

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 element under some similarity criterion has a vast number of applications in many branches of computer science, from pattern recognition to textual and multimedia information retrieval. We are interested in the rather general case where the similarity criterion defines a *metric space*, instead of the more restricted case of a vector space. 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 some basic results that explain the intrinsic difficulty of the search problem. This includes a quantitative definition of the elusive concept of “intrinsic dimensionality”. We also 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. We present experiments validating our results and comparing the existing approaches. 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 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

*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. Partially supported by CONACyT under grant R-28923A

[‡]Depto. de Ciencias de la Computación, Universidad de Chile. Blanco Encalada 2120, Santiago, Chile. [{gnavarro,rbaeza}@dcc.uchile.cl](mailto:gnavarro,rbaeza@dcc.uchile.cl). Partially supported by Fondecyt Grant 1-000929.

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

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 element¹. 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, computational geometry, 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 comparisons have been done. More importantly, there have been no attempts to conceptually unify all those solutions.

In some applications the metric space turns out to be of a particular type called “vector space”, where the elements consist of k real-valued coordinates. A lot of work has been done on vector spaces by exploiting their geometric properties, but normally these cannot be extended to general metric spaces where the only available information is the distance among objects. In this general case, moreover, the distance is normally quite expensive to compute, so the general goal is to reduce the number of distance evaluations. In contrast, the operations in vector spaces tend to be simple and hence the goal is mainly to reduce I/O. Some important advances have been done for general metric spaces, in general around the concept of building an *index*, i.e. a data structure to reduce the number of distance evaluations at query time. Some recent work [26, 52] tries to achieve at the same time the goals of reducing the number of distance evaluations and the amount of I/O performed.

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. Two main techniques based on equivalence relations, namely pivoting and Voronoi partitions, are shown to encompass all the existing methods. 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 efficiency. 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 quantitative methods to assert the intrinsic difficulty in searching on a given metric space and provide lower bounds on the search problem. This includes a quantitative definition of the up to now conceptual notion of “intrinsic dimensionality”, which we show to be very appropriate. We present some experimental results that help to validate our assertions.

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

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

time, time and space cost to build the indexes, nearest neighbor queries, dynamic capabilities of the indexes and I/O considerations. There are some features which we definitely do not cover in order to keep our scope reasonably bounded, such as (1) complex similarity queries involving more than one similarity predicate [28], as few works on them exist and they are an elaboration over the simple similarity queries (a particular case is polygon searching in vector spaces); (2) 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; and (3) inverse queries (i.e. find the elements for which q is their closest neighbor) and total queries (e.g. find all the closest neighbors) since the algorithms are, again, built over the simple ones.

This paper is organized as follows. A first part (Sections 2–5) is a pure survey of the state of the art in searching metric spaces, with no attempt to provide a new way to think on the problem. A second part (Sections 6–8) presents our basic results on the difficulty of the search problem and our unifying model that allows understanding the essential features of the problem and its existing solutions. Finally, Section 9 gives our conclusions and points out future research directions.

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 errors or not.

Possible types of searches are *point* or *key* search (all the key information is given), *range* search (only some fields are given or only a range of values is specified for them) and *proximity* search (in addition, records “close” to the query are considered interesting). These types of search are 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* [49]. 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, 70].

Think for example in a repository of images. Interesting queries are of the type “*find an image of a lion 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 searching 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 [54, 38, 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” [56]) 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 [64, 56].

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 [33].

Other applications of this universal function approximator are density estimation [32] and reinforcement learning [59]. 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 or in wireless communication. 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 *valid* objects. A finite subset of it, \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:

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

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

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

and in most cases

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

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 *metric*, i.e. if it satisfies

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

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

If the distance 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.

In the rest of paper we use the term *distance* in the understanding that we refer to a metric.

3.2 Proximity Queries

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

Range query $(q, r)_d$. Retrieve all elements which are within distance r to q . This is, $\{u \in \mathbb{U} / d(q, u) \leq r\}$.

Nearest neighbor query $NN(q)$. 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 anyway). 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.

k -Nearest neighbor query $NN_k(q)$. 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 the range query. The left part of Figure 1 illustrates a query on a set of points which will be our running example, using \mathbb{R}^2 as the metric space for clarity.

A *range query* will be therefore a pair $(q, r)_d$ with q an 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 range query.

We use “NN” as an abbreviation of “nearest neighbor”, and give the generic name “NN-query” to the last two types of queries and “NN searching” to the techniques to solve them. As we see later, NN-queries can be systematically built over range queries.

The total time to evaluate a query can be split as

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

and we would like to minimize T . In many applications, however, evaluating $d()$ is so costly that the other components of the cost 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 in n (but reasonable) amount of CPU work and a linear traversal over the database on disk, 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*. The I/O time can be the dominant factor in some applications and negligible in others, depending on the amount of main memory available and the relative cost to compute the distance function. We cover the little existing work on relating metric spaces and I/O considerations in Section 5.3.2.

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 index 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 (called *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).

4 The Case of 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, \dots, 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

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

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 L_2 distance is better known as “Euclidean” distance, as it corresponds to our notion of spatial distance. 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:

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

Searching with the L_∞ distance corresponds directly to a classical range search query, where the range is the k -dimensional hyper-rectangle. This distance plays a special role in this survey.

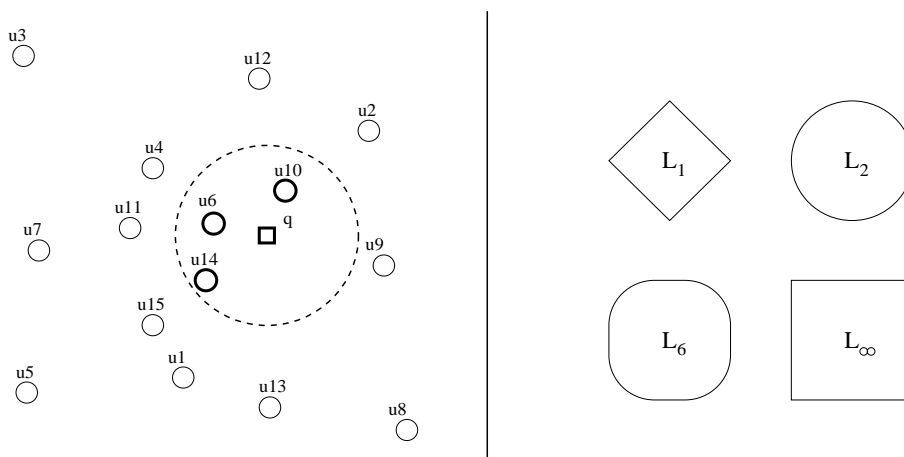


Figure 1: On the left, an example of a range query on a set of points. On the right, the set of points at the same distance to a center point, for different Minkowski distances.

In many applications the metric space is indeed a vector space, i.e. the objects are k -dimensional points and the similarity is interpreted geometrically. A vector space permits more freedom than

a general metric space when designing search approaches, since it is possible to use geometric and coordinate information which is unavailable in a general metric space.

In this framework optimal algorithms (on the database size) exist in both the average and the worst case [10] for closest point search. Search structures for vector spaces are called *spatial access methods* (SAM). Among the most popular are *kd-trees* [8, 9], *R-trees* [40], quad-trees [55] and the more recent *X-trees* [11]. These techniques make extensive use of coordinate information to group and classify points in the space. For example *kd-trees* divide the space along different coordinates and *R-trees* groups points in hyper-rectangles. Unfortunately the existing techniques 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*).

Vector spaces may suffer from large differences between their *representational* dimension (k) and their *intrinsic* dimension (i.e. the real number of dimensions in which the points can be embedded while keeping the distances among them). For example a plane embedded in a 50-dimensional space has intrinsic dimension 2 and representational dimension 50. This is in general the case of real applications, where the data is clustered, and it has led to attempts to measure the intrinsic dimension such as the concept of “fractal dimension” [35]. Despite that no techniques can cope with intrinsic dimension higher than 20, much higher representational dimensions can be handled by dimensionality reduction techniques [36, 30, 41].

Since efficient techniques to cope with vector spaces exist, application designers try to give their problems a vector space structure. However, this is not always easy or feasible at all. For example, experts in image processing try to express the similarity between images as the distance between “vectors” of features extracted from the images, although in many cases better results are obtained by coding specific functions that compare two images, despite that they cannot be easily expressed as the distance between two vectors (e.g. cross-talk between features [34]). Another example that resists conversion into a vector space is similarity functions between strings, to compare DNA sequences for instance.

For this reason several authors resort to general metric spaces, even knowing that the search problem is much more difficult. Of course it is also possible to treat a vector space as a general metric space, by using only the distances between points. One immediate advantage is that the intrinsic dimension of the space shows up, independent of any representational dimension (this requires extra care in vector spaces). It is interesting to remark that in [26] they present preliminary results showing that a metric space data structure (the M-tree) can outperform a well known vector space data structure (the R*-tree) when applied to a vector space.

Specific techniques for vector spaces is a whole different world which we do not intend to cover in this work (see [55, 65, 39] for good surveys). However, we discuss in the next section a technique which, instead of treating a vector space as a metric space, tries to embed a general metric space into a vector space. This concept is central in this survey, despite that specific knowledge on specific techniques for vector spaces is, as we shortly show, not necessary to understand it.

4.1 Resorting to Vector Spaces

An interesting and natural reduction of the similarity 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 process range queries in the projected space with the same radius. Since some spurious elements can be

captured in the target space, the outcome of the query in the projected space is a *candidate list*, which is later verified element-wise with the original distance to obtain the actual outcome of the query.

Intuitively, the idea is to “invent” k coordinates and map the points onto a vector space, using some vector space technique as a first filter to the actual answer to a query. One of the main thesis of this work is a large subclass of the existing algorithms can be regarded as relying on some mapping of this kind. A widely used method (explained in detail in Section 6.6) is to select $\{p_1 \dots p_k\} \subseteq \mathbb{U}$ and map each $u \in \mathbb{U}$ to (\mathbb{R}^k, L_∞) using $\Phi(u) = (d(u, p_1), \dots, d(u, p_k))$. It can be seen that this mapping is contractive but not proximity preserving.

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 NN-queries can be directly performed in the projected space. Indeed, most current algorithms for NN-queries are based in range queries, and with some care they can be done in the projected space if the mapping is contractive, even if it is not proximity preserving.

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, but it makes no difference in the number of evaluations of the d distance.

How well do metric space techniques perform in comparison to vector space methods? It is difficult to give a formal answer because of the different cost models involved. In metric spaces we use the number of distance evaluations as the basic measure of complexity, while vector space techniques may very well use many coordinate manipulations and not a single evaluation of the distance. Under our model, the cost of a method that maps to a vector space to trim the candidate list is measured as the number of distance evaluations to realize the mapping plus the final distances to filter the trimmed candidate list, while the work on the artificial coordinates is seen as just extra CPU time.

A central question related to this reduction is: how well can a metric space be embedded into a vector space? How many coordinates have to be considered so that the original metric space and the target vector spaces are similar enough so that the candidate list given by the vector space is not much larger than the actual outcome of the query in the original space? This is a very difficult question that lies behind all this paper, and we return to it in Section 7.

The issue is better developed in vector spaces. There are different techniques to reduce the dimensionality of a set of points while preserving the original distances as much as possible [30, 41, 36], that is, to find the intrinsic dimension of the data.

5 Current Solutions for Metric Spaces

In this section we explain the existing indexes to structure metric spaces and how are they used for range and NN searching. 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 add a final subsection devoted to more advanced issues such as dynamic capabilities, I/O considerations and approximate and probabilistic algorithms.

5.1 Range Searching

We divide the presentation in three parts. The first one deals with tree indexes for *discrete* distance functions, that is, functions that deliver a small set of values. The second part corresponds to tree indexes for *continuous* distance functions, where the set of alternatives is infinite or very large. Third, we consider other methods that are not tree-based.

Table 1 summarizes the complexities of the different structures. These are obtained from the source papers, which use different (and incompatible) assumptions and in many cases give just gross analyses or no analysis at all (just heuristic considerations). Therefore, we give the complexities *as claimed by the authors of each paper, not as a proven fact*. At best, the results are analytical but rely on diverse simplifying assumptions. At worst, the results are based on a few incomplete experiments. Keep also in mind that there are hidden factors depending (in many cases exponentially) on the dimension of the space, and that the query complexity is always on average, as in the worst case we can be forced to compare all the elements. Even in the simple case of orthogonal range searching on vector spaces there exist $\Omega(n^\alpha)$ lower bounds for the worst case [44].

5.1.1 Trees for Discrete Distance Functions

We start by describing tree 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 with these trees.

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*.

When 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 u which are at distance $d(u, p) = i$ from some node p , where $|d(p, q) - i| > r$. By the triangular inequality, $d(p, q) \leq d(p, u) + d(u, q)$, and therefore $d(u, q) \geq d(p, q) - d(p, u) > r$.

Figure 2 shows an example, where the element u_{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.

Data Structure	Space Complexity	Construction Complexity	Claimed Query Complexity	Extra CPU query time
BKT [19, 58]	n pointers	$O(n \log n)$	$O(n^\alpha)$	—
FQT [5]	$n..n \log n$ pointers	$O(n \log n)$	$O(n^\alpha)$	—
FHQT [5, 4, 6]	$n..nh$ pointers	$O(nh)$	$O(\log n)$ (*)	$O(n^\alpha)$
FQA [24]	nhb bits	$O(nh)$	$O(\log n)$ (*)	$O(n^\alpha \log n)$
VPT [61, 67, 25]	n pointers	$O(n \log n)$	$O(\log n)$ (**)	—
MVPT [16, 15]	n pointers	$O(n \log n)$	$O(\log n)$ (**)	—
VPF [68]	n pointers	$O(n^{2-\alpha})$	$O(n^{1-\alpha} \log n)$ (**)	—
BST [43, 51]	n pointers	$O(n \log n)$	not analyzed	—
GHT [61, 18]	n pointers	$O(n \log n)$	not analyzed	—
GNAT [16]	nm^2 distances	$O(nm \log_m n)$	not analyzed	—
VT [31, 50, 62]	n pointers	$O(n \log n)$	not analyzed	—
MT [26]	n pointers	$O(n(m..m^2) \log_m n)$	not analyzed	—
SAT [47]	n pointers	$O(n \log n / \log \log n)$	$O(n^{1-\Theta(1/\log \log n)})$	—
AESA [63]	n^2 distances	$O(n^2)$	$O(1)$ (***)	$O(n)..O(n^2)$
LAESA [46, 45, 48, 23]	kn distances	$O(kn)$	$k + O(1)$ (***)	$O(\log n)..O(kn)$

(*) If $h = \log n$.

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

(***) Empirical conclusions without analysis, in case of LAESA for “large enough” k .

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.

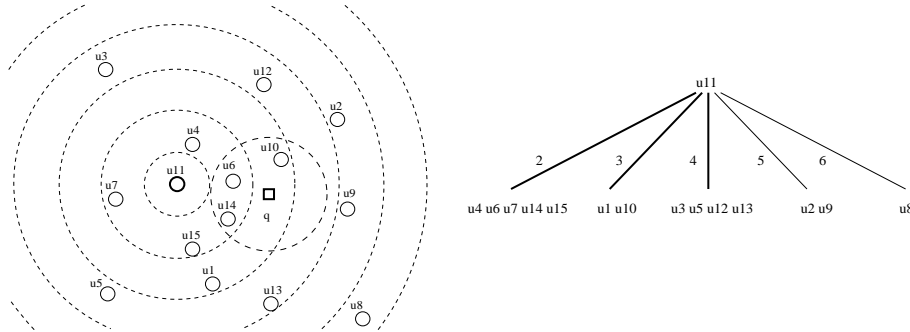


Figure 2: On the left, the division of the space obtained when u_{11} is taken as a pivot. On the right, the first level of a BKT with u_{11} as root. We also show a query q and the branches that it has to traverse. We have discretized the distances so they return integer values.

The results of Table 1 for BKTs are extrapolated from those made for Fixed Queries Trees [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 $O(\log n)$.

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 [4, 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 [38]. This idea of using fewer 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, trying that about the same number of elements lies in each cell of the next level. This allows using 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.

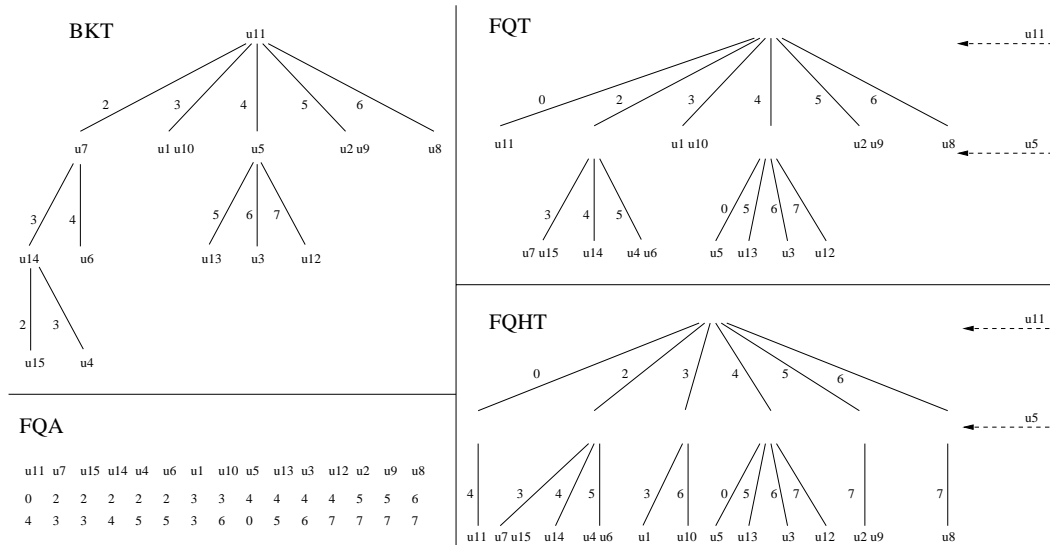


Figure 3: Example BKT, FQT, FHQT and FQA for our set of points. We use $b = 2$ for the BKT and FQT, and $h = 2$ for FHQT and FQA.

Hybrid In [58], the use of more than one element per node of the tree is proposed. Those k elements allow eliminating 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 child 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 lose their sublinear extra CPU time.

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 class.

5.1.2 Trees for Continuous Distance Functions

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

VPT The first tree designed for continuous distance functions is called “Metric Trees” in [61]. A more complete work on the same idea [67, 25] calls them “Vantage-Point Trees” or VPTs. They

build a binary tree recursively, taking any element p as the root and taking the *median* of the set of all distances, $M = \text{median}\{d(p, u) \mid u \in \mathbb{U}\}$. 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 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.

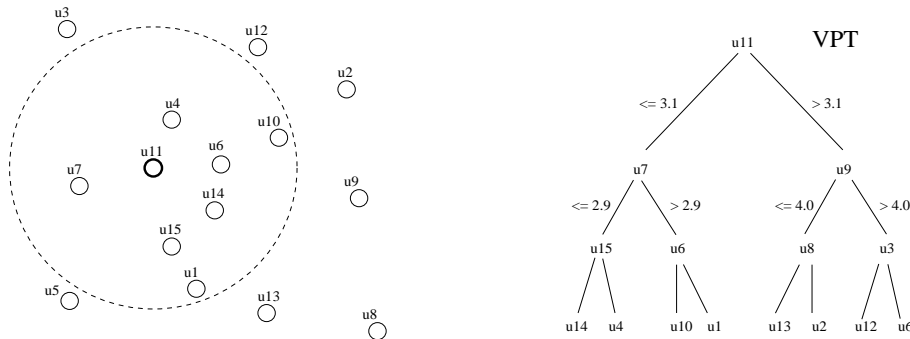


Figure 4: Example VPT with root u_{11} . We plot the radius M used for the root. For the first levels we show explicitly the radii used in the tree.

The query complexity is argued to be $O(\log n)$ in [67], 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 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 [67], to prune more elements before checking them. Finally, the author of [67] considers the problem of pivot selection and argues that it is better to take elements far away from the set.

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 [58]. 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 of [15] 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) [68]. This algorithm is designed for radii limited NN search (an $NN(q)$ query with a maximum radius r^*), but in fact the technique is perfectly compatible with a range search query. 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 quite small.

BST In [43], the “Bisector Trees” (BSTs) are proposed. The BST is a binary tree built recursively as follows. At each node, two “centers” c_1 and c_2 are selected. The elements closer to c_1 than to c_2 go into the left subtree and those closer to c_2 into the right subtree. For each of the two centers, its “covering radius” is stored, i.e. the maximum distance from the element to any other element in its subtree. At search time, we enter into each subtree if $d(q, c_i) - r$ is not larger than the covering radius of c_i . That is, we can discard a branch if the query ball (i.e. the hypersphere of radius r centered in the query) does not intersect the ball that contains all the elements inside that branch. In [51], the “Monotonous BST” is proposed, where one of the two elements at each node is indeed the parent center. This makes the covering radii to decrease as we move downward in the tree. Figure 5 illustrates the first step of the tree construction.

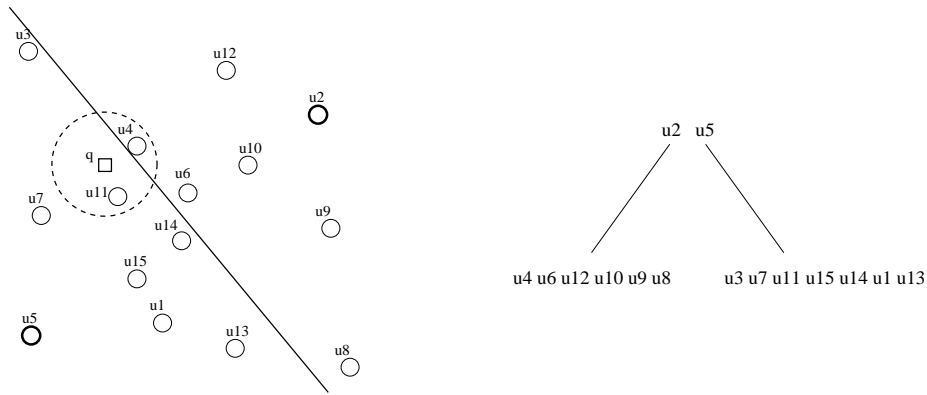


Figure 5: Example of the first level of a BST or GHT and a query q . Either using covering radii (BST) or hyperplanes (GHT), both subtrees have to be considered in this example.

GHT Proposed in [61], the “Generalized-Hyperplane Tree” (GHT) is identical in construction to a BST. However, the algorithm uses the hyperplane between c_1 and c_2 as the pruning criterion at search time, instead of the covering radius. At search time we enter into the left subtree if $d(q, c_1) - r < d(q, c_2) + r$ and into the right subtree if $d(q, c_2) - r \leq d(q, c_1) + r$. Again, it is possible to enter into both subtrees. In [61] it is argued that GHTs could work better than VPTs in high dimensions. The same idea of reusing the parent node is proposed in [18], this time to avoid performing two distance evaluations at each node.

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 centers $c_1 \dots c_m$, and define $\mathbb{U}_i = \{u \in \mathbb{U}, d(c_i, u) < d(c_j, u), \forall j \neq i\}$. That is, \mathbb{U}_i are the elements closer to c_i than to any other c_j . From the root, m children numbered $i = 1..m$ are built, each one recursively as a GNAT for \mathbb{U}_i . 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].

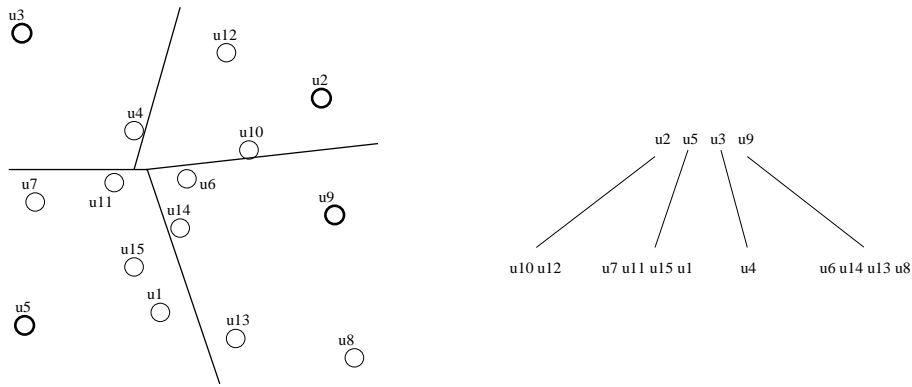


Figure 6: Example of the first level of a GNAT with $m = 4$.

The search algorithm, however, is quite different. At indexing time, the GNAT stores at each node an $O(m^2)$ size table $range_{ij} = [\min_{u \in \mathbb{U}_j}(c_i, u), \max_{u \in \mathbb{U}_j}(c_i, u)]$, which stores minimum and maximum distances from each center to each class. At search time the query q is compared against some center c_i and then it discards any other center c_j such that $d(q, c_i) \pm r$ does not intersect $range_{i,j}$. All the subtree \mathbb{U}_j can be discarded using the triangle inequality. The process is repeated with random centers until no one can be discarded. The search then enters recursively in each non discarded subtree. In the process, any center close enough to q is reported.

The authors use a gross analysis to show that the tree takes $O(nm^2)$ space and is built in close to $O(nm \log_m n)$ 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.

VT The “Voronoi Tree” (VT) is proposed in [31] as an improvement over BSTs, where this time the tree has 2 or 3 elements (and children) per node. When a new tree node has to be created to hold an inserted element, its closest element from the parent node is also inserted in the new node. VTs have the property that the covering radius is reduced as we move downwards in the tree, which provides better packing of elements in subtrees. It is shown in [31] that VTs are superior and better balanced than BSTs. In [50] they show that balanced VTs can be obtained by insertion procedures similar to those of B-trees, a fact later exploited in M-trees (see next).

MT The M-tree (MT) data structure is presented in [26], aiming at providing dynamic capabilities and good I/O performance in addition to few distance computations. The structure has some resemblances with a GNAT, since it is a tree where a set of representatives are chosen at each node and the elements closer to each representative² are organized into a subtree rooted by that representative. The search algorithm, however, is closer to BSTs. Each representative stores its covering radius. At query time, the query is compared against all the representatives of the node and the search algorithm enters recursively into all those that cannot be discarded using the covering radius criterion.

The main difference of the MT is the way in which insertions are handled. An element is inserted into the “best” subtree, defined as that causing the subtree covering radius to expand less (zero expansion is the ideal), and in case of ties selecting the closest representative. Finally,

²There are many variants but this is reported as the most effective.

the element is added to the leaf node and if the node overflows (i.e. becomes of size $m + 1$) it is split in two and one node element is promoted upwards, as in a B-tree or an R-tree [40]. Hence the MT is a balanced data structure, much as the VP family. There are many criteria to select the representative and to split the node, the best results being obtained by trying a split that minimizes the maximum of the two covering radii obtained. They show experimentally that the MT is resistant to the dimensionality of the space and that it is competitive against R*-trees.

SAT The algorithm SAT (“Spatial Approximation Tree”) [47] does not use centers 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 a set of “neighbors” N , defined as a subset of elements $u \in \mathbb{U}$ such that u is closer to p than to any other element in N (note that the definition is self-referential). The other elements (not in $N \cup \{p\}$) are assigned to their closest element in N . Each element in N is recursively the root of a new subtree containing the elements assigned to it.

This allows searching 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 $r > 0$ is allowed, then we consider that an unknown element $q' \in \mathbb{U}$ is searched with tolerance zero, from which we only know that $d(q, q') \leq r$. Hence, we search as before for q and consider that any distance measure may have an “error” of at most $\pm r$. 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. That is, if $c \in N$ is the closest neighbor of q , we enter into all $c' \in N$ such that $d(q, c') - r \leq d(q, c) + r$. The tree is built in $O(n \log n / \log \log n)$ time, takes $O(n)$ space and inspects $O(n^{1-\Theta(1/\log \log n)})$ elements at query time. Covering radii are also used to increase pruning. Figure 7 shows an example and the search path for a query.

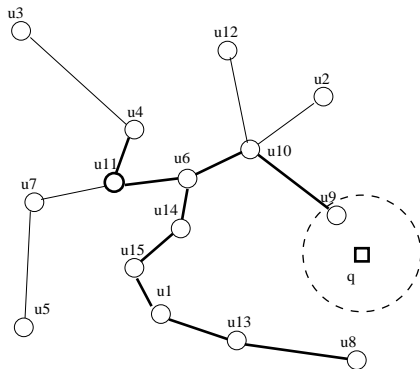


Figure 7: Example of a SAT and the traversal towards a query q , starting at u_{11} .

5.1.3 Other Techniques

AESA An algorithm which is close to many of the presented ideas but performs surprisingly better by an order of magnitude is [63] (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 8 shows an example with a first pivot u_{11} .

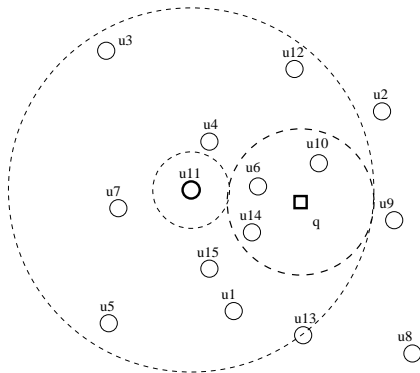


Figure 8: Example of the first iteration of AESA. The points between both rings centered at u_{11} qualify for the next iteration.

Although this idea seems very similar to FQTs, there are some 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 [63] 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 [57], although in this latter case they may take fewer 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) [46], 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 reduce the extra CPU time, while in this case no such an 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 way to reduce the extra CPU time is presented later in [45], 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 losing information on distances are presented in [48, 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 searched³. Extra pointers are added to be able to trace an element across the different orderings for each pivot (this needs more space, however).

Clustering approaches Clustering is a very wide area with lots of applications [42]. 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 c_i for each. They compute covering radii r_i . To search for the closest neighbor, the

³Although in [48] they consider only vector spaces, the same technique can be used here.

query q is compared against all the c_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 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 and MTs.

5.2 Nearest Neighbor Queries

We have concentrated in range search queries up to now. This is because, as we show in this section, most of the existing solutions for NN-queries are built systematically over range searching techniques, and indeed can be adapted to any of the data structures presented (despite having been originally designed for specific ones).

5.2.1 Increasing Radius

The simplest NN search algorithm is based on using a range searching algorithm as follows. Search q with fixed radii $r = a^i \varepsilon$ ($a > 1$), starting with $i = 0$ and increasing it until at least the desired number of elements (1 or k) lies inside the search radius $r = a^i \varepsilon$. Later, the radius is refined between $r = a^{i-1} \varepsilon$ and $r = a^i \varepsilon$ until the exact number of elements is included.

Since the complexity of the range searching normally grows sharply on the search radius, the cost of this method can be very close to the cost of a range searching with the appropriate r (which is not known in advance). The increasing steps can be made smaller ($a \rightarrow 1$) to avoid searching with a radius much larger than necessary.

5.2.2 Backtracking with Decreasing Radius

A more elaborated technique is as follows. We first explain the search for the closest neighbor. Start the search on any data structure using $r^* = \infty$. Each time q is compared against some element p , update the search radius as $r^* \leftarrow \min(r^*, d(q, p))$ and continue the search with this reduced radius. This has been for example proposed for BKTs and FQTs [19, 5].

As closer and closer elements to q are found, we search with smaller radius and the search becomes cheaper. For this reason it is important to try to find quickly elements that are close to the query (which is unimportant in range queries). The way to achieve this is dependent on the particular data structure. For example, in BKTs and FQTs we can 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 (other heuristics may be better). Therefore, the 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 of them.

$NN_k(q)$ queries are solved as an extension of the above technique, where we keep the k elements seen that are closest to q and set r^* as the maximum distance between those elements and q (clearly we are not interested in elements farther away than the current k -th closest element). Each time a new element is seen whose distance is relevant, it is inserted as one of the k nearest neighbors known up to now (possibly displacing one of the old candidates out of the list) and r^* is updated. In the beginning we start with $r^* = \infty$ and keep this value until the first k elements are found.

A variant of this type of queries is the *limited radius* NN searching. Here we start with the maximum expected distance between the query element and its nearest neighbor. This type of queries has been the focus of [68, 69].

5.2.3 Priority Backtracking

The previous technique can be improved by a smarter selection of which elements to consider first. For clarity we consider backtracking in a tree, although the idea is general. Instead of following the normal backtracking order of the range query, modifying at most the order in which the subtrees are traversed, we give much more freedom to the traversal order. The goal is to increase the probability of quickly finding elements close to q and therefore reduce r^* fast. This technique has been used in vector and metric spaces several times [60, 53, 26].

At each point of the search we have a set of possible subtrees that can be traversed (not necessarily all at the same level). We select among them using some heuristic (e.g. first traverse subtrees whose root is closest to q). Once a subtree has been selected we compare q against its root, update r^* and the candidates for output if necessary, and determine which of the children of the considered root deserve traversal. Unlike the normal backtracking, those children are not immediately traversed but added to the set of subtrees that have to be traversed at some moment. Then we select again a subtree from the set using the optimization heuristic.

The best way to implement this search is with a priority queue ordered by the heuristic “goodness”, where the subtrees are inserted and removed. We start with an empty queue where we insert the root of the tree. Then, we repeat the step of removing the most promising subtree, processing it, and inserting the relevant subtrees until the queue gets empty.

If applied to a BKT or a FQT, this method yields the same result as the previous section, but this technique is superior to deal with continuous distances.

5.2.4 Specific NN Algorithms

The techniques described above cover almost all the existing proposals for solving NN-queries. The only exception we are aware of was presented in [29], which is a GNAT-like data structure where the points are inserted into more than one subtree to limit backtracking (hence the space requirement is superlinear).

After selecting the representatives for the root of the tree, each element u is not only inserted into the subtree of its closest representative p , but also in the tree of any other representative p' such that $d(u, p') \leq 3d(u, p)$. At search time, the query q enters not only into its nearest representative p but also into every other representative p' such that $d(q, p') \leq 3d(q, p)$. As shown in [29] this is enough to guarantee that the nearest neighbor will be reached.

By using subsets of size $n^{1/2^{k+1}}$ at depth k in the tree, the search time is polylogarithmic in n and the space requirement is $O(n \text{ polylog } n)$ if some conditions hold in the metric space.

5.3 Extensions

We cover in this section the work that has been pursued on extensions of the basic problems or in alternative models. None of these are the main focus of our survey.

5.3.1 Dynamic Capabilities

Many of the data structures for metric spaces are designed to be built on a static data set. In many applications this is not reasonable because elements have to be inserted and deleted dynamically. Some data structures tolerate insertions well, but not deletions.

We first consider insertions. Among the structures that we have surveyed, the least dynamic is SAT, which needs full knowledge of the complete set at index construction time and has difficulty in handling later insertions (some workarounds are described in [47]). The VP family (VPT, MVPT,

VPF) has the problem of relying on global statistics (such as the median) to build the tree, so later insertions can be performed but the performance of the structure may deteriorate. Finally, the FQA needs in principle insertion in the middle of an array, but this can be handled by using standard techniques. All the other data structures can handle insertions in a reasonable way. There are some structure parameters that may depend on n and thus require periodical structural reorganization, but we disregard this issue here (e.g. adding or removing pivots is generally problematic).

Deletion is a little more complicated. In addition to the above structures, which present the same problems as for insertion, BKTs, GHTs, BSTs, VTs, GNATs and the VP family cannot tolerate deletion of an internal tree node because it plays an essential role in organizing the subtree. Of course this can be handled as just marking the node as removed and actually keeping it for routing purposes, but the quality of the data structure is affected over time.

Therefore, the only structures that fully support insertions and deletions are the FQ family (FQT, FQHT, FQA, since there are no truly internal nodes), AESA and LAESA approaches (since they are just vectors of coordinates), the MT (which is designed with dynamic capabilities in mind and whose insertion/deletion algorithms remind those of the B-tree), and a variant of GHTs designed to support dynamic operations [62]. The analysis of this latter structure shows that dynamic insertions can be done in $O(\log^2 n)$ amortized worst case time, and that deletions can be done at similar cost under some restrictions.

5.3.2 I/O Considerations

Most of the research on metric spaces deals with reducing the number of distance evaluations or at most the total CPU time. However, depending on the application, the I/O cost may play an important role. As most of the research on metric spaces has focused on algorithms to discard elements, I/O considerations have been normally left aside.

The only exception to the rule is MT, designed specifically for secondary memory. The tree nodes in the MT are to be stored in a single disk page (indeed, the MT does not fix an arity but rather a node capacity in bytes). Earlier balanced trees exist (such as the VP family), but the purpose of this balancing is to keep low the extra CPU costs. As we show later, unbalanced data structures perform much better in high dimensions and it is unlikely that the reduced CPU costs may play an important role. The purpose of balancing the MT, on the other hand, is to keep I/O costs low, and depending on the application this may be even more important than the number of distance evaluations.

Other data structures could probably be adapted to perform well in secondary memory, but the authors simply have not considered the problem. For instance, it is not hard to imagine strategies for the tree data structures to pack “compact” subtrees in disk pages, so as to make as good use as possible of a page that is read from disk. When a subtree grows larger than a page it is split in two pages of approximately the same number of nodes. Of course, a B-tree like scheme as that of MT has to be superior in this respect. Finally, array oriented approaches such as FQA, AESA and LAESA are likely to read all the disk pages of the index for each query, hence having bad I/O performance.

5.3.3 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. We do not cover them in depth here but present a few examples. Approximation algorithms for similarity searching are considered in depth in [65].

As a first example, we mention an approximate algorithm for NN 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 such that $d(u, q) \leq d(v, q) \quad \forall v \in \mathbb{U}$, they find an element u^* , an $(1 + \varepsilon)$ -nearest neighbor, differing from u by a factor of $(1 + \varepsilon)$, i.e. u^* such that $d(u^*, q) \leq (1 + \varepsilon)d(v, q) \quad \forall v \in \mathbb{U}$.

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 $O(\lceil 1 + 6k/\varepsilon \rceil^k k \log n)$.

A second example is a probabilistic algorithm for vector spaces [69]. The data structure is like a standard kd -tree, using “aggressive pruning” to improve the performance. The idea is to increase the number of branches pruned at the expense of losing some candidate points in the process. This is done in a controlled way, so the probability of success is always known. In spite of the vector space focus of the algorithm, it could be generalized to metric spaces as well. The data structure is useful for finding only limited radius nearest neighbors, i.e. neighbors within a fixed distance to the query. Finally, an example of a probabilistic NN algorithm for general metric spaces is that of [29]. The original intention is to build a Voronoi-like data structure on a metric space. As this is not possible in general because there is not enough knowledge of the characteristics of the queries that will come later [47], the author of [29] chooses to have a “training set” of queries and to build a data structure able to answer correctly only queries belonging to the training set. The idea is that this is enough to answer correctly an arbitrary query with high probability. Under some probabilistic assumptions on the distribution of the queries, it is shown that the probability of not finding the nearest neighbor is $O(\log n)^2/K$, where K can be made arbitrarily large at the expense of $O(Kn\rho)$ space and $O(K\rho \log n)$ expected search time. Here ρ is the logarithm of the ratio between the farthest and the nearest pairs of points in the union of \mathbb{U} and the training set.

6 A Unifying Model

At first sight, the indexes and the search algorithms 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 9. All the indexing algorithms

partition the set \mathbb{U} into subsets. An index is built which allows determining a set of candidate subsets where the elements relevant to the query can appear. At query time, the 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).

The last two subsections describe the two main approaches to similarity searching in abstract terms.

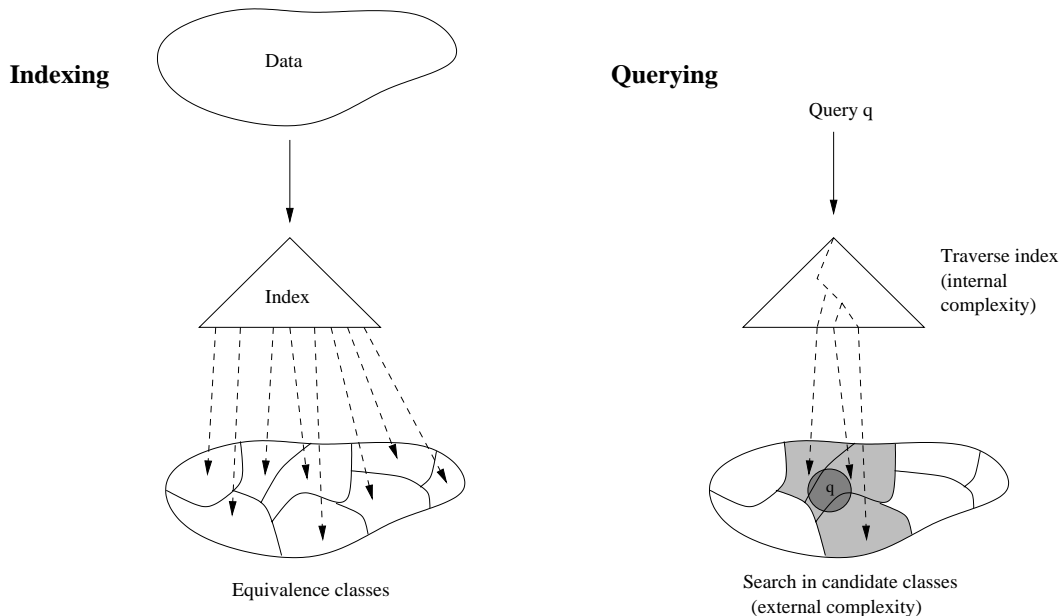


Figure 9: The unified model for indexing and querying metric spaces.

6.1 Equivalence Relations

The relevance of equivalence classes for this paper comes from the possibility of partitioning a metric space so that a new metric space is derived from the quotient set. Readers familiar with equivalence relations can safely skip this short section.

Given a set \mathbb{X} , a *partition* $\pi(\mathbb{X}) = \{\pi_1, \pi_2, \dots\}$ is a collection of pairwise disjoint subsets whose union is \mathbb{X} , 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 ($x \sim y \wedge y \sim z \Rightarrow x \sim z$).

Every partition $\pi(\mathbb{X})$ induces an equivalence relation \sim and, conversely, every equivalence relation induces a partition: two elements are related if they belong to the same partition element. 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 just classes), obtained when applying the equivalence relation to the set \mathbb{X} .

6.2 Indexing and Partitions

The equivalence classes in the quotient set $\pi(\mathbb{X})$ of a metric space \mathbb{X} can be considered themselves as objects in a new metric space, provided we define a distance function in $\pi(\mathbb{X})$.

We introduce a new function $D_0 : \pi(\mathbb{X}) \times \pi(\mathbb{X}) \rightarrow \mathbb{R}$ now defined in the quotient.

Definition 1. *Given a metric space (\mathbb{X}, d) and a partition $\pi(\mathbb{X})$, the extension of d to $\pi(\mathbb{X})$ is defined as $D_0([x], [y]) = \inf_{x \in [x], y \in [y]} \{d(x, y)\}$.*

D_0 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 metric D that lower bounds D_0 (i.e. $D([x], [y]) \leq D_0([x], [y])$). Since D is a metric, $(\pi(\mathbb{X}), D)$ is a metric space and therefore we can make queries in $\pi(\mathbb{X})$ in the same way we have done in \mathbb{X} . We redefine the outcome of a query in $\pi(\mathbb{X})$ 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 d), 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 (recall Figure 9). In other words:

Proposition. *All the existing indexing algorithms for proximity searching consist in building an equivalence relation, so that at search time some classes are discarded and the others are exhaustively searched.*

As we see shortly, the most important tradeoff when designing the partition is to balance the cost to find $([q], r)_D$ and the cost to verify this candidate list.

In Figure 10 we can see a schematic example of the idea. We divide the space in several regions (equivalence classes). The objects inside each region become 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).

We recall that this property is not enough for an arbitrary NN search algorithm to work (since the mapping would have to preserve proximity instead), but most existing algorithms for NN are based on range queries (recall Section 5.2), 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. Say that 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 ball centered at q with radius r (recall Figures 10 and 8). This is the familiar rule used in many independent algorithms to trim the space.

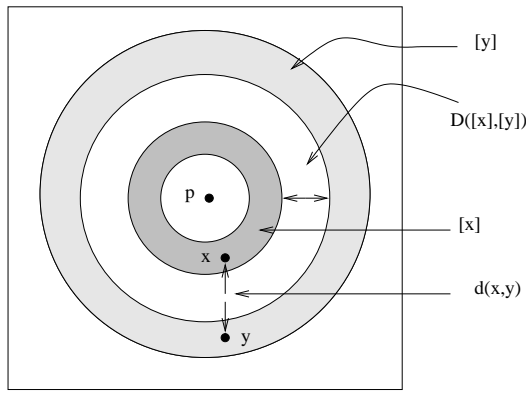


Figure 10: Two points x and y , and their equivalence classes (the shaded rings). D gives the minimal distance among rings, which lower bounds the distance between x and y .

Example 2. 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 partition 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) they inspect this set of candidate cells exhaustively to find the actual points inside the query range⁴. In this case the equivalence classes are the cells, and the tradeoff is that the larger the cells, the cheaper it is to find the appropriate ones, but the more costly is the final exhaustive search.

6.3 Coarsening and Refining a Partition

We start by defining the concept of refinement and coarsening.

Definition 2. Let \sim_1 and \sim_2 be two equivalence relations over a set \mathbb{X} . We say that \sim_1 is a refinement of \sim_2 or that \sim_2 is a coarsening of \sim_1 if for any pair $x, y \in \mathbb{X}$ such that $x \sim_1 y$ it holds $x \sim_2 y$. The same terms can be applied to the corresponding partitions $\pi^1(\mathbb{X})$ and $\pi^2(\mathbb{X})$.

Refinement and coarsening are important concepts for the topic we are discussing. The following lemma shows the effect of coarsening on the effectiveness of the partition for searching purposes.

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

Proof. Let us denote $[x]_i$ and $[y]_i$ the equivalence classes of x and y under equivalence relation \sim_i . Then, $D_1([x], [y]) = \inf_{x \in [x]_1, y \in [y]_1} \{d(x, y)\} \leq \inf_{x \in [x]_2, y \in [y]_2} \{d(x, y)\} = D_2([x], [y])$, since $[x]_2 \subseteq [x]_1$ and $[y]_2 \subseteq [y]_1$.

An interesting idea arising from the above lemma 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.

⁴The 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.

6.4 Discriminative Power

As sketched previously, most indexing algorithms rely on building an equivalence relation. 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 1 above. It may also involve some extra CPU time (which although not the central point in this paper, must be kept reasonable). The second part consists of directly comparing the query against the candidate list. The following definition gives a name to both parts of the search cost.

Definition 3. *Let A be a search algorithm over (\mathbb{X}, d) based on a mapping to $(\pi(\mathbb{X}), D)$, and let $(q, r)_d$ be a range query. Then the internal complexity of A is the number of evaluations of d necessary to compute $([q], r)_D$, and the external complexity is $|([q], r)_D|$.*

We recall that $|([q], r)_D|$ refers to the number of elements in the original metric space, not the number of classes retrieved.

There is a concept related to the external complexity of a search algorithm, which we define next.

Definition 4. *The discriminative power of a search algorithm based on a mapping from (\mathbb{X}, d) to $(\pi(\mathbb{X}), D)$, with regard to a query $(q, r)_d$ of nonempty outcome, is defined as $|(q, r)_d|/|([q], r)_D|$.*

Although the definition depends on q and r , we can speak in general terms of the discriminative power by averaging over the q 's and r 's of interest. The discriminative power serves as an indicator of the performance or fitness of the equivalence relation.

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.

Let us consider Example 1. 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 lemma.

Lemma 2. *If A_1 and A_2 are search 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, for any r , $([q], r)_{D_2} \subseteq ([q], r)_{D_1}$. 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 next.

6.5 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. We define “locality” as a property of a partition that stands for how much the classes resemble cells. More formally,

Definition 5. The non-locality of a partition $\pi(\mathbb{X}) = \{\pi_1, \pi_2, \dots\}$ with respect to a finite dictionary \mathbb{U} is defined as $\max_i \{\max_{x,y \in \pi_i \cap \mathbb{U}} d(x,y)\}$, that is, as the maximum distance between elements lying in the same class.

We say that a partition is “local” or “non local” meaning that it has high or low locality. Figure 11 shows an example of a non local partition (u_5 and u_{12} lie in separate fragments of a single class). It is natural to expect more discriminative power from a local partition than from a non-local one. This is because in a non local partition the candidate list tends to contain elements actually far away from the query.

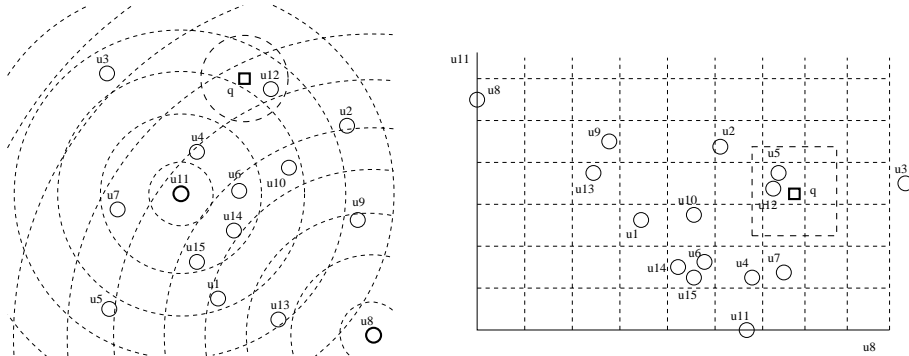


Figure 11: An equivalence relation induced by intersecting rings centered in two pivots, and how a query is transformed.

Notice that in Figure 11 the locality would improve sharply if we added a third pivot. In a vector space of k dimensions, it suffices to consider $k + 1$ pivots in general position⁵ to obtain a highly local partition. In general metric spaces we can also take a sufficient number of pivots so as to obtain highly local partitions.

However, obtaining local partitions may be expensive in internal complexity and not enough to achieve low external complexity, otherwise the bucketing method for vector spaces [10] explained in Example 2 would have excellent performance. Even with such a local partition and assuming uniformly distributed data, a number of empty cells are verified, whose volume grows exponentially with the dimension. We return later to this issue.

6.6 The Pivot Equivalence Relation

A large class of algorithms to build the equivalence relations are based on *pivoting*. This consists in considering the distances between an element and a number of preselected “pivots” (i.e. elements of \mathbb{U} or even \mathbb{X} , called also reference points, 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 the 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 correspond to the intuitive notion of the family of sphere shells with center p . Points falling in the same sphere shell (i.e. at the same distance to p) are equivalent from the view point of p .

⁵That is, not lying on a $(k - 1)$ -hyperplane.

The above equivalence relation is easily generalized to k pivots.

Definition 6. The pivot equivalence relation based on elements $\{p_1, \dots, p_k\}$ (the k pivots) is defined as

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

A graphical representation of the class in the general case corresponds to the intersection of several sphere shells centered at the points p_i (recall Figure 11).

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 . With k pivots, this becomes $D([x], [y]) = \max_i \{|d(x, p_i) - d(y, p_i)|\}$. 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 6, the indexing algorithm will consist in finding the set of equivalence classes such that they fall inside the radius of the search when using D in the quotient 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 12 illustrates this concept (Figure 11 is also useful).

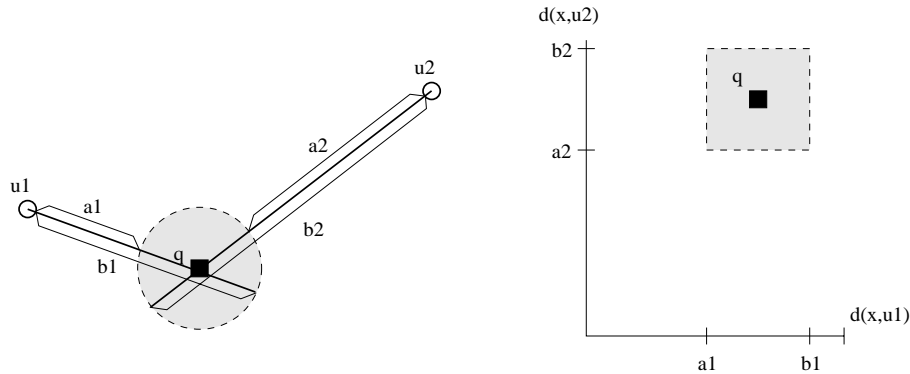


Figure 12: Mapping from a metric space onto a vector space under the L_∞ metric, using two pivots.

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 $d(q, u) > r$ without need to actually evaluate $d(q, u)$. 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 .

6.7 The Voronoi Equivalence Relation

A different type of equivalence relation, used by another large class of algorithms, is defined with respect to the proximity to a set of elements (that we call “centers” to distinguish them from the pivots of the previous section).

Definition 7. The Voronoi equivalence relation based on $\{c_1, \dots, c_m\}$ (the centers) is

$$x \sim_{\{c_i\}} y \iff \text{closest}(x, \{c_i\}) = \text{closest}(y, \{c_i\})$$

where $\text{closest}(z, S) = \{w \in S, \forall w' \in S, d(z, w) \leq d(z, w')\}$. The associated partition is called a Voronoi partition.

That is, we divide the space with one partition for each c_i and the class of c_i is that of the points that have c_i as their closest center⁶. Figure 13 shows an example in (\mathbb{R}^2, L_2) . In particular note that we can select \mathbb{U} as the set of centers, in which case the partition has optimal locality. Even if $\{c_i\}$ is not \mathbb{U} , Voronoi partitions have good locality.

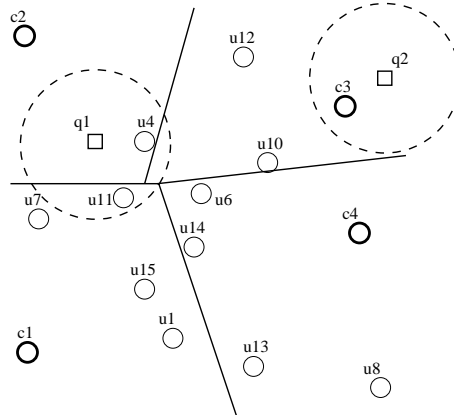


Figure 13: A Voronoi partition using four centers and two query balls intersecting some classes. Note that the class of c_3 could be excluded from consideration for q_1 by using the covering radius criterion but not the hyperplane criterion, while the opposite happens to discard c_4 for q_2 .

In vector spaces the Voronoi partition is assumed to use \mathbb{U} as the set of centers. Its associated concept, the “Delaunay tessellation” is a graph whose nodes are the elements of \mathbb{U} and whose edges connect nodes whose classes share a border. The Delaunay tessellation is the basis of very good algorithms for proximity searching [3, 66]. For example, an $O(\log n)$ NN algorithm exists in two dimensions. Unfortunately, this algorithm does not generalize efficiently to more than two dimensions. The Delaunay tessellation, which has $O(n)$ edges in two dimensions, can have $O(n^2)$ edges in three and more dimensions.

In a general metric space, the $D_0([x], [y])$ distance in the space $\pi(\mathbb{X})$ of the Voronoi classes is, as before, the smallest distance between points $x \in [x]$ and $y \in [y]$. To find $[q]$ we basically need to find the nearest neighbor of q in the set of centers. The outcome of the query $([q], r)_{D_0}$ is the set of classes intersected by query ball (see Figure 13).

A problem in general metric spaces is that it is not easy to bound a class so as to determine whether or not the query ball intersects it. From the many possible criteria, two are the most popular:

Hyperplane criterion: this is the most basic one and the one that best express the idea of a Voronoi partition. In essence, if c is the center of the class $[q]$ (i.e. the center closest to q), then (1) the query ball of course intersects $[c]$; (2) the query ball does not intersect $[c_i]$ if $d(q, c) + r <$

⁶We are using our own definition of “Voronoi partition”, which matches with the concept of “Dirichlet domain” if all the elements of the set are centers.

$d(q, c_i) - r$. Graphically, if the query ball does not intersect the hyperplane dividing its closest neighbor and another center c_i , then the ball is totally outside the class of c_i .

Covering radius criterion: this tries to bound the class $[c_i]$ by considering a ball centered at c_i that contains all the elements of \mathbb{U} that lie in the class. We define

Definition 8. *The covering radius of c for \mathbb{U} is $cr(c) = \max_{u \in [c] \cap \mathbb{U}} d(c, u)$.*

Now it is clear that we can discard c_i if $d(q, c_i) - r > cr(c_i)$.

7 The Curse of Dimensionality

As explained, one of the major obstacles for the design of efficient search techniques on metric spaces is the existence and ubiquity in real applications of the so called *high dimensional spaces*. Traditional indexing techniques for vector spaces (such as *kd-trees*) have an exponential dependency on the representational dimension of the space (as the volume of a box or hypercube containing the answers grows exponentially with the dimension).

More recent indexing techniques for vector spaces and those for generic metric space can get rid of the representational dimension of the space. This makes a big difference in many applications that handle vector spaces of representational high dimension but intrinsic low dimension (e.g. a plane immersed in a 50-dimensional vector space, or simply clustered data). However, in some cases even the intrinsic dimension is very high and the problem becomes intractable for exact algorithms, and we have to resort to approximate or probabilistic algorithms (Section 5.3.3).

Our aim in this section is (a) to show that the concept of intrinsic dimensionality can be conceived even in a general metric space; (b) to give a quantitative definition of the intrinsic dimensionality; (c) to show analytically the reason for the so-called “curse of dimensionality”; and (d) to discuss the effects of pivot selection techniques.

7.1 Intrinsic Dimensionality

Let us start with a well-known example. Consider a distance such that $d(x, x) = 0$ and $d(x, y) = 1$ for all $x \neq y$. Under this distance (in fact an equality test), 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, no matter how smart is our indexing technique.

Let us consider the *histogram* of distances between points in the metric space \mathbb{X} . This can be approximated by using the dictionary \mathbb{U} as a random sample of \mathbb{X} . This histogram is mentioned in many papers, e.g. [16, 22, 27], as a fundamental measure related to the intrinsic dimensionality of the metric space. The idea is that, as the space has higher intrinsic dimension, the mean μ of the histogram grows and its variance σ^2 is reduced. Our previous example is an extreme case.

Figure 14 gives an intuitive explanation of why the search problem is harder when the histogram is concentrated. If we consider a random query q and an indexing scheme based on random pivots, then the possible distances between q and a pivot p are distributed according to the histogram of the figure. The elimination rule says that we can discard any point u such that $d(p, u) \notin [d(p, q) - r, d(p, q) + r]$. The grayed areas in the figure show the points that we *cannot* discard. As the histogram is more and more concentrated around its mean, less and less points can be discarded using the information given by $d(p, q)$.

This phenomenon is independent on the nature of the metric space (vectorial or not, in particular) and gives us a way to quantify how hard is to search on an arbitrary metric space.

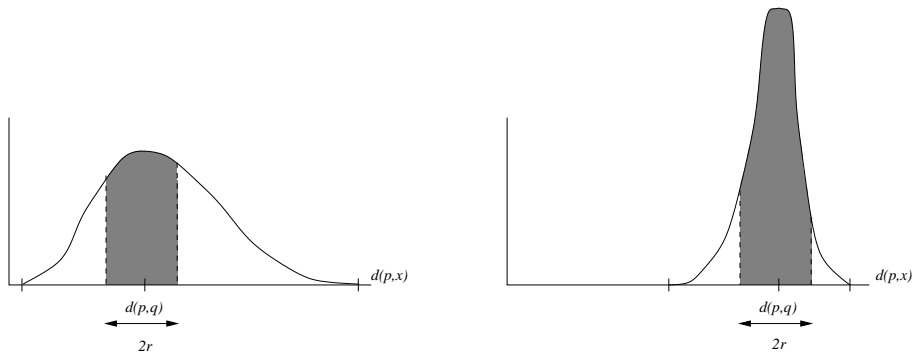


Figure 14: A low-dimensional (left) and high-dimensional (right) histogram of distances, showing that on high dimensions virtually all the elements become candidates for the exhaustive evaluation. Moreover, we should use a larger r in the second plot in order to retrieve some elements.

Definition 9. *The intrinsic dimensionality of a metric space is defined as $\rho = \frac{\mu^2}{2\sigma^2}$, where μ and σ^2 are the mean and variance of its histogram of distances.*

The technical convenience of the exact definition is made clear shortly. The important part is that the intrinsic dimensionality grows with the mean and shrinks with the variance of the histogram.

The particular cases of the L_s distances in vector spaces are useful illustrations. As shown in [68], a uniformly distributed k -dimensional vector space under the L_s distance has mean $\Theta(k^{1/s})$ and standard deviation $\Theta(k^{1/s-1/2})$. Therefore its intrinsic dimensionality is $\Theta(k)$ (although the constant is not necessarily 1). So the intuitive concept of dimensionality in vector spaces matches our general concept of intrinsic dimensionality.

7.2 A Lower Bound for Pivoting Algorithms

Our main result in this section relates the intrinsic dimensionality with the difficulty of searching with a given search radius r using a pivoting equivalence relation that chooses the pivots at random. As we show next, the difficulty of the problem is related to r and the intrinsic dimensionality ρ .

We are considering independent identically distributed (i.i.d.) random variables for the distribution of distances between points. Although not accurate, this simplification is optimistic and hence can be used to lower bound the performance of the indexing algorithms. We come back to this shortly after the discussion.

Let $(q, r)_d$ be a range query over a metric space indexed by means of k random pivots, and let u be an element of \mathbb{U} . The probability that u cannot be excluded from direct verification after considering the k pivots is exactly

$$Pr(|d(q, p_1) - d(u, p_1)| \leq r, \dots, |d(q, p_k) - d(u, p_k)| \leq r)$$

Since all the pivots are assumed to be random and their distance distributions i.i.d. random variables, this expression is the product of probabilities

$$Pr(|d(q, p_1) - d(u, p_1)| \leq r) \times \dots \times Pr(|d(q, p_k) - d(u, p_k)| \leq r)$$

which for the same reason can be simplified to

$$Pr(\text{not discarding } u) = Pr(|d(q, p) - d(u, p)| \leq r)^k$$

for a random pivot p .

If X and Y are two i.i.d. random variables with mean μ and variance σ^2 , then the mean of $X - Y$ is 0 and its variance is $2\sigma^2$. Using Chebyshev's inequality⁷ we have that $Pr(|X - Y| > \varepsilon) < 2\sigma^2/\varepsilon^2$. Therefore,

$$Pr(|d(q, p) - d(u, p)| \leq r) \geq 1 - \frac{2\sigma^2}{r^2}$$

where σ^2 is precisely the variance of the distance distribution in our metric space. The argument that follows is valid for $2\sigma^2/r^2 < 1$ (large enough radii). Then, we have

$$Pr(\text{not discarding } u) \geq \left(1 - \frac{2\sigma^2}{r^2}\right)^k$$

We have now that the total search cost is the number of internal distance evaluations (k) plus the external evaluations, whose number is on average $n \times Pr(\text{not discarding } u)$. Therefore

$$Cost \geq k + n \left(1 - \frac{2\sigma^2}{r^2}\right)^k$$

is a *lower bound* to the average search cost by using pivots. Optimizing we obtain that the best k is

$$k^* = \frac{\ln n + \ln \ln(1/t)}{\ln(1/t)}$$

where $t = 1 - 2\sigma^2/r^2$. Using this optimal k^* , we obtain an absolute (i.e. independent on k) lower bound for the average cost of any random pivot-based algorithm:

$$Cost \geq \frac{\ln n + \ln \ln(1/t) + 1}{\ln(1/t)} \geq \frac{\ln n}{\ln(1/t)} \geq \frac{r^2}{2\sigma^2} \ln n$$

which shows that the cost depends strongly on σ/r . As r increases t tends to 1 and the scheme requires more and more pivots and it is anyway more and more costly. We have just proved

Theorem 1. *Any pivot based algorithm using random pivots has a lower bound $\frac{r^2}{2\sigma^2} \ln n$ in the average number of distance evaluations performed for a range query (q, r) with random q , where σ is the standard deviation of the distance distribution. If $r = \mu f$, where μ is the mean of the distance distribution, then the lower bound can be expressed as $\rho f^2 \ln n$, where ρ is the intrinsic dimension of the space.*

This result matches that of [4, 6] on FHQTs, about obtaining $\Theta(\log n)$ search time using $\Theta(\log n)$ pivots.

The theorem shows clearly that the parameter governing the performance of range searching algorithms is ρf^2 , not just ρ . As the dimension ρ grows we have to reduce the search radius (seen as a fraction f of the average distance) in order to keep the same performance. Of course, this is not reasonable for many applications. For example, in vector spaces we need to keep f constant (r growing) in order to get a fixed percentage of the database elements. As another example, many of the algorithms for NN searching use random distances to q as temporary upper bounds to the

⁷For an arbitrary distribution Z with mean μ_z and variance σ_z^2 , $Pr(|Z - \mu| > \varepsilon) < \sigma_z^2/\varepsilon^2$.

distance to the nearest neighbor. As the mean distance is larger in higher dimensions, those bounds are worse and the search cost is higher. Moreover, if q is random then its actual distance to its nearest neighbor is indeed larger.

On the other hand, it is possible that in some applications the query q is known to be a perturbation of some element of \mathbb{U} and therefore we can keep a constant search radius r as the dimension grows. Even in those cases ρf^2 may grow if σ shrinks with the dimension, as in the L_s vector spaces with $s > 2$.

We have considered i.i.d. random variables for each pivot and the query. This is a reasonable approximation, as we do not expect much difference between the “view points” from the general distribution of distances to the individual distributions (see Section 7.4). The expression given in Eq. (7.2) cannot be obtained without this simplification.

A stronger assumption comes from considering all the variables as independent. This is an optimistic consideration equivalent to assuming that in order to discard each element u of the set we take k new pivots at random. The real algorithm fixes k random pivots and uses them to try to discard all the elements u of the set. The latter alternative can suffer from dependencies from a point u to another, which cannot happen in the former case (for example, if u is close to the 3rd pivot and u' is close to u then the distance from u' to the third pivot carries less information). Since the assumption is optimistic, using it to reduce the joint distribution in Eq. (7.2) to the expression given in Eq. (7.2) keeps the lower bound valid.

Figure 15 shows an experiment on the search cost in (\mathbb{R}^ℓ, L_2) using different number of pivots k and dimensions ℓ . The $n = 100,000$ elements are generated at random and the pivots are randomly chosen from the set. We average over 1,000 random queries whose radius is set to retrieve 10 elements of the set. We count the number of distance evaluations. The left plot shows the existence of an optimum $k^* = 110$, while the right plot shows the predicted $O(n(1 - 1/\Theta(\ell))^k)$ behavior. We have not enough memory in our machine to show the predicted growth in $k^* \approx \Theta(\ell) \ln(n)$.

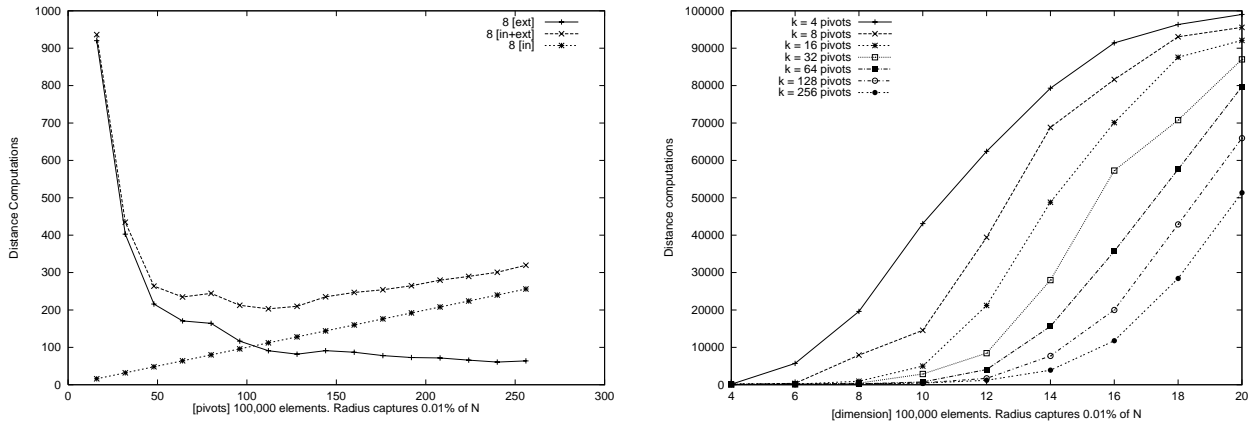


Figure 15: On the left, internal, external and overall distance evaluations in 8 dimensions, using different number of pivots k . On the right, overall distance evaluations as the dimension grows for fixed k .

Figure 16 shows the effect in a different way. As the dimension grows, the histogram of L_2 moves to the right ($\mu = \Theta(\sqrt{\ell})$). Yet the pivot distance D (in the projected space $(\mathbb{R}^\ell, L_\infty)$) remains about the same for fixed k . Increasing k from 32 to 512 moves the histogram slightly to the right. This shift is effective in low dimensional metric spaces, but it is ineffective in high dimensions. The plots of these two histograms can measure how good are the pivoting algorithms. Intuitively, the

overlap between the histogram for the pivot distance and the histogram for the original distance is directly proportional to the discriminative power of the pivot mapping. As the overlap increases the algorithms becomes more effective.

The particular behavior of D in figure 16 is due to the fact that D is the maximum of k random variables whose *mean* is $\Theta(\sigma)$ (i.e. $|d(p, q) - d(p, x)|$). The fact that D does not grow with ℓ means that, as a lower bound for d , it gets less effective in higher dimensions.

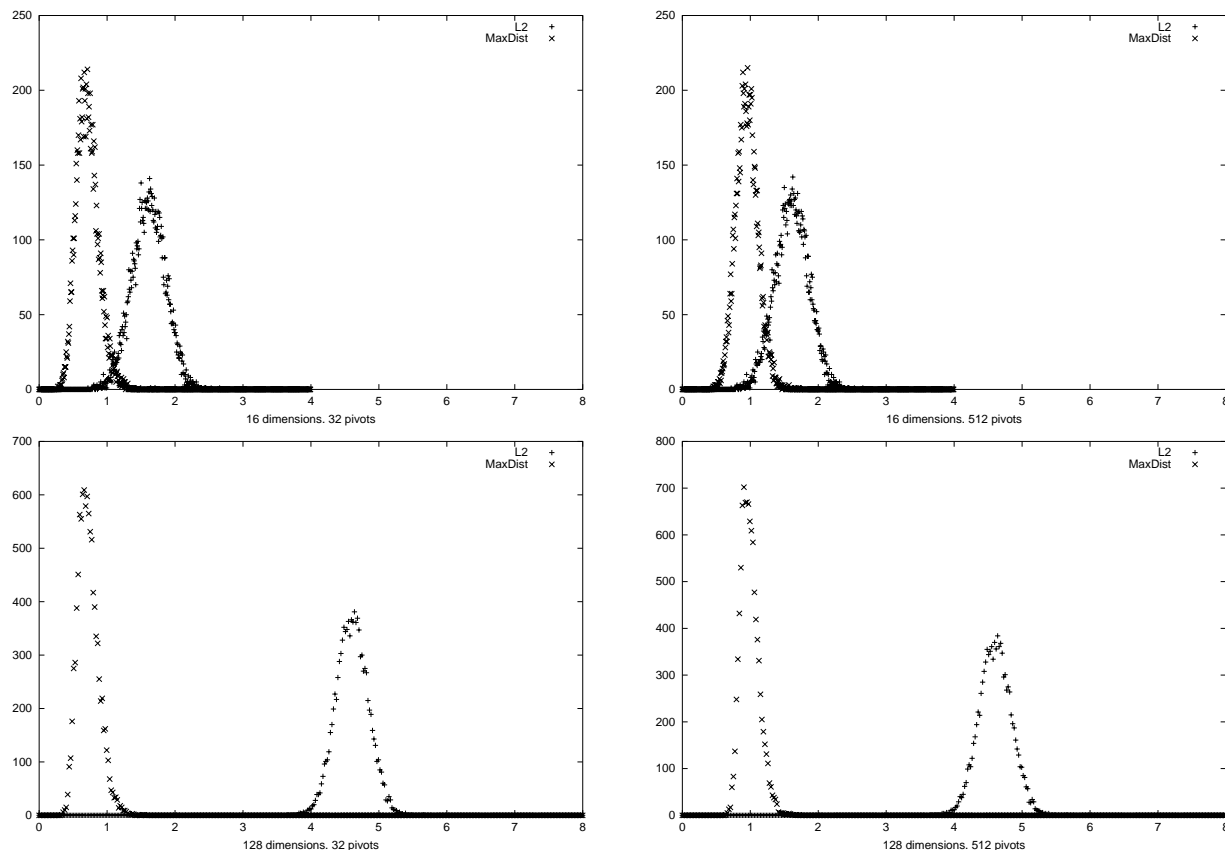


Figure 16: Histograms comparing the L_2 distance in different ℓ -dimensional Euclidean spaces and the pivot distance (MaxDist) obtained using different numbers k of pivots. In the top row $\ell = 16$ and in the bottom row $\ell = 128$. On the left $k = 32$ and on the right $k = 512$.

7.3 A Lower Bound for Voronoi Algorithms

We now try to obtain a lower bound to the search cost of algorithms based on the Voronoi partition. The result is surprisingly similar to that of the previous section. Our lower bound considers only the hyperplane criterion, which is the most purely associated to the Voronoi partition. We assume just the same facts about the distribution of distances as in the previous section: all of them are i.i.d. random variables.

Let $(q, r)_d$ be a range query over a metric space indexed by means of m random centers $\{c_1 \dots c_m\}$. The m distances $d(q, c_i)$ can be considered random variables $X_1 \dots X_m$ whose distribution is that of the histogram of the metric space. The distribution of the distance from q to its closest center c is that of $Y = \min\{X_1 \dots X_m\}$. The hyperplane criterion specifies that a

class $[c_i]$ cannot be excluded if $d(q, c) + r \geq d(q, c_i) - r$. The probability that this happens is $Pr(Y \geq X_i - 2r)$. But since Y is the minimum over m variables with the same distribution, the probability is $Pr(Z \geq X - 2r)^m$, where X and Z are two random variables distributed according to the histogram. Using Chebyshev's inequality and noticing that if $Z < X - 2r$ then X or Z are at distance at least r from their mean, we can say that

$$Pr(\text{not discarding } [c_i]) = Pr(Z \geq X - 2r)^m \geq \left(1 - \frac{\sigma^2}{r^2}\right)^m$$

On average each class has n/m elements, so that the external complexity is $n \times Pr(\text{not discarding } [c_i])$. The internal cost to find the intersected classes deserves some discussion. In all the hierarchical schemes that exist, we consider that the real partition is that induced by the leaves of the trees, i.e. the most refined ones. We see all the rest of the hierarchy as a mechanism to reduce the internal complexity of finding the small classes (hence the m we use here is not, say, the m of GNATs, but the total number of final classes). It is difficult to determine this internal complexity (an upper bound is m), so we call it $C_I(m)$, knowing that it is between $\Omega(\log m)$ and $O(m)$. Then a lower bound to the search complexity is

$$Cost \geq C_I(m) + n \left(1 - \frac{\sigma^2}{r^2}\right)^m$$

which indeed is very similar to the lower bound on pivot based algorithms. Optimizing on m yields

$$m^* = \frac{\ln n + \ln \ln(1/t') - \ln i'(m^*)}{\ln(1/t')}$$

where $t' = 1 - \sigma^2/r^2$. Using the optimal m^* the search cost is lower bounded by

$$Cost = \Omega\left(C_I(\log_{1/t'} n)\right) = \Omega\left(C_I\left(\frac{r^2}{\sigma^2} \ln n\right)\right)$$

which also shows an increase in the cost as the dimensionality grows. We have just proved

Theorem 2. *Any Voronoi based algorithm based on random centers has a lower bound $C_I((r/\sigma)^2 \ln n)$ in the average number of distance evaluations performed for a range query (q, r) with random q , where σ is the standard deviation of the distance distribution and $C_I()$ is the internal complexity to find the relevant classes, where $\Omega(\log m) = C_I(m) = O(m)$. If $r = \mu f$, where μ is the mean of the distance distribution, then the lower bound can be expressed as $C_I(2\rho f^2 \ln n)$, where ρ is the intrinsic dimension of the space.*

This result is weaker than Theorem 1 because of our inability to give a good lower bound on i , so we cannot ensure more than a logarithmic increase with respect to ρ . However, even assuming $C_I(m) = \Theta(m)$ (i.e. exhaustive search of the classes), when the theorem becomes very similar to Theorem 1, there is an important reason that explains why the Voronoi based algorithms can in practice be better than pivot based ones. We can in this case achieve the optimal number of centers m^* , which is impossible in practice for pivot-based algorithms. The reason is that it is much more economical to represent the Voronoi partition using m centers than the pivot partition using k pivots.

Figure 17 shows an experiment on the same dataset, where we have used different m values and a hierarchical Voronoi partitioning based on them. We have used the hyperplane and the maximum

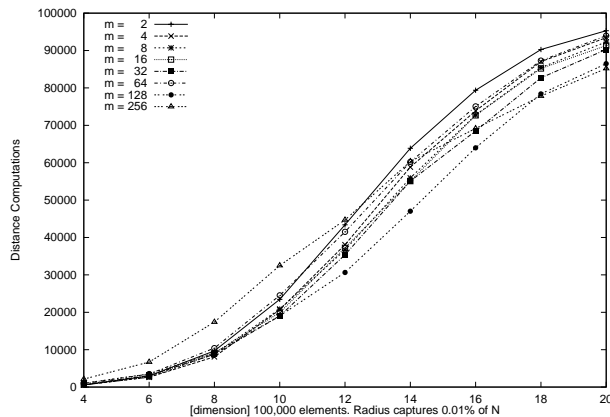


Figure 17: Overall distance evaluations using a hierarchical Voronoi partitioning with different arities.

radius criteria to prune the search. As can be seen, the dependency on the dimension of the space is not so sharp as for pivot based algorithms, and is closer to a dependency of the form $\Theta(\ell)$.

The general conclusion is that, even if the lower bounds using pivot based or Voronoi based algorithms look similar, the first ones need much more space to store the classes resulting from k pivots than the last ones using the same number of partitions. Hence, the latter can realistically use the optimal number of classes, while the former cannot. If pivot based algorithms are given all the necessary memory, then using the optimal number of pivots they can improve over Voronoi based algorithms, because t is better than t' , but this is more and more difficult as the dimension grows.

Figure 18 compares both types of partitioning. As can be seen, the pivoting algorithm improves over the Voronoi partitioning if we give it enough pivots. However, “enough” is a number that increases with the dimension and with the search radius (i.e. ρ and f). For ρf^2 large enough, the required number of pivots will be unacceptably high in terms of memory requirements.

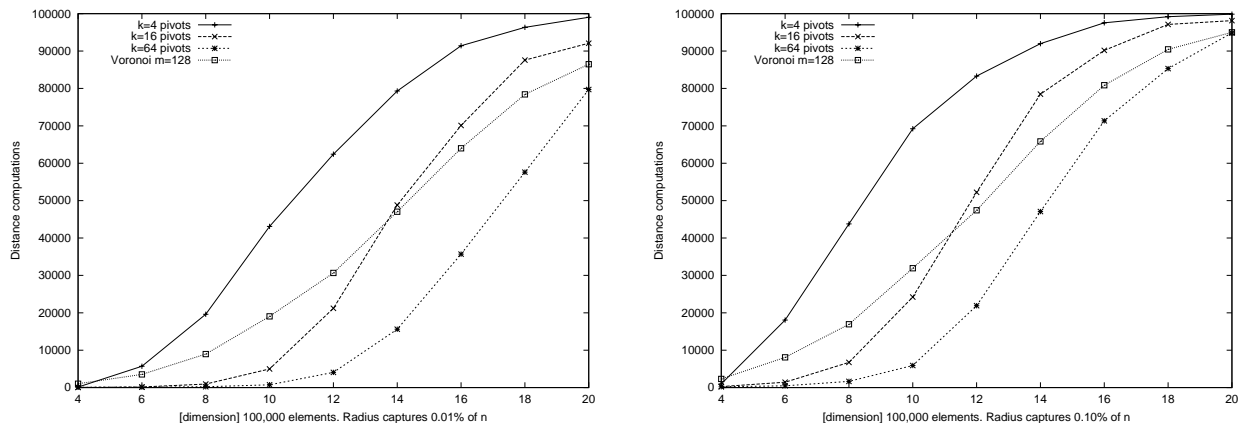


Figure 18: Distance evaluations for increasing dimension. We compare the Voronoi algorithm of Figure 17 using 128 centers per level against the filtration using k pivots for $k = 4, 16, 64$. On the left plot the search radius captures 0.01% of the set, on the right 0.1%.

7.4 Pivot and Center Selection Techniques

In [37], they prove formally that if the dimension is constant, then after properly selecting (not at random!) a constant number k of pivots the exhaustive search costs $O(1)$. This contrasts with our $\Omega(\log n)$ lower bound. The difference is that they do not take the pivots at random but select a set of pivots which is assumed to have certain selectivity properties. This shows that the way in which the pivots are selected can affect the performance. Unfortunately, their theorem is not constructive and does not show how to select such a set of pivots.

Little is known about pivot/center (let us call them collectively “references”) selection policies, and in practice most methods choose them at random, with a few exceptions. For instance, in [58] 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 references should be far apart from each other, which is evident since close references will give almost the same information. On the other hand, references selected at random are already far apart in a high-dimensional space.

The histogram of distances gives a formal characterization of good references. Let us start with a definition.

Definition 10. *The local histogram of an element u is the distribution of distances from u to every $x \in \mathbb{X}$.*

A good reference has a flatter histogram, which means that it will discard more elements at query time. The measure $\rho = \mu^2/(2\sigma^2)$ of intrinsic dimensionality (now defined on the local histogram of u) can be used as a good parameter to evaluate how good is a reference (good references have local histograms with small ρ).

This is related to the difference in viewpoints (histograms) between different references, a subject discussed in depth in [27]. The idea is that the local histogram of a reference u may be quite different from the global histogram (especially if u is not selected at random). This is used in [27] to show that if the histograms for different references are similar then they can accurately predict the behavior of instances of their data structure (the MT), a completely different issue.

Note also that good reference selection becomes harder as the intrinsic dimension grows. As the global histogram reduces its variance, it becomes more difficult to find references that deviate significantly from it, as already noted in [27].

The histogram characterization explains a well-known phenomenon: to discriminate among the elements in the class of a local partition (or in a cluster), it is a good idea to select a pivot from the same class. 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 [46] 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. On the other hand, outliers can be good at a first approximation, in order to discriminate among clusters, but later they are unable to separate well the elements of the same cluster.

Selecting good individual references, i.e. with a flatter histogram, is not enough to ensure a good performance. For example, a reference close to a good one is probably good as well, but using both of them gives almost the same information as using only one. It is necessary to include a variety of independent viewpoints in the selection. If we consider the selection of references as an optimization problem, the goal is to obtain a set of references that are collectively good for indexing purposes. The set of references is good if it approximates the original distance well. This is more evident in the case of pivoting algorithms, where a contractive distance is implicitly used. A practical way to decide whether our set of centers is effective is to compare the histograms of

the original distance and the pivot distance as in Figure 16. A good reference set will give a pivot distance histogram with a large intersection with the histogram of the original distance.

7.5 The Effect on the Discriminative Power

Another reason that explains the curse of dimensionality is related to the decrease in the discriminative power of a partition, because of the odd “shapes” of the classes.

As explained before, a non local partition suffers from the problem of being unable to discriminate between points that are actually far away from each other, which leads to unnecessarily high external complexity. The solution is to select enough pivots or to use a Voronoi method that yields a local partition. However, even in a local partition, the shape of the cell is fixed at indexing time, while the shape of the space region defined as interesting for a query q is a ball dynamically determined at query time. The discriminative power can be visualized as the volume of the query ball divided by the total volume of the classes intersected by that ball.

A good example is that of a ball inside a box in a k -dimensional L_2 space. The box is the class obtained after using k orthogonal pivots, and the partition obtained is quite local. Yet the volume of the ball is smaller, and the ratio with respect to the volume of the box (i.e. the discriminative power) decreases exponentially with k . This means that we have to examine a volume (and a number of candidate elements) that, with respect to the size of the final result, grows exponentially with the dimension k . This fact is behind the exponential dependency on the dimension in data structures for vector spaces such as the kd -tree or the R-tree.

The same phenomenon occurs in general metric spaces, where the “shapes” of the cells cannot possibly fit an unknown query ball. We can add more pivots to bound better any possible ball, but this increases the internal complexity and becomes more difficult as the intrinsic dimension grows (the smaller the variance, the more pivots are needed to make a difference).

8 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 concentrate on pivoting algorithms. They differ in their method to select the pivots, in when is the selection made, and in how much information on the comparisons is used. Later, we consider the Voronoi algorithms. These differ in their methods to select the centers and to bound the equivalence classes.

Figure 19 summarizes our classification of methods attending to their most important features (explained throughout the section).

8.1 Pivot Based Algorithms

8.1.1 Search Algorithms

Once we have determined the equivalence relation to use (i.e. the k pivots), we preprocess the dictionary 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 storing $d(u_i, p_j)$.

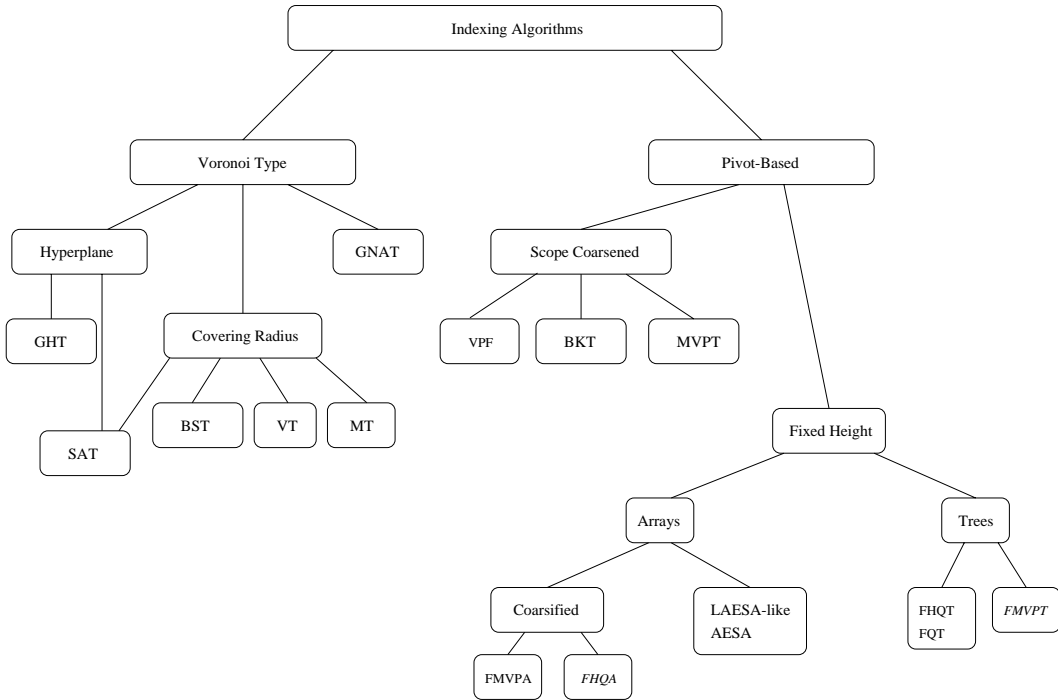


Figure 19: Taxonomy of the existing algorithms. The methods in italics are combinations that appear naturally as soon as the taxonomy is built.

At query time, we first compare the query q against the k pivots, hence obtaining its k coordinates $[q] = (y_1, \dots, y_k)$ in the target space, i.e. its equivalence class. 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, \dots, x_k) of the table) and see if the triangular inequality allows discarding 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 [48, 23]. This is possible because we are interested, at column i , in the values $[y_i - r, y_i + r]$. The extra CPU cost gets closer to $O(k \log n)$ than to $O(n)$ by using this technique.

This is not the only improvement 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 [37, 4, 6], one cannot normally understand the application well enough to predict the actual optimal number of pivots k^* (i.e. the point where it is better to switch to exhaustive evaluation).

Another improvement that can be used with column-wise evaluation is that the selection of the pivots can be done on the fly instead of beforehand 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 p we have to store the coordinates of all the elements of the set for p (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 sections).

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 the 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.

8.1.2 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 *scope coarsening*. Their ideas are roughly illustrated in Figure 20.

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

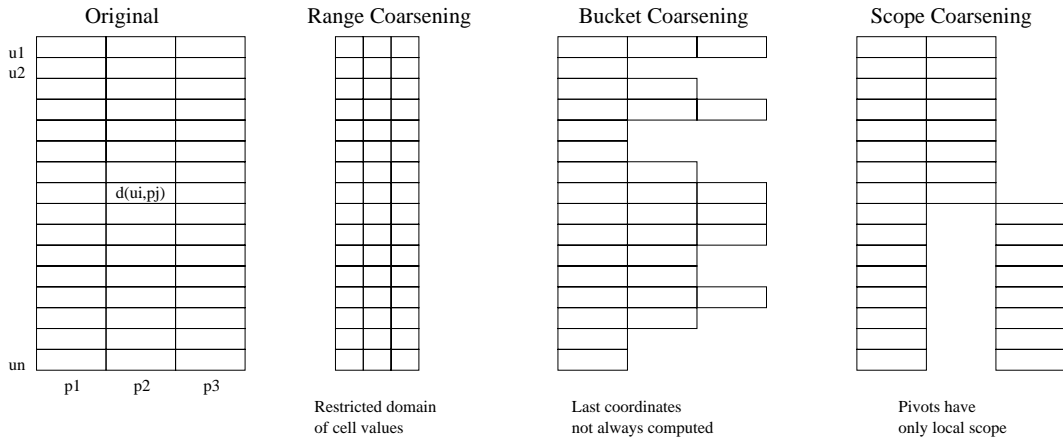


Figure 20: Different coarsification methods.

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 with the same amount of memory 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 amount of memory to use.

8.1.3 Range Coarsening

The auxiliary data structures proposed by most authors for continuous distance functions are aimed at reducing 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

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

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 and can be naturally extended to more than one pivot.

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, and that the tree has the same 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.

		Fixed percentiles	Fixed width
Different pivot per node (scope coarsening)		VPT, MVPT	BKT
Different pivot per level	Different slice per node	<i>FMVPT</i>	FQT, FHQT
	Different slice per level	FMVPA (FQA)	<i>FHQA</i>

Table 2: Different options for range coarsening. We put in italics the new structures created to fill the empty holes.

Therefore, we can have slices of fixed width (BKT, FQT, FHQT) or determined by percentiles (VPT, MVPT, FQA). We may have a different pivot per node (BKT, VPY, MVPT) or per level (FQT, FHQT, FQA). Among the last, we can define the slices at each node (FQT, FHQT) or for the whole level (FQA). All these choices are drawn in Table 2. The empty slots have been filled with new structures that are defined now.

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. In the FHQT, instead, each *node* has a different slice width.

FMVPT Is a cross between an MVPT and a FHQT. 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. The particular case where $m = 2$ will be called **FHVPT**.

FMVPA Is just a new name for the FQA, more appropriate for our discussion since it is to MVPts as FHQAs to FHQTs: the FMVPA uses variable width slices to ensure that the subtrees are balanced in size, but the same slices are used for all the nodes of a single level, so that the balance is only level-wise.

The combinations we have just created allow us to explain some important concepts.

Amount of range coarsening. Let us consider FHQAs and FMVPAs. They are no more than LAESA with different forms of range coarsening. 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 then reduced to kbn bits.

However, range coarsening is not just a technique to reduce space, but the same space can be used to accommodate more pivots. 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.

Percentiles versus fixed width. Another unclear issue is whether fixed slices is better or worse than percentile splitting. On one hand, a balanced data structure has obvious advantages because

the internal complexity may be reduced. Fixed slices produce unbalanced structures since the outer rings have much more elements (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 ring). If the rings are too narrow, many rings will be frequently included in the radius of interest of the queries (see Figure 21). An alternative idea 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.

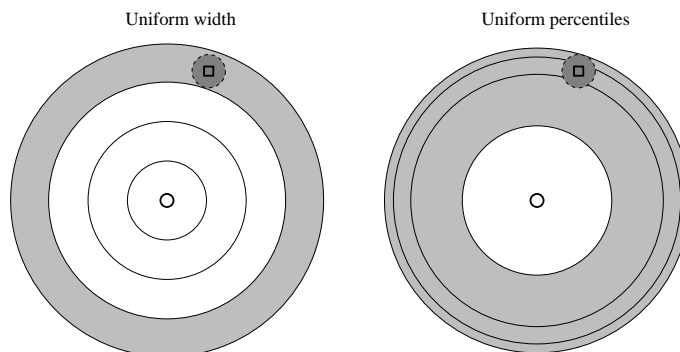


Figure 21: The same query intersects more rings when using uniform percentiles.

Trees versus arrays FHQTs and FMVPTs are almost tree versions of FHQAs 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 may 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 the i -th ring, then we just traverse the i -th subtree. This reduces the extra CPU time. If the distance is too fine-grained, however, the root will have nearly n children and the subtrees will have just 1 child. The structure will be very similar to a table of k coordinates per element and the search will degenerate into a linear row-wise traversal. Hence, range coarsening is also a tool to reduce the extra CPU time.

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.

Summarizing, range coarsening can be applied using fixed-width or fixed-percentile slices. They can reduce the space necessary to store the coordinates, which can allow the use of 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.

8.1.4 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 8.1.1) 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.

8.1.5 Scope Coarsening

The last and least obvious form of coarsening is the one we call “scope 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 us 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 scope 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 these structures 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 [58, 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.

With hyperplanes	GHT, SAT
With balls	BST, VT, MT, SAT
With rings	GNAT

Table 3: Different options for limiting classes.

8.2 Voronoi Partition Algorithms

All the remaining algorithms (GHTs, BSTs, GNATs, VTs, MTs, SATs) rely on a hierarchical Voronoi partition of the metric space. A first source of differences is in how the centers are selected at each node. GHTs and BSTs take two elements per level. VTs repeat previous centers when creating new nodes. GNATs select m centers far apart. MTs try to minimize covering radii. SATs select a variable number of close neighbors of the parent node.

The main difference, however, lies in the search algorithm. While GHTs use purely the hyperplane criterion, BSTs, VTs and MTs use only the covering radius criterion. SATs use both criteria to increase pruning. In all these algorithms the query q is compared against all the centers of the current node and the criteria are used to discard subtrees.

GNATs are a little different, as they use none of the above criteria. Instead, they apply an AESA-like search over the m centers considering their “range” values. That is, a center c_i is selected, and if the query ball does not intersect a ring around c_i that contains all the elements of c_j , then c_j and all its class (subtree) can be safely discarded. In other words, GNATs limit the class of each center by intersecting rings around the other centers. This way of limiting the extent of a class is different from both the hyperplane and the covering radius criteria. It is probably more efficient, but it requires storing $O(m^2)$ distances at each level.

Table 3 summarizes the differences. It is clear that there are many possible combinations that have not been tried, but we do not attempt to enumerate all them.

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.

9 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 understanding 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. We have also analyzed the main factors that affect the efficiency when searching metric spaces. The main conclusions of our work are summarized as follows

1. The concept of intrinsic dimensionality can be defined on general metric spaces as an abstract

and quantifiable measure that affects the search performance.

2. The main factors that affect the efficiency of the search algorithms are the intrinsic dimensionality of the space and the search radius.
3. Equivalence relations are a common ground underlying all the indexing algorithms, and they divide the search cost in terms of internal and external complexity.
4. 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.
5. The equivalence relations can be coarsened to save space or to improve the overall efficiency by making better use of the pivots. A hierarchical refinement of classes can improve performance.
6. Although there is an optimal number of pivots to use, this number is too high in terms of space requirements. In practical terms, a pivot based index can outperform a Voronoi based index if it has enough memory.
7. As this amount of memory becomes unfeasible as the dimension grows, Voronoi based algorithms normally outperform pivot based ones in high dimensions.
8. In high dimension the search radius needed to retrieve a fixed percent of the database is very large. This is the reason of the failure to overcome the brute force search with an exact indexing algorithm.

A number of open issues require further attention. The main ones follow.

- For pivot based algorithms, 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.
- For Voronoi based algorithms, work more on clustering schemes in order to select good centers. Find ways to reduce construction times (which are in many cases too high).
- Search for good hybrids between Voronoi 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 smaller, 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 clusters have high locality.
- 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.
- Take I/O costs into account, which may very well dominate the search time in some applications. The only existing work on this is the M-tree [26].
- 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, 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, 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)*, pages 38–46. IEEE CS Press, 1999.
- [24] E. Chávez, J. Marroquín, and G. Navarro. Overcoming the curse of dimensionality. In *European Workshop on Content-Based Multimedia Indexing (CBMI'99)*, pages 57–64, 1999. <ftp://garota.fismat.umich.mx/pub/users/elchavez/fqa.ps.gz>.
- [25] T. Chiueh. Content-based image indexing. In *Proc. of the 20th Conference on Very Large Databases (VLDB'94)*, pages 582–593, 1994.
- [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] P. Ciaccia, M. Patella, and P. Zezula. Processing complex similarity queries with distance-based access methods. In *Proc. 6th International Conference on Extending Database Technology (EDBT'98)*, 1998.

- [29] K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.
- [30] T. Cox and M. Cox. *Multidimensional Scaling*. Chapman and Hall, 1994.
- [31] F. Dehne and H. Nolteimer. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987.
- [32] L. Devroye. *A Course in Density Estimation*. Birkhauser, 1987.
- [33] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.
- [34] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *J. of Intelligent Information Systems*, 3(3/4):231–262, 1994.
- [35] C. Faloutsos and I. Kamel. Beyond uniformity and independence: analysis of R-trees using the concept of fractal dimension. In *Proc. 13th ACM Symposium on Principles of Database Principles (PODS'94)*, pages 4–13, 1994.
- [36] 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.
- [37] 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, 1993.
- [38] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.
- [39] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
- [40] 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.
- [41] J. Hair, R. Anderson, R. Tatham, and W. Black. *Multivariate Data Analysis with Readings*. Prentice-Hall, 4th edition, 1995.
- [42] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.
- [43] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9(5), 1983.
- [44] K. Melhorn. *Data Structures and Algorithms*, volume III - Multidimensional Searching and Computational Geometry. Springer, 1984.
- [45] 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.
- [46] 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.

- [47] G. Navarro. Searching in metric spaces by spatial approximation. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, pages 141–148. IEEE CS Press, 1999.
- [48] 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.
- [49] J. Nievergelt and H. Hinterberger. The grid file: an adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, 1984.
- [50] H. Nolteimer. Voronoi trees and applications. In *Proc. International Workshop on Discrete Algorithms and Complexity*, pages 69–74, Fukuoka Recent Hotel, Fukuoka, Japan, 1989.
- [51] H. Nolteimer, K. Verbarg, and C. Zirkelbach. Monotonous Bisector* Trees – a tool for efficient partitioning of complex schenes of geometric objects. In *Data Structures and Efficient Algorithms*, LNCS 594, pages 186–203. Springer-Verlag, 1992.
- [52] S. Prabhakar, D. Agrawal, and A. El Abbadi. Efficient disk allocation for fast similarity searching. In *Proc. ACM SPAA '98*, Puerto Vallarta, Mexico, 1998.
- [53] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. ACM SIGMOD '95*, pages 71–79, 1995.
- [54] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
- [55] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.
- [56] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.
- [57] D. Sasha and T. Wang. New techniques for best-match retrieval. *ACM Trans. on Information Systems*, 8(2):140–158, 1990.
- [58] M. Shapiro. The choice of reference points in best-match file searching. *Comm. of the ACM*, 20(5):339–343, 1977.
- [59] R. Sutton and A. Barto. *Reinforcement Learning : an Introduction*. MIT Press, 1998.
- [60] J. Uhlmann. Implementing metric trees to satisfy general proximity/similarity queries. Manuscript, 1991.
- [61] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.
- [62] K. Verbarg. The C-Tree: a dynamically balanced spatial index. *Computing Suppl.*, 10:323–340, 1995.
- [63] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.
- [64] M. Waterman. *Introduction to Computational Biology*. Chapman and Hall, 1995.

- [65] 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.
- [66] A. Yao. *Computational Geometry*, chapter 7, pages 345–380. Elsevier Science, 1990. J. Van Leeuwen, editor.
- [67] 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.
- [68] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge, ALENEX'99*, Baltimore, MD, 1999.
- [69] P. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. In *Proc. 11st ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, 2000. To appear.
- [70] A. Yoshitaka and T. Ichikawa. A survey on content-based retrieval for multimedia databases. *IEEE Trans. on Knowledge and Data Engineering*, 11(1):81–93, 1999.