UNIVERSIDADE DA CORUÑA

Departamento de Computación, Universidade da Coruña

An Effective Approach for Selecting Indexing Objects in Metric Spaces

Tese Doutoral

Doutorando: Óscar Pedreira Fernández

Directoras: Nieves Rodríguez Brisaboa, Ángeles Saavedra Places

A Coruña, novembro de 2009

UNIVERSIDADE DA CORUÑA

Departamento de Computación, Universidade da Coruña

An Effective Approach for Selecting Indexing Objects in Metric Spaces

Tese Doutoral

Doutorando: Óscar Pedreira Fernández

Directoras: Nieves Rodríguez Brisaboa, Ángeles Saavedra Places

A Coruña, novembro de 2009

PhD thesis supervised by
Tese doutoral dirixida por

Nieves Rodríguez Brisaboa
Departamento de Computación
Facultade de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1243
Fax: +34 981 167160
`brisaboa@udc.es`

Ángeles Saavedra Places
Departamento de Computación
Facultade de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1249
Fax: +34 981 167160
`asplaces@udc.es`

A meus pais e a miña irmá

Acknowledgments

I would like to start this page thanking my thesis advisors, Nieves and Ángeles, for their help and dedication during this work, and for their support and faith in me from the first day. I owe a special gratitude to Nieves for opening me the doors of this group in which I feel like at home, and for guiding me throughout my way at University, which sometimes turned out to be more difficult than I could expect.

I also thank all my mates from *Laboratorio de Bases de Datos* of *Universidade da Coruña* and *Enxenio*, for making the lab a so nice place, for their friendship, and for being there for everything I could need. I would like to write the names of them all as it was usually done, but we are so many people nowadays that I would surely forget someone. So, thank you very much.

I do not want to forget about my friends, the ones from school and high school, and those I made at University, for encouraging me with this thesis, gently and patiently listening me when I tried to tell them what I was doing, and for something as important as reminding me that there is life outside the school.

My most special acknowledgment is for my family, for their unlimited support and for encouraging me to keep on getting always further. Particularly to the ones who were closest to me during this time, my sister, Natalia, my girlfriend, María, and very specially to my parents, Manuel and Amalia, for the enormous effort they did for making me possible to be writing this lines today, and for teaching me with their example how important is to do my best every day.

Agradecementos

Gustaríame comezar esta páxina agradecéndolle ás miñas directoras de tese, Nieves e Ángeles, a súa axuda e dedicación durante este traballo, e o seu apoio e confianza desde o primeiro día. A Nieves débolle un agradecemento especial por abrirme as portas deste grupo no que me atopo tan a gusto, e por guiarme neste camiño da Universidade, que, ás veces, resultou menos doado do que eu pensaba.

Grazas tamén a todos os compañeiros do Laboratorio de Base de Datos da Universidade da Coruña e de Enxenio, por facer que este sexa un lugar de traballo tan agradable, pola súa amizade, e por estar sempre aí para todo o que faga falla. Gustaríame escribir os nomes de todos eles como se facía normalmente, pero somos tantos que seguro que esquezo alguén. Así que, moitas grazas a todos.

Tampouco quero esquecer aos meus amigos de toda a vida, e os que fixen durante a carreira, polos seus ánimos, por aguantarme cando tentaba contarlles de que vai, e por algo tan importante como lembrarme que hai vida fora da facultade.

O meu agradecemento máis especial vai dedicado á miña familia, da que nunca me faltou apoio nin ánimos para seguir traballando e chegar sempre máis alá. En especial aos que estiveron sempre máis cerca de min, a miña irmá, Natalia, a miña moza, María, e, moi especialmente, a meus pais, Manuel e Amalia, polo enorme esforzo que fixestes para que eu poida estar aquí, e por ensinarme co exemplo o importante que é esforzarse cada día.

Abstract

Nowadays, there are many applications that manage very large databases of objects in which the searches do not rely on comparisons of the equality/inequality of two objects, but on how similar the objects are. This is the case of problems such as searching for similar fingerprints to one given as a query, content-based image retrieval in multimedia databases, sequence searching in genome databases, duplicate document detection in web search engines, and spam detection, just to name a few of them. *Metric spaces* provide a generic and useful framework for those problems where the exact comparison of two objects is not possible or does not make sense since it is useless. Inside this framework different methods for indexing and search have been proposed. In this work we provide an extensive description of the state of the art.

In all these domains where a similarity search is needed, a *distance function* to compute the value of the *similarity* or proximity between any two objects in the database is available. The naive implementation of similarity search would consist in sequentially scanning the entire database. However, that is not a feasible solution in practice, since the computation of the distance between two objects, that is, the use of the distance function, is in general very costly. Methods for searching in metric spaces make similarity search efficient by *indexing* the database and reducing as much as possible the number of distance computations needed for solving a query. That is, methods for searching in metric spaces avoid to compare (using the distance function) all the objects in the database with the query.

In order to index the database, all methods using the metric space approach select a set of *reference objects* from the database and store in the index the distances between those reference objects and the rest of objects in the database. The way these distances are stored and used allows us to classify existing methods in *pivot-based methods*, which store the distances from the reference objects to the rest of objects in the database, and *clustering-based methods*, which partition the space into a set of clusters around the reference objects. In both cases, the selection of good reference objects determines the effectiveness of the index for *pruning* the search space and thus reduce the cost of the search.

The main contribution of this thesis is a new method for the selection of effective reference objects. An important difference with previous methods is that the reference objects selected with our method are well distributed in the space. Our method for selecting reference objects has also other interesting properties: it automatically determines the optimal number of reference objects, it works with both discrete and continuous distances, it adapts the set of reference objects to the characteristics of the space, and the references are selected dynamically as new objects are inserted into an initially empty database with no extra cost.

Other contributions of this thesis are:

- A pivot-based method that uses our method for the selection of reference objects. This method is competitive when compared with previous methods in search cost, and is clearly better in other aspects: it is dynamic and adaptive, it determines by itself the optimal number of pivots, and it does not impose an additional cost for the selection of pivots.

- A clustering-based method that uses our method for the selection of reference objects to create an unbalanced index structure adapted to the characteristics of the space. By using an unbalanced structure to index the space, it clearly outperforms previous clustering-based methods.

- A criteria for refining the set of references by removing those that do not contribute to increase the capability of the set of references for discarding objects. The result is a smaller set of references that conserves its capacity for pruning the search space.

- Finally, we introduce the concept of *nested metric spaces* to explain certain irregularities that appear in real problems. We show how these irregularities can affect the search performance of some methods, and we show that by detecting and treating them is it possible to improve the search performance.

Therefore, this thesis contributes to the state of the art in this field with methods that combine an improvement of the search cost with other practical aspects important for their application to real problems.

Resumo

Hoxe en día existen moitas aplicacións que xestionan grandes bases de datos nas que as buscas non se basean en comparacións de igualdade ou desigualdade entre dous obxectos, senón no similares que son. É o caso de problemas como a busca de pegadas dixitais semellantes a unha dada como consulta, a recuperación de imaxes por contido en bases de datos multimedia, a busca de secuencias en bases de datos xenéticas, a detección de documentos duplicados en buscadores Web, ou a detección de spam, entre outros. Os *espazos métricos* proporcionan un marco de traballo xenérico para os problemas nos que a comparación exacta de dous obxectos non é posible ou non ten sentido porque é inútil. Dentro deste marco de traballo podemos atopar diferentes métodos de indexación e busca. Neste traballo proporcionamos unha descrición extensiva do estado da arte.

En todos estes dominios nos que se necesita facer buscas por similitude, hai unha *función de distancia* para calcular o valor da *similitude* ou proximidade entre dous obxectos. A implementación trivial da busca por similitude consistiría na comparación do obxecto de consulta con todos os obxectos da base de datos. Porén, esta non é unha solución factible na práctica, xa que a comparación de dous obxectos, isto é, o uso da función de distancia, é moi custosa en xeral. Os métodos de busca en espazos métricos fan a busca por similitude máis eficiente mediante a indexación da base de datos, reducindo así o número de comparacións necesarias para resolver unha consulta. Isto é, os métodos de busca en espazos métricos evitan a comparación (utilizando a función de distancia) de todos os obxectos da base de datos coa consulta.

Para indexar a base de datos, todos os métodos de busca en espazos métricos seleccionan un conxunto de *obxectos de referencia* da base de datos, e almacenan no índice as distancias entre estes obxectos de referencia e o resto dos obxectos da base de datos. Pola forma en que se almacenan estas distancias, podemos clasificar os métodos existentes en *baseados en pivotes*, que almacenan as distancias entre os obxectos de referencia e o resto dos obxectos da base de datos, e *baseados en clusters*, que particionan o espazo nun conxunto de clusters arredor dos obxectos de referencia. En ambos os casos, a selección de bos obxectos de referencia determina a efectividade do índice para *podar* o espazo de busca e reducir así o custo da busca.

A principal contribución desta tese é un novo método para a selección de obxectos de referencia efectivos. Unha diferenza importante con métodos previos é que os obxectos de referencia seleccionados están ben distribuídos no espazo. O noso método para a selección de obxectos de referencia ten outras propiedades interesantes: determina automaticamente o número óptimo de obxectos de referencia, traballa tanto con funcións de distancia discretas como continuas, adapta o conxunto de obxectos de referencia ás características do espazo, e permite que o conxunto de obxectos de referencia se seleccione dinamicamente a medida que se insiren novos obxectos nunha base de datos inicialmente baleira.

Outras contribucións desta tese son:

- Un método baseado en pivotes que utiliza o noso método para a selección de obxectos de referencia. É un método competitivo con propostas anteriores en custo de busca, e é claramente mellor noutros aspectos: é dinámico e adaptativo, determina por si mesmo o número óptimo de pivotes, e non impón un custo adicional para a selección de pivotes.

- Un método baseado en clusters que utiliza o noso método de selección de obxectos de referencia para crear unha estrutura de índice non balanceada e adaptada ás características do espazo. Usando unha estrutura non balanceada para indexar o espazo, este método supera claramente a outros métodos baseados en clusters.

- Un criterio para refinar o conxunto de obxectos de referencia mediante a eliminación daqueles que non contribúen a mellorar a capacidade do conxunto para descartar obxectos do espazo de busca. O resultado é un conxunto de referencias máis pequeno que conserva a capacidade para podar o espazo de busca.

- Finalmente, introducimos o concepto de *espazos métricos anidados* para explicar certas irregularidades que aparecen en bases de datos reais. Mostramos como estas irregularidades poden afectar ao rendemento de certos métodos, e mostramos como se pode mellorar o rendemento na busca ao detectalas e tratalas.

Por tanto, esta tese contribúe ao estado do arte neste campo con novos métodos que combinan a mellora no custo da busca con outros aspectos prácticos importantes de cara á súa utilización en aplicacións reais.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Data-centric systems have substantially evolved during the last years and no longer manage only numeric or alphanumeric data organized into tuples and relations in a relational database. Nowadays, data-centric systems face the problem of managing very large collections of objects of semistructured and unstructured complex data types, that is, data types which could not have a well-defined and semantically clear structure [Manning et al., 2008]. In these contexts, the classical exact-match search model is not applicable, and more general search models are needed in order to manage and exploit the data.

When working with structured data, searches usually involve comparisons of equality or inequality. For example, in a relational database, we could be interested in retrieving the records of the customers whose city is exactly equal to a city given as a query. But when working with semistructured or unstructured data, this type of search is no longer useful, or even possible. For example, if we were interested in searching for images with a similar content to another image given as a query, it would not make sense to compare pixel by pixel the contents of the images, since only the images exactly equal to the query would be in the result. The same problem arises when working with databases that contain objects as fingerprints, videos, or text documents, for example.

Similarity search can be defined as searching for objects that are similar under some criterion to another object given as a query. For any application like the ones we have already mentioned, if we can define a distance function that determines the value of the similarity between any two objects of the database, similarity search turns into a very useful search model. Similarity search is the underlying search model in applications like content-based image retrieval in multimedia databases, sequence searching in genome databases, or duplicate document detection in search engines. Efficiency is an implicit requirement in order to fulfill the search needs of a wide range of applications.

Specific solutions for efficient similarity search have been developed for particular problems. Most research on information retrieval has focused on the retrieval of similar textual documents, and different models have been developed, as the Boolean model, the vector space model [Salton and Lesk, 1968], and the probabilistic model [Robertson and Jones, 1976] (see [Baeza-Yates and Ribeiro-Neto, 1999] for a complete survey). Other research lines have addressed the development of methods for similarity search in databases of images, sound, or video. However, there are other problem domains in which similar search capabilities are needed for different data types. It is the case of computational biology, pattern recognition, or recommendation systems, and, surely, more cases will appear in the future.

The naive way of implementing similarity search would consist in sequentially comparing the query object with all the objects in the database, adding to the result those objects satisfying the similarity criterion with the query. However, in most cases the comparison of two objects using the distance function involves a high computational cost. Thus, a sequential scan of the entire database is not a feasible solution in practice, since its computational cost would be prohibitive for real applications.

Therefore, the main goal of methods for similarity search is make similarity search efficient by avoiding the sequential scan of the database, that is obtained by reducing as much as possible the number of distance computations needed to solve a query. To achieve this goal, these methods preprocess the database and build an *index* with useful information that will be used during the search to discard as many objects as possible without comparing them with the query.

Metric spaces

Although it would be possible to develop specific solutions for each problem domain, it is also possible to address all of them with the same, unified approach. That is, instead of developing a specific method for each particular problem, it is possible to develop application-independent methods for efficient similarity search that are not tied to a specific data type or problem. Since these methods must work for different data types, a formalization of the problem of similarity search is necessary, abstracting the basic properties of the data space, and making no assumptions either on the internal representation of the objects or on the definition of the distance function.

The mathematical abstraction of *metric spaces* provides a generic framework for searching in large collections of any data type for which a distance function that measures the dissimilarity or distance between two objects exists. The computation of the distance between two objects using the distance function is assumed to be expensive, since, in most real applications, comparing two objects involves a high computational cost. A *metric space* is a pair (X, d) composed of a universal set of objects X, and a *metric* defined on it, that is, a distance function $d : X \times X \to \mathbb{R}$ that holds the following properties:

- (Strictly positiveness) $d(x, y) \geq 0$ and $d(x, y) = 0 \Leftrightarrow x = y$
- (Symmetry) $d(x, y) = d(y, x)$
- (Triangle inequality) $d(x, y) \leq d(x, z) + d(z, y)$

These properties ensure the consistency of the values of distance returned by the distance function. Methods for similarity search can only use the information provided by the definition of the metric space, that is, the existence of a distance function that holds the properties of strictly positiveness, symmetry, and the triangle inequality. Methods for similarity search do not use any other information about the objects or about how the distance function is defined.

Several methods for similarity search in metric spaces have been proposed. A unified taxonomy of methods for searching in metric spaces can be found in [Chávez et al., 2001b]. At a first glance, they can be classified in pivot-based or clustering-based methods:

- *Pivot-based methods* select a subset of objects from the collection to be used as reference objects, called pivots. During the preprocessing of the collection, the distances from these pivots to the rest of the objects in the database are computed and stored in the index.

  During the search, the query object is compared with the pivots. The distances from the query object to the pivots, the precomputed distances stored in the index and the property of triangle inequality of the distance function are used to discard as many objects as possible from the result without comparing them with the query.

- *Clustering-based methods* select a set of objects from the database as reference objects, in this case, called cluster centers. The cluster centers are used to divide the space into a set of partitions or clusters. The index stores useful information about each cluster, as the covering radius, that is, the distance from the cluster center to its furthest object in the cluster.

  During the search, the query object is compared with the centers of each cluster. These distances and the information stored in the index are used to discard complete clusters from the result, so none of the objects contained in the discarded clusters has to be compared with the query object.

Pivot-based methods are significantly more efficient than clustering-based methods in what refers to the number of distance computations needed for solving a query. However, the amount of memory they require to store the distances from the pivots to the rest of objects is significantly higher than that needed by clustering-based methods. While clustering-based algorithms require usually linear space, pivot-based algorithms store a large matrix of distances.

In both cases, reducing the number of distance computations needed for solving a query is the main measure of search efficiency, since it is the main component in the overall search cost. The *complexity* of the search is given by the sum of the *internal complexity*, which is the number of comparisons of the query object with the reference objects, and the *external complexity*, which is the number of comparisons of the query object with the objects that could not be directly discarded.

Although the main goal of methods for searching in metric spaces is to reduce as much as possible the number of distance computations for solving a query, there are other aspects involved on the index performance that have also to be taken into account. First, the overall search performance depends also on the extra I/O time for loading the index from secondary memory, and the extra CPU time for processing the information it stores during the search. The memory requirements of the index are important since they can be significant, depending on the number of objects in the database.

Some methods can only work with discrete distances, while others can work with both, continuous and discrete distances. The edit distance between two strings (computed as the number of symbols to be added, removed, or replaced to transform a string into another) is an example of discrete distance function. The Euclidean distance between two vectors is an example of a continuous distance function. Methods designed for working with discrete distances can not be applied in problems where the distances between objects are continuous.

Another important aspect is whether the indexing of the collection is static or dynamic. Static methods build an index on a complete collection, and further insertions of objects are either not possible, or possible at the cost of degrading the index performance. A dynamic method should start from an initially empty database, and build the index and transform it as new objects are inserted into, or removed from the database. All these aspects are important and determine the applicability of a method to certain problem domains.

Selection of effective indexing objects

A key issue for all methods is how the reference objects, pivots or clusters, are selected. Although most existing methods select them at random, it has been shown that the specific set of objects used as references and the way they are selected, significantly affect the search performance. Although this problem is present in both pivot-based and clustering-based methods, most existing proposals refer to the selection of effective pivots.

Several techniques have been proposed for the selection of effective pivots (a detailed description of them is provided in Chapter 2). They are based on the ideas of selecting as pivots objects that are far away from each other and from the rest of objects of the database, and/or optimizing some given criterion of effectiveness.

Although these techniques significantly improve the search performance against a random pivot selection, they present some drawbacks. The most important is perhaps that they all are static: the user has to specify the number of pivots to select, and the algorithm obtains them from a complete database through a usually costly process. Therefore, the database can not be initially empty and grow later (which is its natural behavior). The user has to obtain the optimal number of pivots by trial and error from the complete collection. Further insertions of objects may be possible at the cost of degrading the search performance obtained by the index. Other important drawback is the computational cost of the selection of pivots in most of these techniques, which can be very high in some cases.

Less attention has been paid to the selection of effective cluster centers in clustering-based methods. However, it seems obvious that the number of cluster centers, their position in the space with respect to each other and their position with respect the rest of objects in the database determine the partition of the space and thus the search performance of the method during the search.

1.2 Contributions of this thesis

The main contribution of this thesis is *Sparse Spatial Selection* (SSS), a new method for the selection of effective reference objects adapted to the characteristics and distribution of the objects in the space. This method is fully dynamic, that is, the reference objects are selected as new objects are inserted into an initially empty database, and it works with continuous and with discrete distance functions. In addition, it is not necessary to specify how many reference objects are needed, since our method selects new references as they are needed when the database grows. An important difference with previous proposals is that the cost of selecting new reference objects in our method is the minimum possible.

Other contributions of this thesis are:

- We present a new pivot-based method that uses Sparse Spatial Selection (SSS) for obtaining the set of pivots of the index. Our experimental evaluation shows that the search performance it obtains is better or at least equal than the obtained by previous techniques. In addition, it is completely dynamic: the database is initially empty, and the index is built as objects are inserted in the database, adapting the structure and information of the index to the content of the database in each moment.

- We address the problem of the selection of effective cluster centers, and propose a new recursive tree-like clustering-based method, *Sparse Spatial Selection Tree* (SSSTree). The cluster centers are selected with Sparse Spatial Selection. This gives as result a tree structure in which the information and resources of the index are adapted to the characteristics and complexity of the space. An important difference with previous proposals is that the tree is not necessarily balanced, since its structure is adapted to the characteristics and complexity of the space.

- We introduce *Non-Redundant Sparse Spatial Selection* (NR-SSS), a new criterion for detecting and replacing *redundant* reference objects, that is, references that do not contribute to improve the effectiveness of the overall set of reference objects as a whole. This method obtains a smaller set of pivots than the original SSS, but maintains the effectiveness of SSS, therefore reducing the search complexity and the space requirements of the index.

- We introduce the concept of nested metric spaces as an explanation of certain irregularities of the space that can appear in real problems. We analyze how those nested spaces can affect the performance of some techniques, and propose a method for detecting these situations and adapt the indexing to those complex spaces.

1.3 Structure of this work

The rest of the thesis is organized as follows:

- Chapter 2 introduces the basic concepts of indexing and searching in metric spaces and sets thus the basis for this work. It also presents the more relevant proposals of the state of art to the date, with a special focus in the previous work on algorithms for pivot selection.

- Chapter 3 presents *Sparse Spatial Selection* (SSS), our main contribution, and a new pivot-based method that uses this method for the selection of the pivots used in the index.

- In Chapter 4 we address the problem of the selection of effective reference objects to clustering-based methods, and propose Sparse Spatial Selection Tree *(SSSTree)*, a new recursive and unbalanced tree-like structure that improves the results of previous techniques by taking advantage of the good distribution of the reference objects provided by SSS.

- Chapter 5 presents *Non-Redundant Sparse Spatial Selection* (NR-SSS). It is a modification of the original SSS method. The idea is to detect and remove reference objects that do not contribute to increasing the capacity of the index for discarding objects. In this chapter we describe the criteria for evaluating the contribution of each reference object and the policy for their removal or replacement.

- Chapter 6 introduces the concept of *nested metric spaces*, showing how this kind of space affects the search performance of some methods, and presents a new method for detecting and dealing with these irregularities of the space.
- Chapter 7 presents the conclusions of this work, and lines of future work.
- Appendix A lists the publications and other research results derived from this thesis, and the works published by other researchers that take our proposals into consideration and quote our work.
- Appendix B summarizes the notation used throughout the thesis.
- Appendix C describes with detail the common test environment used for all the experiments described in this work.

Chapter 2

Indexing and searching in metric spaces

2.1 Overview of the chapter

In this chapter we introduce the background and basic concepts related with searching in metric spaces. We start by presenting the basic concepts and notation of metric spaces and how they are used to formalize the problem of similarity search. We also present the most important similarity queries, and typical metrics that can be used in a wide range of problems.

Searching in metric spaces could be trivially implemented as a sequential scan of the entire database. However, this is not feasible in practice, due to the high computational cost of computing the distance between two objects. Therefore, the main goal of methods for searching in metric spaces is to avoid the comparison of the query object with the whole content of the database. In this chapter we provide a description of the most important state-of-art methods, and we study the policies they apply in order to avoid a sequential scan of the database. We also review the concept of intrinsic dimensionality and how it can affect the performance of search methods in certain metric spaces.

All methods for searching in metric spaces try to improve the efficiency of the search by building an index on the collection, based on some objects selected as indexing objects, also called reference objects. We end the chapter with an analysis of the how the selection of indexing objects affects the search performance and other parameters of the indexes. We provide a detailed description of the previous work on this issue, since our proposals are compared to them in next chapters.

2.2 Basic concepts on metric spaces

Metric spaces are one of the mathematical tools that can be used to formalize the problem of similarity search. A *metric space* is a pair (X, d) composed of a universe of objects X and a metric d. The *universe* is the universal set of objects of a specific type, and a *metric* is a distance function $d : X \times X \longrightarrow \mathbb{R}^+$ that satisfies the following properties:

- (Strictly positiveness) $d(x, y) \geq 0$, and $d(x, y) = 0 \Leftrightarrow x = y$
- (Symmetry) $d(x, y) = d(y, x)$
- (Triangle inequality) $d(x, y) \leq d(x, z) + d(z, y)$

These properties ensure the consistency of the metric. For every $x, y \in X$, the number $d(x, y)$ is called the *distance* between the objects x and y with respect to the metric d. The distance $d(x, y)$ is a measure of the difference or dissimilarity between x and y with respect to the metric d. The more similar the objects are, the smaller the distance between them. The definition and meaning of the distance function depend on the application domain and on the goals of the application. Note that although we use the term distance, it is not necessarily a spatial distance (actually, it is not in most cases). For example, considering that X is a set of words in natural language, and that d is the edit distance (computed as the minimum number of characters to be inserted, replaced, or removed in a word to transform it into the other), the distance between the words "tip" and "trip" is 1, because the difference between them is given by one character. In this example, it is clear that the distance is not spatial.

The *database* or *collection* of objects is represented by a finite subset $U \subseteq X$ of size $n = |U|$. A *query* is expressed as a query object $q \in U$, and a constraint about its similarity to the objects in the database. The *result set* is the subset of objects in U that, using the distance function, satisfy the constraint of similarity to the query object. A query could consist, for example, in obtaining all the objects up to a certain distance to the query object q.

A set of all the words of a given language and the edit distance form a metric space. Dictionaries are possible databases of objects. In this example, we could be interested into retrieving all the words up to a given distance of a specific word given as a query for spelling correction.

The internal details of the objects and the distance used to compare them are abstracted by the formalization of the problem. No matter if we are searching in a database of biological sequences, or comparing words for spelling correction in a text editor, any method for efficiently search in metric spaces could be applied.

There are other alternatives to formalize similarity search. Vector spaces are one of them. In some cases, complex objects can be represented as vectors of features, where each component of the vector is a numeric representation of a feature of interest of the object. The distance between two objects is computed as the Euclidean distance between their feature vectors. For example, images can be represented by feature vectors obtained with computer vision algorithms.

When similarity search is formalized using vector spaces, methods for similarity search can take advantage of the properties of the Euclidean geometry. But there are many data types that can not be represented by feature vectors. Since metric spaces are much more general than vector spaces (actually, vector spaces are a particular case of metric spaces), they can be applied to a wider range of problems. As we have already explained, methods for searching in metric spaces only know the existence of a distance function that holds the properties of strictly positiveness, symmetry, and the triangle inequality.

2.2.1 Metrics

Several metrics could be defined for objects of the same data type. A metric that is useful for a certain domain can be useless in another. In some cases it is difficult to find distance functions that hold the three necessary properties to be a metric. When the distance function does not hold the symmetry property it is called a *quasi-metric*. In the case that the function does not hold the property of strictly positiveness it is called a *pseudo-metric*. In both cases the distance function can be transformed in other function that holds the metric properties.

This section describes typical metrics that can be applied in many application domains. As we will see, some of them are discrete and other continuous. This distinction is important since some algorithms can only work with discrete metrics, which restricts the problems in which they can be applied.

Metrics for vectors

In some applications the objects are represented by feature vectors, where each component represents a certain feature of interest of the object. It is the case of applications for content-based image retrieval, where each image is represented by features as the colors, textures, or the presence of shapes of interest. In this section we describe several metrics that can be used when working with vector spaces.

The family of *Minkowski distances* is a parametric set of distance functions for vector spaces. They are defined as:

$$L_p(\vec{x},\vec{y}) = \left(\sum\nolimits_{1\leq i\leq k}|x_i - y_i|^p\right)^{\frac{1}{p}}$$

where k is the dimensionality of the vector space, and x_i and y_i are the components of the vectors $\vec{x} \in \mathbb{R}^k$ and $\vec{y} \in \mathbb{R}^k$respectively.

Each value of the parameter p between 0 and ∞ gives as a result a different distance function, and all of them satisfy the conditions for being metrics. L_1 is the *Manhattan distance*. For instance, when $k = 2$, L_1 measures the distance between two points as the number of vertical and horizontal movements from one point to another. L_2 is the Euclidean distance, the usual choice.

An interesting metric is $L_\infty = max_{1 \leq i \leq k}|x_i - y_i|$, called the *maximum distance*, since it measures the maximum pairwise difference between the coordinates of two vectors.

Minkowski distances assume that the components of vectors are independent, that is, that each component of a vector is only compared with its corresponding component of the other vector. However, in some cases there are correlations between different coordinates, and the comparison of the objects should take them into account. *Quadratic form distances* are defined as [Hafner et al., 1995]:

$$d_M(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^T \cdot M \cdot (\vec{x} - \vec{y})}$$

where $\vec{x} \in \mathbb{R}^k$ and $\vec{y} \in \mathbb{R}^k$ are two vectors of dimension k, and M is a $k \times k$ matrix which components take values between 0 and 1. The matrix M represents the correlations between the components of each vector. The component $M(i, j)$ represents the weight of the correlation between the component x_i of $\vec{x}$ and the component y_j of $\vec{y}$.

Quadratic form distances are used, for example, for the comparison of images represented by feature vectors in which a certain feature has a relation with another. If some features correspond to colors, there will be relationships between similar colors and, therefore, different components of the feature vector. The weight of the relations between colors (or any other feature) can be reflected in the matrix.

All the metrics of the Minkowski and quadratic form families are continuous.

Metrics for strings

The *edit distance* or *Levenshtein distance* [Levenshtein, 1965] between two strings of symbols from a given alphabet, is defined as the minimum number of symbols to insert, replace, or remove to transform one string into the other. Therefore, it is a discrete metric. The edit distance can be used when working with dictionaries of words for spelling correction (as it is the case of many text editors and search engines). Sequences of proteins or DNA are also strings of symbols that can be compared using the edit distance.

There are several variations on this definition. A direct generalization consists in assigning different weights to each edit operation (addition, removal, or replacement of a symbol), depending on the cost of each of them in a particular application. In some cases not all the edit operations make sense. The *Hamming distance* allows only replacements of symbols. In the same way, the *Episode distance* allows removals of symbols. In these cases the distance between two strings can be ∞. The *contextual normalized edit distance* [de la Higuera and Micó, 2008] takes into account the length of the strings to normalize the final result.

Metrics for trees

The *tree edit distance* between two labeled trees is the minimum number of edit operations necessary for transforming a tree into the other. In this case the edit operations consist in adding, removing, or relabeling nodes [Bille, 2005]. This definition can be generalized as in the case of the edit distance for strings, allowing only some operations or assigning different costs to them. For example, the cost assigned to the insertion of a node can be higher for deeper levels of the tree, since it involves a high computational cost that may be significant for the application.

The edit distance for trees has been used for comparing XML documents [Guha et al., 2002], represented as trees. As in the case of the edit distance for strings, the edit distance for trees is a discrete metric.

Metrics for sets

Some applications need to compare objects that are represented as sets that contain or not certain features or elements of interest. For example, the description of a user profile in a virtual shop could be represented as the set of all the products the user has viewed or bought. The user profiles could be compared with sets of products for obtaining the most promising users for a direct mail campaign. User profiles in social networks can be also viewed as sets.

Given two arbitrary sets A and B, the *Jaccard coefficient* is defined as:

$$Jaccard(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

That is, the Jaccard coefficient measures the dissimilarity between two sets as the ratio between the cardinalities of the intersection and the union of the compared sets. The higher the number of objects common to both sets is, the smaller the distance between them. The distance between two sets according to the Jaccard coefficient is measured as the percentage of objects they have in common.

However, in some applications it is not possible to know if two elements of the sets A and B are exactly equal, so it is not always applicable. For example, if we compare two sets of images, it does not make sense to compare if two images of those sets are exactly equal.

The Hausdorff distance considers the distances from objects in a set to objects in the other set, and therefore obtains a more refined distance than the Jaccard coefficient, that considers only the exact equality between the elements of the two sets compared.

Given a metric space (X, d) and two sets $A, B \subseteq X$, the *Hausdorff distance* between the sets A and B is defined as:

$$\textit{Hausdorff}\,(A, B) = max\{sup_{x \in A} d(x, B),\ sup_{y \in B} d(A, y)\}$$

where $d(x, B)$ is the distance from the object x to the set B (that is, the minimum distance from x to any element of B), and $d(A, y)$ is the distance from the object y to the set A (that is, the minimum distance from y to any element of A).

Complexity

Table 2.1 shows the computational complexity of the metrics we have described in this section, in terms of the size of he objects compared. As we can see in the table, most of them have a high computational cost. This is specially important when they are used to compare very large objects. For example, comparing two words in natural language with the edit distance may not be very costly, but the edit distance is also used with sequences of proteins or DNA with hundreds of thousands of symbols. In this case, the comparison of two objects involves a significant cost.

Distance function	**Complexity**
Minkowski distances	$O(n)$
Quadratic form distances	$O(n^2 + n)$
Edit distance	$O(n \times m)$
Tree edit distance	$O(n^4)$
Hausdorff distance	$O(n \times m)$

Table 2.1: Complexity of metrics for vectors, strings, trees and sets.

Note that the most costly operation when processing a similarity query is the evaluation of the distance between two objects. That is the reason why the main goal of methods for indexing and searching in metric spaces is to reduce as much as possible the number of distance computations.

2.2.2 Similarity queries

Several interesting types of query have been proposed for similarity search. In this section we describe the most important types of query.

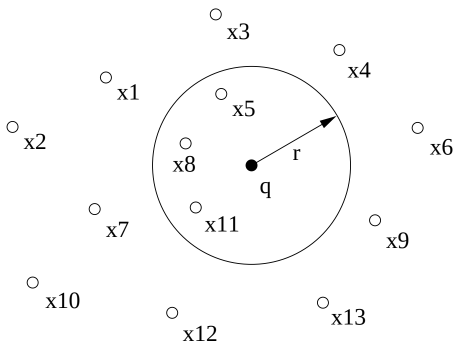


Figure 2.1: Example of range search in $\mathbb{R}^2$.

Range search

The most important type of query is *range search*, since it is the more general and the implementation of other types of queries relies on it. Given a query object $q \in X$ and a search radius $r \in \mathbb{R}^+$ (also called *range*), a range search retrieves all the objects in the database up to a distance r from q:

$$Range(q, r) = \{x \in U,\ d(x, q) \leq r\}$$

Note that q is an object in X, but not necessarily an object in the database U. Retrieving the words up to a certain distance of another one is an example of range search. As we will see later, methods for searching in metric spaces discard objects by estimating if they are out of the region in the space defined by the query object and the search radius, avoiding the direct comparison with the query. Therefore, the higher the search radius is, the more difficult the search.

Figure 2.1 shows an example of range query in a two dimensional scenario. In this example, the query object q and the radius r define a two-dimensional query ball. The result set is the subset of objects in the database that are contained in this ball, that is, the region of the space around q encircling those objects that answer the query. In this case, x_5, x_8 and x_{11} are in the result set.

Nearest neighbors search

In some domains range search is not adequate since it can be difficult to set an appropriate value of the search radius. In a collection of words we know the meaning of retrieving the words at distance two of another one: the words with at most two spelling errors. But in the case of a collection of images compared with the Euclidean distance, the radius does not have a clear meaning. In some cases the problem is

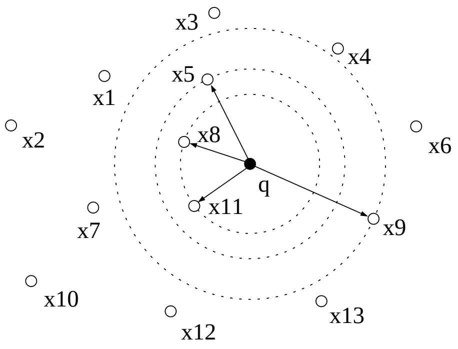


Figure 2.2: Example of nearest neighbors search in $\mathbb{R}^2$.

that the size of the result set can be very variable for a given radius depending on the query object. This is very common when working with discrete distance functions. Some applications just need a fixed number of results and searching for them can be even more efficient than a range search. *k-nearest neighbor search* (kNN-search) retrieves the k closest objects to the query object q:

$$kNN(q) = \{A \subseteq U,\ |A| = k,\ \forall x \in A,\ y \in U - A,\ d(x,q) \leq d(y,q)\}$$

A particular case of kNN-search is 1NN-search, usually called *nearest neighbor search*, that retrieves just the closest object to the query. kNN-search can be implemented upon range search by using dynamic search radius. That is, the kNN-search is answered by carrying out as many range searches as needed, starting with a small search radius and increasing it until the result of the range search contains the k closest objects for q [Chávez et al., 2001b]. However, there are also specific algorithms for this operation that can achieve better performance, as [Clarkson, 1999].

Figure 2.2 shows an example of 4NN-search, the four nearest neighbors search in a two dimensional scenario. In this case the result set contains the objects x_8, x_{11}, x_5 and x_9, the four most similar objects to the query, despite of the values of the distances from them to the query.

In some problems the search must proceed in the opposite direction. That is, given a query object q, find the objects for which q is among their nearest neighbors. This operation is called *reverse nearest neighbor search.* Decision support systems can use this operation to detect influence sets around an object of interest. Several variations of these types of query exist.

Similarity join

Another interesting type of query in metric spaces is similarity join. In relational databases, a join query returns the objects in two or more relations that are linked through common values in equal domain attributes. This idea can be extrapolated to the case of metric spaces. Given a metric space (X, d) and two databases $U \subseteq X$ and $V \subseteq X$, similarity join retrieves all the pairs $(u, v) \in U \times V$ for which u and v are up to a distance r:

$$SimJoin(U, V, r) = \{(u, v) \in U \times V,\ d(u, v) \leq r\}$$

In [Bayardo et al., 2007] the search of all similar pairs is used to find all the similar user profiles in a social network. Similarity join could also be used in databases of objects from different universes if a metric to compare them is available.

The join operation in relational databases is a particular case of similarity join. Given two relations R and S, with attributes U and V respectively, such that $dom(U) = dom(V)$, the similarity join $Sim(U, V, 0)$ searches all the pairs of objects $(x, y) \in U \times V$ where $d(x, y) = 0$, that is, where x and y are equals.

2.3 Search in metric spaces

The naive implementation of similarity search consists in performing a sequential comparison of the query object with all the objects in the database. However, this implementation is not feasible in practice. Due to the high computational cost of evaluating the distance between two objects using the distance function, and the typically very large size of the databases, a sequential scan of the database would result in a prohibitive cost for real applications.

The goal of methods for searching in metric spaces is to solve the queries without comparing the query object with all the objects in the database. That is, the goal is to solve the queries by comparing the query object only with a small fraction of the objects in the database. To achieve this goal, methods for searching in metric spaces build indexes on the database to avoid the comparison of the query object with all the objects in the database.

An *index* is a data structure that maintains useful information about the collection that will be used during the search to discard as many objects as possible from the result without comparing them with the query object. It is important to note that when we use the term *method for searching in metric spaces*, we refer to the set made up by the index data structure and the algorithms to build the index, to modify the index when an object is inserted into or removed from the database, and to process the index during the search to prune the search space.

Although reducing the number of comparisons of objects needed to solve a query is the main goal of methods for searching in metric spaces, there are other aspects that affect the overall search efficiency obtained with a method:

- Processing the information stored in the index to prune the search space during the search involves an extra CPU time. Although it is usually supposed to be less costly than the comparison of two objects, it affects the overall search performance.
- The space requirements of the index have also to be taken into account. If the index does not fit in main memory, the possibility of efficiently store the index into secondary memory and the number of I/O operations needed to load it are also important.
- Some methods can only work with discrete distance functions, while others can work with continuous distances too. This constraint can restrict the applicability of the method in some problems.
- Some methods are static, that is, the index is built on the complete collection of objects and the index does not admit further insertions or deletions of objects in the database. Others admit insertions or deletions of objects although the search performance can degrade if the number of changes is not small. Dynamic methods build and adapt the index as objects are inserted into or removed from an initially empty database.

Many methods for searching in metric spaces have been proposed to the date. Some of these methods were developed as solutions for specific problems in different areas, as data engineering, statistics, pattern recognition or computational biology, for example. This situation led to some ideas and concepts being reinvented. Some authors [Chávez et al., 2001b] proposed a unified taxonomy of methods for searching in metric spaces. That taxonomy groups all these proposals under a common, application-independent framework. At a first level, the taxonomy distinguishes between *pivot-based* and *clustering-based* methods, also known as *Voronoi-type* methods.

In this section we describe in general terms how methods of each group work. In section 2.5 we review the most important methods of the state of the art.

2.3.1 Pivot-based methods

Pivot-based methods select a small subset of objects from the database to be used as reference objects, called *pivots*, $P = \{p_1, \ldots, p_m\}$, $p_i \in U$. The distances $d(x, p_i)$, $1 \leq i \leq m$, from the objects in the database $x_i \in U$ to the pivots $p_i \in P$ are computed and stored in the index. The distances $d(x_i, p_j)$ are used during the search to prune the search space by discarding as many objects as possible from the result without comparing them with the query.

When given a range query (q, r), the query object is compared with the pivots to obtain the distances $d(q, p_j)$, $1 \leq j \leq m$. Let $x_i \in U$ be an object in the database, and $p_j \in P$ be a pivot. Applying the triangle inequality, we know that:

$$d(x_i, p_j) \leq d(x_i, q) + d(q, p_j)$$

Note that we know the value of the distance $d(x_i, p_j)$, that is stored in the index, and the value of the distance $d(q, p_j)$, that is obtained when the query object is compared with the pivots. By rearranging the triangle inequality, we obtain a *lower bound* on the distance from the object x_i to the query object q as:

$$d(x_i, q) \geq |d(x_i, p_j) - d(q, p_j)|$$

If the lower bound is greater than the search range, that is, if:

$$|d(x_i, p_j) - d(q, p_j)| > r$$

then $d(x_i, q) > r$, and the object x_i is discarded from the result without comparing it with the query.

Note that for each object in the database we can obtain as many lower bounds on its distance to the query objects as pivots used in the index. If an object is not discarded by a particular pivot, it may be discarded by another one.

Figure 2.3 shows how pivot-based methods proceed to prune the search space. In this example, the database contains ten points in a two-dimensional space. Four of them are selected as pivots, $P = \{p_1, \ldots, p_4\}$. The algorithm for building the index computes and stores in a table (for example) the distances from the pivots to the rest of objects in the database $U - P = \{x_1, \ldots, x_6\}$. When given a range query (q, r), the query object is compared with the four pivots. The distances from the pivots to the rest of objects and the distances from the query to the pivots are used during the search to discard as many objects as possible from the result.

Figures 2.4 and Figure 2.5 show how the triangle inequality and the distances stored in the index are used to discard objects from the result without comparing them with the query. The value of the distance $d(x_i, p_j)$ has been computed during the indexing of the collection and it is available in the index. The value of the distance $d(q, p_j)$ has been obtained when the query object was compared with the pivots. In the example shown in the figure, the lower bound obtained as $d(q, x_i) \geq |d(x_i, p_j) - d(q, p_j)| > r$ is greater than the search radius r, and therefore, the object x_i can be directly discarded from the result without being compared with the query object q.

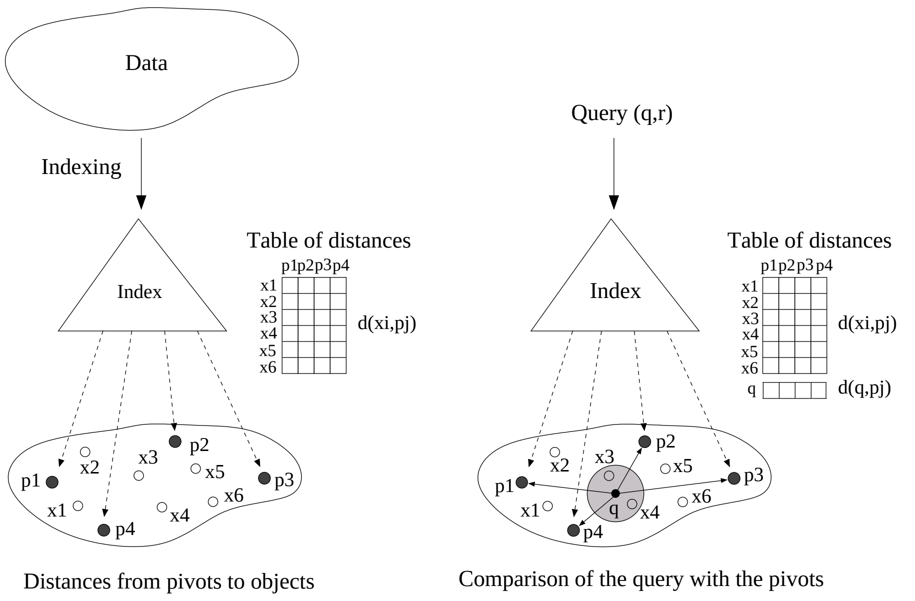


Figure 2.3: Indexing and searching with pivot-based indexes.

Since we are working with values of positive distances, it does not matter which of them is greater than the other. That is the reason for introducing the absolute value of the difference between the distances in the equation of the lower bound. In Figure 2.4 the distance $d(q, p_j)$ is greater than the distance $d(x_i, p_j)$. In Figure 2.5 the situation is the opposite. However, the value of the lower bound is the same in both cases.

Obviously, if the object is in the result set, it is not possible to discard it applying this criterion, that is, there are no false negatives. But, note that even if the object x is not in the result set, it may not be possible to discard it without comparing it with the query. Figure 2.6 shows two examples. In the first case the lower bound is greater than the search radius and x can be discarded. In the second case, the lower bound is not greater than r and the object can not be discarded. In this second case, although the object x is not in the result set, we can not discard it since the lower bound we obtain for it does not guarantee us that this object is not in the result set. Therefore, it is necessary to compare the objects that could not be discarded with the query in order to avoid false positives.

The condition for discarding an object can be expressed in other form. Given an object $x \in U$, a pivot $p \in U$, and a query (q, r), x can be discarded from the result set if $d(x, p) \notin [d(p, q) - r, d(p, q) + r]$. Figure 2.7 shows graphically this idea. The dashed circles delimit the area of the objects that can not be discarded.

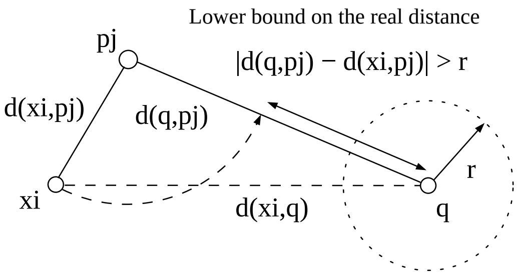


Figure 2.4: Triangle inequality in pivot-based indexes $(d(p,q) > d(p,x))$.

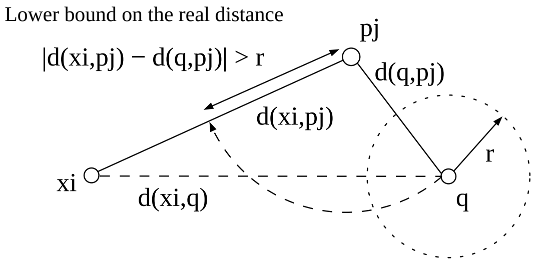
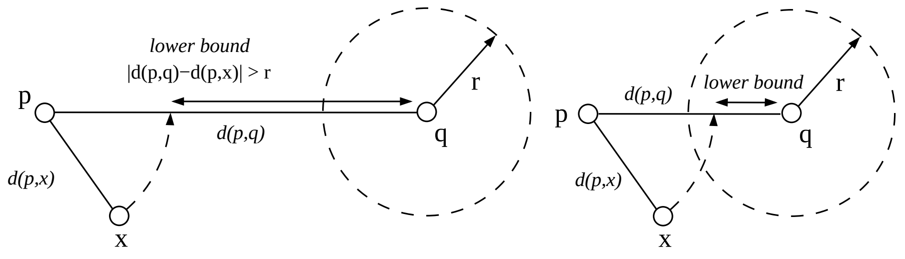


Figure 2.5: Triangle inequality in pivot-based indexes $(d(p,q) < d(p,x))$.

In these examples we showed how to obtain a lower bound of $d(x,q)$ using only one pivot. However, the indexes use a set of several pivots with which they obtain different lower bounds for $d(x,q)$. If a pivot is not able to discard the object, perhaps it can be discarded by another pivot. As we can see in figure 2.7, the list of candidate objects is given by the intersection of the lists of candidates of each pivot. The search space is significantly pruned with each new pivot. The more the pivots, the smaller the candidate list will be.

Existing pivot-based methods differ on how pivots are selected, the information they store, and the data structures they use to store it. Some methods store the distances from objects to pivots in tree-like structures, as FQT [Baeza-Yates et al., 1994], FHQT [Baeza-Yates, 1997], and FMVPT [Chávez et al., 2001b]. Other methods store the distances in tables or other array structures, as AESA [Vidal, 1986], LAESA [Micó et al., 1994], FMVPA [Chávez et al., 2001b], and FHQA [Chávez et al., 2001a]. Some methods follow an approach know as *scope coarsening*, and do not store all the distances from objects to pivots (thus reducing the scope of action of each pivot), as BKT [Burkhard and Keller, 1973], VPT [Uhlmann, 1991], and MVPT [Brin, 1995, Bozkaya and Ozsoyoglu, 1997].



(a) Discarded. (b) Not discarded.

Figure 2.6: Avoiding distance computations with pivot filtering.

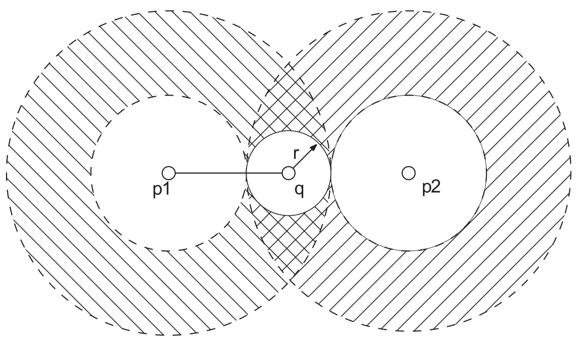


Figure 2.7: Example of pivot filtering with two pivots.

2.3.2 Clustering-based methods

Clustering-based methods follow a different approach. In this case, the space is decomposed in a Voronoi partition. That is, the space is divided in a set of m clusters C_i such that $C_i \cap C_j = \emptyset$, $1 \leq i, j \leq m$, and $\bigcup C_i = U$. To create such partition, a set of objects are $c_1, \ldots, c_m$ taken from the database as reference objects, in this case, cluster centers. Each cluster is defined as the subset of objects closest to its center than to any other center, that is:

$$C_i = \{x \in U,\ d(x, c_i) \leq d(x, c_j),\ 1 \leq j \leq m\}$$

While pivot-based methods try to obtain good lower bounds of the distance from the object to the query for each individual object in the database, clustering-based methods try to obtain good lower bounds for groups of objects.

Although the information stored in the index for each cluster varies for the different clustering-based methods, all methods store at least a reference to the

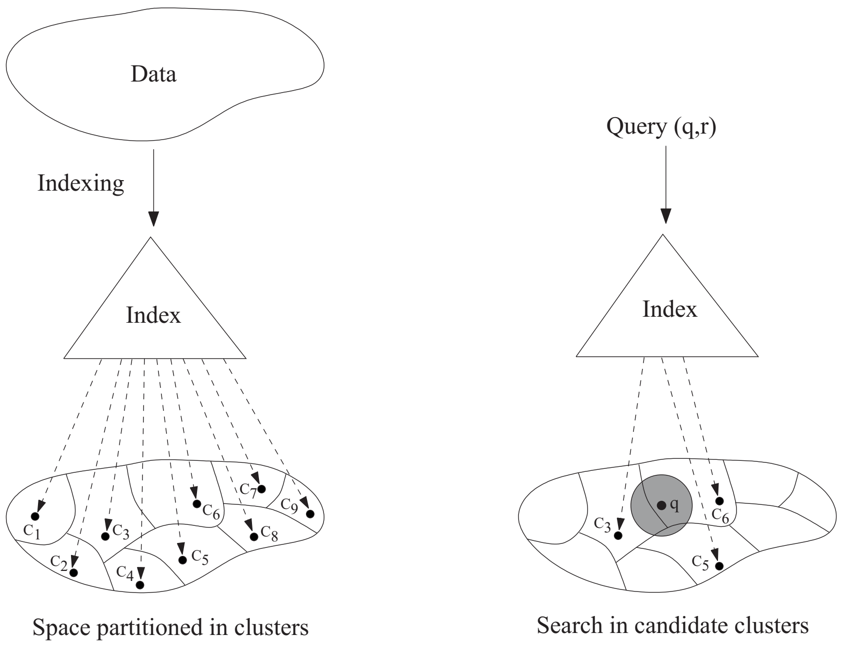


Figure 2.8: Indexing and searching with clustering-based methods.

center of the cluster, and the covering radius, which is the distance from the center of the cluster to its furthest object in the cluster, that is:

$$r_{c_i} = max\{d(x, c_i),\ x \in C_i\}$$

The center and the covering radius define a ball in the space. The *ball* associated to the cluster C_i is the set of all points of X at distance less than r_{c_i} from c_i (note that we say objects in X, not objects in U), that is:

$$(c_i, r_{c_i}) = \{x \in X,\ d(x, c_i) \leq r_{c_i}\}$$

It is important to note the difference between the cluster, and its associated ball. The cluster is a set of objects. The ball is a region in the space. For example, in a two-dimensional vector space, a cluster is a set of (x, y) points in the space, while the ball is a circumference with center c_i and radius r_{c_i}. The intersection of two clusters is always empty, but their associated balls can intersect.

When given a range query (q, r), the query object is compared with the cluster centers. These distances and the covering radius of each cluster permit us to

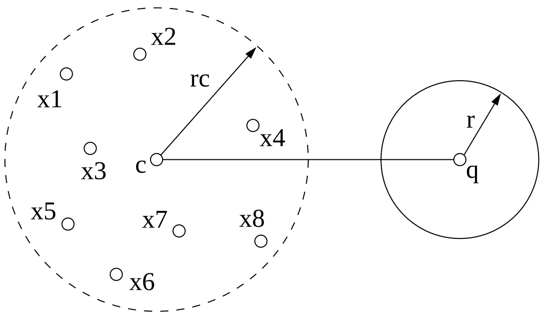


Figure 2.9: Use of ball partitioning in clustering-based methods.

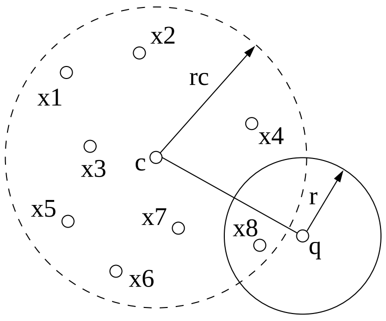


Figure 2.10: Use of ball partitioning in clustering-based methods.

determine if the intersection between the ball corresponding to the cluster and the ball corresponding to the query is empty. The cluster C_i is discarded from the result set if:

$$d(q, c_i) \geq r_{c_i} + r$$

where r_{c_i} is the covering radius of C_i. If this lower bound is greater than the search radius r, no object of the cluster is in the result set and thus the cluster can be pruned from the search. In the example shown in Figure 2.8, all the objects on clusters C_1, C_2, C_4, C_7, C_8, and C_9 are directly discarded from the result without being compared with the query.

Figures 2.9 and 2.10 show two examples of how the policy for discarding clusters works. In the example of Figure 2.9, the lower bound is greater than the search radius and the complete cluster can be discarded. As we can see in the figure, the query ball does not intersect with the ball corresponding to the cluster. However, in the case of the example shown in Figure 2.10, the query ball intersects with the ball corresponding to the cluster and it is not possible to discard it, nor any of the objects in it. Note that in the case of (q_2, r), there is no necessarily any object of the cluster in the result set. The ball corresponding to the cluster is used as an approximation of the real contents of the cluster. The ball is used to try to discard it. Since it is an approximation, the fact that both the cluster and query balls intersect does not guarantee that any object of the cluster is in the result set.

The capacity of pruning the search space depends on the characteristics of the partition. On the one hand, if the clusters are too large, it is more difficult to discard them from the result, since it is more probable that the query ball will intersect them. On the other hand, having a lot of small clusters improves the pruning of the search space but increases the internal complexity of the search, since the query object has to be compared with a larger number of cluster centers.

Most clustering-based methods create a multilevel recursive partition of the space for this reason. In a first level, the space is decomposed in few, large clusters. Each of them is then recursively decomposed in subclusters that will also be decomposed in smaller clusters until the resulting clusters are small enough. During the search, the larger clusters are used first, and the recursive decompositions of each of them is used if they can not be discarded from the result.

The resulting partition of the space depends directly on the set of objects chosen as cluster centers and the criteria for creating the clusters. With this approach, two clusters can have a different number of elements. Some methods use different criteria for obtaining a balanced partition of the space, although it has been shown that it is not the best choice in terms of search performance [Chávez and Navarro, 2005].

Existing clustering-based methods differ in how they choose the cluster centers, how they partition the space, and in the information they store in the index for each cluster. Clustering-based methods can be classified in those based on the use of hyperplanes, as GHT [Uhlmann, 1991], and those based on the use of the covering radius (distance from the center to its furthest object in the cluster) of each cluster, as BST [Kalantari and McDonald, 1983], VT [Dehne and Noltemeier, 1987], M-Tree [Ciaccia et al., 1997] and List of Clusters [Chávez and Navarro, 2005].

Comparison

There are important differences between pivot and clustering based methods. On the one hand, pivot-based methods significantly outperform clustering-based methods in what refers to search cost (that is, the number of distance computations needed for solving a query).

On the other hand, clustering-based methods use linear space for storing the index, while in the case of pivot-based methods the space requirements depend directly on the number of pivots, and, therefore, on the distance computations they store in the index.

2.3.3 Search complexity

As we have already explained, the search performance of a method for searching in metric spaces is measured as the average number of distance computations needed for solving a query. The total number of distance computations carried out during a search is given by the sum of the internal and external complexities:

- *Internal complexity*: is the number of comparisons for comparing the query object with the pivots or cluster centers used by the index. It is called internal since this number of comparisons depends on the resources and internal structure of the index.
- *External complexity*: is the number of comparisons needed for comparing the query object with the objects that could not be discarded by the index during the search. As we have already explained, it is necessary to compare them with the query in order to avoid false positives in the final result.

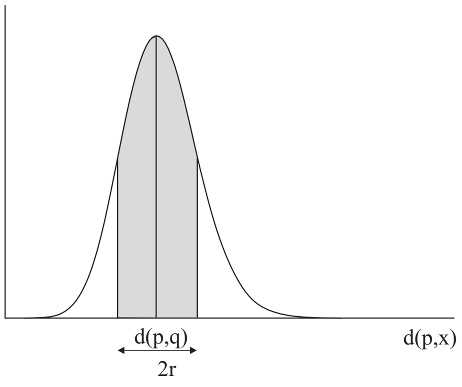
In most methods there is a trade-off between both components of the search cost. Reducing the external complexity by using more pivots or cluster centers increases the internal complexity, and reducing the internal complexity of the method increases the external complexity.

2.4 Intrinsic dimensionality

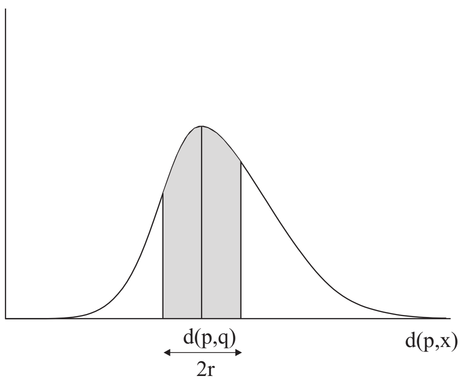
The intrinsic dimensionality of a metric space is an interesting concept that has an important influence on the search performance obtained with methods for searching in metric spaces. In a vector space, the dimension of the space is the number of components of each vector. In general, when vector spaces are indexed with multidimensional access methods, the higher the dimensionality, the more difficult the search.

Though general metric spaces do not have an explicit dimensionality, as vector spaces have, we can talk about their intrinsic dimensionality following the same idea. The *intrinsic dimensionality* of a metric space is a measure of its complexity. The higher the dimensionality, the more difficult the search. That is, discarding objects from the result without comparing them with the query is more difficult in spaces with high intrinsic dimensionality.

This concept has been studied in depth in [Chávez et al., 2001a], that proposes a way of estimating the intrinsic dimensionality of a metric space, and that permits to have an idea of its complexity and the efficiency that could be achieved with indexing algorithms. To obtain this estimation, [Chávez et al., 2001a] uses the histogram of all the distances between objects in the metric space. The idea is that the more concentrated the histogram, the more difficult the search. Therefore, the intrinsic



(a) Concentrated histogram of distances



(b) Flat histogram of distances

Figure 2.11: Spaces with low (left) and high (right) dimensionality.

dimensionality of the metric space will be higher when the mean μ of the histogram is higher and the variance σ^2 lower. Figure 2.11 illustrates this idea.

Given a random range query (q, r), the distances between the query object q and the pivots are distributed according to the histogram of distances. The policy for discarding an object using pivots determines that any object $x \in U$ can be discarded if:

$$d(p, x) \notin [d(p, q) - r, d(p, q) + r]$$

Thus, the shadowed areas in Figure 2.11 represent the objects in the space that the method can not discard. The more concentrated the histogram around the mean, the less the objects that can be directly discarded.

Based on this idea, [Chávez et al., 2001a] proposes a formula for estimating the intrinsic dimensionality of a metric space (we omit the analytical steps that led to this formula, available in [Chávez et al., 2001a]):

$$\rho = \frac{\mu^2}{2\sigma^2}$$

The intrinsic dimensionality of a space is therefore given by the distribution and topology of the objects in the space. Even among collections of objects of the same nature, one of the collections can be more difficult than the other in terms of the number of distance computations that can be avoided for solving the query.

2.5 State of the art

2.5.1 Pivot-based methods

As we explained in Section 2.3, pivot-based algorithms use a subset of objects of the collection as reference objects, called pivots. The indexes maintain distances from these pivots to the objects of the collection. During the search, the query is compared with the pivots, and these distances are used with the index and the triangle inequality to discard objects without comparing them with the query (see Figure 2.3), as we have already explained.

In this section we review the most important proposals in pivot-based methods. As we will see, the existing methods differ mainly on how they choose the pivots in the space, the information they store in the index, and the data structures they use to store that information.

Burkhard-Keller Tree (BKT)

Burkhard-Keller Tree (BKT) [Burkhard and Keller, 1973] was probably the first proposal to the problem of similarity search in metric spaces. It works only with discrete metrics. The information about the space is stored in a tree-like structure, recursively built as follows: an object $p \in U$ is selected at random to be the first pivot, and the root of the tree. For all the values $i > 0$ that can be returned by the distance function, the set U_i is defined as:

$$U_i = \{x \in U,\ d(x,p) = i\}$$

That is, U_i is the set of objects placed at distance i from the pivot p. For each non-empty U_i, a child node is added to the root, and the corresponding branch is labeled with the distance value i. The same process is recursively applied to each new node. When a subset U_i has less than b objects, a leaf node is created for it and the recursive construction of the index stops in that branch. The index uses as many pivots as nodes has the tree, and the distances from the pivots to the rest of objects of the database are stored in the branches of the tree.

Given a range query (q, r), the search is carried out by traversing the tree from the root to the leaves. The query object q is compared with the pivot stored in the root of the tree. Once the distance $d(q, p)$ has been computed, the search proceeds recursively through the branches for which:

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

When the search reaches a leaf node (with up to b objects), the objects of the leaf are directly compared with the query. Each time the query is compared with

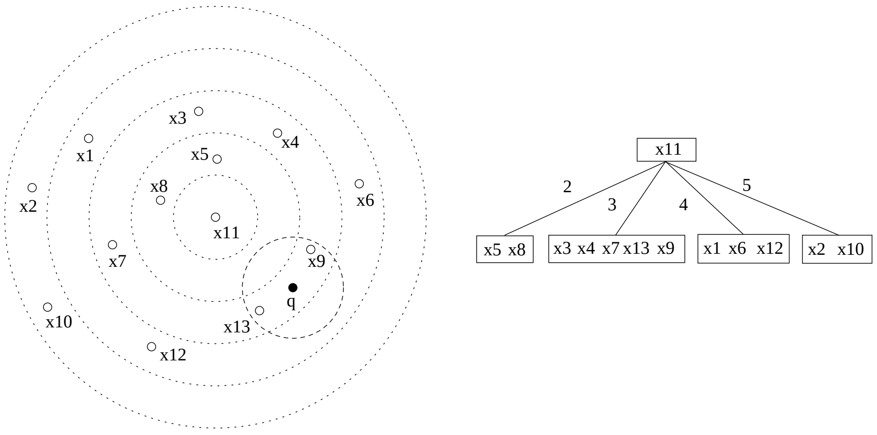


Figure 2.12: First level of a BKT tree for a set of points in $\mathbb{R}^2$.

either a pivot or an object in a leaf node, it is added to the result set if $d(q, x) \leq r$. With this structure, the search process can prune some branches of the tree by using the triangle inequality and the precomputed distances that label each branch of the tree. Therefore, the search space is pruned and the query object is not compared with objects that can be discarded by using the information of the index.

Figure 2.12 shows an example of how BKT works in a two-dimensional scenario. The left side of the figure shows the set of points indexed. The first object selected as a pivot is x_{11}. The figure shows the lines that intersect the objects placed at each distance of this first pivot. For each one of these distances, a new branch is added to the tree. The right side of the figure shows the first level of the tree. As we can see in the figure, the branch of objects placed at distance 5 from the pivot can be pruned, so x_2 and x_{10} can be directly discarded from the result set.

Fixed-Queries Tree (FQT)

Fixed-Queries Tree (FQT) [Baeza-Yates et al., 1994] is a modification of BKT that achieves a smaller number of evaluations of the distance function by reducing the number of comparisons of the query object with the pivots of the index. In FQT, all the nodes of a given level of the tree use the same object as a pivot, and the rest of objects of the database are stored in the leaves of the tree. Note that, in this case, an object used as a pivot in a node does not necessarily belong to the subset of objects processed in that node. In this way, the index reduces the number of comparisons by reducing the internal complexity (comparisons of the query with pivots). Only one comparison is needed in each level of the tree . Avoiding these distance computations can be significant if the distance function can return too

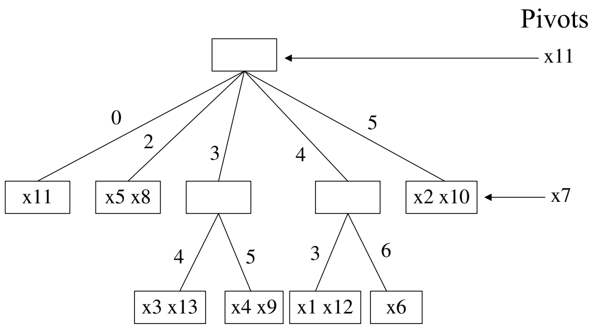


Figure 2.13: First level of a FQT tree for a set of points in $\mathbb{R}^2$.

many different values, something that would generate a very broad tree, or if the collection had a large number of objects, that would make the tree to be very deep. The search proceeds as in a BKT tree.

Figure 2.13 shows the FQT tree corresponding to the set of points we used in the previous example, assuming $b = 2$ (we do not show the set of points with the distances to each pivot for reasons of space). As we can see in the figure, the pivot used in the root of the tree is again x_{11}. However, instead of using a different pivot for each node in the next level, the object x_7 is used as a pivot for all of them. All the rest of objects of the database are referenced in the leaves of the tree.

A modification of this algorithm, Fixed-Height Tree (FHT) [Baeza-Yates, 1997], structures the tree with all the leaf nodes are at the same height h, independently of the number of objects they store. In this way, the shortest paths are extended through additional paths. The fact of having a deepest tree can improve the search, since those deeper paths can be discarded before reaching the leaf nodes (avoiding the comparison of the query with the objects stored in that leaf). Since the number of pivots is the number of levels of the tree, we can easily state how many pivots are going to be used. Although the extra cost for processing the structure is higher in this way, having more pivots permits to discard more objects by using the triangle inequality. While in BKT and FQT the number of pivots depends on the specific collection and the values returned by the distance function, in FHT the number of pivots has to be stated in advance. The experimental results shown in [Baeza-Yates, 1997] show that a deeper tree can reduce the number of evaluations of the distance function.

Fixed Queries Array (FQA)

Fixed Queries Array (FQA) [Chávez et al., 1999, Chávez et al., 2001a] is a compact representation of FHT. That is, instead of implementing the tree with nodes and

Fixed queries array

x11	x5	x8	x3	x13	x4	x9	x1	x12	x6	x2	x10	
0	2	2	3	3	3	3	4	4	4	5	5	Distances to pivot x11
3	4	2	4	4	5	5	3	3	6	3	3	Distances to pivot x7

Figure 2.14: Fixed Queries Array (FQA) for a set of points in $\mathbb{R}^2$.

pointers, it is transformed into a plain vector representation with the associated algorithms for efficiently performing the necessary operations on it.

Given a FQHT tree of height h, the FQA representation of the tree is obtained by performing a traversal of the leaves of the tree from the left to the right, storing the objects of the leaves in an array. For each object in the array, we obtain the h numbers that determine the path for reaching that object in the tree (that is, h distances in the tree). Each of these h numbers is coded in b bits in such a way that the highest levels of the tree correspond to the more significant digits.

The resulting vector is sorted by the hb-bits for each element. In this way, each interval of the FQA corresponds to a subtree of FQHT, and a movement in the FQHT is simulated with binary searches in the FQA. This implies an additional CPU cost for processing the index.

Figure 2.14 shows the FQA corresponding to the FHT index we used in the previous example. The advantage of this structure is that, using the same memory, FQA can use more pivots than FQHT, and this can improve the number of objects discarded by the algorithm. This reduction in the number of evaluations of the distance function compensates the extra CPU time needed for processing the structure. In addition it is more adequate for storing it in secondary memory.

BKT, FQT, FHT, and FQA are designed to work with discrete distance functions that return a finite, and relatively small, set of values. If applied with continuous distance functions, the tree would be completely plain and the search would consist in a sequential scan. Baeza-Yates et al. [Baeza-Yates et al., 1994] proposed a way of using these data structures with continuous distance functions by dividing the range of distances of the metric in a set of intervals and assigning intervals to each branch of the tree.

Vantage Point Tree (VPT)

Vantage Point Tree (VPT) [Yianilos, 1993] is also a tree-like pivot-based index. The index is a binary tree recursively built as follows: a first pivot $p \in U$ is selected at random as the root of the tree, and the distances from this pivot to the rest of

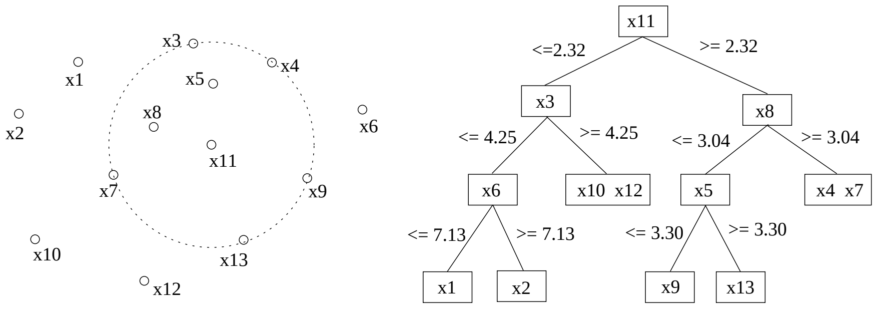


Figure 2.15: Vantage Point Tree, example of tree construction

objects of the collection are computed. If $m = median\{d(p,x),\ x \in U\}$, the objects $x \in U$ for which $d(x,p) \le m$ are processed in the left subtree, and the objects for which $d(x,p) > m$ are processed in the right subtree. Each node stores the pivot and the median distance to the rest of objects processed in that node. This procedure is recursively applied in each node until reaching the stop condition when the number of objects assigned to a subtree is small enough. In this structure the objects are stored in all the nodes of the tree, and not only in the leaves.

Figure 2.15 shows an example of a VPT structure on a set of points in a two-dimensional vector space. As in previous examples, x_{11} is the first object selected as a pivot. The dashed line corresponds to the median of distances from this pivot to the rest of objects of the collection. The recursive construction of the index gives as result the tree shown in the right side of the figure (we assume that $b = 2$).

Given a range query (q,r), the search starts by computing the distance $d(p,q)$ at the root of the tree. If $d(p,q) - r \le m$, the search has to process the left subtree; if $d(p,q) - r \ge m$ the search has to process the right subtree. Note that in any node of the tree, it may be necessary to continue the search in both children nodes if the pruning condition does not hold for any of them. The structure is very simple but the options for pruning a branch depend on the query radius. As argued in [Yianilos, 1993], the index is very efficient for small query radius.

Other important contribution of VPT is that it showed that the way the pivots are selected affects the search performance. In [Yianilos, 1993], the authors concluded that, for VPT, the best pivots are the farthest objects from the rest of objects in the collection. That is, they concluded that outliers are good pivots.

Multi-Vantage Point Trees (MVPT)

VPT tries to prune the search space in each subtree by discarding some of its branches. When this is not possible both the branches have to be explored and backtracking is necessary in order to finish the search. In some cases, this problem can degenerate in an almost sequential search, especially when the search radius is not small. A variant called Multi-Vantage Point Tree (MVPT) was proposed by [Bozkaya and Ozsoyoglu, 1997] as an extension of VPT that tries to solve this problem. MVPT is a vantage point tree in which each node has k children, with $k > 2$. Instead of using the median of the distances from the objects to the pivot for partitioning the space, MVPT uses the $k - 1$ percentiles $d_{m_1}, \ldots, d_{m_{k-1}}$.

Both the algorithms for index construction and search are very similar to the algorithms of VPT. The experimental evaluation presented in [Bozkaya and Ozsoyoglu, 1997] shows that MVPT does not always perform better than VPT. This is the case of high-dimensional spaces where the distances between any pair of objects are very small. In this case it can be necessary to enter in all the children of a node for answering a query.

Approximating Eliminating Search Algorithm (AESA)

Approximating and Eliminating Search Algorithm (AESA) [Vidal, 1986] [Vidal, 1994] and its variants are the most efficient pivot-based algorithms. Instead of using a tree structure as the methods we have already presented in this section, this method stores the distances from the pivots to rest of the objects in the database in a table. The data structure built by AESA is a $n \times n$ matrix which stores the distances between any two of objects in the database. The space needed for storing the matrix of distances can be reduced to $n(n-1)/2$ due to the property of symmetry of the distance function. Having all the distances in the index, any object of the database can be used as a pivot.

Given a query (q, r), the algorithm selects an object p at random and uses it as a pivot. The distance $d(q, p)$ from the pivot to the query is computed and is used to discard as many objects as possible, that is, all the objects for which $|d(q, p) - d(x, p)| > r$. The algorithm selects then another object as a pivot, the closest to the query in order to maximize the possibilities of discarding more objects. This is repeated until the list of objects that could not be discarded is small enough.

Figure 2.16 shows an example of AESA after the first iteration of the search process. In this example, x_{11} is the first object selected as a pivot. The two circles with center x_{11} are defined by the policy for discarding objects $|d(q, p) - d(u, p)| > r$, and enclose between them the objects that cannot be discarded using this first pivot. In the next iteration this process will be repeated with another pivot, the closest to the query, in order to discard as many objects as possible from the result. In this way, the areas that contain the candidate list for each pivot intersect giving as result a smaller candidate list.

Although AESA is very efficient in what refers to distance computations, and easy to implement, the huge space requirements for large databases can prevent it to be used in some cases. If the database contains a high number of objects, the

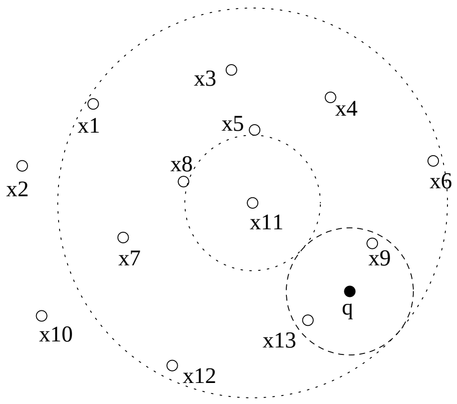


Figure 2.16: First step of the search with AESA on a set of points in $\mathbb{R}^2$.

space needed for storing the precomputed distances can be excessive. For example, if we had to index a database with $100,000$ objects, and the distances among any two of them are stored as floating point numbers of 8 bytes, we would need 37 GB for storing the index.

This does not mean that AESA is not applicable to real problems. It is an adequate method for problems with very costly distance functions and not very large databases. For example, it is a good option for indexing a database with a few thousand DNA sequences that are compared using the edit distance. The evaluation of a distance is very costly in this problem and the index would need only 15 MB of space, which is a reasonable space requirement taking into account the cost of a sequential search for this problem.

Linear AESA (LAESA)

Linear AESA (LAESA) [Micó et al., 1994] is a variation of AESA in which the space requirements are reduced at the cost of an increment of the number of distance computations. In LAESA a subset of m objects of the database are selected as pivots, and the distances from the rest of objects of the database to these pivots are stored in a $n \times m$ matrix. Therefore, the index stores only $n \times m$ distances instead of $n(n-1)/2$ in the case of AESA. For example, for a database of $100,000$ objects, if 50 pivots were selected, the index would require just 19 MB instead of the 37 GB required by AESA. Of course, the less information we have for pruning the search space, the higher the number of distance computations needed to solve a query.

The index construction process is basically the same of AESA. After the pivots are selected, the distances from the objects in the database to the pivots are computed and stored in a table. When given a query (q, r), the query object is compared with the k pivots, and these distances are used with the distances stored in the index and the triangle inequality to discard as many objects as possible. The objects that could not be discarded make up the candidate list and have to be directly compared with the query. An important issue in this algorithm is how the pivots are selected from the database. The number of pivots, the specific objects used as pivots, and their location with respect to each other and to the rest of the objects in the database affect the search performance.

A problem present in both AESA and LAESA is that they are static indexes. This means that the index has to be built on the complete database, and that further insertions or deletions of objects can degrade the index performance.

2.5.2 Clustering-based methods

As we have already explained in Section 2.3, clustering-based methods take a different approach to prune the search space. They decompose the space into a set of clusters, each of them represented by a cluster center. During the search, the information of the partition of the space is used to directly discard complete clusters from the result set without comparing any object in them with the query object.

The existing clustering-based methods differ in the criteria the use to discard clusters from the result set, how they partition the space, and the information they store in the index for each cluster.

Bisector Tree (BST)

Bisector Tree (BST) [Kalantari and McDonald, 1983] is probably one of the first clustering-based methods for searching in metric spaces. It is a very simple tree structure that is built by recursively partitioning the space. For each node, two objects c_1 and c_2 are selected and stored as cluster centers. The objects closer to c_1 form the cluster that will be assigned to the left child of the node, and those closer to c_2 form the cluster that will be assigned to the right child of the node. With these two objects used as centers the space is partitioned in two clusters. This partitioning procedure is recursively applied in each node of the tree until the clusters are small enough and are not further partitioned. The nodes store the cluster center and the covering radius (distance from the cluster center to its furthest object in the cluster) for each cluster.

The center of a cluster and its covering radius define a *ball* (c, r_c) in the space. It is important to note the difference between the cluster (a set of objects), and the ball defined by the cluster (a region in the space). The intersection of any two clusters is empty, while the balls corresponding to different clusters can intersect.

During the search, the tree is recursively processed from the root to the leaves. In each node, the query object is compared with the cluster centers. These distances

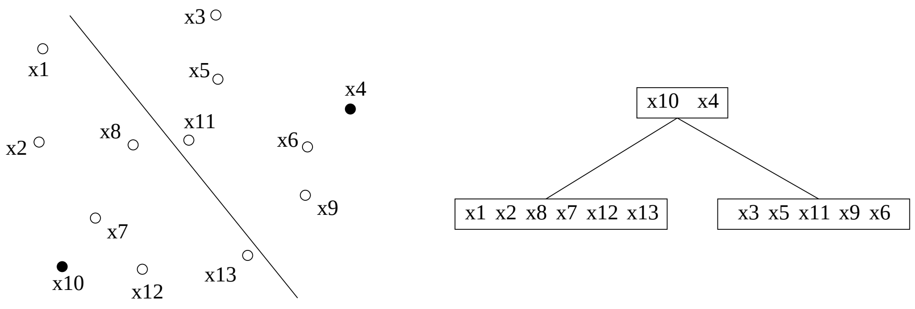


Figure 2.17: Space decomposition for the first level of a BST index.

and the covering radius (r_c) permit to know if the intersection of the cluster and the result set is non-empty. If $d(c, q) \leq r_c + r$, the balls defined by (c, r_c) and (q, r) intersect, so we have to further explore the cluster since it can have objects belonging to the result set. If the intersection is empty, that region of the space is directly pruned.

The idea is to use the tree to directly discard complete regions of the space, as soon as possible. Of course, in a given step of the search it can be necessary to explore both the clusters because the result set intersects the balls defined by each of them. When the search reaches a leaf of the tree, the query is directly compared with each object stored in it.

Figure 2.17 shows an example of BST and how it is used for the search. In the first level, the objects x_{10} and x_4 are used as cluster centers for partitioning the space. The line between them defines the limit of each region. Each cluster would be further partitioned until reaching the leaves.

The recursive partition of the space ensures that clusters of the adequate size will be available during the search to prune the search space. On the one hand, a too large cluster has more chances to intersect with the query ball even if no of its objects are in the result set. On the other hand, working with too small clusters increases the internal complexity since the query has to be compared with more cluster centers. A recursive partition refines the decomposition of the space in each level of the tree, from larger clusters to smaller ones.

Generalized Hyperplane Tree (GHT)

Generalized Hyperplane Tree (GHT) [Uhlmann, 1991] uses the same data structure as BST, but it does not use the same policy for discarding clusters during the search. Instead of using the covering radius of each cluster to discard complete regions, it uses the hyperplane placed between the two cluster centers. Given a query (q, r), in a given level of the tree we have to process the left child if $d(q, c_1) - r \leq d(q, c_2) + r$,

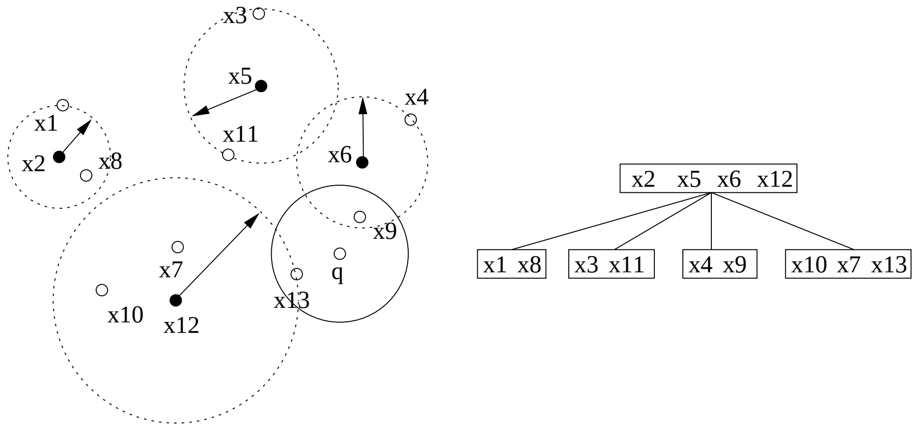


Figure 2.18: Space decomposition in the first level of GNAT ($m = 4$).

and the right child if $d(q, c_1) + r \geq d(q, c_2) + -r$. As in BST, it can be necessary to process both children to complete the search. An advantage of GHT is that it does not need to compute the covering radius of each cluster, and this avoids a lot of distance computations during the index construction.

Both in BST and GHT, the structure can be improved if one of the objects used as a center in a node is the cluster center of its parent. This makes the regions smaller, which improves the search performance, and reduces the internal complexity of the structure (that is, the comparisons of the query with the centers stored in the tree), since there are less cluster centers in the tree.

Geometric Near-neighbor Access Tree (GNAT)

Geometric Near-neighbor Access Tree (GNAT) [Brin, 1995] is a generalization of GHT using a m-ary tree instead of a binary tree. In the first level of the tree m cluster centers $c_1, \ldots, c_m$ are selected, and the space is partitioned in m clusters $C_i = \{x \in U,\ d(c_i, x) < d(u, c_j), \forall j \neq i\}$. The root of the tree stores these centers and each cluster is recursively partitioned.

Figure 2.18 shows the decomposition of a two-dimensional space in the first level of a GNAT index, with $m = 4$. As we can see in the figure, the ball defined by the query intersects with the clusters represented by x_{12} and x_6, but not with the clusters represented by x_2 and x_5, that can be directly discarded from the result.

The construction process is very similar to the construction of GHT, but the search is quite different. When the tree is built, each node stores also a $m \times m$ matrix defined as $range(i, j) = [min_{u \in U_j}(c_i, u), min_{u \in U_j}(c_i, u)]$. That is, each node stores a matrix that contains the minimum and maximum distances from each cluster center to each one of the rest of clusters. Given a query (q, r), q is compared with a center selected at random c_i. This distance and the distances stored in the matrix of the node permit to discard all those clusters for which $d(q, c_i)$ is not in the range defined by $range(i, j)$. This process is repeated with the rest of cluster centers until no more clusters can be discarded. The search process will continue recursively exploring the clusters that could not be discarded.

Partitioning the space in more clusters in each level of the tree gives as result smaller regions. This decreases the chances of the result set to intersect with all of them and thus the need to further explore them. In addition, GNAT combines the cluster and pivot approaches, by storing precomputed distances in the nodes of the tree that permit to discard some clusters from the result without comparing them with the query, thus reducing the internal complexity.

Voronoi Tree (VT)

Voronoi Tree (VT) [Dehne and Noltemeier, 1987] was proposed as a set of modifications to the original BST that significantly improve the search performance. In this case, each node of the tree has two or three cluster centers. The main difference with BST is that when a new node is created to insert a new object in the collection, the closest object of its parent node will also be inserted in the new node. In this way, the clusters are more compact as we move down in the tree structure, which permits to discard more objects during the search.

The experimental results provided by [Dehne and Noltemeier, 1987] show that VT obtains a better search performance than BST. Noltemeier et al. [Noltemeier, 1989] show that the VT trees can be constructed following an insertion criterion similar to that of the B-Trees.

M-Tree (MT)

The M-Tree [Ciaccia et al., 1997] is one of the most important methods for indexing and searching in metric spaces. It was designed for efficiently supporting insertions and deletions on the database without degrading the search performance of the structure. This means that M-Tree is a fully dynamic method: the database can be initially empty and grow later as new objects are inserted or deleted. In addition, it can be efficiently stored in secondary memory and obtains a good search performance.

M-Tree has a very similar structure to that of GNAT. As in GNAT, it uses a tree with several cluster centers stored in each node. However, the search algorithm is different. In the case of M-Tree, the tree stores the covering radius for each cluster. When given a query (q, r), q is compared with all the cluster centers of the node and this information is used to discard as many clusters as possible.

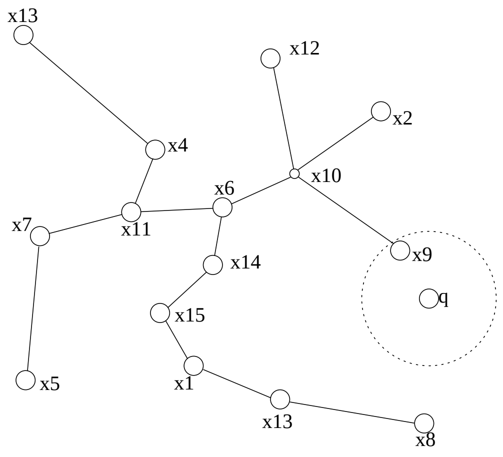


Figure 2.19: Example of indexing and searching with SAT

The main difference with other structures is how the insertion of new objects is managed. When a new object is added to the database, it will be inserted in the best subtree possible, that is, in that for which the covering radius is less increased. As we have seen in other algorithms, the smaller the radius of the clusters, the more the possibilities of discarding more objects during the search. The insertion looks for the leaf node of the tree in which the object should be inserted. If the node has place for this new object, it is just inserted. If not, the node is splited as in a B-Tree. This insertion procedure makes M-Tree a balanced structure, efficient in terms of the number of distance computations for answering a query and in the number of I/O operations needed for loading and processing the index during the search.

Spatial Approximation Tree (SAT)

Spatial Approximation Tree (SAT) [Navarro, 1999] follows a different approach and tries to take advantage of the relationships of proximity between the objects in the space when building the index. The index created with SAT is a tree that approximates a *Delaunay graph* of the space, defined as follows: if the space were divided into a Voronoi partition, a Delaunay graph contains a node corresponding to each cell of the partition, and edges connecting the nodes corresponding to directly neighboring cells in the space. Since the construction of such a graph is a NP-complete problem, SAT tries to obtain an approximation at a reasonable cost.

The construction of the index proceeds recursively as follows. The root of the tree, p, is selected at random. For this root node p, the method obtains the set $N(p)$, defined as the set of all the objects which are nearer to p than to any other

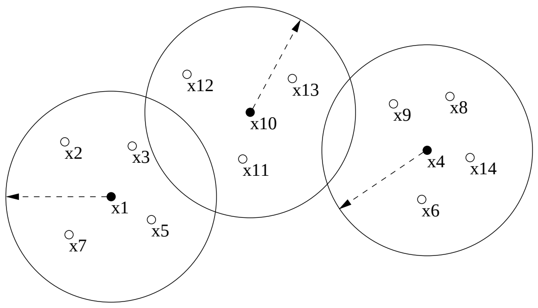


Figure 2.20: List of clusters on a set of two-dimensional points.

object in $N(p)$ (the definition is self-referenced, and for a given object p, many valid $N(p)$ sets are possible). In order to obtain this set, p is compared with the rest of the objects of the database, which are sorted according to their proximity to p and added to $N(p)$ if they hold the condition that defines the set. Each object in $N(p)$ becomes a child node of p, and the same construction procedure is recursively applied to each of them. Each node stores the covering radius, this is, the maximum distance from p to any object in $N(p)$.

When given a query (q, r), the tree is recursively traversed, comparing the query object with the nodes that can not be discarded from the result by applying the ball partitioning principle.

List of Clusters (LC)

Most clustering-based methods organize the index as a tree-like structure that reflects a recursive decomposition of the space. List of Clusters (LC) [Chávez and Navarro, 2005] follows a different approach by creating a Voronoi partition of the space and organizing the resulting clusters in a list, without further partitioning them.

This partition can be obtained by creating clusters of a fixed radius, or by creating clusters with a fixed number of objects. For example, in the case of clusters with a fixed number of elements s, the index algorithm proceeds as follows. A cluster center is selected and a cluster is created from it with its s nearest objects. The process is repeated with the objects that were not processed in this first step, until all objects have been indexed.

Figure 2.20 shows an example of a list of clusters for a set of points in a two-dimensional vector space. In this example all the clusters have the same covering radius. The index is just a list that stores the cluster centers $\{x_1, x_{10}, x_{14}\}$ and the list of objects in each cluster.

The search algorithm proceeds as usual. The query object is compared with all the clusters and the search space is pruned by discarded from the result as many of them as possible.

Although it can work better than other clustering-based methods in its optimal configuration, either the optimal covering radius of the clusters or the optimal number of objects in each of them have to be obtained by trial and error on the collection, a disadvantage with respect to previous methods.

2.6 Selection of reference objects

An important aspect in any method for searching in metric spaces is how the objects used as references (either pivots or cluster centers) are actually selected. Although most methods select them at random, it has been shown that the specific set of objects used as references and the way they are selected affects the overall search performance and other characteristics of the index. The survey *Searching in Metric Spaces* [Chávez et al., 2001b] deeply analyzed the existing methods for searching in metric spaces and proposed a unified taxonomy of methods. The conclusions of this survey already pointed out the selection of effective reference objects as a problem worth of further research.

In this section we review the existing techniques for the selection of reference objects for methods for searching in metric spaces, and the advantages and drawbacks of each of them.

Issues related to the selection of reference objects

The selection of reference objects for methods for searching in metric spaces has many implications in both the search performance and other characteristics of the method. Several issues are associated to the selection of effective reference objects:

- *Selecting the most effective pivots*: The possibility of a pivot to discard an object from the result set depends on its relative position with respect to the query object and to the object we try to discard. If two pivots are more or less in the same position, their effectiveness will also be more or less the same. The position of the pivots with respect to each other, and their position with respect to the objects stored in the database affect the overall index performance. A random pivot selection does not ensure the objects used as pivots to be as best as possible.

- *Determining the optimal number of pivots*: The higher the number of pivots, the more the possibilities the index has for discarding an object from the result. However, since the total complexity is given by the sum of internal and external complexities, there is a point in which the number of comparisons of the query with the pivots does not pay for the gain in the reduction of the candidate list.

There is an optimal number of pivots that optimizes the trade-off between the internal and external complexities. A random pivot selection does not give any insight about the optimal number of pivots.

- *Space requirements*: In general, the larger the number of objects the index uses as pivots, the more space we need for storing the distances from objects to pivots. Given two set of pivots that show the same performance in the search, the smaller is the best one, since it requires less memory. The number of pivots is an important issue since memory requirements is one of the main drawbacks of pivot-based algorithms and can make impossible to apply them in practical applications if the pivots are too many.

- *Static indexing*: Selecting pivots at random inevitably makes the index static. Due to the trade-off between internal and external complexities, the optimal number of pivots has to be determined by trial and error on the whole collection. This means that the database has to be complete before building the index. Some indexes do not allow insertions or deletions of objects after the index is built. Other indexes allow these operations, but the search performance can degrade if the number of insertions and deletions is not small. With a fully dynamic index the database could be initially empty, and the index is built as new objects are inserted or removed from the collection, maintaining a stable search performance while the database evolves.

- *Complexity of the selection*: The cost of selecting pivots also affects the index operation, since it translates in the cost of inserting an object in the collection. If dynamism and interactivity are important for the operation of the system, the complexity of pivot selection is also important.

The problematic of the selection of reference objects was originally thought for pivot-based methods, but the same issues arise for the selection of effective cluster centers in clustering-based methods. The number of cluster centers, their position with respect to each other, and their position with respect to the rest of objects of the database determine the properties of the decomposition of the space, and therefore affect directly the capacity of the index for pruning the search space. For example, the set of cluster centers determines the compactness of each cluster and the overlapping between them. As in the case of pivot-based methods, a random selection of cluster centers has several drawbacks.

Although most techniques for the selection of reference objects have been proposed for pivot-based indexes, in this thesis we consider this problem also for the selection of effective cluster centers.

Previous work on selection of reference objects

Random selection of pivots

Most pivot-based methods select the pivots at random. Obviously, a random selection of reference objects do not give us any guarantee of the obtained set of objects to be effective. Although the selection is not complex since it does not involve any kind of extra computation in order to decide which objects become references, it suffers of all the issues we have described in the previous section.

Selecting far away pivots

The first techniques for pivot selection focused in using as pivots objects that are far away between them, and also far away from the rest of objects of the database, that is, *outliers*. This idea was first explored in LAESA [Micó et al., 1994], where the objects selected as pivots were selected as far as possible between them. [Yianilos, 1993] and [Brin, 1995] extended this idea to the selection of cluster centers, trying to obtain objects maximizing the sum of distances between them to minimize as possible the overlap between the clusters.

Although these works introduced the idea that far objects work well as pivots, this was not the problem in what they focused, so the effect of pivot effectiveness was not studied in much depth. Later contributions fully focused on the effect and techniques for pivot selection. They are presented in the rest of this section in chronological order.

MaxMin

MaxMin [Vleugels and Veltkamp, 2002] was proposed as a technique for pivot selection following the idea of obtaining pivots that are far from each other. In this technique, the first pivot is chosen at random. Then, each pivot p_i, $2 \leq i \leq m$ is chosen as the object maximizing the distance to the previously selected pivots.

The idea is very similar to the proposed in works as [Micó et al., 1994], [Yianilos, 1993], and [Brin, 1995]. Note that this technique tries to obtain pivots far away from each other, but it does not impose the condition that they have to be far from the rest of objects of the database.

In addition, this technique does not provide any guidance on how to obtain the optimal number of pivots for a given space.

Stepwise Forward Leave-One-Out (SFLOO)

Stepwise Forward Leave-One-Out (SFLOO) [Hennig and Latecki, 2003] introduces the concept of the loss of quality introduced by the pivots as a measure of the difference between the real distance between two objects and the distance between their representation in the vector space defined by the pivots. The selection procedure obtains an effective set of pivots by minimizing the loss of quality it introduces.

Let (X, d) be a metric space, $U = \{x_1, \ldots, x_n\}$, $U \subseteq X$, the database of objects, and $P = \{p_1, \ldots, p_m\}$ a set of m pivots. For any object $x \in U$, its representation in the vector space defined by the pivots is given by:

$$\vec{v}(x) = (d(q, p_1), \ldots, d(q, p_m))$$

Given a query object q, the distance between the query object q and an object $x \in U$ in the vector space defined by the pivots (d_v) is given by the Euclidean distance between the vectors that represent them in that space:

$$d_v(q, x) = ||\vec{v}(q) - \vec{v}(x)||$$

From these definitions, the nearest object to the query q in the vector space defined by the pivots (from now, the pivot space), $s(q, U)$, is the object $x \in U$ that minimizes the value of the distance $d_v(q, x)$, that is:

$$s(q, U) = \{\ x \in U\ /\ \forall y \in U - \{x\},\ d(q, x) \leq d(q, y)\}$$

Hennig and Latecki [Hennig and Latecki, 2003] consider that the set of pivots introduces a loss of information in the search if the query's nearest neighbor in the pivot space and the query's nearest neighbor in U using the distance function d are not the same object. The loss of information introduced by the set of pivots for the object $q \in X$ is defined as the distance between the query and its nearest neighbor in the pivot space, that is:

$$l(q, s(q, U)) = d(q, s(q, U))$$

Then, Hennig and Latecki define the loss of quality introduced by the set of pivots as the average loss of information for all the objects in the database [Hennig and Latecki, 2003], that is:

$$L(s) = \frac{1}{n} \sum_{x \in U} l(x, s(x, U - \{x\}))$$

The selection of the pivots tries to optimize the value of the loss of information introduced by this search procedure. The selection is incremental: the first object selected as a pivot is the one that alone gives the smaller loss; then, the next pivot is chosen as the object minimizing the value of the loss together with the first one; and this is repeated until reaching the desired number of pivots.

This selection involves a high computational cost. In [Hennig and Latecki, 2003], the authors try to reduce this cost using a sample of objects to evaluate the loss function, and by taking the pivots also from a sample of objects updated in each iteration of the process.

Incremental and Local Optimum

Incremental and Local Optimum [Bustos et al., 2003] try to obtain an effective set of pivots by iteratively refining an initial set of pivots selected at random, according to a criteria for evaluating the effectiveness of a set of pivots. An important contribution of [Bustos et al., 2003] is a criterion for estimating the efficiency of a set of pivots of a given size.

Let (X, d) be a metric space, and $P = \{p_1, p_2, \ldots, p_m\}$, with $p_i \in U$, a set of pivots. Given an object $x \in U$, Bustos et al. [Bustos et al., 2003] denote with $[x]$ the representation of x in the pivot space, that is, the tuple composed by the distances from x to each pivot in P:

$$[x] = (d(x, p_1), d(x, p_2), \ldots, d(x, p_m))$$

Thus a new space of objects $[P]$ is defined as $[P] = \{[x] \ / \ x \in X\}$, which is a vector space $\mathbb{R}^k$. We can define a distance function on this new space as:

$$D_P([x], [y]) = max_{1 \leq i \leq m} |d(x, p_i - d(y, p_i)|$$

Since the distance function D_P is a metric defined on the objects of $[P]$, the pair $([P], D_P)$ is a metric space. That is, the set of the representations of all the objects in the pivot space and the maximum distance forms a new metric space.

Given a query (q, r), the policy for discarding an object from the result set without comparing it with the query, $|d(p_i, u) - d(p_i, q)| > r$ for some pivot p_i, can be translated to the new metric spaces as:

$$D_P([q], [u]) > r$$

Intuitively, the above expression means that we can discard the object x from the result if the best lower bound obtained with the set of pivots P is greater than the search radius r.

The more objects it can discard, the more effective a set of pivots is. Thus, a set of pivots P_1 is better than other set of pivots of the same size P_2 if the probability of $D_{P_1}([q], [x]) > r$ is higher than the probability of $D_{P_1}([q], [x]) > r$, with $x \in U$. If μ_{D_P} is the mean of the distribution of the distance function D_P, the larger the value of μ_D, the better the set of pivots P is.

Thus the criterion proposed by [Bustos et al., 2003] establishes that a set of pivots $P_1 = \{p_1, p_2, \ldots, p_m\}$ is better than the set $P_2 = \{p'_1, p'_2, \ldots, p'_m\}$ if:

$$\mu_{D_{P_1}} > \mu_{D_{P_2}}$$

[Bustos et al., 2003] also proposes several strategies for pivot selection based on this effectiveness estimator:

- *Selection*: This technique selects N groups of pivots at random, and it finally uses the one that maximizes the effectiveness criterion, that is, the one that obtains the higher value for μ_D. Although it is a very simple strategy, it obtains better results than a simple random selection if the number of samples N is high enough, usually around 50. In order to estimate the value of μ_D for the N samples of pivot sets, instead of computing the value on the whole collection, it is used a set of A pairs of objects selected at random form the collection, where A can be a much smaller value than the total number of possible pairs of objects. Since the value of μ_D is estimated N times, the pivot selection requires $2mAN$ evaluations of the distance function.

- *Incremental*: the incremental pivot selection starts the process with a pivot selected from a subset of N objects from the collection. This first pivot is the object maximizing alone the value of μ_D. Then, a second object is selected from another sample of N objects, in such a way that $\{p_1, p_2\}$ maximizes the value of μ_D, considering p_1 is already selected. The process is repeated until completing the set of m pivots. Each time a new pivot is added to the set, it is necessary to carry out $2AN$ evaluations of the distance function (in order to determine what is the object maximizing μ_D), and thus this selection strategy has the same computational cost than the previous one: $2mAN$. With the same computational cost, the results of [Bustos et al., 2001] show that this strategy selects better pivots than Selection.

- *Local Optimum*: the last technique proposed in [Bustos et al., 2001] follows an iterative local optimum strategy. In this case the selection algorithm starts with a set of m pivots selected at random. Then, in each iteration, the algorithm removes from the set of pivots the object that less contributes to the value of μ_D, and it is replaced by the best pivots among a sample of X objects selected at random from the collection. This process is repeated N' times. To determine what is the pivot replace in each iteration, the algorithm stores a multidimensional matrix M of A rows and m columns, being A the number of objects used to estimate μ_D and m the number of pivots. The initial construction of the matrix requires $2Am$ evaluations of the distance function,. In each iteration, determining the pivot to be replaced does not have additional cost at all, since all the necessary information is in M. Obtaining the pivot that will replace it requires $2AX$ evaluations of the distance function. Thus, the total cost is $2A(k + N'k)$.

[Bustos et al., 2001] considers two versions of this selection strategy. The first one uses $N' = m$ and $X = N - 1$, that gives as result a complexity of $2AmN$. The second one uses $N' = N - 1$ and $X = m$, with the same final complexity. With the same complexity, the first variant uses more objects to obtain the new pivot, while the second performs more iterations. [Bustos et al., 2001] shows that the results obtained in each case do not need to be equal.

The results presented in [Bustos et al., 2001] proof the importance of a good pivot selection for the index performance. It is the first work providing an analytical criterion for comparing the efficiency of two sets of pivots of the same size.

Spacing

The technique proposed by [van Leuken et al., 2006] is based on two criteria which address the number of false positives in the retrieval results directly. The first criterion, the spacing, concerns the relevance of a single pivot; the second criterion, the correlation, deals with the redundancy of a pivot with respect to the other pivots.

The spacing is achieved by avoiding clusters on the vantage axis belonging to the pivot p_j. The spacing between two consecutive objects x_i and x_{i+1} of the database on the axis of the pivot p_j is:

$$d(x_{i+1}, p_j) - d(x_i, p_j)$$

If μ is the average spacing, the variance of spacing is:

$$\frac{1}{n-1}\sum_{i=1}^{n-1}((d(x_{i+1}, p_j) - d(x_i, p_j)) - \mu)^2$$

To ensure that the database objects are evenly spread in the pivots space, the variance of spacing has to be as small as possible. A pivot with a small variance of spacing is said to be a relevant pivot.

However, a low variance of spacing does not guarantee that the database is well spread out in pivot space, since the pivot axes might be strongly correlated. Therefore, they compute all linear correlation coefficients for all pairs of pivots and make sure these coefficients do not exceed a certain threshold.

As can be noticed, the number of pivots and two thresholds (for the variance of spacing and for the correlation coefficients) must be set beforehand. There is a clear new tradeoff there: the stricter these threshold values are, the better the selected pivots will perform but also the higher the chance of a pivot needing to be replaced, resulting in a longer running time to select the whole set of pivots.

Maximum Pruning

Maximum Pruning [Venkateswaran et al., 2008] is an iterative strategy for pivot selection. After initializing the set of pivots, the algorithm replaces in each iteration some pivot by a most promising object taken from the database, an strategy similar to *Local Optimum* [Bustos et al., 2003]. As in other techniques, the number of pivots to select, m, is given as a parameter to the algorithm, and its optimal value has to be determined by trial and error.

The set of pivots contains initially a set of m objects considered promising pivots, which are selected as follows: given an object $x \in U$, the mean and variance of its distances to the rest of objects of the database give an idea of the amount of near and far objects it has. If the variance is small, most objects are more or less at the same distance from x; if the variance is high, x has near and far objects. The set of pivots is initialized with the m objects that show the *maximum variance* in their distances to the rest of objects in the database, assuming that, the higher the variance, the more promising is the object as a pivot.

After the initialization of the set of pivots, the algorithm replaces a pivot by a more promising object in each iteration. Let P be the set of pivots and Q a set of sample queries taken from the database. In each iteration, all the objects in U are candidates for replacing some pivot in P. Basically, the algorithm computes for $x_i \in U$ and each $p_j \in P$, the gain obtained in the effectiveness of the set of pivots by replacing p_j by x_i. In each iteration the algorithm does the replacement $x_i \longleftrightarrow p_j$ that maximizes this gain. The replacement of pivots stops when the effectiveness of the whole set can not be improved with any replacement.

Although the algorithm obtains good results in what refers to search performance, we consider that this method presents two inconveniences. First, it needs to know the search range during the indexing phase in order to select the set Q of sample queries used in each iteration, or at least a range of values bounding this search range. This restricts the search range to that used during the indexing phase, and gives the algorithm an advantage over the rest of methods, since no other method assumes to know this information to index the dataset.

The second inconvenience of the algorithm is the high computational cost involved in the indexing phase: during the initialization, each object has to be compared with the rest of objects or with a significant sample of them; then, each replacement involves a high computational cost too. As noted by the authors [Venkateswaran et al., 2008], the algorithm is impractical for large databases without applying sampling-based optimizations.

[Venkateswaran et al., 2008] proposes two sampling-based optimizations for reducing the cost of index construction. The first one consists in reducing the number of objects to be searched in for each query when estimating the gain in the effectiveness of the set of pivots. The second optimization consists considering a subset of objects in the database as candidates for replacing a pivot, instead of considering the entire database.

2.7 Summary

In this chapter we introduced the problem of similarity search, and how it can be formalized using the concept of metric spaces. We have described the most typical similarity queries, and typical distance functions that can be applied to a wide range of problems.

We then introduced the basic concepts of methods for similarity search in metric spaces, which try to make the operation more efficient by avoiding the comparison of the query object with all the objects in the database. Pivot-based methods select a subset of objects from the database as references and compute and store in the index the distances from those reference objects to the rest of objects in the database. During the search, these distances are exploited with the triangle inequality to obtain lower bounds on the distances from the query to each object in the database and thus discard as many objects from the result.

Clustering-based methods take a different approach by dividing the data space into a set of clusters, storing in the index for each of them the cluster center and the covering radius, which is the distance from the center to its furthest object in the cluster. During the search, the objects in the clusters which enclosing ball have an empty intersection with the query ball are directly discarded from the result set.

After reviewing the most important methods of the state of the art, we analyzed the different issues related to the selection of effective reference objects, and how they can affect the search performance, as well as other parameters as the space requirements of the index, the indexing cost and the possibility of dynamically indexing the database. We have presented a detailed description of the previous techniques proposed for the selection of effective pivots.

Chapter 3

Sparse Spatial Selection

3.1 Overview of the chapter

In this chapter we present Sparse Spatial Selection (SSS), a new pivot-based method for searching in metric spaces. While previous methods tried to obtain pivots as far as possible between them, our method ensures the set of pivots to be well distributed in the space. This is the most outstanding property of our method and it has not been considered before. As we will see in this chapter, the search performance we obtain is better than, or at least as good as the obtained with previous and more complex and expensive techniques for pivot selection.

Sparse Spatial Selection has other important characteristics. It works with continuous and discrete distances. Both the selection of pivots and the construction of the index are dynamic and adaptive. That is, our method builds and adapts the index as new objects are inserted into or removed from an initially empty database. The construction of the index never finishes, and the information it stores depends on the content of the database in each moment. The number of pivots does not have to be fixed beforehand, and the selection is more efficient than in previous techniques. The selection of pivots and the construction of the index are described in Section 3.2, and Section 3.3 describes the changes to be done in the index when an object is deleted from the database. Search is described in Section 3.4

The simplest structure for storing the index is a table that stores in each row the distances from an object in the database to all the pivots. Section 3.5 discusses other alternatives for efficiently storing and retrieving the index in secondary memory, suitable for its dynamic nature. Section 3.6 presents the results and conclusions of the experimental evaluation of the method. Finally, Section 3.7 provides a discussion of the advantages and drawbacks of the proposed method, and Section 3.8 summarizes the contributions presented in this chapter.

3.2 Pivot selection and index construction

Sparse Spatial Selection (SSS) is a new pivot-based method for searching in metric spaces. As methods like AESA [Vidal, 1986] and LAESA [Micó et al., 1994], it selects a set of objects from the database to be used as pivots, and stores the distances from all the objects in the database to the pivots. The main difference with previous methods is its policy for pivot selection. While previous methods try to select as pivots objects as far as possible from each other, our method selects a set of pivots well distributed in the space. Our intuition was that well distributed pivots will have more probability of discarding an object from the result.

In order to obtain a set of well distributed pivots, an object becomes a pivot if it is far enough from the already selected pivots. We consider that the object x is far enough from the current set of pivots, $P = \{p_1, \ldots, p_m\}$, if its distance to any pivot is equal or greater than $M\alpha$, where M is the maximum distance between any two objects of the space, and α is a constant parameter that takes values between 0 and 1, typically around 0.4. That is, we consider that the object is far enough from the already selected pivots if it is up to a fraction of the maximum distance between any two objects in the database.

An important difference of SSS with previous techniques is that it is dynamic. It assumes that the construction of the index starts from an empty database and never finishes. The pivots are selected and the index is built adapting itself as new objects are inserted into the database.

Let (X, d) be a metric space, $U = \{x_1, \ldots, x_n\}$, $U \subseteq X$, the database, and $P = \{p_1, \ldots, p_m\}$, $P \subseteq U$, the set of pivots selected by SSS. To simplify the conceptual approach let assume that the distances from the objects to the pivots are stored in a table with as many rows as objects in the database and as many columns as pivots. The construction of the index proceeds as follows:

- When an object $x_{n+1} \in X$ is inserted into the database, $U \leftarrow U \cup \{x_{n+1}\}$, it is compared with the pivots already selected to obtain the distances $d(x_{n+1}, p_i)$, $1 \leq i \leq m$.
- If $d(x_{n+1}, p_i) \geq M\alpha, 1 \leq i \leq m$, x_{n+1} becomes a pivot, $P \leftarrow P \cup \{x_{n+1}\}$. A column is added to the table of distances and the distances $d(x_{n+1}, x_j)$, $1 \leq j \leq n$, are computed and stored in that column.
- If the new object is too close to an already selected pivot, it does not become a new pivot. In this case, a row is added to the table of distances and the distances $d(x_{n+1}, p_i)$, $1 \leq i \leq m$, (already computed) are stored in it.

When the first object is inserted in the database, it becomes the first pivot since there are no other pivots to compare it with. If the index is built on a complete database, the first pivot can be chosen at random. As we will see in the experimental

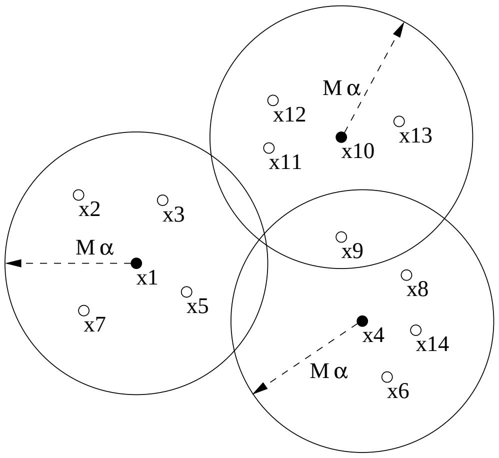


Figure 3.1: Example of indexing in a two-dimensional vector space.

evaluation of the method (Section 3.6), neither the selection of the first pivot nor the order in which the rest of objects of the database are processed affects significantly the search performance of the method, ensuring its robustness.

Figure 3.1 shows an example of how the pivots are selected and how the index is built for a small database where the objects are distributed in a two-dimensional space. In this example, the points $x_1, \ldots, x_{14}$ are sequentially inserted into an initially empty database. The objects selected as pivots are shown in bold.

The first object inserted in the database is x_1. Since no pivot has been selected, it is far enough from all pivots and it becomes the first one. The objects x_2 and x_3 are then inserted in the database. Since $d(x_2, x_1) \leq M\alpha$, and $d(x_3, x_1) \leq M\alpha$, they are not selected as pivots, because they are not far enough from x_1. When x_4 is inserted, it becomes a new pivot, since $d(x_4, x_1) \geq M\alpha$. The objects x_5, x_6, x_7, x_8, and x_9 are then inserted, falling too close to x_1 or x_4, and, therefore, they do not become pivots. The last pivot selected is x_{10}. Finally x_{11}, x_{12}, x_{13}, and x_{14} are inserted and none of them becomes a pivot.

As we can see in this example, the index evolves as the database changes when new objects are inserted. The pivots are selected and the index is adapted as needed. In addition, it is not necessary to state beforehand the number of pivots to use. New pivots are selected as the collection expands to new regions of the space. If the space is completely covered by the set of pivots, no more pivots will be selected despite of the number of new objects inserted. That is, the number of pivots depends on the complexity or dimensionality of the space, and not on the size of the collection.

It is important to take into account that the information used for selecting pivots is the same that is needed to insert an object in the database: the distances from the new object to the pivots already selected. When an object becomes a pivot, the cost of computing the distances from the rest of objects to it is also the needed in any method. Therefore, SSS does not impose an additional computational cost for pivot selection. All previous methods need an extra processing for pivot selection, very significant in some cases.

Algorithm 3.1: Insertion of a new object with Sparse Spatial Selection.

```
Input: $u \in X$, $P$, $M$, $\alpha$
Output: $P$
$selected \leftarrow true$;
i ← 1;
repeat
    $distance = d(x, p_i)$;
    if $distance < M\alpha$ then
        $selected \leftarrow false$;
    end
    i ← i + 1;
until not $selected$ or $i > |P|$ ;
if $selected$ then
    return $P \cup \{x\}$;
end
return $P$;
```

The pseudo-code shown in Algorithm 3.1 summarizes the algorithm for the insertion of a new object in the database with Sparse Spatial Selection.

3.2.1 Pivot selection policy

In order to obtain a set of well distributed pivots, a new object becomes a pivot if its distance to the rest of pivots is equal or greater than $M\alpha$. For instance, if $\alpha = 0.5$, an object becomes a pivot if it is up to a half of the maximum distance from the already selected pivots.

The parameter α controls the density of pivots with which the space is covered, it imposes a lower bound on the distance between any two pivots. The smaller the value of α, the more the pivots selected, and the closer they will be. Since α controls the number of pivots selected, it also controls the trade-off between the internal (comparisons of the query with the pivots) and external complexities (comparisons of the query with the objects that could not be discarded). The experimental evaluation reveals that the optimal values of this parameter are around 0.4 for collections of different nature.

Note that it is not necessary to state beforehand the number of pivots to select.

The method determines by itself the optimal number of pivots from the maximum distance between two objects M, the distribution of objects in the space and the parameter α. SSS adapts both the number of pivots and the index structure to the complexity and distribution of the space.

This is another advantage of this selection policy when compared with previous techniques. In previous methods the number of pivots has to be stated beforehand and its optimal value is obtained by trial and error on the complete collection, which makes the index static. However, the construction of the index in SSS is dynamic and pivots are selected as needed.

3.2.2 Estimating the maximum distance

Although it is not necessary to state beforehand the number of pivots to select, this number is determined by the parameter α and the maximum distance between any two objects. In most cases, the value of the maximum distance can be obtained from the definition of the metric space, that is, from the definition of the objects and the distance function used to compare them. For instance, when comparing words with the edit distance, the maximum distance will be around 21 for most languages, and 27 for German. If we are indexing a collection of images represented by feature vectors, the maximum distance between two objects can be obtained from the maximum and minimum values of each component of the feature vectors.

If it is not possible to obtain this value analytically, a good approximation of the maximum distance can be obtained while conserving the dynamic nature of the method. Instead of indexing every object from an initially empty database, the method can postpone the construction of the index until the database contains a sample of objects (although it does not have to be complete). Those first objects inserted into the database can be used to obtain the estimation of the maximum distance. The selection of pivots would start from that point.

Instead of comparing the objects in the database against each other to obtain the value of the maximum distance M, an approximation can be obtained with the following algorithm, proposed in [Goel et al., 2001]. An object is picked at random and compared with all the other objects, in order to find its furthest neighbor. This furthest neighbor is them compared with the rest of objects to obtain its furthest neighbor. The process is repeated for a given number of iterations. It has been shown that a very good estimation of the maximum distance, if not the exact value of that distance, can be obtained in about four iterations.

Therefore, even if it is not possible to analytically obtain the value of the maximum distance, it can be approached without incurring into a high computational cost. The value of the maximum distance can also be updated as more objects are inserted into the database.

3.3 Deleting an object

In order to delete an object $x \in U$ from the database, the first problem is to find it, to check if it is in the database or not. Unlike in classical data structures, this can not be done by simulating the insertion of the object. In this case, the object can be found by performing a range query with search radius zero, that is, $R(q, r) = R(x, 0)$. The cost of finding the object will be reasonably small, since the smaller the search radius is, the cheaper the range query. Once the object is located in the database, there are two possibilities depending on whether the object to be deleted is a pivot or not.

If the object x to be deleted is not a pivot, it can be deleted from the database without cost. The row that stores the distances from this object to the pivots can be removed, and no further changes are necessary in the index, since the set of pivots remains the same and no other object is affected by the deletion of x. In this case, the deletion of the objects does not imply any additional evaluation of the distance function.

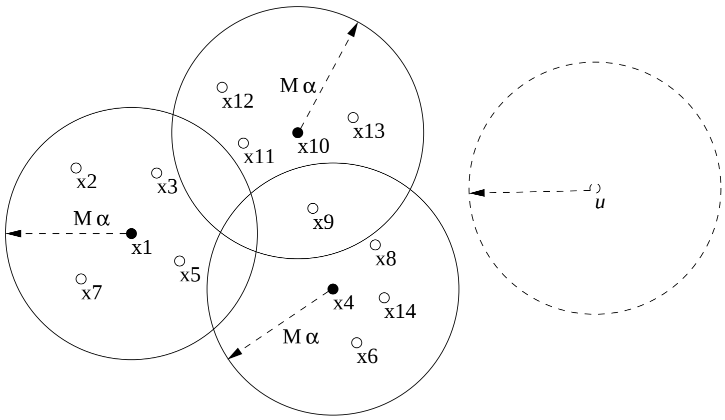


Figure 3.2: Deletion of a pivot without physically removing it.

If the object is a pivot, we have two alternatives to delete it:

- The pivot x can be tagged as a deleted object, so it will not be included into the result of any query. That is, the object no longer belongs to the database, but we use it as an element that gives us information for maintaining the index. Although it is easy, this choice has two drawbacks. First, in some applications the objects are very large and would be convenient to physically delete it. Second, we could reach a situation in which the deleted pivot covers a region of the space that does not need to be covered, since there are no real objects in it. This may led to unnecessarily increasing the internal complexity of the search (see Figure 3.2).

- If the object is physically deleted, another object of the database can become a pivot after removing this one. If $x = p_i$ was the only pivot that prevented other object to become a pivot, that object has now to be added to the set of pivots. In this case, to update the index is enough to check in the index for the objects $y \in U$ such that $d(x, y) \leq M\alpha$, if $d(y, p_j) \geq M\alpha,\ 1 \leq j \leq m$. If y becomes a pivot, the distances $d(y, x_i)$, for $x_i \in U$, must be computed The distances from y to the rest of pivots are already stored in the index, so the deletion of the pivot only implies the computation of distances if another object becomes a pivot. In order to preserve the structure of the set of pivots on the space, the first objects to be checked should be the nearest to the deleted pivot.

Following the second option, the index also adapts its structure and information when objects are removed from the database without increasing the internal complexity with pivots that are no longer needed. This is another important difference with previous techniques.

3.4 Searching

When given a query (q, r), the search proceeds as in general in most pivot-based methods. The query object is compared with the pivots in order to obtain the distances $d(q, p_j),\ 1 \leq j \leq m$. These distances are used with the distances from the objects to the pivots, stored in the index, $d(x, p_j),\ x \in U,\ 1 \leq j \leq m$, to obtain lower bounds on the distance from each object to the query using the triangle inequality, as explained in Section 2.3.1. That is, for each object $x \in U$, if $d(q, x) \geq |d(q, p) - d(x, p)| > r$, for any pivot $p \in P$, the object is directly discarded from the result set.

After processing the information stored in the index, the objects that could not be discarded from the result set are directly compared with the query object to make up the final result set.

3.5 Index structure and storage

The simplest way of storing the distances from objects to pivots is a table with as many rows as objects in the database and as many columns as pivots. However, due to the amount of information stored in the index by pivot-based methods, if the information stored in the index is processed in that way, the extra CPU time for loading the index and processing it can be significant. Several optimizations of this processing are possible depending on how the index is stored in secondary memory.

A first improvement can be obtained if the index is processed column-wise instead of row-wise [Chávez et al., 2001b]. The query object can be compared with the first pivot p_1 to discard as many objects as possible from the result with this first comparison. If the rows corresponding to each object of the collection are ordered by the distance of each object to the first pivot, the first list of discarded objects can be obtained with two binary searches.

Methods that follow this alternative put in the first place the pivot they evaluate as the best one, sort the objects in the database by their distances to this pivot, and they store in the table the distances to the rest of pivots. Thus, binary search is only possible for the first pivot.

Storing the distances in a table has another disadvantage when the method is dynamic. When an object is added to the database, a row is to be added to the table. When a new pivot is selected, a column has to be added to the table. Either if the table is stored by rows or columns, one of the two cases will be a problem for the growth of the index.

The solution we propose for storing the distances from objects to pivots in dynamic pivot-based methods consists in storing the distances from one pivot to the objects in the collection in a B-tree, taking as keys the distances and as values the identifiers of the objects. That is, each pivot has its own B-tree in which the distances from the pivot to the rest of objects are stored. The index can then be seen as a list of B-trees. This structure has several advantages:

- The list of objects discarded by each pivot can be obtained with two accesses to the B-tree it has associated. The final candidate list could then be obtained as the intersection of the candidate lists of each pivot. Thus, instead of processing the $O(n \times m)$ distances stored in the table, the final candidate list can be obtained by checking only $O(m \times \log n)$ distances.

- Using B-trees solves the problems of growth of the index, either when an object is added to the database or when it becomes a new pivot.

- When working with very large databases the amount of information stored in the index can be too high. This structure is also suitable for the application of a technique called scope coarsening [Chávez et al., 2001b], which consists in storing the distances with less precision in order to reduce the space requirements of the index. The experimental exploration of this possibility remains as future work.

  Some methods apply techniques of scope coarsening to reduce that amount of information [Chávez et al., 2001b]. Scope coarsening consists in storing only for each object the distances to its most promising pivots. This structure is also suitable for this technique.

These issues are important when working with very large databases and the system has to deal with a significant amount of queries in each unit of time.

3.6 Experimental evaluation

In this Section we present the results and conclusions extracted from the experimental evaluation of Sparse Spatial Selection.

First, we present our results about different aspects of the behavior of the method in terms of the complexity, size, and intrinsic dimensionality of the collection. Then, we present the results obtained from the experiments carried out in order to evaluate the efficiency obtained with our method in comparison with a random pivot selection and existing techniques.

Appendix C describes with detail the experimental environment in which the experiments were carried out.

3.6.1 Our hypothesis

Pivot selection and growth of the collection

SSS selects a set of pivots well distributed in the space without being necessary to state beforehand how many pivots the algorithm has to select: it selects as many pivots as necessary for adequately covering the space. As we pointed out in Section 3.2, this number of pivots depends on the complexity of the collection, on its actual content, but not on its size. That is, once the space is covered with a good set of pivots, no more pivots will be selected despite how much the collection grows.

The growth of the size of the set of pivots in terms of the growth of the collection is an important aspect. If the number of pivots selected does not stop in some point as the collection grows, the internal complexity can unbalance the trade-off between the internal and external complexities, and the memory requirements for storing the information of the index can become unacceptable too.

We tested SSS with collections of synthetic vectors of dimension 8, 10, 12, and 14, uniformly distributed in hypercubes of side 1: VECTOR8, VECTOR10, VECTOR12, and VECTOR14 respectively (they are all described in Appendix C). For each collection, we obtained the number of pivots selected by SSS for different sizes of the collection. The results obtained are shown in Figure 3.3.

As we can see in Figure 3.3, the number of pivots grows quickly when the first objects are inserted into the collection, and that the number of pivots moderates its growth as the collection becomes larger, until it finally converges to a stable value when all the space is covered with pivots. Although the number of pivots selected can seem to be high for these collections, we have to take into account the dimensionality of each of them.

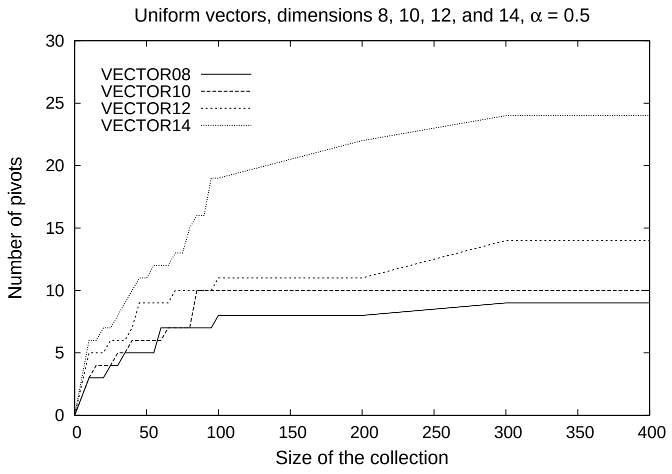


Figure 3.3: Pivots in terms of the size and dimension of the collection.

The second aspect we can see in these results is that our method selects more pivots for the collections of higher dimensionality. This result is coherent with our analysis, since the higher the intrinsic dimensionality, the more difficult the space is to cover, and the more difficult the search is.

Controlling the density of pivots

Although in our method it is not necessary to state beforehand the number of pivots to select, we have to set the value of the parameter α, and the number of pivots selected by Sparse Spatial Selection depends on this parameter. The parameter α controls the *density* of pivots used to cover the space. The higher the value of α, the less pivots selected, since the distance that must separate each pivot from the rest of them is larger. Therefore, different values of α led to different trade-offs between the internal and external complexities during the search.

We run Sparse Spatial Selection on collections of $100,000$ vectors from VECTOR8, VECTOR10, VECTOR12, and VECTOR14 with values of α ranging from 0.30 to 0.50. Figure 3.4 shows the search performance we obtained for each configuration of the method in each collection.

As we can see in Figure 3.4, the optimal results are always obtained for values of α between 0.35 and 0.40, and the search performance is virtually the same for all the values of α included in this interval. We can also observe that when $\alpha > 0.40$, the number of evaluations of the distance function is higher in the spaces with higher dimensionality. The reason for this result is that an increment in the

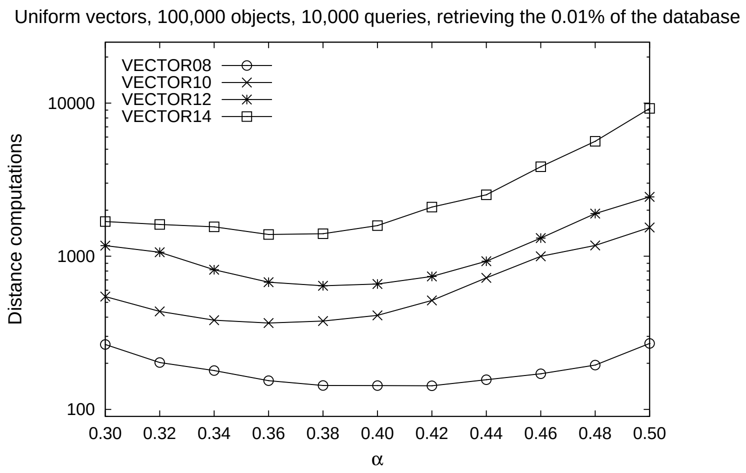


Figure 3.4: Search cost in terms of α and the dimension of the collection.

value of α means a reduction in the number of pivots, and the degradation of the search performance due to having less pivots becomes more important in the most complex spaces.

Effect of the order of objects

As we pointed out in Section 3.2, if the construction of the index is dynamic, the first object inserted in the database becomes the first pivot, and if the construction is static, the first object can be chosen at random.

Immediately we can wonder if the choice of the first pivot, or the order in which the rest of objects inserted into the collection are processed, can affect the final result in search performance or in the number of pivots selected, since SSS goes on in each step with the information it has, and never goes back.

In order to analyze the dependency of the search performance on the order in which the objects of the database are processed, we performed several experiments using the real data collections we call ENGLISH, and NASA (a detailed description of these test collections is provided in Appendix C). We run SSS indexing the 90% of the collection with 10 different orders for each value of α between 0.30 and 0.50, recording for each run the number of pivots and the number of distance evaluations for solving the remaining 10% objects used as queries.

In these experiments we worked with real collections of data, since the objects in them do not have a regular distribution that could hide the effects of the order in which the objects are processed.

α	ENGLISH				NASA			
	Pivots		Search cost		Pivots		Search cost	
	μ	σ	μ	σ	μ	σ	μ	σ
0.30	1557.70	11.91	1590.25	11.76	372.40	4.92	436.22	4.18
0.32	1558.40	11.86	1591.25	11.76	270.90	3.78	339.28	3.63
0.34	813.10	5.57	855.71	5.19	201.80	4.92	276.95	4.92
0.36	813.10	5.57	855.71	5.19	156.20	4.37	235.51	2.90
0.38	813.10	5.57	855.71	5.19	119.50	5.58	205.37	4.72
0.40	404.30	8.10	474.38	6.54	95.50	4.81	187.39	3.57
0.42	404.30	8.10	474.38	6.54	79.10	3.73	178.08	3.52
0.44	204.70	5.36	**354.43**	5.15	64.70	3.06	**171.16**	4.12
0.46	204.70	5.36	354.43	5.15	55.50	2.64	168.07	5.44
0.48	107.60	4.48	506.73	36.60	48.50	3.10	171.19	4.58
0.50	107.60	4.48	506.73	36.60	38.40	2.55	173.57	7.65

Table 3.1: Effect of the order of objects in pivot selection.

For the collections ENGLISH and NASA, Table 3.1 shows the mean and typical deviation of the number of pivots selected and the number of distance computations needed for solving a query, for values of α between 0.30 and 0.50. For both parameters, the mean and typical deviation were obtained when running SSS 10 times on each collection, processing the objects in a different random order each time. In this table we can see how the number of pivots selected decreases when the value of α becomes larger. As we can see observing the typical deviations of each parameter, the effect of the order in which the objects are processed does not introduce a significant deviation in the final results.

Effect of the intrinsic dimensionality of the space

In the description of the method (see Section 3.2) we pointed out an important hypothesis: the set of pivots selected by SSS depends on the topology and characteristics of the space, and not on its size. That is, the set of pivots is adapted to the complexity and distribution of the space. In order to validate this hypothesis, we carried out experiments to study the behavior of the method in terms of the intrinsic dimensionality of the space.

Particularly, we tested SSS with two different collections of words, ENGLISH and SPANISH. Both are collections of words taken from natural language but, according to the estimator of the intrinsic dimensionality $\rho = \mu^2/2\sigma^2$ [Chávez et al., 2001b],

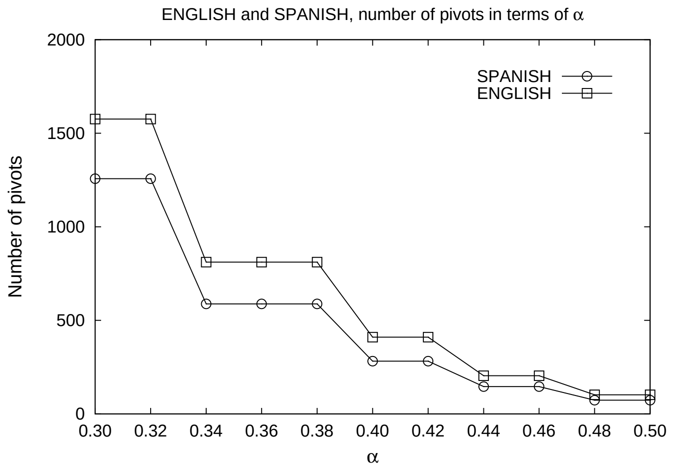


Figure 3.5: Pivots in terms of α in ENGLISH and SPANISH.

they have different intrinsic dimensionality. That is, one of them is more complex than the other. They also differ in the number of objects they contain. For each collection, we obtained the number of pivots selected by SSS and the number of object comparisons needed to solve a query, for different values of α.

Figures 3.5 and 3.6 show the number of pivots selected and the number of evaluations of the distance function for both collections respectively. Since SPANISH contains more objects than ENGLISH, we could expect to need more pivots in order to index the first collection. However, the intrinsic dimensionality of ENGLISH is higher and is therefore the collection is more complex.

In Figures 3.5 and 3.6 we can see that, for the same values of α, the algorithm selects a higher number of pivots in the case of ENGLISH, necessary to cover all the space. However, the results obtained about the search performance are virtually the same for both collections, independently of their size and the number of pivots used to index each of them.

The results of this experiment show that SSS indexes the space not depending on the size of the collection, but on the complexity of the search space.

3.6.2 Efficiency

Comparison with random selection

A first evaluation of the search performance of Sparse Spatial Selection was obtained by comparing it with a random pivot selection, in several synthetic and real

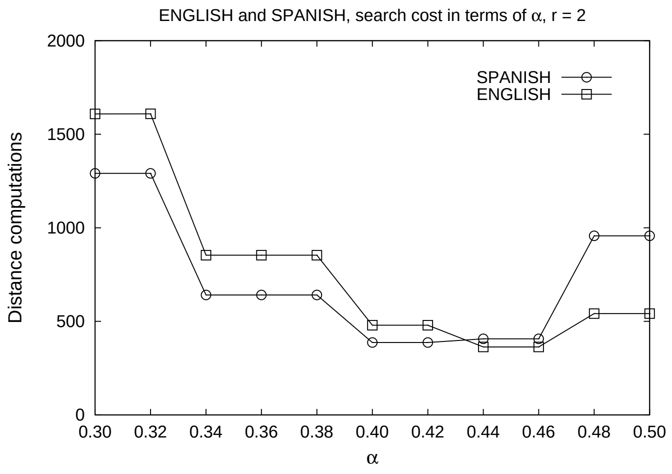


Figure 3.6: Search cost in terms of α in ENGLISH and SPANISH.

collections of data: VECTOR8, VECTOR10, VECTOR12, VECTOR14, ENGLISH, SPANISH, and NASA.

For each collection, the 90% of the objects were used as the database to be indexed, and the 10% of the objects were used as queries. In each experiment, we obtained the average number of distance computations needed to solve a query. In collections of vectors, the search range r was adjusted to retrieve an average of the 0.01% of the objects of the database in each query, and in collections of words, the search range used was $r = 2$ (as usual in most works dealing with distances among words).

Figures 3.7, 3.8, 3.9, and 3.10 show the results obtained for the collections of uniformly distributed vectors of dimensionality 8, 10, 12, and 14, respectively. As we can see in the results, the number of comparisons is higher for collections of higher dimensionality, but SSS always performs better than a random pivot selection, independently of the dimensionality of the space.

Table 3.2 shows the mean and typical deviation of the number of distance computations needed for solving a query, for each collection of vectors. As we can see in these results, the typical deviation is smaller with our method, which is coherent with our hypothesis. That is, not only SSS is always better, but its behavior is also more stable (smaller σ) than when pivots are selected at random. This parameter can be important in systems that receive a large number of queries in each unit of time.

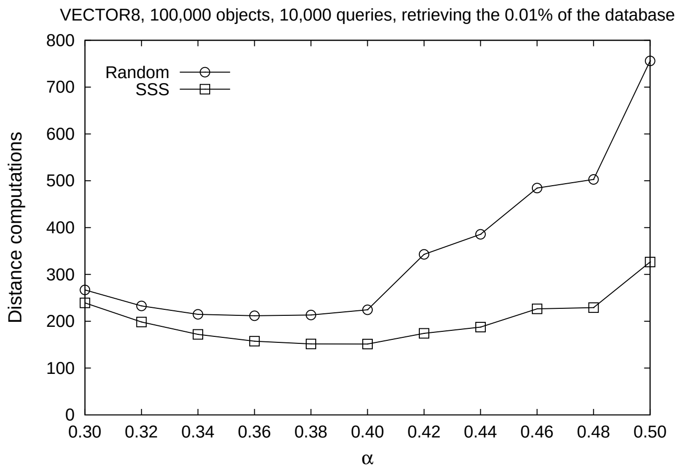


Figure 3.7: Search cost with Random and SSS, collection VECTOR8.

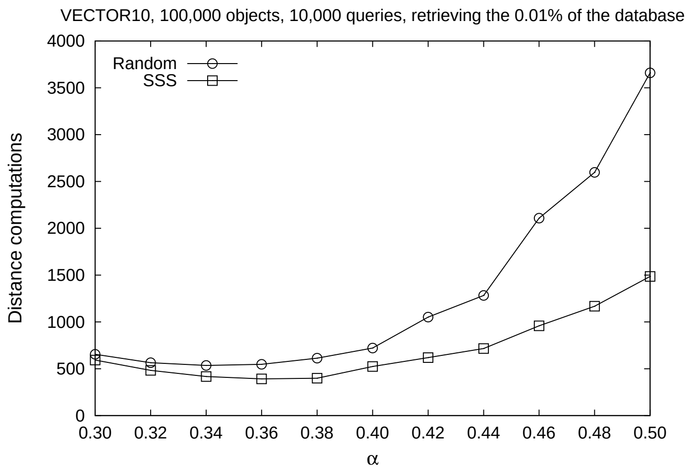


Figure 3.8: Search cost with Random and SSS, collection VECTOR10.

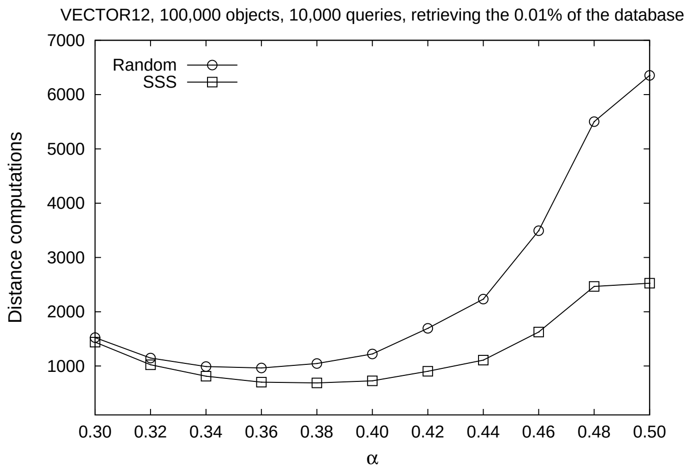


Figure 3.9: Search cost with Random and SSS, collection VECTOR12.

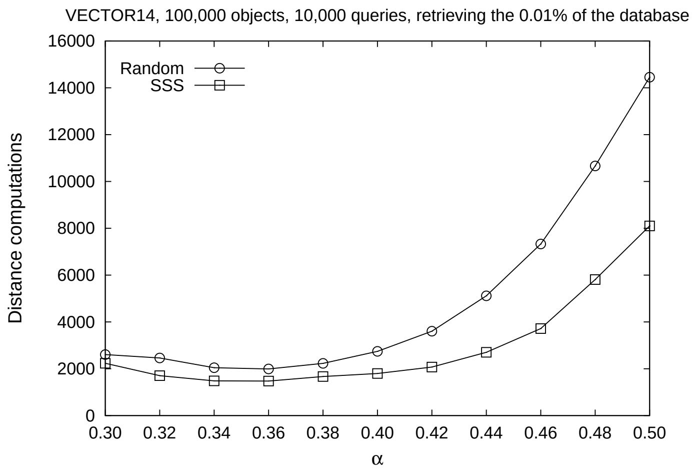


Figure 3.10: Search cost with Random and SSS, collection VECTOR14.

Method	VECTOR8		VECTOR10		VECTOR12		VECTOR14	
	μ	σ	μ	σ	μ	σ	μ	σ
Random	224	53	581	166	1046	316	2087	622
SSS	151	33	390	101	689	193	1452	399

Table 3.2: Variation of the search cost in uniformly distributed vectors.

Comparison with other techniques for pivot selection

We compared SSS with previous techniques to evaluate its competitiveness in what refers to distance computations. We selected the following techniques to compare our method with: Incremental [Bustos et al., 2001], MaxMin [Vleugels and Veltkamp, 2002], SFLOO [Hennig and Latecki, 2003], and Spacing [van Leuken et al., 2006]. We do not considered Maximum Pruning nor Maximum Variance [Venkateswaran et al., 2008, Venkateswaran et al., 2006] since they use the search range in the indexing phase, and the comparison would not be fair. For the comparison we used two real data collections of different nature: ENGLISH, a collection of words, and NASA, a collection of images.

As usual, for each collection, the 90% of the objects of the collection were used as the database to be indexed, and the remaining 10% objects were used as queries. We obtained the average number of distance computations needed for solving a query, using different numbers of pivots. The results are shown in figures 3.11, and 3.12 respectively, as the number of distance computations in terms of the number of pivots used by each algorithm. Tables 3.3 and 3.4 list the relevant subset of the results shown in each figure. The first five columns show the results obtained with previous techniques for fixed numbers of pivots. The last three columns show the results obtained with different values of the parameter α with SSS.

In the collections ENGLISH and NASA, SSS and MaxMin [Vleugels and Veltkamp, 2002] obtain the best search performance. SSS obtains a better result in ENGLISH and MaxMin obtains a better result in NASA, although the difference between them is not very significant.

Although the differences in search performance between Sparse Spatial Selection and previously proposed methods are not very significant, our method is clearly better than the previous ones in other operations and features, important for the operation of a real database system. As we pointed out in previous sections, SSS is the only dynamic pivot-based method, it adapts its structure as the database grows. It obtains the smaller cost for the insertion of an object and the selection of pivots, since the only information it needs is the needed for inserting an object into the database.

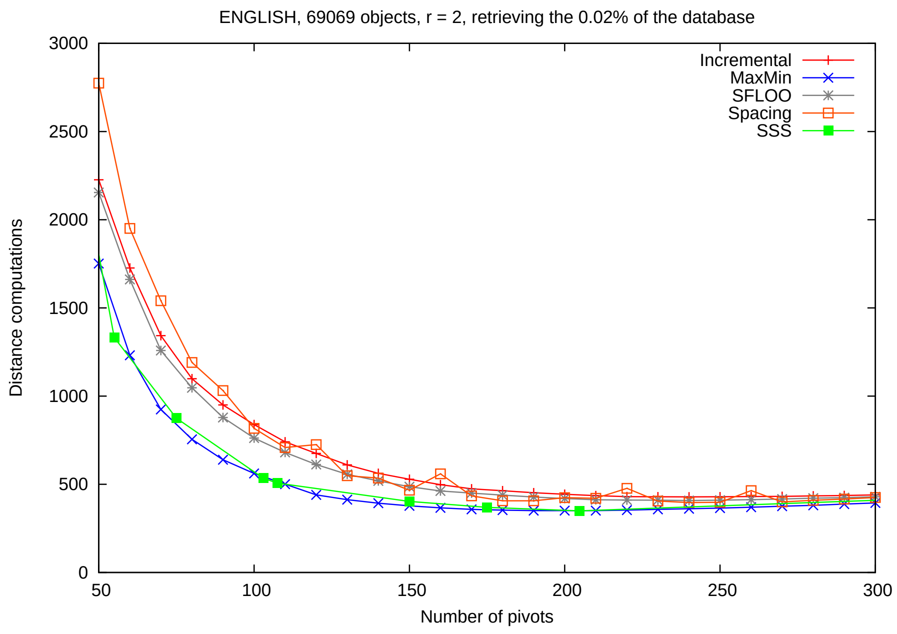


Figure 3.11: Comparison with other techniques, collection ENGLISH.

Piv.	Increm.	Maxmin	SFLOO	Spacing	α	Piv.	SSS
20	6957.48	8214.77	7432.28	11608.43	0.8	3	30060.68
40	3025.13	2659.12	3182.53	4421.69	0.76	5	30060.68
60	1726.65	1231.56	1661.63	1950.95	0.72	5	24377.49
80	1098.74	754.86	1047.32	1190.86	0.68	11	13622.29
100	838.88	561.47	762.26	816.83	0.64	18	8257.96
120	675.00	440.23	612.01	725.13	0.62	18	8257.96
140	562.00	392.53	519.21	534.74	0.58	31	3600.80
160	497.77	366.14	461.88	559.31	0.54	55	1332.30
180	463.51	353.36	438.23	406.70	0.50	107	506.73
200	443.35	**350.62**	419.57	424.51	**0.46**	**204**	**349.43**
220	429.84	353.77	410.68	477.21	0.42	404	474.38
240	**428.86**	361.43	**407.67**	**396.68**	0.38	813	855.71
250	429.37	365.17	410.00	397.64	0.34	813	855.71

Table 3.3: Comparison with other techniques, ENGLISH.

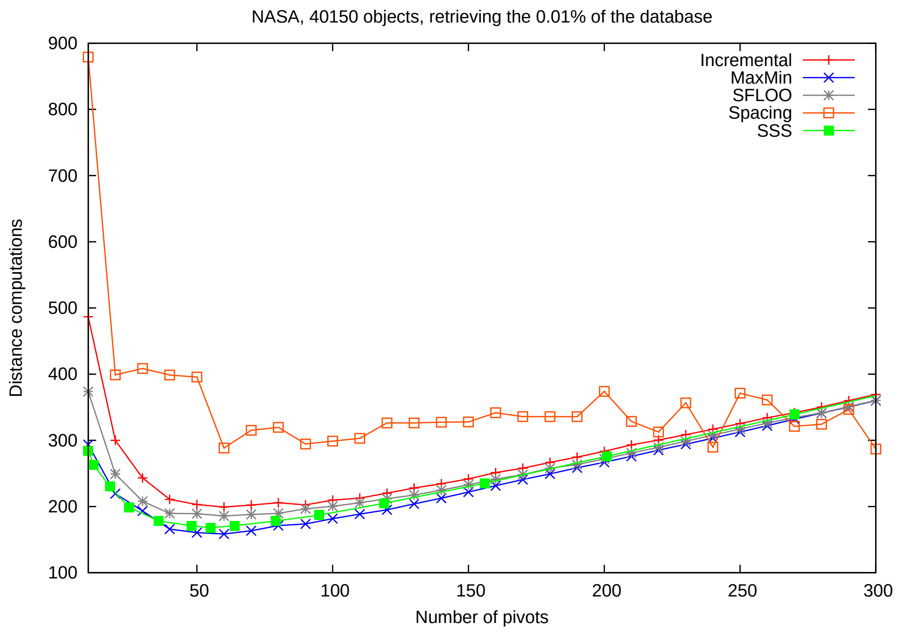


Figure 3.12: Comparison with other techniques, collection NASA.

Piv.	Increm.	Maxmin	SFLOO	Spacing	α	Piv.	SSS
30	243.21	193.24	207.91	408.57	0.64	10	284.97
40	210.78	165.87	189.73	398.72	0.62	12	263.31
50	203.06	160.55	189.05	395.58	0.60	14	270.34
60	**199.23**	**158.52**	**185.64**	**288.47**	0.58	18	230.49
70	202.27	163.42	188.03	315.19	0.56	20	240.11
80	205.75	171.17	189.76	319.61	0.54	25	199.58
90	202.49	173.65	196.59	294.44	0.52	26	213.27
100	209.79	181.77	200.29	298.83	0.50	36	178.40
110	212.72	188.60	205.72	303.10	0.48	48	171.19
120	220.18	195.11	211.85	326.30	**0.46**	**55**	**168.07**
130	228.03	204.00	217.33	326.33	0.44	64	171.16

Table 3.4: Comparison with other techniques, NASA.

3.7 Discussion

In this Section we summarize the main advantages and drawbacks of the method we have proposed in this chapter, in comparison with the characteristics of existing methods for the selection of effective pivots.

Cost of indexing

An advantage of SSS when compared with previous methods is the cost of the selection of reference objects. In other techniques, the selection involves a very high number of distance computations that makes the construction of the index very costly. SSS does not need any extra information for the selection of pivots. When an object is inserted into the database, it has to be compared with all the pivots, and that is the only information that our method needs to determine if a new object becomes a pivot or not.

If an object becomes a pivot, the distances from all the objects in the database to the new pivot it have to be computed and stored, but that distances would need to be computed in any case for inserting the object into the database.

Optimal number of pivots

An important characteristic of SSS is that it is not necessary to state the number of pivots beforehand. The method determines by itself how many pivots it needs for indexing the collection of data. New pivots are selected as needed because the database of objects has grown. The method determines by itself how many pivots are necessary in each moment.

In all previous techniques this number has to be stated before the indexing, and the optimal value for the trade-off between internal and external complexities has to be obtained by trial and error on the complete collection, which inevitably makes the index static.

Dynamic and adaptive

Being dynamic and adaptive are another two important advantages of SSS over previous methods. All previous techniques for pivot selection are static, thus forcing the users to have a complete database before the indexing phase, and limiting the insertions or removals of objects after the index has been built, since they can degrade the performance of the structure.

SSS starts the construction of the index from an empty database. As new objects are inserted, SSS adapts the structure of the index and the information it stores as needed when new objects are inserted into the database and new regions of the space appear. The index is also adapted when an object is removed, removing a pivot if necessary and therefore conserving the trade-off between the internal and external complexities in the search performance.

Search performance

In the experimental evaluation provided in Section 3.6, we showed that SSS is always more efficient than a random pivot selection, and that it is a competitive technique when compared with previous proposals in data collections taken from real problems.

Particularly, we have compared Sparse Spatial Selection with the previous techniques described in Section 2.6: random selection, MaxMin, SFLOO, Incremental, and Spacing. In the experimental comparison we used both synthetic collections of data with which specific parameters of the method were evaluated, and also collections of data extracted from real applications.

Parameter tuning

A possible drawback of our method is the parameter tuning. Although in most scenarios is possible to analytically obtain the value of the maximum distance, or that it can be approximated on a first sample of objects, it can be seen as a problem in some domains. The use of the parameter α can be seen also as a disadvantage, although our experimental results show that it also gives the best results, or results very near to the best for values of α in values from 0.30 to 0.40.

3.8 Summary

In this chapter, we have presented Sparse Spatial Selection (SSS), a new pivot-based method for similarity search in metric spaces. While previous methods select pivots based on the hypothesis that they have to be as far as possible from each other, Sparse Spatial Selection selects a set of pivots well distributed in the space.

An important characteristic of SSS when compared with other methods is that it is dynamic. That is, the method starts with an initially empty database and pivots are selected or removed and the index is adapted as necessary when objects are inserted or removed from the database. Another advantage of Sparse Spatial Selection when compared with previous techniques is the cost of the construction of the index. When an object is inserted in the database, it has to be compared with the already selected pivots to index it. Our method does not need any further information in order to decide if this new objects becomes a new pivot or not. Therefore, the overhead introduced by the pivot selection is 0, the cost for the selection of pivots is minimum.

In this chapter we presented the algorithms for inserting and removing objects in the database, and discussed possible data structures for the construction of the index, suitable for its dynamic nature.

The chapter also presents the results and conclusions taken from the experimental evaluation. It has been shown that the number of pivots selected by the algorithm does not grow infinitely, it stops growing when the set of pivots fully covers the space. It also has been shown that the optimal values of the parameter α are in the range $[0.35, 0.45]$, and that the search cost is virtually the same for all values in that range. The results also show that the order in which the objects are processed does not affect significantly the final search performance. Finally, the experimental evaluation shows that SSS is as efficient as or better in most cases than previous techniques.

Chapter 4

Sparse Spatial Selection Tree

4.1 Overview of the chapter

In clustering-based methods, the specific set of objects used as cluster centers directly determines the structure of the index and its capacity for pruning the search space. Although most methods select the cluster centers at random, the specific set of objects used as cluster centers directly determines the effectiveness of the index during the search. Other important aspect of clustering-based methods is the structure of the index. In most cases it is a completely balanced structure that may not adequately fit the irregular distribution of the data in the space.

In this chapter we provide an analysis of the issues related to the selection of effective cluster centers and the use of unbalanced index structures as a way to improve the search performance obtained with clustering-based methods.

We present *Sparse Spatial Selection Tree* (SSSTree), a new clustering-based method for similarity search. In each level of the tree, it selects as many cluster centers as necessary to cover the space and obtain clusters as compact as possible. In addition, the index structure is completely unbalanced. That is, each node of the tree will have as many children as necessary, and some branches can grow while others have already stopped depending on if a region of the space is worth of further partitioning or not.

Section 4.2 presents our analysis of the effect of cluster center selection and the use of unbalanced index structures for clustering based methods respectively. Section 4.3 presents Sparse Spatial Selection Tree, the algorithms for index construction and searching, and other issues related to how the cluster centers are selected and possible criteria for stopping the partitioning of the space. Section 4.4 presents the experimental evaluation of the method.

4.2 Issues related to the selection of cluster centers

Before introducing our analysis of the issues related to the selection of cluster centers in clustering-based methods, it is worth to remind some important aspects about how these methods proceed to prune the search space and thus reduce the number of comparisons needed for solving a query.

Clustering-based methods select a subset of objects from the database as cluster centers, $c_1, \dots, c_m$. Then, each object in the database is assigned to the cluster corresponding to its nearest cluster center, dividing the space into a set of disjoint clusters, $C_1, \dots, C_m$:

$$C_i = \{x \in U,\ d(x, c_i) \leq d(x, c_j),\ 1 \leq j \leq m\}$$

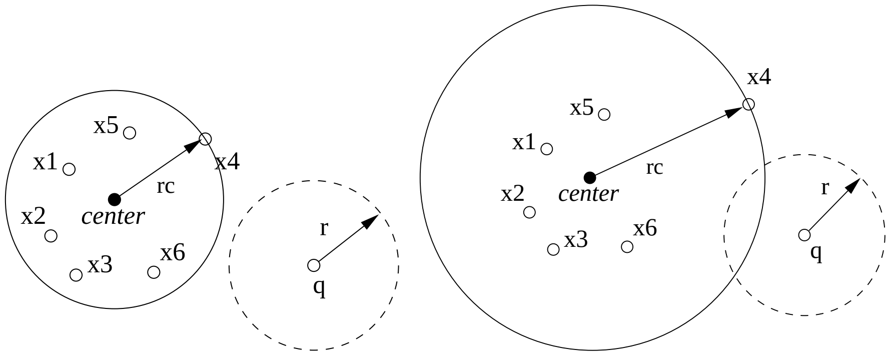
The clusters are disjoint, and the result of the union of them is the complete database. This division of the space is called a Voronoi partition. The information stored in the index for each cluster usually consists in the cluster center c_i and the covering radius of the cluster r_{c_i}, which is the distance from the center to its furthest object in the cluster.

The center of the cluster and the covering radius define a *ball* that encloses the cluster. It is important to note the difference between the cluster (a set), and its enclosing ball defined by the center and the covering radius. A query (q, r) also defines a ball in the space, with center q and radius r. Therefore, the result set must contain all the objects in the database that fall inside the ball.

A cluster is directly discarded without comparing the query with any of the objects it contains if the intersection of the ball enclosing the cluster and the ball defined by the query is empty. When given a query (q, r), the cluster C_i is discarded from the result if:

$$d(q, c_i) > r_c + r$$

The probability of discarding a cluster from the result depends on its size (that is, on its covering radius). The larger a given cluster is, the more the chances that its enclosing ball intersects the query ball. Therefore, if the partition creates small clusters, the probability of discarding them increases. However, the query object would have to be compared with too many cluster centers, which increases the internal complexity. Most clustering-based methods (see Section 2.5.2) address this problem by recursively partitioning the space. That is, the space is first partitioned in large clusters that are then recursively partitioned in smaller clusters and so on. If a cluster is not discarded from the result, it is further processed instead of comparing the query with all the objects it contains.



(a) Compact cluster (b) Non-compact cluster

Figure 4.1: Example of compact (a) and non-compact (b) clusters.

Even in methods that recursively partition the space, the effectiveness of the partition for pruning the search space depends mainly on the compactness of each cluster, and on the overlapping between the balls defined by each cluster:

- A cluster is said to be *compact* if its associated ball does not have wide regions without any object. Figure 4.1 shows examples of compact and non-compact cluster. As we can see in the figure, the compact cluster has more chances to be discarded from the result if it does not contain any object in the result, since it is less probable that its ball intersects the ball defined by the query.

- The overlapping between the balls of clusters also affects the capacity of the index for pruning the search space. If the balls of two clusters intersect, it is more difficult to discard one of them from the result. Figure 4.2 shows an example. In the step of the search shown in the figure, only the cluster C_1 can be discarded from the result. If the intersection between C_2 and C_3 were empty, one of them could be pruned from the search space.

Problems of randomly selected cluster centers

Most clustering-based methods select the cluster centers at random. As happens with the selection of pivots in the case of pivot-based methods, this approach presents several inconveniences:

- First, in order to avoid the non-empty intersection between the enclosing ball of each cluster and the query balls, the cluster centers should be selected in such a way that the clusters are as compact as possible. Clearly, random selected cluster centers do not guarantee the resulting clusters to be compact.

If an object is not really near of its nearest center, this will cause the enclosing ball of the cluster to be much larger than it should. This increases the possibilities of the enclosing ball of the cluster to have an intersection with the ball defined by the query.

- Second, with a random selection of the cluster centers, there are no guidelines for determining the optimal number of centers. Too much centers led to more compact clusters, but also to too much comparisons of the query with cluster centers (increment of the internal complexity). Few cluster centers can cause the enclosing balls of each cluster to be larger than they should. Thus, the optimal number of cluster centers should be determined by trial and error. Again, this inevitably makes the index static.

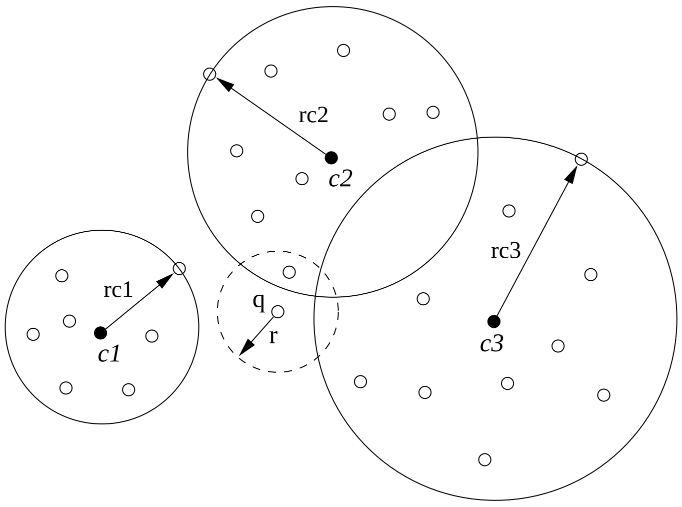


Figure 4.2: Search pruning and overlap between clusters.

Recursive tree-like indexes try to address this problem by recursively partitioning the space. For instance, in the root of the tree the space is partitioned in four clusters, and each of them is further partitioned in four new clusters in the next level of the tree and so on. This strategy obviously helps to obtain compact clusters, but it could be improved by following some strategy to improve the quality of the clusters in each level.

Another aspect of all clustering-based methods is that the number of clusters created in each node is always the same. Again, this can cause the partition on each level to be far from the optimal.

4.3 SSSTree: Sparse Spatial Selection Tree

Most clustering-based methods, as BST [Kalantari and McDonald, 1983], GHT [Uhlmann, 1991], GNAT [Brin, 1995], VT [Dehne and Noltemeier, 1987], or M-Tree [Ciaccia et al., 1997] (all of them described in Setion 2.5.2), create tree-like indexes in order to recursively partition the space. They differ mainly in the structure of the tree and the information about the partition they store in each level of the tree. Something that most of them have in common is that the trees are balanced structures. That is, all the nodes of the tree have the same number of children nodes, and usually all the leaf nodes are all at the same level.

In our opinion, since real collections of data present biases and irregular distributions, this approach trying to index an irregular space with a balanced (regular) data structure is not a good alternative. By contrast, an index structure which does not impose a rigid structure can adapt better to the irregular distribution of the objects in the data space, and obtain therefore more compact and effective clusters.

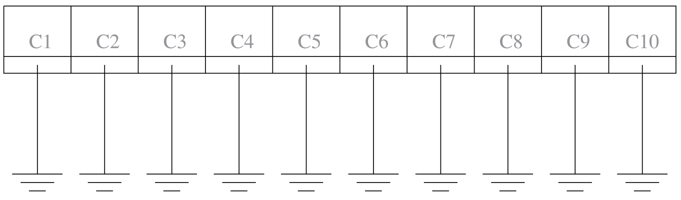
SSSTree is a clustering-based method for similarity search based on a tree structure where the cluster centers of each internal node of the tree are selected applying Sparse Spatial Selection (SSS) (presented in the previous chapter). The structure we propose is unbalanced in order to explicitly adapt to the structure of the space, and tries to select in each level the best possible cluster centers in order to improve the search cost.

Our hypothesis is that, using SSS to select the cluster centers, the partition of the space will be better and the performance of the search operation will be improved. Each node stores the covering radius of its corresponding cluster. Then, that cluster is recursively divided in several clusters, again selecting the cluster centers with SSS. An important difference with methods like GNAT is that SSSTree is not a balanced tree and not all the nodes have the same number of branches. The tree structure is determined by the number of clusters selected by SSS and therefore it depends on the internal complexity of the subspace defined in each node by the objects it contains.

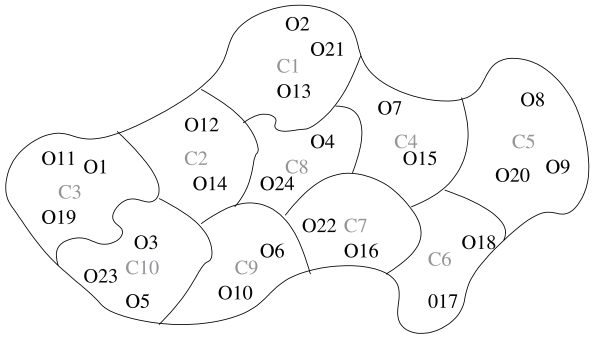
4.3.1 Construction

The construction process starts with all the objects in the database. In the root of the tree, a set of cluster centers is selected applying SSS (the maximum distance can be estimated as in the previous chapter). That is, the number of clusters created in the root of the tree is not stated beforehand. The algorithm selects as many reference objects as necessary to cover the space. All the cluster centers will be at a distance greater than $M\alpha$ from the rest of centers in the first level.

Figures 4.3 and 4.4 show an example of tree construction after the selection of the cluster centers at the root of the tree. As usual, each object is assigned to the



(a) Selected centers in the first level



(b) Decomposition of the space in the first level of the tree.

Figure 4.3: Partition of the space in the root of the tree.

cluster corresponding to its nearest center. For each cluster, the tree stores the cluster center and the covering radius of the cluster.

The process is recursively repeated. That is, each cluster of the first level is partitioned following the same approach, selecting as many cluster centers as necessary applying SSS. The recursive process stops when a cluster has a small number of objects. The minimum number of objects must be established previously (threshold δ). The process stops when a cluster has a number of objects less than or equal to a threshold δ or, alternatively, when the covering radius of the cluster is smaller than a given threshold.

This is the main difference of SSSTree with previous methods. While existing techniques create a balanced tree with the same arity in all the nodes, SSSTree creates a tree structure where the number of children of each node depends on the SSS selection of cluster centers. Each cluster of the first level can be subdivided in a different number of clusters, depending on the number of objects they have, and the distribution of the objects in the subspace.

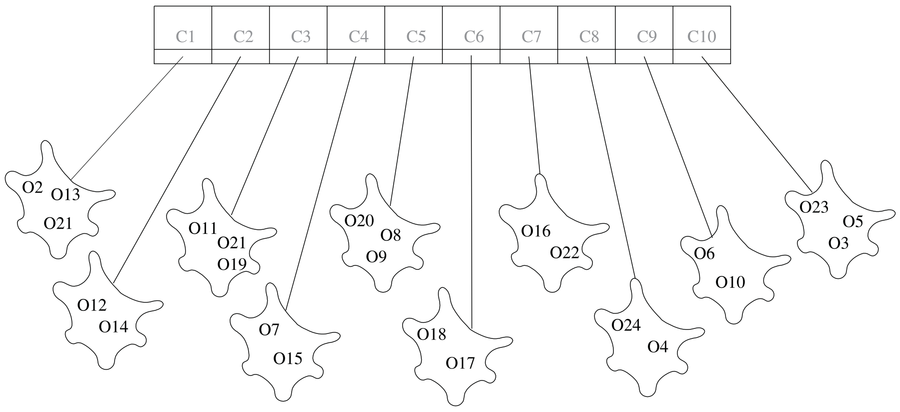


Figure 4.4: SSSTree tree after the first space partition.

In each internal node we have to estimate again the maximum distance in the cluster associated to it (since the maximum distance of the metric space is not valid for each new cluster). The maximum distance can vary even between different clusters in the same level. In the next subsection we explain different ways to efficiently estimate the maximum distance into a cluster. Once the maximum distance inside a cluster is estimated, the new cluster centers are created.

Applying this strategy for the selection of cluster centers, not all the nodes of the tree will have the same number of child nodes. Each cluster is divided in a number of regions which depends on the distribution and complexity of the data of that cluster. This is a very important difference with other structures like GNAT. The index construction adapts the index to the complexity and distribution of the objects of the metric space, and in each level of the tree only those needed clusters will be created. This property is derived from the fact that SSS is able to adapt the set of reference objects to the complexity of each space or subspace.

Estimation of the maximum distance M

One of the problems of the construction process is the need of computing the maximum distance M in each cluster. The naive way to compute this distance is to compare each object in the cluster with all the other objects in the cluster, but this approach is too expensive.

Let M be the maximum distance between any pair of objects of the cluster C_i, and r_{c_i} the covering radius of C_i, the distance from the cluster center to the furthest object in the cluster. Then, we know that $M \leq 2 \times r_{c_i}$. As we can observe in Figure

4.5, if we use $2 \times r_{c_i}$ as the cluster diameter, we can cover the same objects as using M as diameter. Therefore, $2 \times r_{c_i}$ can be used as a good estimation of the maximum distance during the construction process.

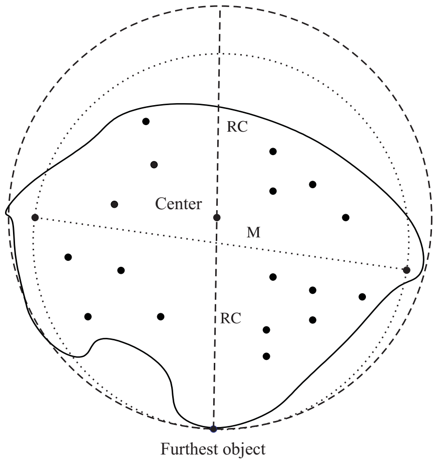


Figure 4.5: Estimation of the maximum distance in each cluster.

Note that with this solution, neither the computation of the covering radius or the estimation of the maximum distance between two objects of the cluster introduce any additional cost. In any level of the tree, once the cluster centers have been selected, each object has to be compared with all the centers in order to assign it to a cluster. The covering radius can be obtained from these distances without additional computation, and therefore, so does the estimation of the maximum distance between two objects in each subcluster.

4.3.2 Searching

When given a query (q, r), the search works as usual. The tree is traversed from the root to the leaves, pruning the search space when possible. In a given node of the tree, the query object is compared with all the cluster centers c_i stored in that node. Using these distances and the covering radius of the cluster corresponding to each center, the algorithm discards from the result all clusters for which:

$$d(q, c_i) > r + r_{c_i}$$

When a branch of the tree is pruned, its corresponding region in the space is discarded from the result. The search algorithm traverses the tree by following the branches that could not be discarded, until reaching the leaves of the tree.

4.4 Experimental results

The performance of SSSTree was tested with several collections of data: a collection of synthetic vectors with uniform distribution in an hypercube of side 1, and the real collections SPANISH (words) and NASA (images). All of them are described with detail in Appendix C.

SSSTree was compared with other well-known clustering-based indexing methods: M-Tree [Ciaccia et al., 1997], and GNAT [Brin, 1995] (described in Chapter 2.5), and EGNAT [Uribe et al., 2006], a modification of GNAT.

Figure 4.6 shows the results obtained when comparing SSSTree with existing methods, with the collection of uniformly distributed vectors. As usual, 90% of the collection was used as the database of objects to be indexed, and the remaining 10% of the objects were used as queries. The figure shows the average number of distance computations needed for solving a query, for different search radius.

We used three different search radius that retrieve for each query the 0.01%, 0.10%, and 1.00% objects of the database respectively. The reason for using different search radius is that, for clustering-based methods, this parameter can affect the results obtained and the comparison of a method with others.

As we can see in these results, SSSTree is significantly more efficient than M-Tree and GNAT in all cases, and also more efficient than EGNAT, for all the search radius. In these results we can see the variation in the search cost in terms of the search radius. Obviously, the larger the search radius, the more difficult to discard objects is for the method.

Figures 4.7 and 4.8 show the results obtained from the comparison of SSSTree with M-Tree, GNAT, and EGNAT, in the collections SPANISH (words) and NASA (images). In the case of the collection SPANISH, we used search radius from 1 to 4, for the same reason as in the previous case. In the case of the collection NASA, we used again three different search radius, that retrieve an average of the 0.01%, 0.10%, and 1.00% of the objects in the database.

The results we obtained are very similar to the obtained for uniformly distributed vectors. SSSTree is systematically more efficient than previous techniques in both collections. In the case of the collection of words, SSSTree is significantly more efficient than all the other methods for all search radius.

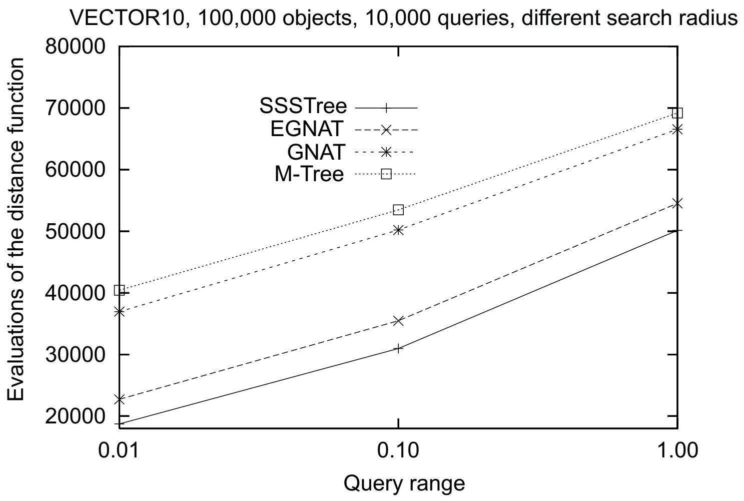


Figure 4.6: Comparison with other methods, collection VECTOR10.

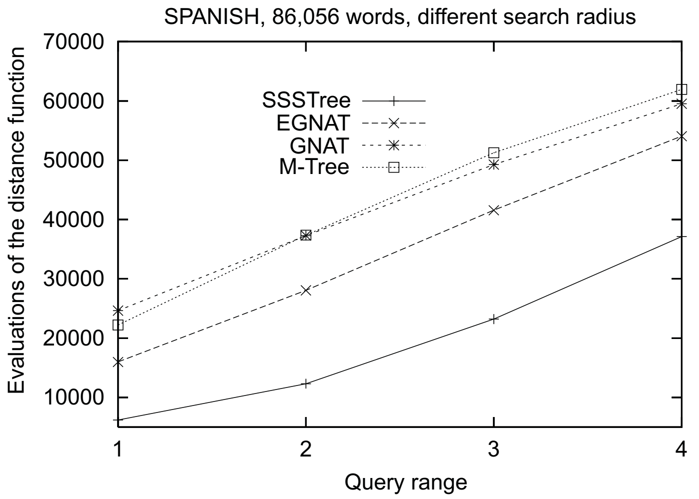


Figure 4.7: Comparison with other methods, collection SPANISH.

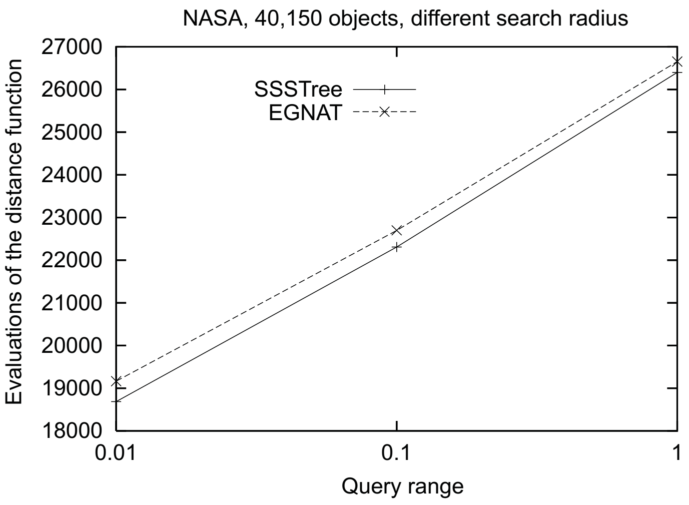


Figure 4.8: Comparison with other methods, collection NASA.

4.5 Summary

In this chapter we addressed the problem of selecting effective indexing objects for clustering-based methods. In Section 4.2 we provided a description of the issues related to this aspect. The main conclusion of that analysis is that, with a random selection and a number of cluster centers fixed beforehand, there is no guarantee of obtaining compact clusters and of their associated balls in the space to have a small overlap. We also analyzed the consequences of the use of balanced or unbalanced tree-like structures in clustering-based methods. In line with the results of our previous analysis, our hypothesis was that the use of balanced structures does not adapt the structure and information stored in the index to the irregular distribution of the data space, present in real applications. That is, a balanced index structure and decomposition of the space is not the optimal configuration for real datasets.

Based on the results of our analysis of the issues related to selecting effective cluster centers and the use of balanced or unbalanced index structures, we proposed a new clustering-based method, *Sparse Spatial Selection Tree* (SSSTree). It is an unbalanced tree-like index that uses in each level the number of cluster centers it considers necessary in terms of the amount of objects in the region of the space it covers and on their distribution in the search space. Thus, the index adapts its structure and its resources to the actual distribution of the data. In Section 4.3 we presented the algorithms of construction and searching.

In Section 4.4 we present the experimental evaluation of the method, in which we compared our method with state-of-art methods. Our experimental results confirm our hypothesis. SSSTree is significantly more efficient than previous methods in both synthetic and real collections, for different search radius in each of them.

Chapter 5

Non-Redundant Sparse Spatial Selection

5.1 Overview of the chapter

In this chapter we present Non-Redundant Sparse Spatial Selection (NR-SSS), an improved version of SSS for the selection of indexing objects. Although an effective set of pivots can be obtained with SSS, it is a greedy algorithm and some of the decisions taken during the selection could be improved as the database evolves. NR-SSS refines the criterion for selecting pivots of SSS at the expense of a higher computational cost of the selection.

When a new object is inserted into the database, it becomes a candidate pivot if it satisfies the selection criterion of SSS. Then, its individual contribution to the overall set of pivots is computed. Based on the contribution of the candidate, it may be discarded, it may be added to the set of pivots, or it may be added to the set of pivots replacing another pivot that is considered worse. The resulting set of pivots is smaller, which reduces the internal complexity of the search, but maintains the capacity for directly discarding objects from the result. As in the case of SSS, the selection is dynamic, that is, the pivots are selected as new objects are inserted an initially empty database.

The rest of the chapter is organized as follows: Section 5.2 presents the details of the motivation for this new method. Section 5.3 presents Non-Redundant Sparse Spatial Selection (NR-SSS): the algorithm for the estimation of the contribution of each candidate pivot and the policies for deciding what to do with each new candidate pivot. Section 5.4 explains how the cost of the selection can be reduced at the expense of a loss in search performance. Finally, Section 5.5 presents the results obtained in the experimental evaluation of the method.

5.2 Motivation

In Chapter 3 we presented Sparse Spatial Selection (SSS) as a new method for pivot selection. As we have seen in our analysis, it is competitive with previous techniques in what refers to search cost (comparisons needed for solving a query), but in addition it is clearly better in other characteristics: it is dynamic, it adapts the index structure to the distribution of the objects in the space, and it has minimum cost for the selection of pivots and the construction of the index.

However, SSS is a greedy algorithm for selecting reference objects. That is, each time an object is inserted into the database, the method decides if it becomes a pivot or not with the information it has up to that moment. Once an object is selected as a pivot, the method does not go back on that decision. In some moment as the database evolves, some of that decisions can become "erroneous" in some way. Two situations are likely to arise:

- An object could be selected as a pivot without being necessary, that is, the set of pivots could achieve the same search performance without that pivot because other pivots do the same work it does, because the objects that are discarded by that pivot can also be discarded by other pivots of the set. In this case, comparing the query with that pivot is useless, and it increases the internal complexity.

- It is possible that an object was selected as a pivot when another object inserted later could be more effective to cover a given region of the space. That is, an object selected as a pivot can prevent another better object to become a pivot. In this case the problem is not only that the internal complexity is increased. The selection of the first pivot prevents to reduce even more the external complexity.

Therefore, by analyzing the set of pivots obtained with SSS when the database has a significant number of objects, it is possible to find pivots that are in some way *redundant*, that is, pivots such that the set of pivots would obtain the search performance without them.

Figure 5.1 shows an example of this situation in a two-dimensional scenario. Lets suppose that the set of points within the rectangle shown in the figure is the universal set, that they are compared using the Euclidean distance, and that the database contains the points shown in the figure, that is, $\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9\}$. The maximum distance between two points is given by the diagonal of the rectangle. With $\alpha = 0.4$, the set of pivots selected by SSS could contain the objects $\{x_1, x_2, x_3, x_4\}$, shown in bold in the figure (we do not shown the regions covered by each pivot in the figure for the sake of clarity in the example). Therefore, the index stores the distances from each object x_i to each of these four pivots.

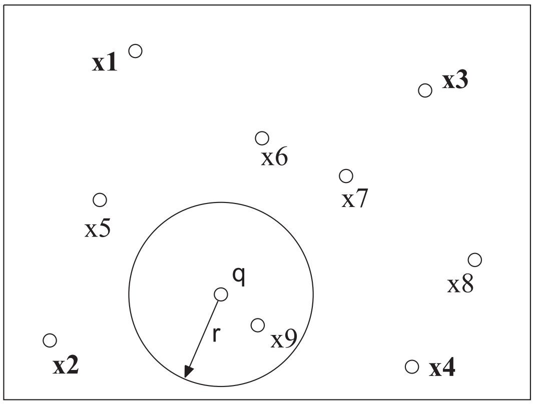


Figure 5.1: Example of *redundant* pivots with Sparse Spatial Selection.

When given the query (q, r), the query object is compared with all the pivots in order to directly discard as many objects as possible from the result. If only the objects x_1 and x_2 were used as pivots, the object x_5 would not be discarded from the result, since its distance to any of those pivots is the same as the distance from the query object q to them (the lower value of the distance from x_5 to q would take a value close to 0). Similarly, if only the objects x_1 and x_3 were used as pivots, the object x_6 would not be discarded from the result for the same reason: its distance to a pivot is more or less the same as the distance from the query to that pivot. However, if the set of pivots contains any subset of three objects from $\{x_1, x_2, x_3, x_4\}$, those pivots would be enough for perfectly locating the position of any point in the space, and thus discard from the result set the objects that are not contained in it.

This is an example in which SSS has selected more pivots than the really needed. Some of the four pivots selected by SSS is redundant in some way, that is, one of the four pivots could be removed from the set of pivots used in the index, and the capacity of the set of pivots for discarding objects from the result would remain the same. This situation unnecessarily increases the internal complexity of the method.

This is a consequence the greedy approach followed by SSS. The method obtains a good set of pivots adapted to the distribution of the space, and in addition the selection is dynamic and with minimum cost. However, if the application admits a higher cost for the selection, the set of pivots obtained by SSS could be refined with further processing of the collection. That is, it is possible to remove some pivots from the set and still conserve its capacity for discarding objects from the result. This gives as a result a smaller set of pivots, which means a reduction of the internal complexity and the space requirements of the index.

If the system can deal with more costly insertions of objects and reasonably small reorganizations of the index, it would be possible to modify the selection criteria in order to identify and remove the redundant pivots, that is, pivots that do not contribute to the overall set of pivots. In this Chapter we present Non-Redundant Sparse Spatial Selection (NR-SSS), a new dynamic technique for the selection of indexing objects that follows this approach.

5.3 Pivot selection and index construction

This section describes Non-Redundant Sparse Spatial Selection (NR-SSS): the criterion and algorithms for evaluating the individual contribution of a pivot, the policies for adding it to the set of pivots in terms of its contribution, and the procedures for pivot selection and index construction.

5.3.1 Estimation of the contribution of a pivot

Let (X, d) be a metric space, $U \subseteq X$ a database of size n, and $P = \{p_1, \ldots, p_m\}$ a set of $m = |P|$ pivots. When a candidate pivot p_{m+1} is selected, the method evaluates the contribution of each pivot in $P \cup \{p_{m+1}\}$ as follows. A set of A pairs of objects from the collection $Pairs = \{(x, y), x, y \in U\}$, $A = |Pairs|$, is selected as a sample of pairs object-query. The set of A pairs is a good sample of queries and objects to be discarded, assuming that the distribution of the queries in the space is similar to the distribution of the objects of the database in the space.

For each pivot p_i and pair (x_j, y_j), the lower bound on the distance $d(x_j, y_j)$ is computed as:

$$d(x_j, y_j) \geq |d(p_i, x_j) - d(p_i, y_j)|$$

We consider that the best pivot p_{max} for each pair (x_j, y_j), the one that obtains the higher lower bound on the real distance, since it is the one with more chances for discarding the object x_j when the y_j object is used as a query or vice versa. The algorithm obtains also the second best pivot for each pair, p_{max2}, as the pivot maximizing the lower bound on the real distance if p_{max} where removed from the set of pivots. The contribution of p_{max} for the pair (x_j, y_j) is defined as:

$$|d(x_j, p_{max}) - d(y_j, p_{max})| - |d(x_j, p_{max2}) - d(y_j, p_{max2})|$$

We consider that the contribution of the other pivots in $P \cup \{p_{m+1}\}$ for this pair is 0. That is, for each pair of objects, all the pivots compete to be the best, and only the one that obtains the best lower bound is given a score that is added to compute its overall contribution for the set of A pairs of objects. Note that by computing the contribution of the best pivot as the difference of its lower bound and the lower bound of the second best pivot, the contribution does not depend on the real value of distance $d(x_j, y_j)$.

The total contribution of a pivot $p_i \in P \cup \{p_{m+1}\}$ is the sum of its contributions for the A pairs of objects. Note that a pivot that has contribution equal to 0 does not maximize the distance in the pivot space for any pair of objects. In that case, we consider that the pivot is redundant (at least for those A pairs), since other elements in the set of pivots can discard the objects it discards.

For each pair (x, y), we consider that the best pivot is the one maximizing the lower bound on the real distance $d(x, y)$, and this is the only pivot that gets a positive score for this pair of objects. However, it is important to note that this does not mean that it is the only pivot able of discarding the object x against the object y. The second, third, or even all the pivots may be able to discard it from the result. Therefore, removing a pivot with a contribution different than 0 does not necessarily mean that the objects for which it is the best pivot can not be discarded by any other pivot. It is also important to note that, since the contribution of the pivots is computed on a sample of queries, the results obtained are an approximation of their real contribution when solving real queries.

5.3.2 Pivot selection

Both the database and the set of pivots are initially empty, and the index is built as new objects are inserted in the database. When an object $x \in X$ is inserted in the database, the algorithm checks if it becomes a candidate pivot applying the criterion of SSS. If the distance from x to the pivots already selected is greater or equal than $M\alpha$ (where M is the maximum distance between any two objects and α a constant parameter that takes values between 0 and 1, usually between 0.35 and 0.45), the object x becomes a candidate pivot.

If the number of pivots is smaller than some small constant c, x is directly added as a pivot. Therefore, the first candidates are directly added to the set of pivots since a number of them is necessary for the computation of the contribution of each pivot. If some of them are redundant, they will be removed in further steps of the algorithm.

If the set of pivots has already more than c objects, the algorithm selects A pairs of objects at random and computes the contribution of each pivot, including the new candidate pivot. The already selected pivot with the lowest contribution is taken as the *victim*, and it can be replaced if the contribution of the new candidate pivot is better than its contribution to the set of pivots.

Based on the values of these contributions, the algorithm applies the second criteria for deciding what to do with the new candidate. Three cases are possible:

- The contribution of the candidate is 0: it means that it has never won the competition for discarding an object. Since the candidate can not discard any pair of objects that is not discarded by the already selected pivots, it does not make sense to add it to the set of pivots since it would be redundant. Therefore, the candidate pivot is discarded.

- The contribution of the candidate is greater than 0, but lower than the contribution of the victim: in this case, there are some pairs of objects that can only be discarded by this new candidate pivot. Since it is not redundant, it is added to the set of pivots. Since the victim pivot discards more objects than the new candidate, it is not removed from the set.

- The contribution of the candidate is greater than 0, and greater than the contribution of the victim: in this case, the new candidate pivot is not redundant, so it is added to the set of pivots. In addition, the algorithm decides to remove the victim pivot because its contribution is smaller than that of the new pivot.

The pseudocode shown in Algorithm 5.1 summarizes the selection of pivots in Non-Redundant Sparse Spatial Selection. Algorithms 5.2 and 5.3 summarize the computation of the contribution of each pivot and the selection of the victim from the current set of pivots respectively.

The selection algorithm computes the contribution of the pivot candidate p, using the rule defined in Section 5.3.1. For each pair of objects (x, y), the algorithm computes the lower bound on the distance $d(x, y)$, using p as pivot. If this distance is greater than the distance obtained with the best pivot for that pair (stored in the array $MaxD$), the algorithm adds the corresponding contribution of p for pair (x, y), otherwise it adds nothing. In this case, the algorithm removes the contribution of the former best pivot for that pair of objects (x, y). The total contribution of p is the sum of its contribution for all pairs of objects. If the contribution of p is 0, it is directly discarded as pivot.

Finally, the algorithm decides if the new pivot p should be added to $Pivots$ or the victim should be replaced. If the contribution of p is greater than the contribution of the victim, the victim is replaced with p. Otherwise, p is added to the $Pivots$, thus incrementing its size in one.

Note that this dynamic algorithm for selecting pivots ensures that it will select only pivots that are at distance at least $M\alpha$ to the other pivots. Thus, the set of selected pivots holds the same property of those selected using Sparse Spatial Selection (SSS).

Algorithm 5.1: Pivot selection in Non-Redundant Sparse Spatial Selection.

```
Input: $u \in X$, $U$, $Pivots$, $M, \alpha$, $A$, $c$
Output: $Pivots$
// Initially, $Pivots \leftarrow \emptyset$
if $\forall p \in Pivots$, $d(u,p) \geq M\alpha$ then
    if $|Pivots| < c$ then
        // The algorithm selects the first $c$ pivots.
        $Pivots \leftarrow Pivots \cup \{u\}$;
    else
        // Compute the victim from $Pivots$
        $(victim, contributionVictim, MaxD, Pairs) \leftarrow$
        $computeVictim(U, Pivots, A)$;
        // Compute the contribution of $u$ (the pivot candidate)
        $contributionNew \leftarrow computeContribution(u, A, MaxD, Pairs)$;
        // If the contribution is positive, decide between adding
            the pivot or replacing an old one
        if $contributionNew > 0$ then
            if $contributionNew > contributionVictim$ then
                // Replace victim with new pivot
                $Pivots \leftarrow (Pivots - victim) \cup \{u\}$;
            else
                // Add pivot to $Pivots$
                $Pivots \leftarrow Pivots \cup \{u\}$;
            end
        end
    end
end
return $Pivots$
```

The space and time complexities of the proposed method are mainly given by the function *computeVictim*, which computes the contribution of each pivot and pivot candidate, and decides if the candidate becomes a pivot or not, and whether it replaces the victim or not in case of becoming a new pivot.

Regarding the space complexity, the algorithm needs space for storing the A pairs of objects $(x, y) \in Pairs$, that is, $2A$. It is also necessary to store the contribution of each pivot ($|Pivots|$), and the array $MaxD$ (A), that stores the best lower bound on the real distance $d(x, y)$ for each pair $(x, y) \in Pairs$. Thus, the total space complexity is $O(A + |Pivots|)$.

Regarding the time complexity, the Algorithm 5.1 for the selection of pivots computes first the distances of the new object inserted in the database to the already selected pivots ($Pivots$). Then it needs to initialize the arrays $Pairs$ (that stores the sample of A pairs of objects), *contribution* (used to store the contribution of

Algorithm 5.2: Function *computeContribution*

```
Input: $p \in X,\ A,\ MaxD,\ Pairs$
Output: $contributionNew$
$contributionNew \leftarrow 0$;
for $i$ from 1 to $A$ do
    $(x, y) \leftarrow i^{th}$ pair of objects in $Pairs$;
    $diff \leftarrow |d(x,p) - d(y,p)|$;
    if $diff > MaxD[i]$ then
        $contributionNew \leftarrow contributionNew + diff - MaxD[i]$;
    end
end
return $contributionNew$
```

each pivot, and $MaxD$ (used to store the best lower bound for each pair of objects in $Pairs$), $(A + |Pivots| + A)$. The algorithm computes then the contribution of each pivot for each pair of objects $(A \cdot |Pivots|)$, and finally it computes the victim pivot (A).

The function *computeContribution* performs only $O(A)$ extra instructions, thus the time complexity of Algorithm 5.1 is $O(A \cdot |Pivots|)$. If one considers a fixed set U that must be indexed, a loose upper bound of the total time complexity for selecting the pivots is $O(A \cdot |Pivots| \cdot |U|)$, because the $O(A \cdot |Pivots|)$ operations are performed only when an object from $|U|$ is sufficiently far away from the previously selected pivots.

Therefore, Non-Redundant Sparse Spatial Selection (NR-SSS) involves both space and time overhead in the selection of pivots if compared with Sparse Spatial Selection (SSS).

5.4 Reduction of the construction cost

An inconvenient of the method as it has been described in the previous section, is that when an object is inserted in the database, the algorithm has to select a set of A pairs of objects from the database and compute the distances from each pivot to the two components of each pair, in order to evaluate the contribution of each pivot. This adds a significant overhead on the insertion of each object. Although this is the optimal process in what respects to the quality of the pivots selected, the cost it introduces in the insertion can be too high for some applications.

The time complexity of the algorithm for computing the victim (Algorithm 5.3) can be improved by reusing most of the pair of distances and by storing the corresponding computed distances (e.g., by changing only one pair of objects on each call of this algorithm). With this approach, the new time complexity of Algorithm 5.3 (and therefore of Algorithm 5.1) is $O(|Pivots|)$, instead of $O(A \cdot |Pivots|)$, which is the minimum cost possible for inserting an object, as in Sparse Spatial Selection

Algorithm 5.3: Function *computeVictim*

```
Input: $U$, $Pivots$, $A$
Output: $victim$, $contributionVictim$, $MaxD$, $Pairs$
$Pairs \leftarrow \emptyset$;
// Select randomly $A$ pairs of objects from $U$
for $i$ from $1$ to $A$ do $Pairs \leftarrow Pairs \cup$ random pair of objects $(x,y) \in U \times U$;
// Initialize array with contribution of each pivot
for $i$ from $1$ to $|Pivots|$ do $contribution[i] \leftarrow 0$;
// Initialize array with distances in pivot space
for $i$ from $1$ to $A$ do $MaxD[i] \leftarrow 0$;
// Compute contributions
for $i$ from $1$ to $A$ do
    $(x,y) \leftarrow i^{th}$ pair of objects in $Pairs$;
    // Compute best pivot for row of $(x,y)$
    $MaxD[i] \leftarrow \max_{j=1}^{|Pivots|} |d(x,p_j) - d(y,p_j)|$;
    $indexMax \leftarrow \arg\max_{j=1}^{|Pivots|} |d(x,p_j) - d(y,p_j)|$;
    // Compute second best pivot for row of $(x,y)$
    $max2 \leftarrow \max_{j=1,\ j \neq indexMax}^{|Pivots|} |d(x,p_j) - d(y,p_j)|$;
    // Add contribution for best pivot of row
    $contribution[indexMax] \leftarrow contribution[indexMax] + MaxD[i] - max2$;
end
// Compute victim and its contribution
$victim \leftarrow \arg\min_{i=1}^{|Pivots|} contribution[i]$;
$contributionVictim \leftarrow \min_{i=1}^{|Pivots|} contribution[i]$;
return ($victim$, $contributionVictim$, $MaxD$, $Pairs$)
```

(it is the minimum cost because in any method the new object has to be compared with all the pivots in order to store the distances from the objects to the pivots).

However, as the A pairs of objects are now not randomly selected on each iteration of Algorithm 5.3, it is possible that the quality of the estimation of the contribution of each pivot decreases. That is why, instead of keeping a fixed sample of pairs of objects for the whole algorithm, we change at least one pair in each iteration, in an attempt of updating that information without incurring in a high computational cost.

As we will see in the experimental evaluation of the method (Section 5.5), the experimental results reveal that the loss of efficiency in the search when the pairs of objects for evaluating the contribution of each pivot are reused, is not significant when compared with the gain in the construction cost. However, the resources devoted to the construction in each step depend on the specific application and users of the method can choose to replace a given number of pairs in each iteration if they can afford it.

5.5 Experimental evaluation

In the experimental evaluation of our proposal we used several collections that represent real similarity search problems. More specifically, we used the collections NASA, and ENGLISH (described in the Appendix C).

In each experiment, the 90% of the collection was used as the database to be indexed, and the remaining 10% objects were used as sample queries, averaging the results obtained in each of them. For collections of images, the search range was adjusted to retrieve an average of 0.01% objects from the database in each query. For the collection of words, the search radius used was 2. As parameters, we used $c = 5$ (minimum size of the set of pivots), and $A = 5.000$ (maximum number of sample pairs of objects), for all the experiments.

5.5.1 Size of the set of pivots

One of the important characteristics of NR-SSS is that its policy for detecting and replacing redundant pivots gives as a result a set of pivots smaller than the obtained with SSS but that conserves its capacity for discarding objects from the result. We carried out several experiments in which for NASA, and ENGLISH, we obtained the number of pivots selected by Sparse Spatial Selection, and Non-Redundant Sparse Spatial Selection, for different values of the parameter α.

Figures 5.2, and 5.3 show the results obtained for NASA, and ENGLISH respectively. As we can see in the results, NR-SSS selects less pivots than SSS, and the size of the set of pivots is significantly smaller.

In these results we can also observe that the number of pivots replaced by NR-SSS is higher for smaller values of α. When the value of the parameter α is small, SSS introduces more objects as pivots, and it is more probable to have redundant pivots that are replaced by NR-SSS. As we can see in Figures 5.2, and 5.3, the number of pivots selected by NR-SSS is very stable when the value of α varies. That is, the number of pivots selected by NR-SSS is more or less the same for all the values of α we considered. This is another advantage of this improvement of SSS. Although the insertion of objects is more costly, the behavior of the method is not so sensible to the value of the α.

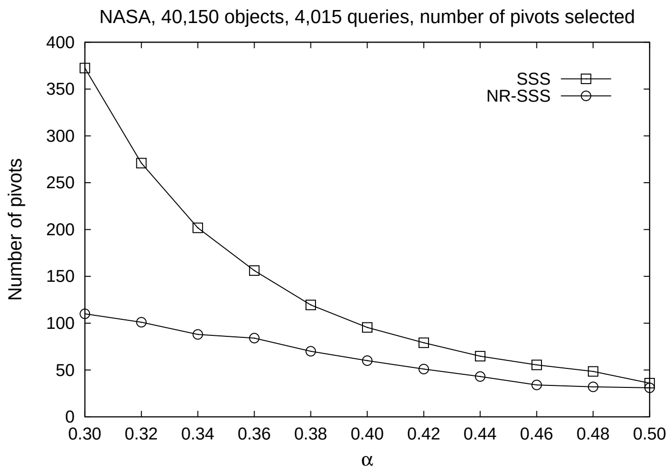


Figure 5.2: Number of pivots with SSS and NR-SSS in NASA.

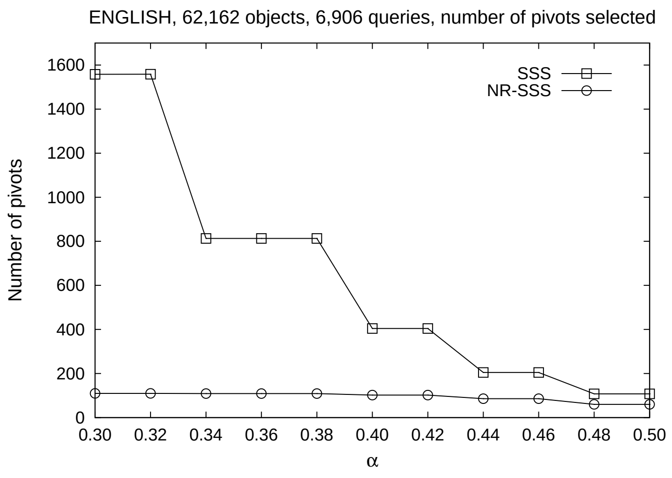


Figure 5.3: Number of pivots with SSS and NR-SSS in ENGLISH.

5.5.2 Search performance

Our hypothesis is that by detecting and removing or replacing pivots that are considered redundant under the criterion we defined in Section 5.3, the set of pivots would be smaller but conserving its capacity for discarding objects. In the previous

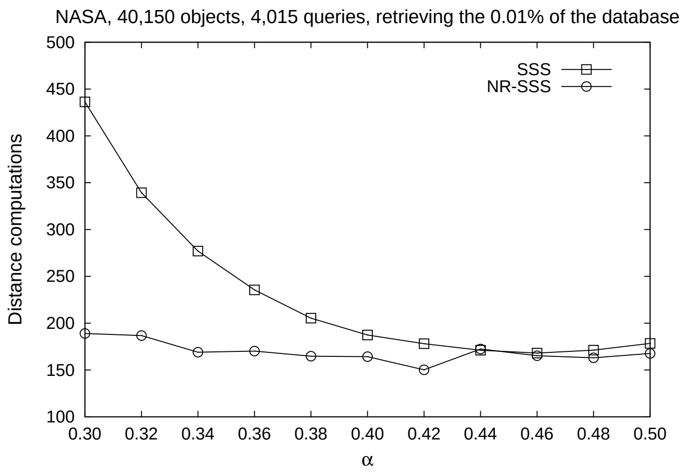


Figure 5.4: Search cost with SSS and NR-SSS in NASA.

subsection we have already seen that the set of pivots obtained by NR-SSS is smaller than the obtained with SSS. Since the total search complexity is given by the sum of the internal and external complexities, if the hypothesis is true, NR-SSS should show a better search performance than SSS for different values of α.

Figures 5.4, and 5.5 show the search performance obtained with SSS and NR-SSS for the collections NASA, and ENGLISH, for values different values of the parameter α.

As we can see in the results, NR-SSS systematically shows a better search performance than Sparse Spatial Selection in the collection NASA. In the case of ENGLISH, the difference in search performance depends on the value of α, although NR-SSS is more efficient for the values of α between 0.30 and 0.40, identified as the optimal range of values for this parameter in the previous chapter. For higher values of α, SSS selects a smaller number of pivots. In this case, the NR-SSS can replace some pivots that contributed to the capacity of the index for discarding objects, obtaining thus a worse result than SSS. This situation could be avoided by refining the criterion for removing pivots, or by establishing a condition for the starting the replacement of pivots, which remains as future work.

5.5.3 Cost of index construction

As we pointed out in the description of the method, selecting A pairs of objects and comparing each component of them with all the pivots can be a computational cost too high for an insertion in systems with dynamic requirements, where objects

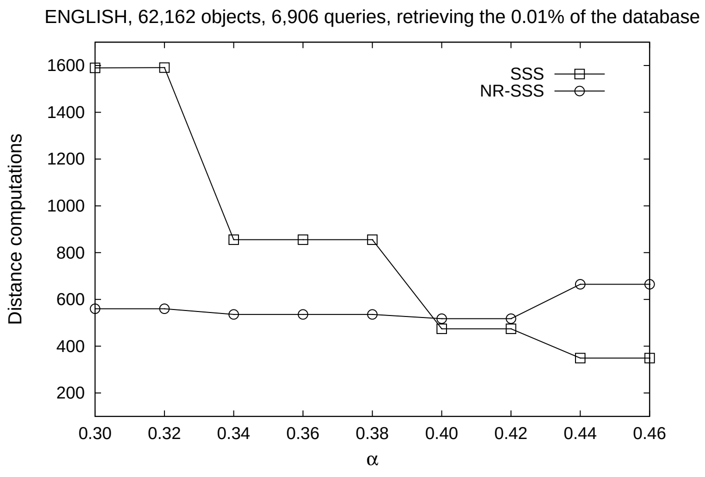


Figure 5.5: Search cost with SSS and NR-SSS in ENGLISH.

are constantly inserted and removed. Section 5.4 described how to reduce the construction cost of the index by keeping the set of pairs of objects during the construction of the index and replacing a minimum number of them in each step as the database evolves.

We compared the cost of construction of the original Non-Redundant Sparse Spatial Selection (NR-SSS) and the lower construction cost (NR-SSS-LCC). Figure 5.6 shows the difference in the cost of construction of the index for a collection of images. The left vertical axis represents the average number of comparisons needed for solving a query with both versions of the method. The right vertical axis represents the number of distance computations needed for indexing the whole collection. As we can see in these results, the loss of efficiency in the search performance is not very significant and the reduction of the cost of building the index can pay for it if this parameter is a problem for the application.

5.5.4 Comparison with previous techniques

Finally, we compared NR-SSS with previous techniques for pivot selection. As in the case of SSS (see Chapter 3), we compared our method with Incremental [Bustos et al., 2001], MaxMin [Vleugels and Veltkamp, 2002], SFLOO [Hennig and Latecki, 2003], and Spacing [van Leuken et al., 2006]. As in Chapter 3, we did not consider Maximum Pruning nor Maximum Variance [Venkateswaran et al., 2008, Venkateswaran et al., 2006] since they use the search range in the indexing phase, and the comparison would not be fair. In this

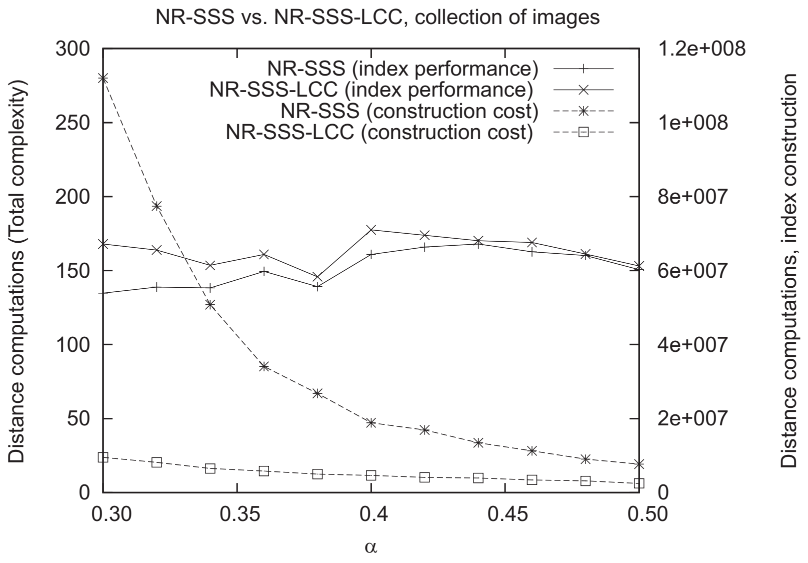


Figure 5.6: Reduction of the construction cost by reusing distances.

comparison we used the collection NASA, in which SSS obtained a worse result than previous methods.

The results are shown in Figure 5.7 and Table 5.1, which lists a relevant subset of the results shown in the figures. The table shows for each method the number of distance computations needed for solving a query, for different numbers of pivots (P). In the results we can see that NR-SSS improves the results obtained with SSS in this collection, in which SSS is not the best method. In addition, it needs less distance computations for solving the query than any previous method. An interesting result is that NR-SSS obtains better results with fewer pivots than any other method. While SSS needs 168 computations of the distance function for solving a query with 55 pivots, NR-SSS needs only 150 comparisons with 51 pivots.

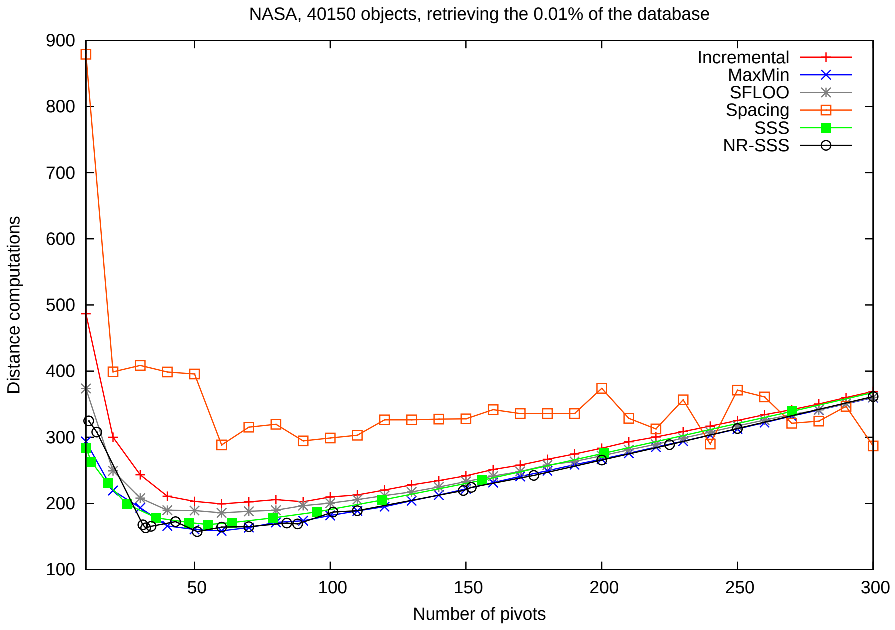


Figure 5.7: Comparison with previous techniques, collection NASA.

						SSS		NR-SSS	
P	**Increm.**	**Maxmin**	**SFLOO**	**Spacing**	α	P	**Dist.**	P	**Dist.**
30	243.21	193.24	207.91	408.57	0.50	38	173.57	31	167.65
40	210.78	165.87	189.73	398.72	0.48	48	171.19	32	163.01
50	203.06	160.55	189.05	395.58	**0.46**	**55**	**168.07**	34	165.27
60	**199.23**	**158.52**	**185.64**	**288.47**	0.44	64	171.16	43	172.45
70	202.27	163.42	188.03	315.19	**0.42**	79	178.08	**51**	**150.24**
80	205.75	171.17	189.76	319.61	0.40	95	187.39	60	164.26
90	202.49	173.65	196.59	294.44	0.38	119	205.37	70	164.78
100	209.79	181.77	200.29	298.83	0.36	156	235.51	84	170.22
110	212.72	188.60	205.72	303.10	0.34	201	276.95	88	168.98
120	220.18	195.11	211.85	326.30	0.32	270	339.28	101	186.81
130	228.03	204.00	217.33	326.33	0.30	372	436.22	110	188.95

Table 5.1: Comparison with previous techniques, collection NASA.

5.6 Summary

In this chapter we have presented Non-Redundant Sparse Spatial Selection (NR-SSS), a new method that improves Sparse Spatial Selection (SSS) for the selection of effective reference objects.

As we have seen in Chapter 3, SSS obtains a good set of pivots well-distributed in the space and it is competitive with previous methods in what refers to search cost. However, as we have explained in Section 5.2, SSS is a greedy algorithm: each time an object is inserted into the database, SSS decides if it becomes a pivot or not with the information up to that moment, and never goes back.

NR-SSS refines the set of pivots selected with SSS by identifying and removing pivots that are redundant, that is, pivots that do not contribute to increase the capacity of the whole set of pivots for discarding objects from the result.

In Section 5.3 we presented the algorithms for selecting pivots and building the index. When a new object is inserted into the database, it becomes a candidate pivot if it satisfies the selection criteria of SSS. Then, we compute its individual contribution to the capacity of the set of pivots for discarding objects from the result. The computation of the individual contribution of each pivot is based in a sample of objects taken at random from the database and used as object-query pairs. Once the contribution of the candidate pivot has been computed, three situations are possible. It that contribution is 0, the candidate is directly discarded and it does not become a new pivot. If the contribution of the pivot is not 0, it is added to the set of pivots. In addition, if its better than the worst current pivot, called victim, it replaces it.

In this way, the set of pivots obtained with NR-SSS is smaller than the set of pivots obtained with SSS, but the capacity for discarding objects is more or less the same. Thus, the search cost is reduced since the internal complexity is smaller.

In Section 5.3 we presented an alternative construction algorithm that reduces the construction cost at the expense of a loss in search efficiency.

The results of the experimental evaluation of the method were presented in Section 5.5, in which we show that the set of pivots of NR-SSS is actually smaller than the set of pivots selected with SSS for different values of α. We have also shown that the number of pivots replaced is greater for smaller values of α, and that the variations of the value of α affect less to NR-SSS. We compared NR-SSS with previous methods with real metric spaces, showing its competitiveness.

Chapter 6

Nested Metric Spaces

6.1 Overview of the chapter

The distribution of the objects in the data space is one of the factors that affects the capacity of methods for searching in metric spaces for pruning the search space. In general, the distribution of the objects in the space is not regular, and methods for searching in metric spaces try to be robust even in the case that irregularities are present in the data space. However, those irregularities are extreme in some cases. In this chapter we continue with the idea of taking the distribution of the objects into account for indexing. However, in this case we focus on the identification and treatment of a particular type of irregularity, which we call *nested metric spaces.*

In some collections of data, the objects are grouped into dense subspaces that contain a large amount of objects in a small region of the space. But, in addition, we have observed that the dimensions or features that explain the dissimilarity between two objects inside those dense clusters are different than the dimensions or features that explain the dissimilarity between any two objects in the rest of the space. We refer to these irregularities as nested metric spaces, since each dense subspace is like an independent metric space nested into a more general one. We have also observed that the presence of nested metric spaces can degrade the search performance of some methods for searching in metric spaces.

In this chapter we introduce the concept of nested metric spaces and why they can appear in real collections of data. We also explain why the presence of this particular type of irregularity of the space can degrade the search performance of methods for searching in metric spaces. We provide experimental results that confirm our hypothesis on the presence of nested metric spaces in real collections of data and their effect on the search performance. Finally, we present an approach for dealing with nested metric spaces.

6.2 Introduction

The search performance obtained by methods for searching in metric spaces is given by the number of distance computations needed for solving a query. The capacity of methods for directly discarding objects from the result, that is, their capacity for pruning the search space, depends on many factors. The distribution of the objects in the data space is one of them. Existing methods assume that the distribution of the objects in real metric spaces is not uniform and that it can present irregularities. For example, the number of objects in a region of the space can be higher than in others, or some objects can be outliers far from the rest of objects in the database. In addition, the type of irregularities that appear in a given space do not necessarily have to appear in another.

Therefore, methods for searching in metric spaces try to be robust in what refers to the dependence of the search performance they achieve with the presence of the many irregularities that can appear in real applications. In previous chapters we have already seen how some methods try to explicitly adapt the structure of the index and the information they store to the distribution of the objects in the space and thus obtain a better search performance.

For pivot-based methods, many pivot selection techniques have been proposed (MaxMin [Vleugels and Veltkamp, 2002], SFLOO [Hennig and Latecki, 2003], Incremental [Bustos et al., 2003], Spacing [van Leuken et al., 2006], Maximum Pruning [Venkateswaran et al., 2008], SSS and NR-SSS) that try to obtain an effective set of pivots for each particular space, instead of choosing them at random. For example, SSS selects an effective set of pivots well-distributed in the space, in order to cover the space in such a way that the dissimilarity between two objects can be detected in spite of the possible irregularities present in the space.

Clustering-based methods usually build tree-like indexes that recursively partition the space. In Chapter 4 we presented Sparse Spatial Selection Tree (SSSTree) and we show how the search performance can be improved by adapting the structure of the index to the distribution of the objects in the data space. Particularly, in the case of SSSTree, the index is adapted to the topology of the space by partitioning each cluster in as many clusters as needed for each region of the space, and not in a fixed number of clusters. With this approach, the index devotes more resources of the index to specific regions of the space.

In this chapter, we continue with the idea of exploring the distribution of the objects in the data space and taking it into account for indexing. But in this case, we focus on the detection and treatment of a specific type of irregularity of the space which we call *nested metric spaces*. We introduce the concept of nested metric spaces and why they can appear in datasets of real applications. We show how the presence of nested metric spaces affects the search performance of some methods for searching in metric spaces, and we show that adapting the resources of the index to these irregularities can improve the search cost.

6.3 The concept of nested metric spaces

6.3.1 Discovering nested metric spaces

During the experimental evaluations of the methods SSS (Chapter 3) and SSSTree (Chapter 4) that we carried out as part of this work, we discovered that there are metric spaces in which the irregularities of the distribution of the objects in the space are extreme.

We found that there are metric spaces in which a significantly large number of objects are grouped in dense subspaces that cover a small region of the space. That is, in some metric spaces there are large groups of objects that are very similar between them. But not only the objects are grouped in very dense subspaces. In addition, the dissimilarity between any two objects in the subspace is explained by dimensions different to the dimensions that explain the dissimilarity between other objects in the general metric space. We refer to this irregularity as *nested metric spaces*. We use this term since each subspace is like an independent metric space nested into a more general metric space.

Figure 6.1 shows an example of the presence of nested metric spaces in a three-dimensional vector space. As we can see in the figure, the database contains a set of points. The space has three explicit dimensions: the main corresponds to the x axis, and the other two correspond to the y and z axis. In this example, there are two subspaces with a large number of objects along the axes y and z. The objects inside a subspace are almost equal according to the main dimension but different according to the specific dimensions of the subspace they belong to. The dimensions that explain the difference between the objects in each nested metric space are different from the dimensions that explain the difference between two objects in the general metric space.

The presence of nested spaces can be caused by several reasons. Typically, the objects in some groups will be closer between them than to the rest of the objects in the space due to their similarity in a relevant feature.

Nested metric spaces can easily appear in real databases of objects. Perhaps the most evident example is a collection of images represented by vectors of features. In content-based image retrieval systems, images are usually analyzed with computer vision algorithms in order to detect and extract certain features of interest. Then, each image is represented by a vector that contains numerical values that represent those features. Typical features of interest for images include points of the histogram of each color of the image, features about the texture of the image, and about the presence of certain shapes. Nested metric spaces are likely to appear in such a database. For example, if a large group of images share the same main color (something easy in a collection of nature photographs, for example), their feature vectors will be grouped in the space, very close between them. In addition, the distance between any two of those images will be small, and the dissimilarity

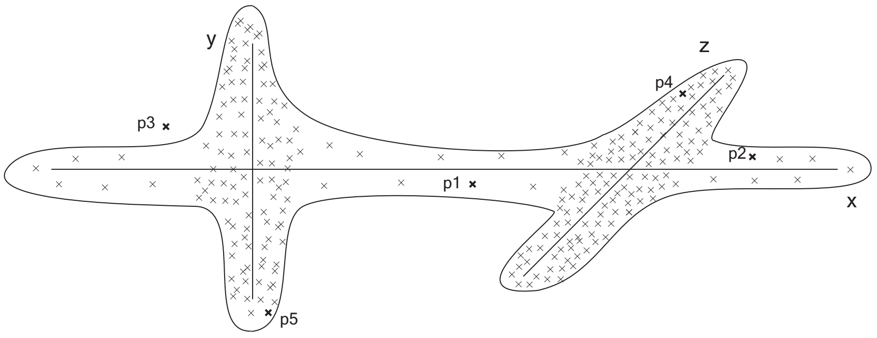


Figure 6.1: Dense subspaces nested into a general metric space.

between them is explained by different features than in the case of other images of the database.

The same situation could happen in a collection of strings. For example, in a database of words, all the words sharing the same root will be very close between them but further from the rest of words in the database. In that case, the groups of words would not be very large. But the same could happen with all the words that share the same word suffixes (as "*-ment*", "*-tion*", or "*-able*", for example). Nested metric spaces could also appear in a database of DNA sequences in which certain patterns repeatedly appear in large groups of sequences.

6.3.2 Effect on the search performance

We have already shown that the distribution of the objects in the data space could be taken into account for improving the structure and performance of the index. Moreover, in some cases, the presence of irregularities as nested metric spaces can degrade the performance obtained with some methods.

During our experiments we found such a situation when testing Sparse Spatial Selection (SSS) with a collection of color images. COLOR is a collection of 112, 544 color images represented by feature vectors of dimension 112. That is, 112 features of interest were extracted from each image. However, if we analyze the contents of the database, most of the coordinates take the value 0 or a value very close to 0 for a large fraction of the images. As a result, all images are concentrated around the origin of coordinates and distributed in groups defined by certain features.

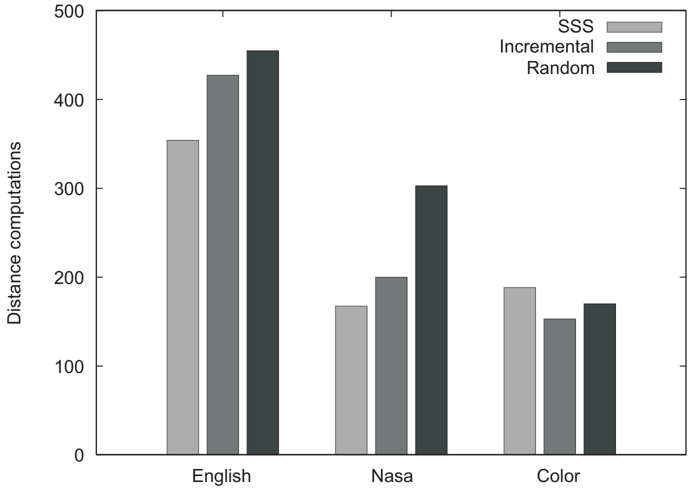


Figure 6.2: Average search cost with Random, Incremental and SSS.

This collection is an example of the presence of nested metric spaces. The collection COLOR is also an example in which the presence of nested metric spaces degrades the performance of some methods. Figure 6.2 shows a summary of the average search cost for solving a query in the collections ENGLISH, NASA, and COLOR (the details of each collection are described in Appendix C) with a random pivot selection, an Incremental pivot selection [Bustos et al., 2003], and SSS. As usual, the 90% of the objects of each collection were used as the database to be indexed, and the remaining 10% of the objects were used as queries in order to obtain the average search cost. As we can see in the results, SSS gets the best result in ENGLISH and NASA, followed by the method for we call Incremental [Bustos et al., 2003]. However, in the case of the collection COLOR, the results are different. The advantage of both Incremental and SSS over a random pivot selection is substantially reduced. Actually, in this collection, a random pivot selection performs very well.

In a collection like COLOR, SSS, or Incremental, or any other method trying to distribute the pivots in the space, is only able to put a pivot in each dense subspace. The maximum distance between two objects is given by the main dimensions of the space, and therefore, few pivots fit inside each subspace if they are selected with a method that distributes the pivots in the space (since the distance between objects inside them is much smaller). However, a random pivot selection has more chances to place more pivots inside each subspace. This explains the difference in search performance between a random pivot selection and other approach in this collection of images.

This is the same situation we draw on Figure 6.1, which shows an example of the presence of nested metric spaces in a three-dimensional space. As we can see in the figure, the maximum distance between any two objects is given by the dimension x of the general metric space. Therefore, SSS is only able to place five pivots in the space. Only one pivot is placed near to each nested metric space. In contrast, a random pivot selection has more chances of placing more pivots in each subspace, thus devoting more resources of the index to them.

6.3.3 Validating our hypothesis

In the previous section we introduced the concept of nested metric spaces as a particular type of irregularity of the distribution of the objects in metric spaces. We also argued that their presence in a database can affect the behavior of methods for searching in metric spaces, as it happens in the case of the collection COLOR, with methods as Incremental and SSS.

In this section we present experimental results that confirm the validity of our hypothesis. That is, we show experimentally that nested metric spaces can appear in real collections of data and that they affect the search performance of methods for searching in metric spaces. Particularly:

- We studied the distribution of the objects in the collection COLOR and compared it with the distribution of the objects in the data space of other collections in which we did not observe the presence of nested metric spaces.
- We created an artificial collection of vectors that contains nested metric spaces to confirm that they affect the search performance of methods for searching in metric spaces.

Test environment

For the validation of our hypothesis we used three collections of real data: ENGLISH, NASA, and COLOR (described in Appendix C). As usual, the 90% of the objects of each collection was used as the database to be indexed, and the remaining 10% of the objects were used as queries. The search radius was adjusted in each case to retrieve the 0.01% of the objects of the collection in average. In the case of the collection of words, the search radius was set to $r = 2$, as usual.

Additionally, we generated two synthetic collections of vectors with known distribution. We refer to the first one as REGULAR, a collection of $100,000$ vectors of dimension 12, uniformly distributed in an hypercube of side 1. We refer to the second collection as IRREGULAR. It is again a collection of $100,000$ vectors of dimension 12, but, as the name of the collection suggests, their distribution is completely biased. Three nested metric spaces are present in the collection. Each

Collection	$\#O$	$\#C$	$\#O/\#C$
ENGLISH	69,069	69	1001.0000
NASA	40,150	40	1003.7500
COLOR	112,544	112	1004.8571

Table 6.1: Experimental setup for each collection.

of them contains a 30% of the objects in the database. The objects of the first nested metric space are very similar according to the first three dimensions. The objects in the second nested metric space are very similar according to another three different dimensions, and the objects in the third nested metric space are very similar according to another three different dimensions. The remaining 10% objects in the collection are uniformly distributed in the space. The covering radius of each nested metric space is twice the typical search radius for such a collection.

Nested metric spaces in real collections of objects

Our initial hypothesis was that, when working with real collections of data, we can not assume the objects to have a regular distribution in the space. We argued that in real collections, the objects can be grouped in clusters that contain a significant amount of objects of the database in small regions of the space.

To validate this hypothesis, we used the three real collections described above: ENGLISH, NASA, and COLOR. For each of them we partitioned the data space into a set of clusters. Table 6.1 shows for each collection: the number of objects it contains ($\#O$), the number of clusters in which the collection was partitioned ($\#C$), and the relation between the number of objects in the collection and the number of clusters in which the collection was partitioned ($\#O/\#C$). In each collection, we chose a number of clusters ($\#C$) such that the number of objects in each cluster is more or less the same in all collections, that is, we partitioned each collection in such a way that $\#O/\#C$ takes more or less the same value in all collections (see the last column in Table 6.1). In this way, the results obtained in a given collection are comparable with the results obtained in the other collections.

For each collection, the cluster centers used to partition the space were obtained with SSS. After partitioning each collection, we obtained for each cluster the number of objects it contains. Figures 6.3, 6.4, and 6.5 show the histogram of the number of objects into the clusters in the collections ENGLISH, NASA, and COLOR respectively. The x axis represents the number of objects into the clusters, expressed as the percentage of objects of the database they contain. The y axis represents the relative frequency.

As we can see in the results, the number of objects contained into each cluster follows the same distribution in the collections ENGLISH and NASA. Most clusters

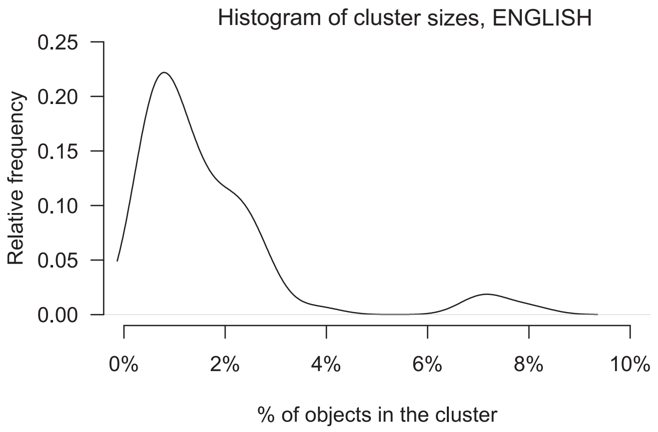


Figure 6.3: Histogram of clusters density for ENGLISH.

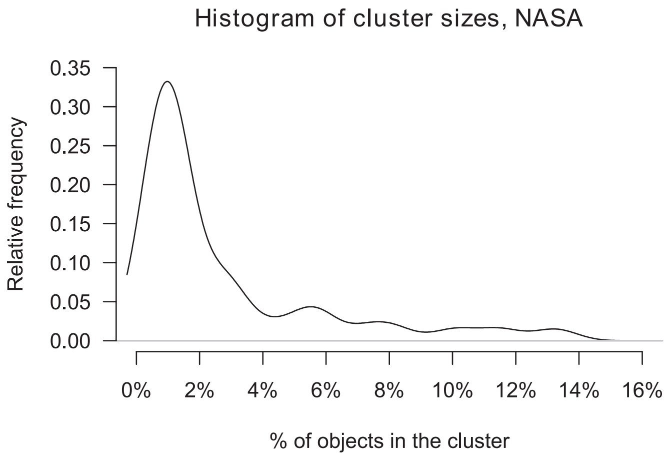


Figure 6.4: Histogram of clusters density for NASA.

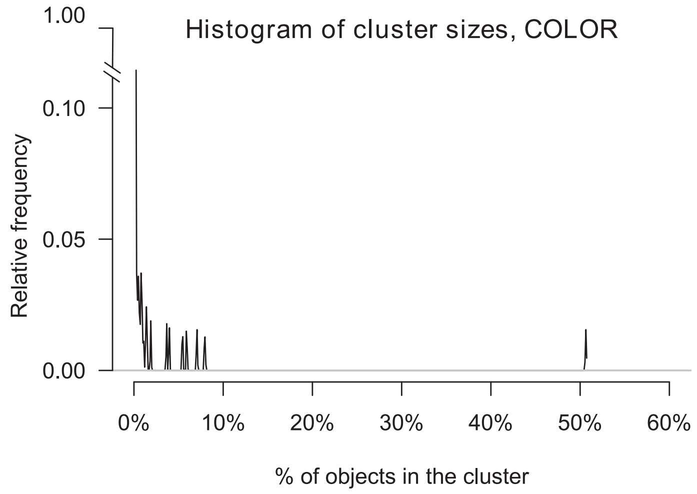


Figure 6.5: Histogram of clusters density for COLOR.

contain between the 1% and the 2% of the objects in the database, and some of them contain a larger number of objects, between the 8% and 14% of the objects in the database respectively. There are no nested metric spaces in these collections.

However, the number of objects into each cluster follows a completely different distribution in the case of the collection COLOR. As we can see in Figure 6.5, there is a cluster that contains half the objects in the database, clearly representing a nested metric space, while the rest of clusters contain very few objects. This is a clear case in which nested metric spaces are present in the collection.

Figure 6.6 shows together the histogram of cluster densities of the collections ENGLISH, NASA, and COLOR. Although in all of them there are clusters significantly larger than the mean, in the figure we can see that the collection COLOR is an extreme case.

Table 6.2 shows a summary of the results obtained in the experiments we have described. For each collection, the table shows the minimum, maximum, mean, and standard deviation of the number of objects in each cluster, expressed as a percentage of the objects in the database.

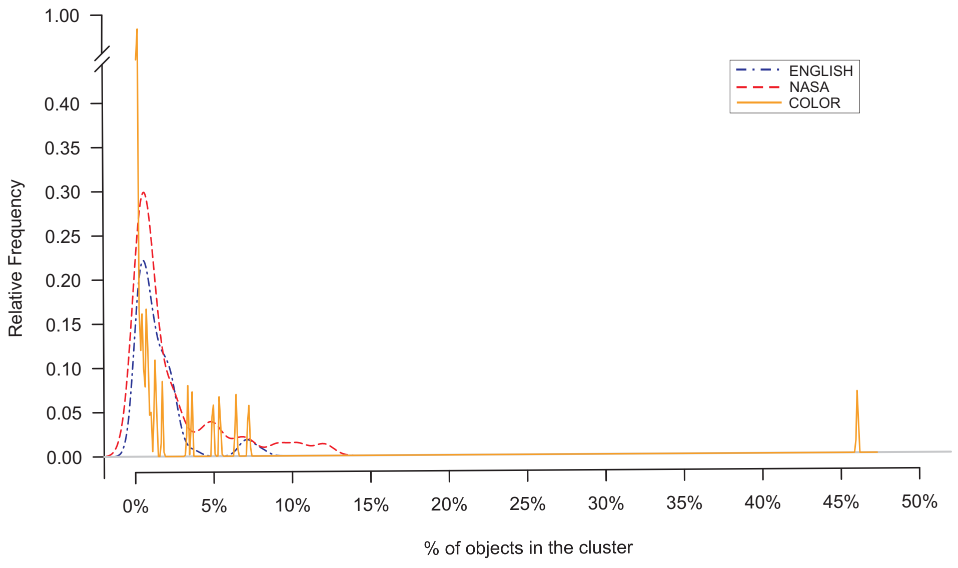


Figure 6.6: Histogram of clusters size in real collections.

Collection	% *min*	% *max*	μ	σ
ENGLISH	0.079631	8.009382	1.447828	1.672078
NASA	0.01925	13.21544	2.497509	3.284655
COLOR	0.000889	50.64775	0.8919686	4.927192

Table 6.2: Distribution of the percentage of objects in each cluster.

Effect on the search performance

Our second hypothesis was that the presence of nested metric spaces can significantly affect the search performance of the methods for searching in metric spaces. A random pivot selection obtains a better result than other methods that, like SSS, distribute the pivots in the space when there are nested metric spaces.

In order to validate this hypothesis we used the collections REGULAR, and IRREGULAR, which was synthetically created with three nested metric spaces. We obtained the average search performance obtained with a random pivot selection and SSS for different values of the parameter α in each collection. As usual, the 90% of the objects of the collection were used as the database to be indexed, and the remaining 10% were used as query objects (of course, the objects in the collection IRREGULAR were randomly unsorted, in other case only the objects in the nested

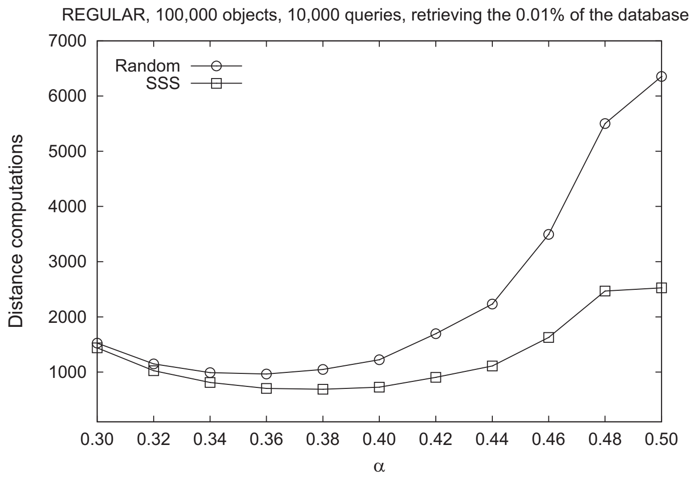


Figure 6.7: Average search cost with random and SSS in REGULAR.

metric spaces would be indexed). Figures 6.7 and 6.8 shows the results we obtained.

In the case of the collection REGULAR, SSS obtains always a better result than a random pivot selection. However, as we can see in Figure 6.8, the results in the collection IRREGULAR are very similar to the results obtained with the collection COLOR. The random pivot selection sistematically obtains a better result than SSS for all values of the parameter α. This result is due to the presence of nested metric spaces, since the distribution of the objects is controlled and there are no other features of the collection that could cause this result.

6.4 Indexing nested metric spaces

In this section we present an approach for detecting and indexing nested metric spaces. Our method uses Sparse Spatial Selection (SSS) to detect the presence of highly dense subspaces in the database, in order to further adapt the structure and information stored in the index to the distribution of the objects. With this method we will show that the search cost can be improved if the presence of nested metric spaces is detected and the resources and structure of the index are adapted to them. We refer to this method as Sparse Spatial Selection for Nested Metric Spaces (SSS-NMS).

SSS-NMS works in two levels. In the first level, the method creates a Voronoi partition of the space. The cluster centers used to create such a partition are selected with SSS, which guarantees the cluster centers to be well distributed in the space. Then, the density of each of these clusters is computed in order to detect

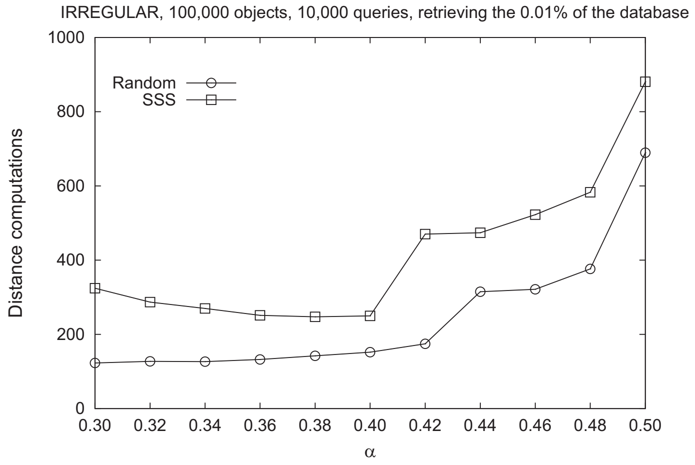


Figure 6.8: Average search cost with random and SSS in IRREGULAR.

the presence of nested metric spaces. In the second level, the method indexes only those clusters considered dense by applying a pivot-based schema inside each of them. The objects used as pivots in each cluster are again selected with SSS inside each nested metric space.

With this approach, SSS-NMS is able to detect complex groups of objects in the database in order to then devote more resources of the index to them, according to their complexity.

6.4.1 Construction

As we have already explained, the construction of the index is carried out in two levels, and SSS is applied in both of them: in the first level SSS is used to obtain effective cluster centers adapted to the data space in order to create a Voronoi partition of the space; in the second level, SSS is applied to select the set of objects used as pivots inside each subspace that was considered to have a high density in the first level of the index.

Since SSS is used in each of them, we will refer to the constants that control the density of objects selected by SSS as α and β for the first and second level of the method respectively.

First level: Voronoi partition of the space with SSS

In the first level of the index, the space is decomposed into a Voronoi partition. The objects used as cluster centers are selected with SSS, so they will be well distributed in the space. This is important in order to partition the space according to the distribution of the objects in the space. Once the cluster centers $\{c_1, \ldots, c_m\}$ are selected (m is not a fixed value, it depends on the value of α), each object is compared with them to create the Voronoi partition. The space is decomposed in a set of clusters C_i. The objects in each cluster are:

$$C_i = \{x \in U,\ d(x, c_i) \leq d(x, c_j),\ 1 \leq j \leq m\}$$

With this procedure for partitioning the space, an object becomes a cluster center if its distance to the already selected centers is equal or greater than $M\alpha$, where M is the maximum distance between two objects and α a constant parameter that takes values between 0 and 1. The value of α should be small in this first phase because a larger α would produce fewer clusters and having few clusters could result in dense clusters that contain also objects that do not belong to the real dense cluster of objects. This situation would increase the covering radius of the cluster, giving as a result a non-compact cluster.

This gives as a result a set of disjoint clusters which union gives as result the complete space. Since the cluster centers are not close to each other, because they have been selected with SSs, and they are well distributed in the space, the resulting clusters are more compact and less overlapping than if they were selected with a random pivot selection.

Second level: Indexing dense clusters with SSS

After partitioning the space into a set of clusters according to the main dimensions of the space, the second level of the index tries to identify nested metric spaces and further index them. In order to detect the presence of nested metric spaces we need some way of measuring how dense a cluster is, that is, if the number of elements it contains is too large for the region of the space it covers.

We define the density of a cluster as the relation between the number of elements assigned to the cluster and the covering radius of the cluster, a measure of the region covered by the clsuter:

$$density(C_i) = \frac{|C_i|}{r_{c_i}}$$

Computing the density of all clusters could be very costly if the maximum distance of each of them is obtained by comparing all the objects in the cluster with each other.

Although we have defined a way of measuring the density of objects in each cluster, we still do not have any kind of criteria for deciding which clusters are worth of further indexing and which not. We could think on establishing a threshold as a criterion, but its value should be different for each collection, so it is not a valid choice. Instead of using a threshold, we decide if a cluster is too dense or not by analyzing how dense it is if compared with the rest of clusters. If μ and σ are the mean and typical deviation of the density of a cluster, we consider that a cluster is dense enough to be further indexed if:

$$density(C_i) > \mu + 2\sigma$$

If the distribution of the objects in the space is more or less regular, the densities of the clusters will be around the mean, and few clusters will be dense enough to be further indexed. If nested metric spaces are present in the space, their density will be for sure far from the mean, so they will be easily detected.

For each cluster considered dense enough to be further indexed, a set of objects is obtained with SSS to be used as pivots, and the table of distances from all the objects of the cluster to the pivots is computed and stored. In this second level the index stores more information for the dense complex subspaces. In this case, the value of β should be around 0.4.

The clusters that are not considered dense are no further indexed. During the search, they are pruned from the result if possible with the information of the partition of the space. If they are not discarded, the query object is compared with all the objects they contain.

6.4.2 Search

Given a query (q, r), the query object is compared with all the cluster centers of the first level in order to discard as many objects as possible from the result without comparing them with the query object. Those clusters C_i for which:

$$d(q, c_i) > r + r_c$$

are directly discarded from the result set, since the intersection of the cluster and the result set is empty. For the clusters that could not be discarded there are two possibilities. If the cluster is not a dense cluster and therefore does not have associated a table of distances from its objects to pivots, the query has to be directly compared with all the objects of the cluster (as happens, for example, in list of clusters [Chávez and Navarro, 2005]). If the cluster has associated a table of distances, the query is compared with the pivots and the table is processed to discard as many objects as possible. The objects that can not be discarded are directly compared with the query.

6.4.3 Preliminary comparison

It is difficult to carry out a fair comparison of SSS-NMS with previous methods, as Incremental or SSS. While pivot-based methods work with a large table of distances, SSS-NMS stores precomputed distances for only some regions of the space. In order to obtain a fair comparison, all methods should be given the same amount of memory, that is, the same amount of information. A similar problem arises if we think in comparing SSS-NMS with clustering-based methods, since they use linear space, while SSS-NMS uses more information than them.

However, a preliminary comparison with previous methods, using the same amount of memory, can be enough for evaluating if the negative effect of nested metric spaces in the search performance can be overcome by detecting them and devoting more resources of the index to them. We compared SSS-NMS with SSS, Incremental [Bustos et al., 2003] and a random pivot selection [Micó et al., 1994].

Again, 90% of the objects of each collection were indexed and 10% were used as queries, retrieving an average of 0.01% of objects of the database for each query in the case of Nasa and Color, and using a search radius $r = 2$ for English.

Figure 6.9 shows the results we obtained. For each collection and method we show the average distance computations needed for solving a query. These results show that SSS-NMS is more efficient in terms of distance computations than the other methods when using the same amount of space. As we can see in the results, SSS-NMS obtains better results than SSS and Incremental in all collections, including COLOR, in which a random pivot selection performs very well.

Although the comparison in the search result is difficult due to the difference in the amount of information used by each method, the results show that devoting more resources of the index to the more complex regions of the space the search performance can be improved.

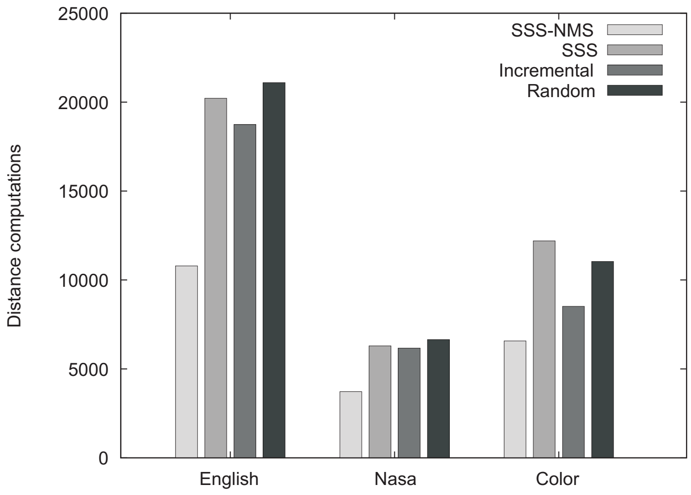


Figure 6.9: Comparison of SSS-NMS with previous techniques.

6.5 Summary

Many factors determine the capacity of a method for searching in metric spaces for pruning the search space, that is, for discarding objects from the result set without comparing them with the query. The distribution of the objects in the data space is one of those factors. In this chapter we continued with the idea of taking advantage of the distribution of the objects in the space in order to improve the search performance of methods for searching in metric spaces. However, in this case we focused in the detection and treatment of a particular type of irregularity that can appear in real metric spaces, which we call nested metric spaces.

We introduced the concept of nested metric spaces as subspaces that contain a large amount of objects in a small region of the space, and in which the difference between two objects is explained by dimensions different to the dimensions that explain the difference between two objects in the general metric space. We use the term nested metric spaces because each subspace is in some way like an independent metric space nested into a more general one. As we have seen in this chapter, this is an extreme irregularity of the space that can appear in real collections of data.

We have also explained how the presence of nested metric spaces can negatively affect the search performance obtained with pivot-based methods as SSS and Incremental. We carried out an experimental evaluation with both synthetic and real collections of data that confirms our hypothesis on the effect of nested metric spaces in the search performance of methods for searching in metric spaces.

Finally, we presented Sparse Spatial Selection for Nested Metric Spaces (SSS-NMS), an approach for the detection and indexing of nested metric spaces. This method works in two levels: in the first one the space is decomposed into a Voronoi partition and the density of each cluster is computed; in the second level, those clusters considered dense are further indexed with a pivot-based approach. Although the comparison of this approach with previous methods is difficult due to the differences in memory requirements, the preliminary results we presented show that the effect of nested metric spaces can be overcome if more resources of the index are devoted to these complex regions of the space.

Chapter 7

Conclusions

7.1 Summary of contributions

The number of applications of similarity search has significantly grown in the last years. Most of these applications appeared in systems as multimedia databases, search engines, or social networks, characterized for managing very large collections of objects of complex data types, in constant interaction with the user. Therefore, reducing the comparisons needed for solving a query as much as possible is mandatory in order to provide efficient search capabilities.

In this thesis, we have presented several methods for similarity search in metric spaces that aim not only to improve the search performance, but also other aspects important for real applications, as the construction cost, the space requirements of the index, the possibility of working with both discrete and continuous metrics, and the possibility of dynamically adapting the information in the index as the collection of objects evolves from an initially empty database.

This section summarizes the main contributions of this thesis:

- We have presented *Sparse Spatial Selection* (SSS), a new method for the selection of effective indexing objects for searching in metric spaces. While previous techniques based the selection on the idea that good pivots have to be far away from each other and far away from the rest of objects of the database, our proposal selects a set of pivots well distributed in the space. That is, pivots are not near to each other, but they are not necessarily very far away from each other. The experimental evaluation shows that the performance obtained with this approach is better or at least equal than the obtained with previous techniques. Therefore, we proved that good pivots do not need to be far from the rest of objects in the database.

Sparse Spatial Selection has other important characteristics that make it suitable for real-life applications. The most important is perhaps that it is dynamic. The method starts with an empty database and the pivots are selected as needed for building the index as objects are inserted into or removed from the database. Since each pivot covers a region of the space, the index adapts its structure and the information it stores to the content of the database in each moment.

An important difference of Sparse Spatial Selection with previous methods is that it is not necessary to state the number of pivots to use before the indexing. While in previous methods the optimal number of pivots had to be obtained by trial and error on the whole collection, our method determines by itself how many pivots it needs in each moment depending on the content and complexity of the collection.

Other important characteristic of this method is that it does not impose any additional cost for the selection of pivots. The only information it needs is the same information needed for inserting an object in the database (the comparisons of the object with the pivots). This is an important difference with previous methods, that require a significant amount of distance computations during the preprocessing of the collection.

- In this thesis we analyzed the selection of indexing objects for clustering-based methods. As in the case of pivot-based methods, the number of cluster centers, their position in the space with respect to each other, and their position with the rest of the objects of the database, determine the pruning capacity of the method. Most existing clustering-based methods select the cluster centers at random. As in the case of pivots, this approach has several inconveniences. A random selection does not ensure the best search performance, and the number of cluster centers has to be stated beforehand.

  We have presented *Sparse Spatial Selection Tree* (SSSTree), a tree-like clustering-based method that selects in each node the cluster centers by applying Sparse Spatial Selection. Thus, the number of cluster centers in each node is adjusted as necessary depending on its size and the objects it contains. This approach gives as a result an unbalanced tree structure adapted to the topology of the space. The experimental results confirm that this unbalanced structure obtains a better search performance than existing, balanced, methods.

- We have proposed *Non-Redundant Sparse Spatial Selection* (NR-SSS), a method to detect redundant pivots and remove or replace them from the set of pivots. We introduced the concept of *redundant pivot* as follows: a pivot is considered to be redundant if it does not improve the effectiveness of the set of pivots as a whole, that is, if the set of pivots has the same capacity for pruning the search space with or without that pivot. We have also defined a criterion for estimating the contribution of each pivot to the whole set.

By detecting and removing the redundant pivots, Non-Redundant Sparse Spatial Selection obtains sets of pivots smaller than those obtained with Sparse Spatial Selection, while conserving the capacity for pruning the search space. Having less pivots reduces the internal complexity without increasing the external complexity. It also reduces the space requirements of the index, something important for pivot-based methods.

Non-Redundant Sparse Spatial Selection is also dynamic and adaptive, and it is not necessary to state beforehand the number of pivots the method has to select.

- Finally, we have introduced the concept of *nested metric spaces*. A nested metric space is subspace that contains a large number of objects in a small region of the space, and in which the dimensions that explain the dissimilarity between any two objects are different from the dimensions that explain the dissimilarity between other two objects in the general metric space. We use the term nested metric space, since they are as independent metric spaces nested into a more general metric space. Nested metric spaces are an extreme irregularity of the distribution of the objects in the space. Their presence can cause bad results in the search performance of some methods. We have experimentally shown the existence of these subspaces.

  We have proposed an hybrid method that tries to detect these subspaces and take advantage of them during the indexing. This method indexes the space in a first level, and then devotes further resources to that special subspaces.

7.2 Future work

This section summarizes the next steps considered for future work after this thesis. First, with respect to the methods proposed in this work:

- The experimental evaluations carried out in this work used a set of common test collections used by most researchers working in similarity search in metric spaces. Although these collections represent a good sample of the problems in which methods for searching in metric spaces are applied, we plan to extend the experimental evaluation of the methods we have proposed using large collections of data taken from real applications, in order to reinforce the robustness of our proposals when working with real, very large collections of data.

- We plan to extend our work in methods for searching in metric spaces that take advantage of the distribution of the objects in the space. We have proposed a method for detecting the presence of nested metric spaces in the database, and we have shown that devoting more resources of the index to them can improve the search performance. However, the method we have presented is still not competitive when compared with pivot-based methods that use as much memory as they need.

- We plan to design and build a product that solves a real application using the methods we have proposed in this thesis.

In addition, we are already working in other aspects of methods for searching in metric spaces that were not mentioned in this thesis. Particularly, the optimization of the space requirements of pivot-based methods.

As we introduced in the revision of the state of the art, pivot-based and clustering-based methods have two important differences. On the one hand, pivot-based methods can solve a query with much less distance computations than clustering-based methods. For instance, the number of distance computations of a pivot-based method can be in the order of hundreds, and the number of distance computations of a clustering-based method can be in the order of thousands. On the other hand, the space requirements of clustering-based methods are linear with the number of objects in the collection, while pivot-based methods may require high amounts of space for storing the information of the index. The space requirements make pivot-based methods impractical for some problems.

Up to now, the decision of which type of method to use depended on how costly the comparison of two objects is and the size of the database. If the comparison of two objects involves a very high computational cost, as happens with the comparison of DNA sequences using the edit distance, the space needed by a pivot-based method can be compensated by the optimization in the search cost. If that is not the case and the database is expected to be very large, clustering-based methods are a good option because the index needs a very small amount of space and will surely fit in memory.

A promising line of research for similarity search in metric spaces is the reduction of the space requirements of pivot-based methods, ideally making them linear with the size of the database, while conserving their capacity for pruning the search space. We are currently working on methods for achieving this goal by storing for each object in the database only the distance to the most promising pivot for it. Preliminary results of our work on this research line have been already published in [Ares et al., 2009a] and [Ares et al., 2009b].

Bibliography

[Ares et al., 2009a] Ares, L. G., Brisaboa, N. R., Esteller, M. F., Pedreira, O., and Ángeles S. Places (2009a). Optimal pivots to minimize the index size for metric access methods. In *Proceedings of the Second International Workshop on Similarity Search and Applications (SISAP 2009)*, pages 74–80, Prague, Czech Republic. IEEE Press.

[Ares et al., 2009b] Ares, L. G., Brisaboa, N. R., Esteller, M. F., Pedreira, O., and Ángeles S. Places (2009b). Reducción del tamaño del índice en búsquedas por similitud en espacios métricos. In *Actas de las XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD'09)*, pages 249–260, San Sebastián, Spain.

[Baeza-Yates, 1997] Baeza-Yates, R. (1997). Searching: An algorithmic tour. *Encyclopedia of Computer Science and Technology*, 37:331–359. A. Kent and J. Williams, Marcel Drekker, New York.

[Baeza-Yates et al., 1994] Baeza-Yates, R., Cunto, W., Manber, U., and Wu, S. (1994). Proximity matching using fixed-queries trees. In *Proc. of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM'94)*, volume 807 of *Lecture Notes in Computer Science*, pages 198–212, Asilomar, C.A., USA. Springer.

[Baeza-Yates and Ribeiro-Neto, 1999] Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval.* ACM Press.

[Bayardo et al., 2007] Bayardo, R., Ma, Y., and Srikant, R. (2007). Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*, pages 131–140, Banff, Alberta, Canada. ACM Press.

[Bille, 2005] Bille, P. (2005). A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239. Elsevier.

[Bozkaya and Ozsoyoglu, 1997] Bozkaya, T. and Ozsoyoglu, M. (1997). Distance-based indexing for high-dimensional metric spaces. In *Proc. of the ACM*

International Conference on Management of Data (SIGMOD'97), pages 357–368, Tucson, Arizona, USA. ACM Press.

[Brin, 1995] Brin, S. (1995). Near neighbor search in large metric spaces. In *Proc. of the 21st Conference on Very Large Databases (VLDB'95)*, pages 574 – 584, Zurich, Switzerland. Morgan Kaufmann Publishers Inc.

[Brisaboa and Pedreira, 2007] Brisaboa, N. and Pedreira, O. (2007). Spatial selection of sparse pivots for similarity search in metric spaces. In *Proc. of the 33rd Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM'07)*, volume 4362 of *Lecture Notes in Computer Science*, pages 434–445, Harrachov, Czech Republic. Springer.

[Brisaboa et al., 2008] Brisaboa, N., Pedreira, O., Uribe, R., Solar, R., and Seco, D. (2008). Clustering-based similarity search in metric spaces with sparse spatial centers. In *Proc. of the 34th Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM'08)*, volume 4910 of *Lecture Notes in Computer Science*, pages 186–197, High Tatras, Slovakia. Springer.

[Brisaboa et al., 2006a] Brisaboa, N. R., Fariña, A., Pedreira, O., and Reyes, N. (2006a). Selección espacial de pivotes dispersos para la búsqueda por similitud en espacios métricos. In *Actas del XII Congreso Argentino de Ciencias de la Computación (CACIC 06)*, San Luis, Argentina.

[Brisaboa et al., 2006b] Brisaboa, N. R., Fariña, A., Pedreira, O., and Reyes, N. (2006b). Similarity search using sparse pivots for efficient multimedia information retrieval. In *Proceedings of the 8th IEEE International Symposium on Multimedia (ISM2006)*, pages 881–888, San Diego, California, USA. IEEE CS Press.

[Brisaboa et al., 2007a] Brisaboa, N. R., Fariña, A., Pedreira, O., and Reyes, N. (2007a). Indexación dinámica para la recuperación de información basada en búsqueda por similitud. In *Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07)*, pages 134–143, Zaragoza, Spain.

[Brisaboa et al., 2007b] Brisaboa, N. R., Fariña, A., Pedreira, O., and Reyes, N. (2007b). Spatial selection of sparse pivots for similarity search in metric spaces. *Journal of Computer Science & Technology*, 7(1):8–13.

[Brisaboa et al., 2009] Brisaboa, N. R., Luaces, M. R., Pedreira, O., Ángeles S. Places, and Seco, D. (2009). Indexing dense nested metric spaces for efficient similarity search. In *Proc. of the 7th International Andrei Ershov Memorial Conference - Perspectives on System Informatics (PSI'09)*, Novosibirsk (Russia). Accepted, to be published in Lecture Notes in Computer Science, Springer.

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

[Bustos et al., 2001] Bustos, B., Navarro, G., and Chávez, E. (2001). Pivot selection techniques for proximity search in metric spaces. In *Proc. of the XXI Conference of the Chilean Computer Science Society (SCCC'01)*, pages 33–40. IEEE CS Press.

[Bustos et al., 2003] Bustos, B., Navarro, G., and Chávez, E. (2003). Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357–2366. Elsevier.

[Bustos et al., 2008] Bustos, B., Pedreira, O., and Brisaboa, N. (2008). A dynamic pivot selection technique for similarity search in metric spaces. In *Proc. of ICDE Workshops (ICDEW'08), 1st International Workshop on Similarity Search and Applications (SISAP'08)*, pages 105–112, Cancún (México). IEEE Press.

[Chávez et al., 1999] Chávez, E., Marroquín, J. L., and Navarro, G. (1999). Overcoming the curse of dimensionality. In *Proc. of the European Workshop on Content-based Multimedia Indexing (CBMI'99)*, pages 57–64.

[Chávez et al., 2001a] Chávez, E., Marroquín, J. L., and Navarro, G. (2001a). Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135. Springer.

[Chávez and Navarro, 2005] Chávez, E. and Navarro, G. (2005). A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376. Elsevier.

[Chávez et al., 2001b] Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. L. (2001b). Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321. ACM Press.

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

[Clarkson, 1999] Clarkson, K. L. (1999). Nearest neighbor queries in metric spaces. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing (STOC'97)*, pages 609 – 617, El Paso, Texas, United States. ACM Press.

[Cole et al., 2004] Cole, R., Gottlieb, L., and Lewenstein, M. (2004). Dictionary matching and indexing with errors and don't cares. In *Proc. of the Annual ACM Symposium on Theory of Computing (STOC'04)*, 91-100, Chicago, USA. ACM Press.

[de la Higuera and Micó, 2008] de la Higuera, C. and Micó, L. (2008). A contextual normalized edit distance. In *Proc. of ICDE Workshops (ICDEW'08), First*

International Workshop on Similarity Search and Applications (SISAP'08), pages 61–68, Cancún, México. IEEE Press.

[Dehne and Noltemeier, 1987] Dehne, F. and Noltemeier, H. (1987). Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175. Wiley.

[Goel et al., 2001] Goel, A., Indyk, P., and Varadarajan, K. R. (2001). Reductions among high dimensional proximity problems. In *Proc. of the ACM Symposium on Discrete Algorithms (SODA'01)*, pages 769–778, Washington, D.C., United States. ACM Press.

[Guha et al., 2002] Guha, S., Jagadish, H. V., Koudas, N., Srivastava, D., and Yu, T. (2002). Approximate xml joins. In *Proc. of the ACM SIGMOD international conference on Management of data (SIGMOD'02)*, pages 287 – 298, Madison, Wisconsin, USA. ACM Press.

[Hafner et al., 1995] Hafner, J., Sawhney, H. S. ., Equitz, W., Flickner, M., and Niblack, W. (1995). Efficient color histogram indexing for quadratic form distance functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(7):729–736. IEEE Press.

[Hennig and Latecki, 2003] Hennig, C. and Latecki, L. J. (2003). The choice of vantage objects for image retrieval. *Pattern Recognition*, 36:2187–2196. Elsevier.

[Hetland, 2009] Hetland, M. L. (2009). *Swarm Intelligence for Multi-objective Problems in Data Mining*, chapter The Basic Principles of Metric Indexing. Springer.

[Kalantari and McDonald, 1983] Kalantari, I. and McDonald, G. (1983). A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9:631–634. IEEE Press.

[Levenshtein, 1965] Levenshtein, V. (1965). Binary codes capable of correcting spurious insertions or deletions of ones. *Problems of Information Transmission*, 1:8–17. Kluwer Academic Publishing.

[Linden et al., 2003] Linden, G., Smith, B., and York, J. (2003). Amazon.com recommendations: item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80. IEEE Press.

[Manku et al., 2007] Manku, G. S., Jain, A., and Sarma, A. D. (2007). Detecting nearduplicates for web crawling. In *Proc. of International Worl Wide Conference (WWW'07)*, pages 141–149, Banff, Canada. ACM Press.

[Manning et al., 2008] Manning, C. D., Raghavan, P., and Schütze, H. (2008). *An introduction to information retrieval.* Cambridge University Press.

[Micó et al., 1994] Micó, L., Oncina, J., and Vidal, R. E. (1994). A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear pre-processing time and memory requirements. *Pattern Recognition Letters*, 15:9–17. Elsevier.

[Navarro, 1999] Navarro, G. (1999). Searching in metric spaces by spatial approximation. In *Proc. of String Processing and Information Retrieval (SPIRE'99)*, pages 141–148. IEEE CS Press.

[Noltemeier, 1989] Noltemeier, H. (1989). Voronoi trees and applications. In *Proc. of the International Workshop on Discrete Algorithms and Complexity*, pages 69–74, Fukuoka, Japan.

[Pedreira and Brisaboa, 2007] Pedreira, O. and Brisaboa, N. R. (2007). Spatial selection of sparse pivots for similarity search in metric spaces. In *Proc. of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'07)*, volume 4362 of *Lecture Notes in Computer Science*, pages 434–445, Harrachov, Czech Republic. Springer.

[Robertson and Jones, 1976] Robertson, S. E. and Jones, K. S. (1976). Relevance weighting of search terms. *Journal of the America Society for Information Sciences*, 27(3):129–146.

[Salton and Lesk, 1968] Salton, G. M. and Lesk, M. E. (1968). Computer evaluation of indexing and text processing. *Journal of the ACM*, 15(1):8–36.

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

[Uribe et al., 2006] Uribe, R., Navarro, G., Barrientos, R. J., and Marín, M. (2006). An index data structure for searching in metric space databases. In *Proc. of International Conference on Computational Science 2006 (ICC'06)*, volume 3991 of *Lecture Notes in Computer Science*, pages 611–617. Springer.

[Uribe et al., 2007] Uribe, R., Solar, R., Brisaboa, N. R., Pedreira, O., and Seco, D. (2007). Ssstree: Búsqueda por similitud basada en clustering con centros espacialmente dispersos. In *Actas del Encuentro Nacional de Computación (ECC'07)*, pages 1–13, Iquique (Chile).

[van Leuken et al., 2006] van Leuken, R. H., Veltkamp, R. C., and Typke, R. (2006). Selecting vantage objects for similarity indexing. In *Proc. of the 8th International Conference on Pattern Recognition (ICPR'06)*, pages 453–456, Hong Kong, China. IEEE CS Press.

[Venkateswaran et al., 2008] Venkateswaran, J., Kahveci, T., Jermaine, C. M., and Lachwani, D. (2008). Reference-based indexing for metric spaces with costly distance measures. *The Very Large Databases Journal (VLBDJ)*, 17(5):1231–1251. Springer.

[Venkateswaran et al., 2006] Venkateswaran, J., Lachwani, D., Kahveci, T., and Jermaine, C. M. (2006). Reference-based indexing of sequence databases. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB'06)*, pages 12–15, Seoul, Korea. ACM Press.

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

[Vidal, 1994] Vidal, E. (1994). New formulation and improvements of the nearest-neighbor approximating and eliminating search algorithm (aesa). *Pattern Recognition Letters*, 15(1):1–7. Elsevier.

[Vleugels and Veltkamp, 2002] Vleugels, J. and Veltkamp, R. C. (2002). Efficient image retrieval through vantage objects. *Pattern Recognition*, 35(1):69–80. Elsevier.

[Yianilos, 1993] Yianilos, P. (1993). Data structures and algorithms for nearest-neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321. ACM Press.

Appendix A

Publications and other research results

This appendix summarizes the publications and research stays directly related with this thesis. For each publication, we include references to relevant works in which they have been cited (these citations were updated in November 2009).

Publications

Submitted papers

- Brisaboa, N., Bustos, B., Pedreira, O., and Reyes, N. *Adaptive Pivot-Based Indexes for Efficient Near Neighbour Search*. Manuscript submitted to *IEEE Transactions on Pattern Analysis and Machine Intelligence.*

Journals

- Brisaboa, N. R., Fariña, A., Reyes, N., and Pedreira, O. (2007). *Spatial Selection of Sparse Pivots for Similarity Search in Metric Spaces*. Journal of Computer Science & Technology, 7(1), pp. 8-13. Albuquerque (USA), 2007.

International conferences

- Ares, L. G., Brisaboa, N. R., Esteller, M. F., Pedreira, O., Places, A. S. (2009). Optimal Pivots to Minimize the Index Size for Metric Access Methods. In *Proc. of the 2nd International Workshop on Similarity Search and Applications (SISAP'09)*, pp. 74-80, Prague, Czech Republic. IEEE Press.

- Brisaboa, N. R., Luaces, M. R., Pedreira, O., Places, A. S., and Seco, D. (2009). Indexing Dense Nested Metric Spaces for Efficient Similarity Search. In *Proc. of the 7th International Andrei Ershov Memorial Conference (Perspectives of System Informatics) (PSI'09)*, Novosibirsk, Russia, 2009. To appear in Lecture Notes in Computer Science, Springer.

- Bustos, B., Pedreira, O., and Brisaboa, N. R. (2008). A dynamic pivot selection technique for similarity search in metric spaces. In *Proc. of ICDE Workshops (ICDEW'08), First International Workshop on Similarity Search and Applications (SISAP'08)*, pp. 105-112, Cancún, México. IEEE Press.

- Brisaboa, N. R., Pedreira, O., Uribe, R., Solar, R., and Seco, D. (2008). Clustering based similarity search in metric spaces with sparse spatial centers. In *Proc. of the 34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'08)*, Lecture Notes in Computer Science (4910), pp. 186-197, High Tatras, Slovakia. Springer.

  This paper has been cited by:

  - Marin, M., Gil-Costa, V., and Uribe, R. (2008b). Hybrid index for metric space databases. In *Proc. of Int. Conference on Computational Sciences (ICC'08)*, Lecture Notes in Computer Science (5101), pp. 327-336, Kraków, Poland. Springer.
  - Hetland, M. L. (2009). *Swarm Intelligence for Multi-objective Problems in Data Mining*, chapter of *The Basic Principles of Metric Indexing.* Springer.

- Brisaboa, N. R., and Pedreira, O. (2007) Spatial Selection of Sparse Pivots for Similarity Search in Metric Spaces. In *Proc. of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'07)*, Lecture Notes in Computer Science (4362), pp. 434-445, Harrachov, Czech Republic. Springer.

  This paper has been cited by:

  - Marín, M., Gil-Costa, V., Hernández, C. (2009). Dynamic P2P Indexing and Search Based on Compact Clustering. In *Proc. of the 2nd Int. Workshop on Similarity Search and Applications (SISAP'09)*, pp. 124-131, Prague, Czech Republic. IEEE Press.
  - Hetland, M. L. (2009). *Swarm Intelligence for Multi-objective Problems in Data Mining*, chapter of *The Basic Principles of Metric Indexing.* Springer.

  - Paredes, R. and Reyes, N. (2009). Solving similarity joins and range queries in metric spaces with the list of twin clusters. *Journal of Discrete Algorithms*, 7(1), pp. 18-35. Elsevier.
  - Marin, M., Gil-Costa, V., and Bonacic, C. (2008a). A search engine index for multimedia content. In *Proc. of Euro-Par 2008 Ũ Parallel Processing (Euro-Par'08)*, Lecture Notes in Computer Science (5168), pp. 866-875, Las Palmas de Gran Canaria, Spain. Springer.
  - Marin, M., Gil-Costa, V., and Uribe, R. (2008b). Hybrid index for metric space databases. In *Proc. of Int. Conference on Computational Sciences (ICC'08)*, volume 5101 of *Lecture Notes in Computer Science*, pp. 327-336, Kraków, Poland. Springer.

- Brisaboa, N. R., Fariña, A., Pedreira, O., Reyes N. (2006). Similarity search using sparse pivots for efficient multimedia information retrieval. In *Proc. of the 8th IEEE International Symposium on Multimedia (ISM'06)*, pp. 881-888, San Diego, USA. IEEE Press.

  This paper has been cited by:

  - Marín, M., Gil-Costa, V., Hernández, C. (2009). Dynamic P2P Indexing and Search Based on Compact Clustering. In *Proc. of the 2nd Int. Workshop on Similarity Search and Applications (SISAP'09)*, pp. 124-131, Prague, Czech Republic. IEEE Press.
  - Marin, M., Gil-Costa, V., and Bonacic, C. (2008a). A search engine index for multimedia content. In *Proc. of Euro-Par 2008 Ũ Parallel Processing (Euro-Par'08)*, Lecture Notes in Computer Science (5168), pp. 866-875, Las Palmas de Gran Canaria, Spain. Springer.
  - Venkateswaran, J., Kahveci, T., Jermaine, C. M., and Lachwani, D. (2008). Reference-based indexing for metric spaces with costly distance measures. *The Very Large Databases Journal (VLBDJ)*, 17(5), pp. 1231-1251. Springer.

National conferences

- Ares, L. G., Brisaboa, N. R., Esteller, M. F., Pedreira, O., Places, A. S. (2009). Reducción del Tamaño del Índice en Búsquedas por Similitud sobre Espacios Métricos. In *Actas de las XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD'09)*, pp. 249-260, San Sebastián, Spain.

  This paper received the best paper award of the conference.

- Brisaboa, N. R., Fariña, A., Pedreira, O., and Reyes, N. (2007). Indexación dinámica para la recuperación de información basada en búsqueda por similitud. In *Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07)*, pp. 134-143, Zaragoza, Spain.

- Uribe, R., Solar, R., Brisaboa, N. R., Pedreira, O., and Seco, D. (2007). SSSTree: búsqueda por similitud basada en clustering con centros espacialmente dispersos. in *Actas del Encuentro Nacional de Computación (ECC'07)*, pp. 1-13. Iquique, Chile.

- Brisaboa, N. R., Fariña, A., Nora Reyes, and Pedreira, O. (2006). Selección espacial de pivotes dispersos para la búsqueda por similitud en espacios métricos. In *Actas del XII Congreso Argentino de Ciencias de la Computación (CACIC'06)*, San Luis, Argentina.

Research stays

- *July 1st, 2008 - July 30th, 2008.* Research stay at Universidad de Chile (Santiago, Chile), under the supervision of Prof. Benjamín Bustos, and at Yahoo! Research Latin America (Santiago, Chile), under the supervision of Prof. Mauricio Marín.

- *October 26th, 2007 - November 8th, 2007.* Research stay at Universidad Michoacana de San Nicolas Hidalgo (Morelia, Mexico), under the supervision of Prof. Edgar Chávez.

Appendix B

Summary of notation

The following table summarizes the notation used throughout the text:

Symbol	**Meaning**
X	Universe of valid objects (universal set)
(X, d)	Metric space
d	Metric or distance function
U	Database or collection of objects
n	Size of the database
(q, r)	Range query
q	Query object
r	Search radius
$kNN(q)$	k-nearest neighbor query
k	Dimension of the vector space $\mathbb{R}^k$
L_p	Minkowski distances
$\rho = \mu^2/2\sigma^2$	Estimation of the intrinsic dimensionality [Chávez et al., 2001b]
P	Set of pivots
m	Size of the set of pivots
M	Maximum distance between any two objects of the space
C	Cluster
c_i	Center of the cluster C_i
r_{c_i}	Covering radius of C_i, $r_{c_i} = max\{d(x, y)x, y \in C_i\}$
(c_i, r_{c_i})	Enclosing ball of the cluster C_i, (c_i, r_{c_i})

Appendix C

Experimental Environment

In the experimental evaluations carried out during this thesis, we used the materials available at the *Metric Spaces Library*[1], a library developed as part of the *Similarity Search and Applications Conference (SISAP).* It provides implementations of the most relevant methods for searching in metric spaces, and a set of collections of different nature. It is, therefore, a common test framework used by most researchers in this field. In particular, we used the following collections:

- VECTOR8, VECTOR10, VECTOR12 and VECTOR14: Collections of synthetic vectors of dimension 8, 10, 12, and 14, uniformly distributed in an hypercube of size 1. Vectors are compared using the Euclidean distance.

  Since these collections are synthetic and all of them have the same distribution of distances, they permit us to test the behavior of methods with collections of known dimensionality. The higher the dimensionality, the more difficult the search.

We also worked with collections of images. Although the images are represented by feature vectors, their distribution is not uniform, and they are useful in order to test how the methods perform in data distributions taken from real applications:

- NASA: A collection of 40,150 images extracted from the archives of image and video of the NASA. Each image is represented by a feature vector of dimension 20. The distance between two images is the Euclidean distance between their feature vectors. This collection was used in the 1999's of the *DIMACS Implementation Challenge.*

  The similarity of two images is based in terms of their color histograms, using the Munsell space color (Hue,Saturation,Intensity). Each image is divided

[1]The *Metric Spaces Library* can be accessed at http://sisap.org (November, 2009)

in four regions of the same size, and the color histogram of each region is obtained. Each histogram is further divided in nine subspaces corresponding to black, white, and other six colors. By concatenating the result obtained for the four histograms, a vector of 36 components is obtained. This vector is then reduced to a vector of dimension 20 by an analysis of main features.

- COLOR: A collection of 112,120 color images, each of them represented by a feature vector of dimension 102. The distance between two images is the Euclidean distance between their feature vectors.

  The procedure to obtain the feature vector for each image is also based on the color histogram, and it is very similar to the used in the previous collection.

When working with collections of vectors, the 90% of the collection was used as the database to be indexed, and the remaining 10% objects were used as query objects. The search cost was computed as the average search cost for each object belonging to that 10%. For each collection, the search radius was adjusted to retrieve an average of the 0.01% of the objects of the database in each query.

We also worked with collections of words. Finding words similar to another one for spelling correction is another common example of similarity search:

- ENGLISH: A collection of 69,069 words taken from the English dictionary, and compared using the edit distance.
- SPANISH: A collection of 86,056 words taken from the Spanish dictionary, and compared using the edit distance.

We used two different collections because, although the objects in both collections are words, the distribution of the distances is not the same in each collection. In the case of the collections of words, the 90% of the collection was used as the database to be indexed, and the remaining 10% objects were used as queries. The search radius used was always $r = 2$.

Chapter 6 addresses the problem of nested metric spaces in real collections of objects, and how it can affect the performance obtained with methods for searching in metric spaces. In order to validate the hypothesis established in that chapter, we worked two synthetic collections of vectors:

- REGULAR: it is exactly equal to VECTOR12, that is, it contains $100,000$ vectors of dimension 12 uniformly distributed in an hypercube of side 1. In Chapter 6 we refer with this name to this collection for the sake of clarity, to remark that the objects in the collection have a regular distribution in the data space.
- IRREGULAR: is another collection of $100,000$ of dimension 12. However, the distribution of the objects in the space is not uniform, it is completely biased.

We generated this collection in order to show that the presence of nested metric spaces affects the search performance of methods for searching in metric spaces. The collection contains three nested metric spaces with the 30% of the objects of the collection in each one. The objects of the first nested metric space are very similar according to the first three coordinates. The objects in the second nested metric space are very similar according to another three different dimensions, and the objects in the third nested metric space are very similar according another three different dimensions. The remaining 10% of the objects are uniformly distributed in the space.

Table C.1 shows some statistics about the data distribution of each collection. For each collection, the table shows the size of the collection (the number of objects in the collection), and the following parameters that characterize the histogram of distances of the collection: mean, typical deviation, variance, minimum value, and maximum value of the distance between any two objects in the collection.

Collection	Size	μ	σ	σ^2	*min*	*max*
UV14	100,000	1.5086	0.2452	0.0601	0.4187	2.5317
UV12	100,000	1.4032	0.2456	0.0603	0.3173	2.5090
UV10	100,000	1.2652	0.2450	0.0600	0.2600	2.3067
UV08	100,000	1.1244	0.2469	0.0610	0.1463	2.1891
NASA	40,150	1.2342	0.3424	0.1172	0.0012	2.5079
COLOR	112,544	0.4005	0.1704	0.0290	0.0000	1.1517
ENGLISH	69,069	8.3176	2.0260	4.1048	1.0000	18.0000
SPANISH	86,061	7.4311	2.0168	4.0676	1.0000	19.0000
IRREGULAR	100,000	0.3830	0.1447	0.0209	0.0069	0.8977

Table C.1: Statistics on the distance distribution of each test collection.