESA and FMVPA need 16 and 2 times more memory, respectively, than MVPTs to beat them; LAESA and FHQA need 128 and 32 times, respectively,

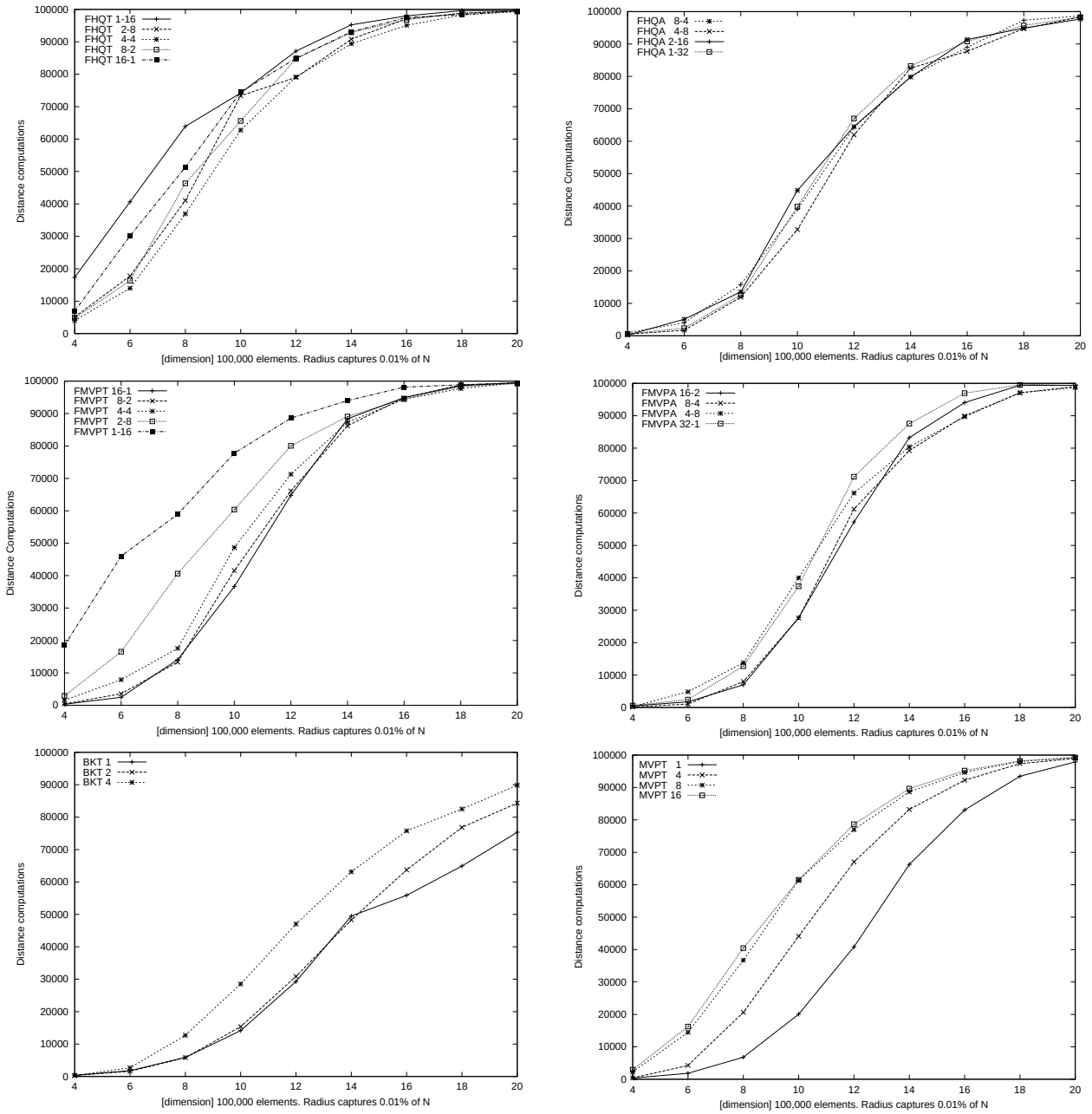

Figure 19: Optimal arity for the different structures, using fixed memory for those without adaptive coarsening. The legends for the top four structures use the format STR h-b, where STR is the structure, the arity is 2^b and the number of pivots is h. The last two use the format STR b, with the same meaning.

more memory to beat BKTs. In particular, beating BKTs becomes harder as the dimension grows.

Finally, we note that the results for BKTs are too good to leave them without further analysis. Studying them in more detail, we find that the height of the trees is extremely large, so the trees are very ill balanced. The reason is quite clear: as the dimension grows, dividing the distances in two equal slices leaves very few elements in one branch of the tree and almost all the rest in the other. What we finally obtain is a *clustering* (or Voronoi-like) scheme, where all the internal nodes are the cluster representatives (indeed, they are good candidates for centroids), and the smallest of the two subtrees has all the elements of the cluster (see Figure 21). There is a clear criterion to select the clusters and their radii and to put the other elements in the clusters. This shows an interesting (although loose) connection between clustering and pivoting algorithms. The turning point where this pivoting schemes behaves more like a clustering scheme seems to be 14 dimensions.

We therefore consider the binary BKT as a new clustering (or Voronoi-like) algorithm, byproduct of this survey. This structure takes linear space and close to quadratic construction time. This cost could be lowered using another structure to build the tree, so as to perform the required queries to fill the subtrees more efficiently.

7.4 Voronoi Equivalence Algorithms

Figure 22 shows a comparison between GHTs and FGHTs, letting FGHTs use different amounts of memory (GHTs are equivalent in memory usage to FGHTs of height 32). It is clear that FGHTs only behave well for very low dimensions. The reason is that the partitions have high locality, and selecting a new partition from the global set of elements (instead of the local set of elements) makes it quite possible that all the objects lie in a single partition at the second level, and so on.

With respect to GNATs, we need also to determine the best arity. Figure 22 shows the results with GNATs using different arities. In general $m = 128$ seems a good choice. Another interesting fact is that GNATs seem more resistant do the dimension than other structures. SATs, on the other hand, do not have parameters to tune, and we defer this structure for the final comparison.

7.5 Final Comparison

We compare now the different resulting approaches, using their optimal setup. These are: BKTs with arity 2, FHQAs, FMVPAs, LAESA, GNATs with arity 128, and SAT. These are compared in two ways. First, we make a comparison using the same amount of memory for all. This turns out to be 32 bits per element. Second, we show how much more space do FHQAs, FMVPAs and LAESA need to beat the other approaches.

Figure 23 shows the comparative results. Using the same amount of memory, the binary BKT is definitively the most efficient data structure, followed in high dimensions by other Voronoi-like schemes. It becomes clear that the Voronoi-like techniques are more efficient than pivot-based algorithms on high dimensions. This shows that the locality of the partition becomes important at some point.

On the other hand, FHQAs, FMVPAs and LAESA have the ability to improve their performance by using more pivots. LAESA needs 128 times more memory and FHQAs and FMVPAs need 32 times more memory to beat the binary BKT in 20 dimensions.

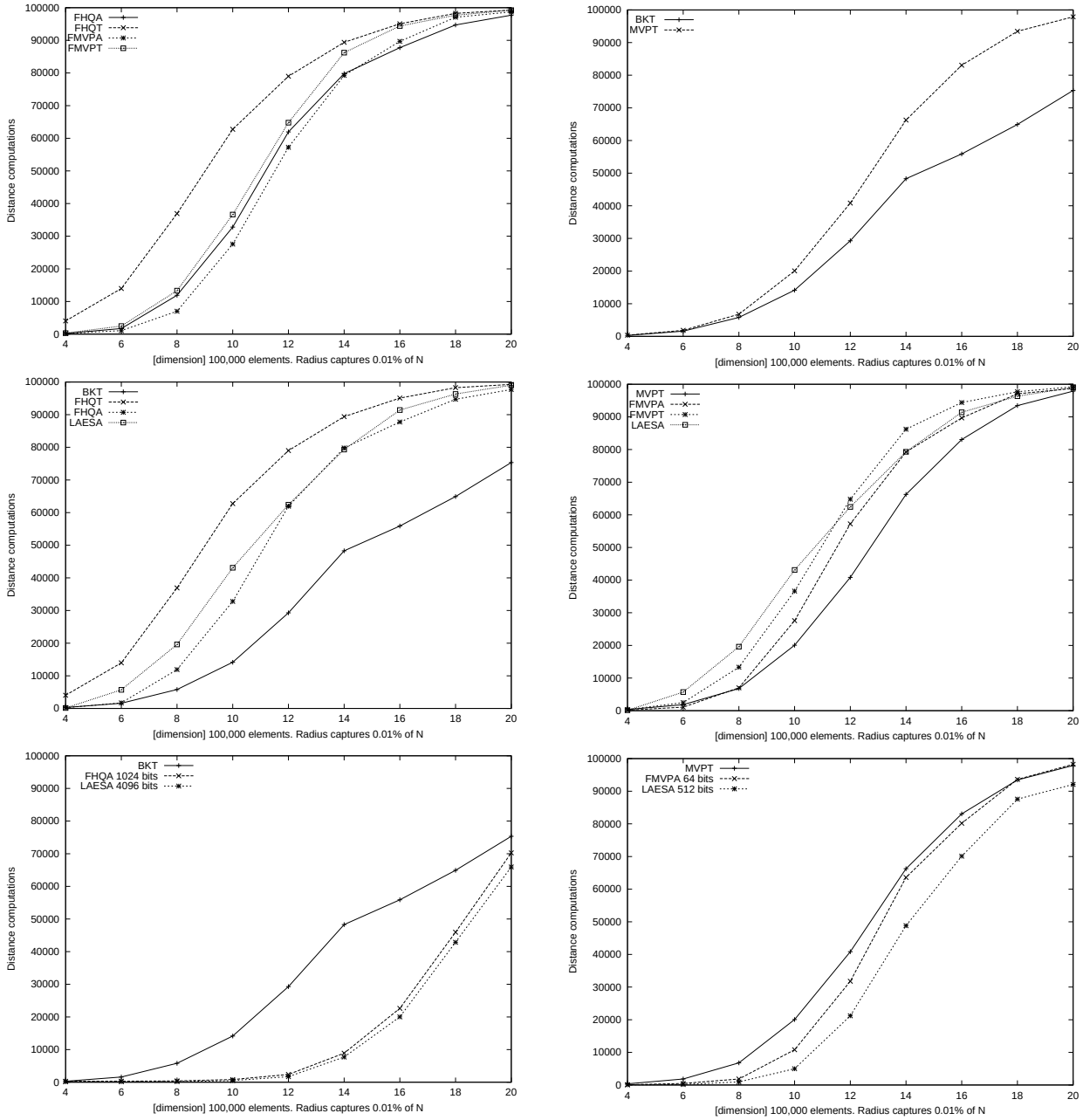

Figure 20: The first two plots compare the different range coarsening techniques. The middle plots compare the use or not of adaptive coarsening, at the same space usage. The lower plots show how much space need the structures that do not use adaptive coarsening to beat those that use. The left plots show range coarsening of fixed slices and the right plots show fixed percentile slices.

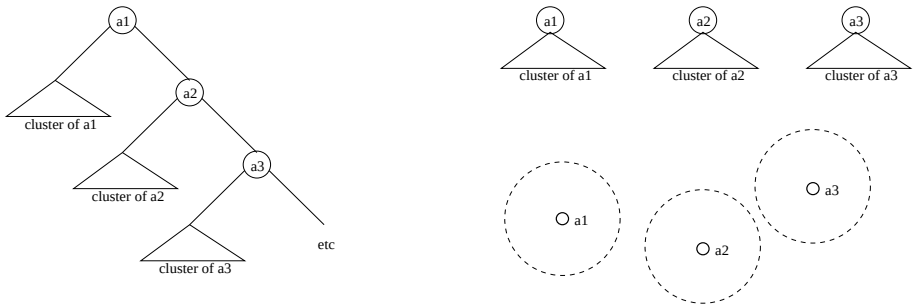

Figure 21: The biased binary BKT is similar to a clustering scheme.

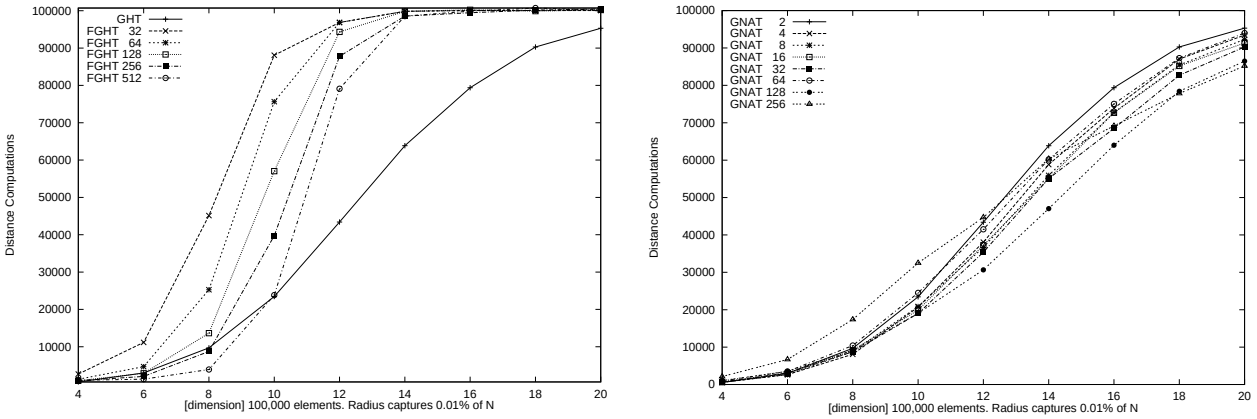

Figure 22: The left plot compares GHTs and FGHTs using different amounts of memory. The right plot compares GNATs using different arities.

It is interesting to notice that, as the dimension grows, pivoting algorithms need more and more pivots to beat Voronoi-like partition algorithms. On the other hand, there are no good methods to allow Voronoi-like algorithms using more memory to increase their efficiency, but if they existed probably they would need less space to obtain the same results of the other algorithms on high dimensions. Note also that only array implementations of FHQTs and FMVPTs make the necessary heights possible in practice.

We believe that a data structure where the idea of a Voronoi-like partition is combined with a k-pivoting algorithm may provide the best tradeoff.

Figure 24 shows a comparison among the structures for increasing n and 8 dimensions, using the same amount of memory. The goal is to study how the costs grow with n.

It can be seen that the costs grow linearly if we retrieve 0.01% of the data set. It is not hard to see that the cost cannot be sublinear if we retrieve a linear proportion of the set. Since our set has limited volume in the space (recall that all points lie in $[0,1]^k$ for k dimensions) the radius to retrieve 0.01% of the set remains the same as n grows, since all the volume gets uniformly denser and therefore a fixed volume remains holding a fixed proportion of the points. If the space were unbounded (e.g. uniform density as n grows) a fixed radius would retrieve lower and lower

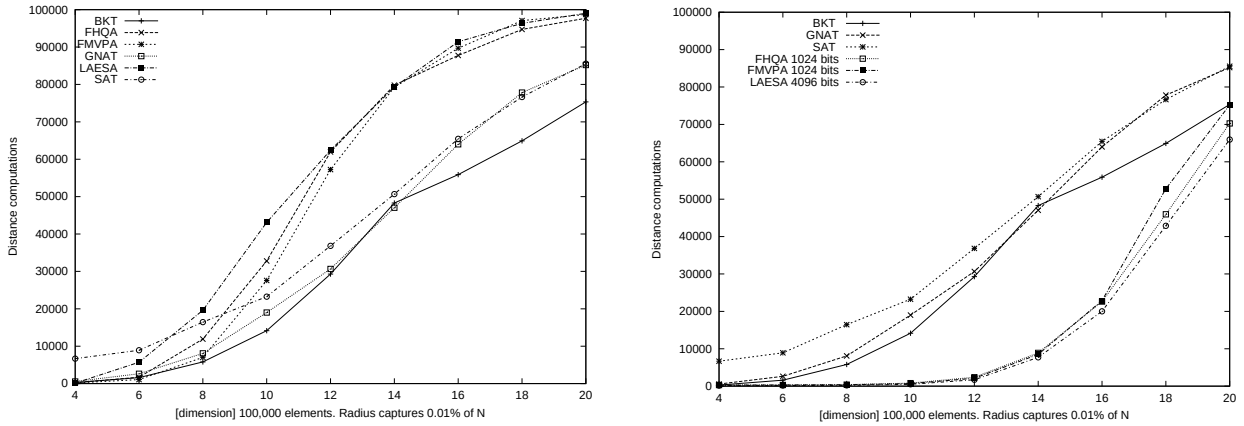

Figure 23: Comparing the best data structures. The left plot uses the same amount of memory for all, while in the right one the pivoting algorithms use more space.

percentages of the set as n grew. We see shortly an example of such a set.

Figure 24 also shows how time grows as the proportion of retrieved elements increases. As it can be seen, the efficiency degrades slowly as the radius grows. Some structures, like SATs, adapt better to larger search radii.

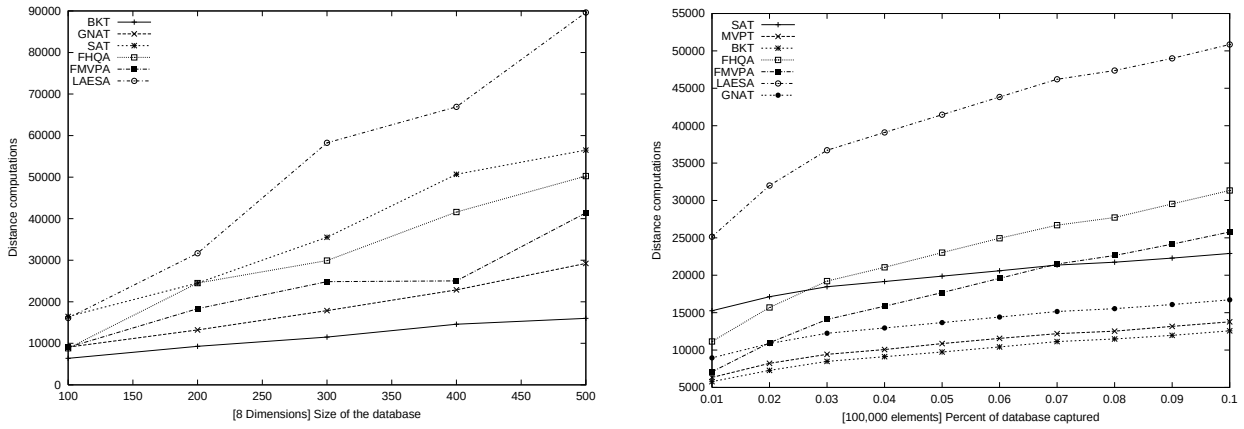

Figure 24: Comparing the data structures using the same amount of memory and in terms of n, for 8 dimensions. The left plot retrieves 0.01% of the set, while the right plot retrieves from 0.01% to 0.1% of the set.

7.6 Construction Time

We have left aside the cost to build the data structures. Since this is done off-line, it has less importance than querying complexity, but nevertheless it is worhwhile to present some results.

Figure 25 shows the construction complexity (number of distance evaluations) for the structures compared in the previous subsection, using the same amount of memory. We show the growth in

terms of the set size and the dimension. As it can be seen, Voronoi-like schemes (SATs, GNATs and binary BKTs) are much more expensive to build than the other data structures.

With respect to the dimension, the schemes with fixed pivots of course do not depend on the dimension, while Voronoi-like schemes become more expensive to build as the dimension grows. An exception seems to be GNATs, which are fixed with the dimension because the arity is so large (128) that the height is always two. With less arity they would be cheaper to build but less competitive. In particular, the cost of the best structure, BKTs, increases sharply as the dimension grows. This shows that there is a high price at construction time for Voronoi-like schemes, which are the most competitive in high dimensions.

With respect to the set size, most schemes are linear (except SATs, which are slightly superlinear as predicted). The only one which deviates from the prediction is the binary BKT, for which we predicted a quadratic construction time. More specifically, if the left subtree holds M elements (where M is much smaller than n) then the tree will be of height n/M and the construction time will be n^2/M. However, our set is compact in $[0,1]^k$, and therefore the distance that divides the left and right subtree keeps basically unchanged as n grows, and so does the volume of the ball that corresponds to the left subtree. Hence, as n grows, M grows proportionally. If we double n and M, the construction cost doubles too: $(2n)^2/(2M) = 2(n^2/M)$.

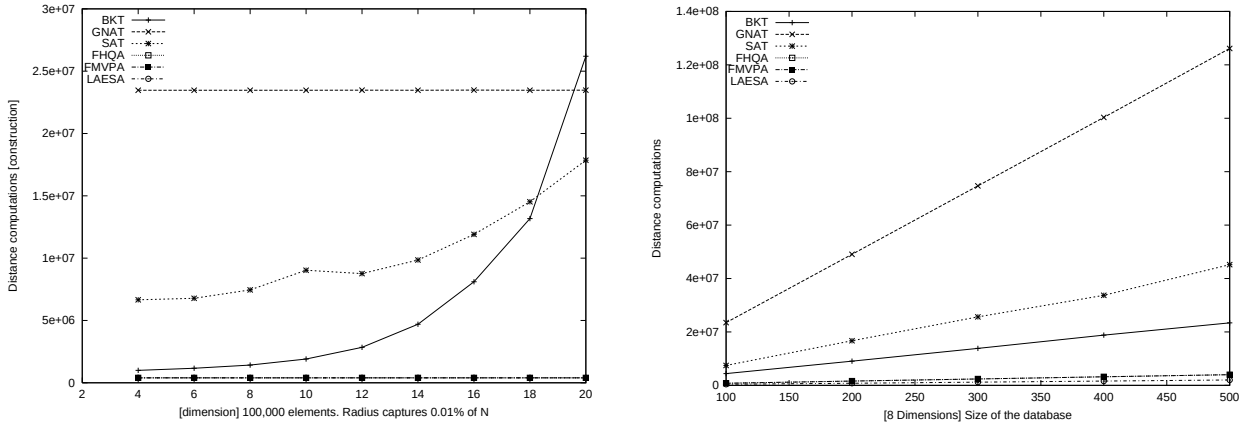

Figure 25: Construction times, as a function of the dimension (left) and as a function of the set size (right). LAESA, FHQAs and FMVPAs are very close and superimposed in some cases.

7.7 A Real-World Discrete Case

We finish our experiments with a real-world example related to text processing. Our set is composed of 500,000 different lower-case words, namely the vocabulary of a subset of the TREC collection [38]. The distance is the *edit distance* (recall Section 2.3), which is the minimal number of character insertions, deletions and replacement needed to make two strings equal. This returns a small integer number in most cases, and the histogram is quite concentrated.

We compare the different data structures on this set, searching with radius 1, 2 and 3. The queries were selected at random from the same dictionary. The structures included are BKT, FQT, FHQT ($h = 16$, so that an FQA implementation uses roughly the same space as the rest), GNAT

($m = 50$, which had the best performance) and SAT. No range coarsening was performed, since the outcome of the distance function has just a few different values.

It can be seen that GNATs and SATs are not competitive in this case (probably because of the discretization). FQTs, BKTs and FHQTs are quite similar, but FQTs and FHQTs are slightly better.

Their time sublinearity is also clear from the figures, which may be unexpected because we are using fixed radii (1, 2 and 3) and therefore a fixed proportion of the set should be retrieved. However, the space is discrete and the number of elements in $\mathbb{X}$ at small distance from a string is severely limited, so it does not grow with n. As n grows, fixed radii retrieve lower percentages of the set.

This shows that the structure of the metric space has a strong influence in all aspects of the the behavior of the algorithms.

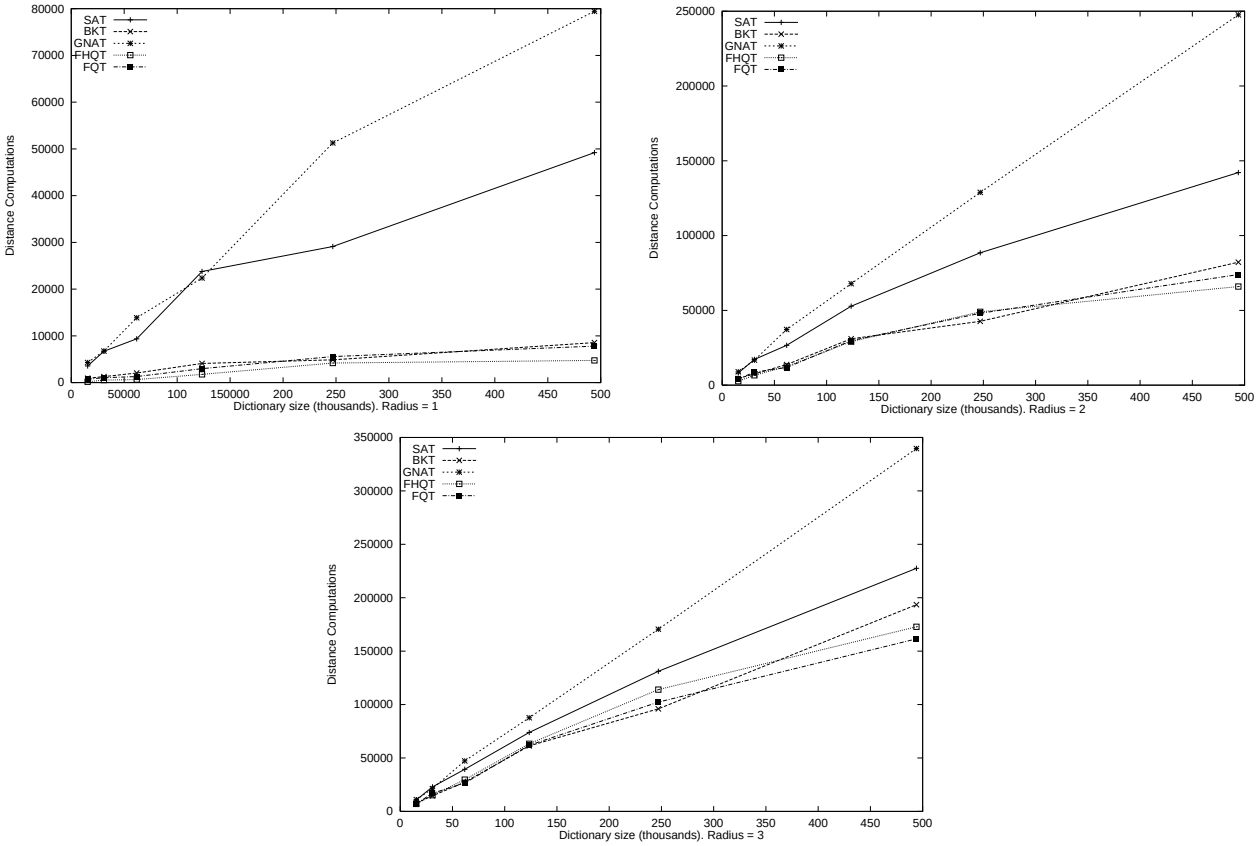

Figure 26: Comparison on the TREC dictionary, for radius from 1 to 3. FHQT uses $h = 16$ and GNAT uses $m = 50$. No range coarsening is done.

8 Conclusions

Metric spaces are becoming a popular model for similarity retrieval in many unrelated areas. We have surveyed the algorithms that index metric spaces to answer proximity queries. We have not just enumerated the existing approaches to discuss their good and bad points. We have, in addition, presented a unified framework that allows to understand the existing approaches under a common view. It turns out that most of the existing algorithms are indeed variations on a few common ideas, and by identifying them, previously unnoticed combinations have naturally appeared, some of them extremely efficient. We have also analyzed the main factors that affect the efficiency when searching metric spaces. Finally, we have presented experimental results validating our assertions and comparing the existing approaches. As a result, we have been able to recommend the best choices among the existing solutions.

The main conclusions of our work are summarized as follows

1. The factors that affect the efficiency of the search algorithms are the intrinsic dimensionality of the space and the proportion of the set that is retrieved.

2. We have identified the use of equivalence relations as the common ground underlying all the indexing algorithms, and classified the search cost in terms of internal and external complexity.

3. A large class of search algorithms rely on taking k pivots and mapping the metric space onto $\mathbb{R}^k$ using the L_∞ distance. Another important class uses Voronoi-like partitions.

4. The equivalence relations can be coarsened to save space or to improve the overall efficiency by making better use of the pivots.

5. Although there is an optimal number of pivots to use, this number is too high in terms of space requirements. Hence, in practical terms, this type of index will outperform those that use fixed space if it has enough memory. However, among them, range coarsening allows to make better or worse use of the same amount of memory. We found that intermediate values for the number of bits per pivot are the best options.

6. The algorithms based on a Voronoi-like partition of the space are the most resistant to the dimensionality.

7. Among the structures considerered, our experimental results show that the best one is our adaptation of BKTs to continuous spaces with arity 2. We have shown that this structure "degenerates" into a very efficient clustering scheme. If more memory is available, other structures finally improve over binary BKTs. Among these, those that use best the available memory are FHQAs and FMVPAs.

This work, however, has left a number of open issues requiring further attention. The main ones follow.

- Work more on clustering schemes in order to devise new algorithms, to find ways to reduce construction times (which is extremely high to be practical in many cases) and to allow them using more memory in order to reduce search times.

- Search for good hybrids between clustering and pivoting algorithms. The first ones cope better with high dimensions and the second ones improve as more memory is given to them. After the space is clustered the intrinsic dimension of the clusters is lower, so a top-level clustering structure joined with a pivoting scheme for the clusters is an interesting alternative. Those pivots should be selected from the cluster because the cluster is already compact in the space.

- Understand better how the structure of the space affects the efficiency of the search algorithms. Parameters like the histogram of distances may allow us to find worst or average case bounds to the complexity of the problem. Another issue is related to whether the space is limited in volume or not as n grows. We found along the survey that in many cases the behavior of the algorithms strongly depends on the fact that if there are more points they must be closer to each other, which is true in the $[0,1]^k$ space but false in the space of strings.

- Understand better the effect of pivot selection, devising methods to choose effective pivots. The subject of the appropriate number of pivots and its relation to the intrinsic dimensionality of the space plays a role here. The histogram of distances may be a good tool for pivot selection. Another related issue is the optimum arity, which is always fixed and could depend on the level in the tree or other parameters, as suggested in [16]. Finally, space overlapping techniques have not been attempted in metric spaces and they yielded good results in vector spaces (R-trees).

- Take extra CPU complexity into account, which we have barely considered in this work. In some applications the distance is not so expensive that one can disregard any other type of CPU cost. The use of specialized search structures in the mapped space (especially $\mathbb{R}^k$) and the resulting complexity tradeoff deserves more attention. Another area left aside is I/O costs.

- Focus on nearest neighbor search. Most current algorithms for this problem are based on range searching, and despite that the existing heuristics seem difficult to improve, truly independent ways to address the problem could exist.

- Consider approximate and probabilistic algorithms, which may give much better results at a cost that, especially for this problem, seems acceptable.

References

[1] P. Apers, H. Blanken, and M. Houtsma. *Multimedia Databases in Perspective*. Springer, 1997.

[2] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimension. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 573–583, 1994.

[3] F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3), 1991.

[4] R. Baeza-Yates. Searching: an algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker Inc., 1997.

[5] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[6] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proc. 5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*, pages 14–22. IEEE CS Press, 1998.

[7] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[8] J. Bentley. Multidimensional binary search trees used for associative searching. *Comm. of the ACM*, 18(9):509–517, 1975.

[9] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. on Software Engineering*, 5(4):333–340, 1979.

[10] J. Bentley, B. Weide, and A. Yao. Optimal expected-time algorithms for closest point problems. *ACM Trans. on Mathematical Software*, 6(4):563–580, December 1980.

[11] S. Berchtold, D. Keim, and H. Kriegel. The X-tree: an index structure for high-dimensional data. In *Proc. 22nd Conference on Very Large Databases (VLDB'96)*, pages 28–39, 1996.

[12] B. Bhanu, J. Peng, and S. Qing. Learning feature relevance and similarity metrics in image databases. In *Proc. IEEE Workshop on Content-Based Access of Image and Video Libraries*, pages 14–18, Santa Barbara, California, 1998. IEEE Computer Society.

[13] A. Del Bimbo and E. Vicario. Using weighted spatial relationships in retrieval by visual contents. In *Proc. IEEE Workshop on Content-Based Access of Image and Video Libraries*, pages 35–39, Santa Barbara, California, 1998. IEEE Computer Society.

[14] S. Blott and R. Weber. A simple vector-approximation file for similarity search in high-dimensional vector spaces. Technical report, Institute for Information Systems, ETH Zentrum, Zurich, Switzerland, 1997.

[15] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 357–368, 1997. Sigmod Record 26(2).

[16] S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[17] M. Brito, E. Chávez, A. Quiroz, and J. Yukich. Connectivity of the mutual k-nearest neighbor graph in clustering and outlier detection. *Statistics & Probability Letters*, 35:33–42, 1996.

[18] E. Bugnion, S. Fhei, T. Roos, P. Widmayer, and F. Widmer. A spatial index for approximate multiple string matching. In R. Baeza-Yates and N. Ziviani, editors, *Proc. 1st South American Workshop on String Processing (WSP'93)*, pages 43–53, 1993.

[19] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Comm. of the ACM*, 16(4):230–236, April 1973.

[20] M. La Cascia, S. Sethi, and S. Sclaroff. Combining textual and visual cues for content-based image retrieval on the world wide web. In *Proc. IEEE Workshop on Content-Based Access of Image and Video Libraries*, pages 24–28, Santa Barbara, California, 1998. IEEE Computer Society.

[21] E. Chávez. Optimal discretization for pivot based algorithms. Manuscript. `ftp://garota.fismat.umich.mx/pub/users/elchavez/minimax.ps.gz`, 1999.

[22] E. Chávez and J. Marroquín. Proximity queries in metric spaces. In R. Baeza-Yates, editor, *Proc. 4th South American Workshop on String Processing (WSP'97)*, pages 21–36. Carleton University Press, 1997.

[23] E. Chávez, J. Marroquín, and R. Baeza-Yates. Spaghettis: an array based algorithm for similarity queries in metric spaces. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, Cancun, Mexico, September 1999. To appear. `ftp://garota.fismat.umich.mx/pub/users/elchavez/spa.ps.gz`.

[24] E. Chávez, J. Marroquín, and G. Navarro. Overcoming the curse of dimensionality. In *European Workshop on Content-Based Multimedia Indexing (CBMI'99)*, 1999. To appear. `ftp://garota.fismat.umich.mx/pub/users/elchavez/fqa.ps.gz`.

[25] L. Childs. *A Concrete Introduction to Higher Algebra.* Springer-Verlag, 1995.

[26] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of the 23rd Conference on Very Large Databases (VLDB'97)*, pages 426–435, 1997.

[27] P. Ciaccia, M. Patella, and P. Zezula. A cost model for similarity queries in metric spaces. In *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98)*, 1998.

[28] K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.

[29] T. Cox and M. Cox. *Multidimensional Scaling.* Chapman and Hall, 1994.

[30] L. Devroye. *A Course in Density Estimation.* Birkhauser, 1987.

[31] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* Wiley, 1973.

[32] C. Faloutsos and K. Lin. Fastmap: a fast algorithm for indexing, data mining and visualization of traditional and multimedia datasets. *ACM SIGMOD Record*, 24(2):163–174, 1995.

[33] A. Faragó, T. Linder, and G. Lugosi. Fast nearest-neighbor search in dissimilarity spaces. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 15(9):957–962, September 1993.

[34] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[35] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[36] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[37] J. Hair, R. Anderson, R. Tatham, and W. Black. *Multivariate Data Analysis with Readings*. Prentice-Hall, 4th edition, 1995.

[38] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[39] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[40] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[41] L. Micó, J. Oncina, and R. Carrasco. A fast branch and bound nearest neighbour classifier in metric spaces. *Pattern Recognition Letters*, 17:731–739, 1996.

[42] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

[43] J. Munkres. *Topology, A First Course*. Prentice-Hall, 1975.

[44] G. Navarro. Searching in metric spaces by spatial approximation. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, Cancun, Mexico, September 1999. To appear. `ftp://-ftp.dcc.uchile.cl/pub/users/gnavarro/metric.ps.gz`.

[45] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, 1997.

[46] J. Nievergelt and H. Hinterberger. The grid file: an adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, March 1984.

[47] S. Prabhakar, D. Agrawal, and A. El Abbadi. Efficient disk allocation for fast similarity searching. In *Proc. ACM SPAA'98*, Puerto Vallarta, Mexico, 1998.

[48] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[49] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[50] D. Sasha and T. Wang. New techniques for best-match retrieval. *ACM Trans. on Information Systems*, 8(2):140–158, 1990.

[51] M. Shapiro. The choice of reference points in best-match file searching. *Comm. of the ACM*, 20(5):339–343, May 1977.

[52] R. Sutton and A. Barto. *Reinforcement Learning : an Introduction.* MIT Press, 1998.

[53] J. Uhlmann. Implementing metric trees to satisfy general proximity/similarity queries. Manuscript, 1991.

[54] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[55] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[56] M. Waterman. *Introduction to Computational Biology.* Chapman and Hall, 1995.

[57] D. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, La Jolla, California, July 1996.

[58] A. Yao. *Chapter 7*, pages 345–380. Elsevier Science, 1990. J. Van Leeuwen, editor.

[59] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*, pages 311–321, 1993.

[60] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge, ALENEX'99*, Baltimore, MD, 1999.