UNIVERSITY OF PIRAEUS

DEPARTMENT OF INFORMATICS

Trajectory Data Management in Moving Object Databases

PhD Thesis

ELIAS K. FRENTZOS

Dipl. Civil Engineer, NTUA (1997)
MSc in Geoinformatics, NTUA (2002)

Athens, July 2008

UNIVERSITY OF PIRAEUS

Advisory Committee:

Supervisor:

Yannis Theodoridis
Asst. Professor U. Piraeus

Members:

Georgios Vasilakopoulos
Professor U. Piraeus

Timos Sellis
Professor NTUA

Thesis
submitted for the degree of
Doctor of Philosophy
at the Department of Informatics,
University of Piraeus

ELIAS K. FRENTZOS

"Trajectory Data Management in Moving Object Databases"

Examination Committee:

Nikolaos Aleksandris
Professor U. Piraeus

Themistoklis Panagiotopoulos
Professor U. Piraeus

Emmanuel Stefanakis
Asst. Professor Harokopio University

Charampos Konstantopoulos
Lecturer U. Piraeus

Preface

The domain of Moving Object Databases (MODs) is an important research area that has received a lot of attention during the last decade. The objective of moving object databases is to extend database technology to support the representation and querying of moving objects and their trajectory. MODs have become an emerging technological field due to the development of the ubiquitous location-aware devices, such as PDAs, mobile phones etc., as well as the variety of the information that can be extracted from such databases. However, the development of mechanisms that enable MODs to efficiently support trajectory data involves several physical aspects of the database technology, such as indexing, advanced query processing and query optimization.

The challenge accepted in this thesis is to provide mechanisms that enable MODs to manage trajectory data efficiently. Towards this goal, we develop a series of access methods, and dedicated query processing techniques which are subsequently implemented in prototypes in order to demonstrate their efficiency. Following the proposals of our thesis, existing moving object indexing techniques are enabled to support a wide range of standard and advanced queries. Beyond that, by applying the suggestions of related work we develop a model for the prediction of the effect of uncertainty in spatio-temporal querying. The results of our research may be directly employed in the context of spatial and spatio-temporal databases and warehouses, as well as, for query optimization purposes over distributed data with uncertainty. Finally, we provide a model that estimates the effect of trajectory compression in spatio-temporal querying. Our model expose interesting details regarding the error distribution of compressed trajectories which may lead to a new generation of more efficient compression algorithms, while it can be used as an additional criterion in order for a user to decide whether the compressed data are suitable for his / her needs.

July 2008 Elias Frentzos

Table of Contents

List of Tables

List of Figures

1. Intoduction

This chapter highlights the background of the thesis and outlines its structure. In Section 1.1 we introduce some basic knowledge about trajectories and motivate the thesis. Section 1.2 presents the notion of trajectories, which are the general subject of this work. In Section 1.3 we set the problems that we will cope with, and Section 1.4 sketches the contributions of this thesis. In Section 1.5 the datasets used throughout the thesis are introduced and finally, Section 1.6 outlines the rest of the thesis.

1.1. Moving Object Databases

The domain of Moving Object Databases (MODs) is an important research area that has received a lot of interest during the last decade. The objective of moving object databases is to extend database technology to support the representation and querying of moving objects and their trajectory. MODs have become an emerging technological field due to the development of the ubiquitous location-aware devices, such as PDAs, mobile phones etc., as well as the variety of the information that can be extracted from such databases. Currently, a number of decision support tasks can exploit the presence of MODs, such as traffic estimation and prediction, analysis of traffic congestion conditions, fleet management systems, battlefield and animal immigration habits analysis [GS05].

Traditionally, the following taxonomy exists in the spatio-temporal database literature: (a) work on the present and future positions of moving objects, such as [SJLL00], [BJKS02], [MXA04] and (b) work on the past positions of objects, asking historical queries, such as [TVS96], [PJT00]. The latter category, can be also classified into two other categories: (a) approaches that model and treat spatial data changing discretely over time, with examples including management of multimedia data [TVS96], simple spatial [NST99] and more comlex, spatial referenced data, such as cadastral data [ACNV99], and, (b) approaches that deal with data changing continuously their position with time [GBE+00], [PJT00]; the latter is the category into which this thesis belongs.

Moving objects are geometries, which may be points, lines, areas or volumes, changing over time, while a trajectory is the description of the movement of those objects. As the geographical space per se is continuous, the physical movement is described by a continuous change of position, i.e., a function from time to geographical space. Movement also implies a temporal dimension as we can only perceive movement through comparison at two different instants. Therefore, a trajectory can be equivalently defined as the recording of a time-varying spatial phenomenon.

According to the previous discussion, a historical trajectory can be quite simply defined as a function from time to geographical space; on the other hand, its description, representation and

manipulation are much more complex. Indeed, from an application point of view, a trajectory is the recording of the movement of some object i.e., the recording of the positions of the object at specific moments in time. Thus, while we naturally think of a well-shaped curve representing the trajectory of the object, in reality the trajectory has to be built from a set of sample points, i.e., the sampled positions of the object; then the trajectory curve is obtained by applying interpolation methods on the set of sample points. However, whichever interpolation method is being employed, the resulting curve will only be a guess of the actual trajectory; a guess that is even worse when considering the possible measurement errors that inevitably happen when recording the original sampled points. There is thus an inherent uncertainty associated with trajectories. In order to model and manage adequately uncertainty, different modelling concepts have been proposed in the literature [TWHC04], [TWZC02], [PJ99].

Moreover, given that trajectories have to be a first-class modeling construct, rather than computable derived data, their concept was introduced in some early papers [CR99], [EGSV99], [FGNS00], which addressed the need for capturing and modeling the complete history of objects' movement. Assessing the fact that location data may change over time, the respective database must contain the whole history of this development; and the Database Management System (DBMS) should be allowed to go back in time at any particular timestamp, and retrieve the state of the database at that time.

Specifically, according to [GBE+00] moving points (*mpoints*) and moving regions (*mregions*) are described as 3D (2D space + time) or higher-dimensional entities whose structure and behaviour is captured by modeling them as abstract data types. Such types and their operations for spatial values changing over time can be integrated as base (attribute) data types into an extensible DBMS. [GBE+00] introduced a type constructor τ which transforms any given atomic data type a into a type $\tau(a)$ with semantics $\tau(a) = time \rightarrow a$. In this way, the two aforementioned basic types, namely *mpoint* and *mregion*, may be also represented as $\tau(point)$ and $\tau(region)$, respectively. [GBE+00] also provided an algebra with data types (such as moving point, moving region, moving real, etc.) together with a comprehensive set of operations, supporting a variety of queries over spatio-temporal trajectory data. The realization of such data models proposed in the literature, as well as packaging corresponding functionality to specific technical solutions results in moving object database engines. In the literature, one can find at least two MOD engines developed to realize the model proposed by Gutting et al. [GBE+00], namely the SECONDO prototype [AGB06] and the HERMES engine [PT06], [PTVP06].

Then again, the development of such engines involves physical aspects of database technology, such as indexing, and dedicated query processing and query optimization techniques. The challenge thus accepted in this thesis is to provide efficient mechanisms that allow MOD Engines to efficiently store and query trajectories. Towards this goal, a number of access methods and dedicated advanced query processing techniques are developed in this thesis and are subsequently implemented and shown to be efficient. All these methods are initially implemented as prototypes in independent development environments, while their porting in commercial DBMS is left as future work; nevertheless, a number of the proposed techniques have been already implemented in the HERMES engine [PFGT08] and the PostgreSQL [Post08b] together with the PostGIS spatial extension [Post08a].

Briefly outlining the main topics that we will cope with in this thesis, which are physical subjects of a MOD engine, they include indexing techniques for moving object trajectories, dedicated query processing techniques, models for querying under the presence of uncertainty, and, finally, issues on trajectory compression.

1.2. Basic Concepts of Trajectories

Generally speaking, spatio-temporal trajectories can be classified into two major categories, according to the nature of the underlying spatial object: (i) objects without area represented as moving points, and (ii) objects with area, represented as moving regions; in this case the region extent may also change with time. Among the above two categories, the former has attracted the main part of the research interest, since the majority of the real-world applications involving spatio-temporal trajectories consider objects represented as points, e.g., fleet management systems monitoring cars in road networks. It is therefore the former type on which this thesis is focused; as such, in the followings our discussion is restricted to trajectories of moving points.

Under this perspective, a trajectory can be straightforwardly defined as a function from the temporal $I \subseteq \mathbb{R}$ domain to the geographical space $\mathbb{R}^2$, i.e., the 2D plane. Formally, a trajectory T is a continuous mapping from the temporal $I \subseteq \mathbb{R}$ to the spatial domain ($\mathbb{R}^2$, the 2D plane):

$$I \subseteq \mathbb{R} \to \mathbb{R}^2 : t \mapsto a(t) = \left(a_x(t), a_y(t)\right), \tag{1.1}$$

and,

$$T = \left\{\left(a_x(t), a_y(t), t\right) \mid t \in I\right\} \subset \mathbb{R}^2 \times \mathbb{R} \tag{1.2}$$

On the other hand, from an application point of view, a trajectory is the recording of an object's motion, i.e., the recording of the positions of an object at specific timestamps; while the actual trajectory consists of a curve, real-world requirements imply that the trajectory has to be built upon a set of sample points, i.e., the time-stamped positions of the object. Thus, trajectories of moving points are often defined as sequences of (x, y, t) triples:

$$T = \left\{(x_1, y_1, t_1), (x_2, y_2, t_2), ..., (x_n, y_n, t_n)\right\}, \tag{1.3}$$

where $x_i, y_i, t_i \in \mathbb{R}$, and $t_1 < t_2 < .. < t_n$, and the actual trajectory curve is approximated by applying spatio-temporal interpolation methods on the set of sample points (Figure 1.1).

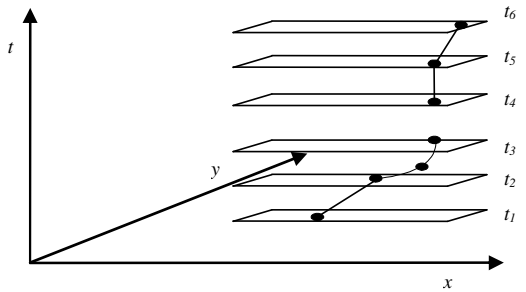


Figure 1.1: The spatio-temporal trajectory of a moving point: dots represent sampled positions and lines in between represent alternative interpolation techniques (linear vs. arc interpolation). Unknown type of motion can be also found in a trajectory (see $[t_3, t_4)$ time interval)

The first and foremost restriction posed by such spatio-temporal interpolation methods, is that a trajectory connected to a data sample should contain the sample points. i.e., for all points (x_i, y_i, t_i) in the sample it holds that $(x_i, y_i, t_i) = (a_x(t_i), a_y(t_i), t_i)$. Secondly, given a data sample, there is an infinite number of trajectories connected to that data sample, which implies that the trajectory is by no means unique. Finding a suitable curve connecting the sample points, is called interpolation. Interpolation brings along its own problems; we wish it to be fast, easily manageable, flexible and accurate. Unfortunately improving one property doesn't necessarily improve another. Linear interpolation is the fastest and easiest of them all (Figure 1.2). The idea is to connect the sample points with straight lines; the linearity is expressed in the fact that equal jumps in time (between the same sample points) lead to equal jumps in space. For example, the segment between the points (x_i, y_i, t_i) and $(x_{i+1}, y_{i+1}, t_{i+1})$ is given by

$$(x, y, t) = (x_i, y_i, t_i) + \frac{t - t_i}{t_{i+1} - t_i}(x_{i+1} - x_i, y_{i+1} - y_i, t_{i+1} - t_i), \text{ and } t_i \le t \le t_{i+1}, \quad (1.4)$$

which is a straight line segment in $\mathbb{R}^2 \times \mathbb{R}$ parameterized by $t \in [t_i, t_{i+1}]$. Finally, the trajectory consists of the concatenation of all these segments. Therefore, a trajectory may be also regarded as a collection of n-1 3D-line segments $T = \{L_1, L_2, ..., L_{n-1}\}$ with $L_i = \{(x_i, y_i, t_i), (x_{i+1}, y_{i+1}, t_{i+1})\}$.

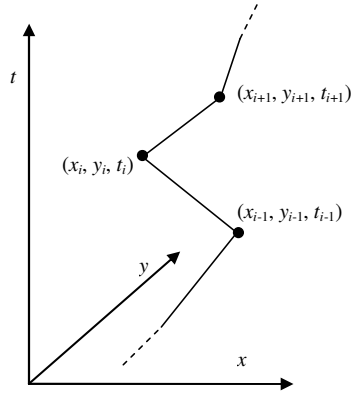


Figure 1.2: Linear interpolation

Linear interpolation in this manner is not so innocent; along the way some assumptions have been made. The first one is that the moving object keeps constant speed and direction between the sample points. Moreover, this speed is the average speed needed to cover the distance between (x_i, y_i) and (x_{i+1}, y_{i+1}) in time $t_{i+1} - t_i$. Secondly, changes in speed and direction at sample points are often abrupt and discontinuous, due to the sharp corners of the trajectory at the sample points. On the other hand, linear interpolation is fast to construct and to handle, and this is the main reason why it has been widely adopted in the trajectory database literature. Hereafter in this thesis, the term 'trajectories' will be used to describe such sets of triplets as in Eq.(1.3), applying linear interpolation in-between them as determined by Eq.(1.4).

1.3. Research Problems and Challenges in Trajectory Databases

Among the variety of technologies involved in the development of MODs for supporting historical trajectories of moving points, in this thesis we focus on a number of physical aspects, namely, indexing, advanced query processing, uncertainty support, and finally, trajectory compression. In the next sections, we briefly present the main research problems and challenges on trajectory databases that we will cope with in this thesis.

1.3.1. Indexing

Querying in MODs could be very expensive due to the nature of the underlying data and the complexity of the query processing algorithms. Given also the ubiquitousness of location-aware devices, trajectory databases will, sooner or later, face enormous volumes of data. It consequently arises that performance in the presence of vast data sizes, will be a significant problem for trajectory databases and the only way to deal with such enormous sizes is the exploitation of specialized access methods used for spatio-temporal indexing purposes.

The domain of spatio-temporal indexing is dominated by the presence of the R-tree [Gut84], along with its variations and extensions; this is actually an expected phenomenon given the popularity of the R-tree in spatial databases. The variations and extensions of the R-tree in the spatio-temporal domain include, among others, 3D R-trees [TVS96], TB-trees and STR-trees [PJT00], PA-trees [NR07], MON-trees [AG05], while SETI [CEP03] is a hybrid R-tree-based and partition-based technique. Since our interest in this thesis focuses on historical MODs, we restrict our discussion to indexing techniques recording past locations. The reader interested in indexing current locations and motion vectors can find very interesting work in [SJ02], [SJLL00], [TPS03], and [XP03].

However, as pointed out in [PJT00], the vast majority of the proposed spatio-temporal indexes overlook the challenges posed by the nature of trajectory data, and they just index collections of line segments in the spatio-temporal space, only concerning about the processing of traditional *coordinate-based* queries ignoring at the same time other useful types, such as topological and navigational queries, which are *trajectory-based.* Moreover, existing spatio-temporal indexes not preserving moving object trajectories and dealing with the spatio-temporal data as collection of line segments in the 2+1 dimensional space (such as SETI [CEP03] and 3D R-tree [TVS96]), overlook the need for deletion operations; albeit the deletion of a line segment from trajectory database may sound meaningless, the deletion of an entire trajectory is a very useful operation which has to be supported by any real-world trajectory index. The same need for trajectory preservation arises when dealing with compression mechanisms, which as we will see in the next chapter, by definition requires treating each trajectory as a single object.

Two index structures presented in [PJT00], namely the Spatio-Temporal R-tree (STR-tree) and the Trajectory Bundle tree (TB-tree), try to fulfill these needs and to efficiently support trajectory-based operations. The outcome of this work was that the TB-tree could support non-traditional queries much more efficiently than the traditional 3D R-tree and the STR-tree. Unfortunately, in spite of its clear advantages in trajectory-based query processing, the TB-tree has a crucial drawback due to its insertion strategy: new trajectory data are always inserted at the right ‘end’ of the tree, leading its performance to heavily depend on the data insertion ordering. However, in real-world applications, this

assumption is not guaranteed to be always true. For example, consider an application with the need to support real-time insertions, and a situation where the moving object enters an area where the position transmission system does not function; then its trajectory could be stored locally in the object and be transmitted to the central server – where the index operates – at a later time. Meanwhile, other moving objects could have transmitted their positions, violating the above TB-tree assumption. Furthermore, the structure of the TB-tree is not suitable for supporting deletion and compression operations; a trajectory deletion would leave 'holes' in the nodes, and trajectory compression as we will discuss in the sequel, requires the index to handle data inserted in non-chronological order.

Another interesting approach regarding the indexing of spatio-temporal trajectories, arise acknowledging that trajectories are more likely to be network – constraint. As pointed in [KGT99], the existence of restrictions in the space in which moving objects realize their movement is a condition that can be used to improve the performance of spatio-temporal indexes. Actually, this is the case in most real-world applications: planes fly in air-paths, cars and pedestrians move on road networks, while trains have fixed trajectories on railway networks. These kinds of special conditions (moving restrictions) have been the subject of research interest [KGT99], [PTKZ02].

More specifically, according to Kollios et al. [KGT99], the domain of the object's trajectories moving on a network is not the 2+1 dimensional space, rather than, a space with 1.5 dimensions, as line segments comprising the network can be stored in a conventional index of spatial data (such as the R-tree). Then, indexing of objects moving in a network is reduced to a one-dimensional indexing problem. In [KGT99], the problem of network-constraint trajectory indexing is studied under a more theoretical view rather than actually proposing an access method that could be used in real-world applications. On the other hand, following the directions provided by [KGT99], in this thesis, we show how this intuition can be realized by developing novel access methods for indexing network-constraint trajectory data.

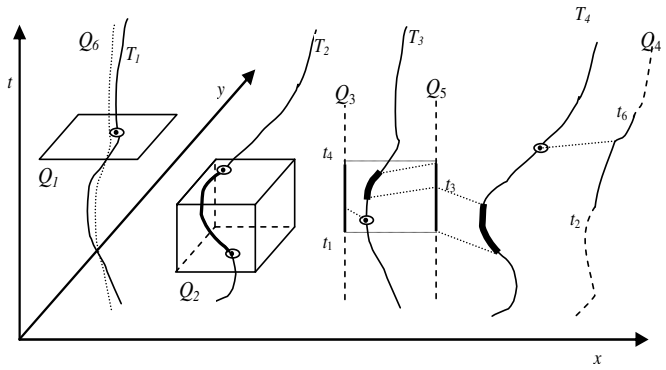


Figure 1.3: Querying trajectory databases

1.3.2. Advanced Query Processing

Advanced query processing over MODs storing historical trajectory information aims at developing specialized query processing techniques suitable for executing advanced queries, which may (or may not) exploit existing index structures being present to support more traditional queries. Here we have to point out that, queries of the form "*find all objects located within a given area during a certain time interval*", i.e., *range* queries (Q_2 in Figure 1.3), are regarded as traditional queries, and they are by

definition supported by any index; in the same category fall also the queries of the form "*find all objects' locations within a given area at a certain time instance*", which are called *timeslice* queries, and constitute a specialization of simple range queries having their lifespan set to zero (Q_1 in Figure 1.3) . The execution of range queries is usually a straightforward task; for example, the execution of a range query over R-tree-like (such as, the 3D R-tree [TVS96], the TB- and STR-trees [PJT00] and the TB*-tree) structures storing historical trajectory information is a straightforward extension of the *FindLeaf* algorithm, originally proposed in [Gut84], in the 3D space formed by the two spatial and the one temporal dimension.

On the other hand, there is a variety of spatio-temporal operators, which require more sophisticated query processing techniques in order to be efficiently processed; often these operators are extensions of the respective spatial ones. Among them, an important class of queries that has been introduced in the MOD directly from the spatial domain is the so-called *k nearest neighbor (k-NN) search*, where one is interested in finding the k closest trajectories to a predefined query object Q. To the best of our knowledge, the database literature regarding such queries primarily deal with either static ([RKV95], [CF98], [HS99]) or continuously moving query points ([SR01], [TPS02]) over stationary datasets, or queries about the future or current positions of a set of continuously moving points ([BJKS02], [TP02], [ISS03], [YPK05], [XMA05], [MHP05]). Apparently, these types of queries do not cover NN search on historical trajectories. Thus, one of the challenges being present in the domain of trajectory databases is to develop mechanisms to perform k-NN search on MODs exploiting spatio-temporal indexes storing historical information.

Moreover, the complexity of the underlying data makes the possible nearest neighbor operators over MODs storing historical trajectory data to be classified as follows: (a) according to the nature of the query object, which may be either a *stationary* or a *moving* point, i.e., another trajectory not contained in the MOD and, (b) according to the requested output of the operator, i.e., between the nearest to the query object during the query lifespan, and the nearest(s) at any time instance during the query lifespan; the latter are called *historical continuous nearest neighbor queries*.

To make the previous taxonomy more intelligible, recall Figure 1.3 illustrating a trajectory database containing four trajectories $\{T_1, T_2, T_3, T_4\}$, and several queries posed against it. Query Q_3 asks for the nearest trajectory to the query object (which is a stationary point) during the time period $[t_1, t_4]$; this is the simple case, and the answer to the query is trajectory T_3. Similarly, Q_4 is equivalent with Q_3, with the single difference that the query object is another trajectory, not contained in the database; in this case, the answer is trajectory T_4. Now, consider query Q_5 which is a historical continuous nearest neighbor query; in this case the query output should be a list of tuples containing the nearest trajectories along with the time period during which they were the nearest trajectory, i.e., $\{(T_4, [t_1,t_3)), [T_3, [t_3, t_4])\}$.

Another interesting query type that is useful in MOD search is the so-called *trajectory similarity* problem, which aims to find 'similar' trajectories of moving objects. To handle such queries efficiently, MOD systems should include methods for answering the so-called *Most-Similar-Trajectory* (MST) search also discussed in [The03]; an example of an MST query is Q_6 in Figure 1.3, which retrieves trajectory T_1 as its most similar. Trajectory similarity search is a relatively new topic in the literature;

the majority of the methods proposed so far are based on either the context of time series analysis or the Longest Common Subsequence (LCSS) model [VKG02] and the recently proposed Edit Distance on Real Sequence (EDR) [COO05].

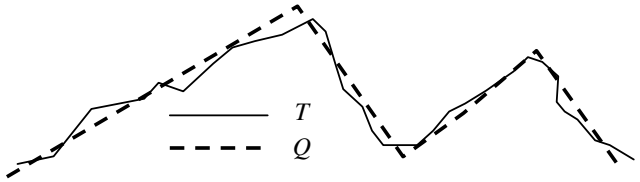


Figure 1.4: Trajectories with different sampling rates

However, the majority of the proposed methods either ignore the time dimension of the movement, therefore calculating the spatial (and not the spatio-temporal) similarity between the trajectories, or assume that the trajectories have the same sampling rate. To exemplify the problem derived when different sampling rates are present, consider Figure 1.4 illustrating trajectories *T* and *Q* with their position being sampled in different rates; while these two trajectories are obviously similar, methods based on the LCSS or the EDR model cannot detect this kind of similarity since they try to match trajectory sampled positions one by one, which clearly does not happen in the above (real world) example. What is more, the majority of the proposed approaches exploit specialized index structures in order to prune the search space and retrieve the most similar to a query trajectory. Thus, one of the challenges being present in the domain of trajectory databases is to develop mechanisms to perform *k*-MST search on MODs exploiting existing spatio-temporal indexes that support other type of queries as well.

1.3.3. Supporting Uncertainty

In the literature, uncertainty has been defined as the measure of the difference between the actual contents of a database, and the contents that the current user or application would have created by direct and perfectly accurate observation of reality [ZG02]. Sources of uncertainty may be one of the followings:

- Imperfect observation of the real world,
- Incomplete representation language,
- Ignorance, laziness or inefficiency.

Pfoser and Jensen [PJ99] propose a representation of location uncertainty due to measurement and sampling errors, which fall into the first and the third of the above error sources, respectively. According to [PJ99] the spatial projection of the trajectory of an object can be modeled as a 2D elliptical area, defined by the two consecutive tracked positions. On the other hand, a model that simultaneously captures both kinds of uncertainty is described by [TWHC04], [TWZC02]. In this model an *uncertainty threshold* is introduced, denoting the maximal distance of the object to the assumed location on the trajectory. Specifically, given the sampled points, after applying linear interpolation between them, this model assigns to each point on the trajectory a disc, parallel to the *XY*-plane, of radius equal to the threshold. Taking all those discs together in the 3D *space-time*, they finally result in a tube around the polyline connecting the sample points (Figure 1.5). This threshold

incorporates interpolation uncertainty and measurement errors all at once, and it does not discriminate sample points from interpolated points.

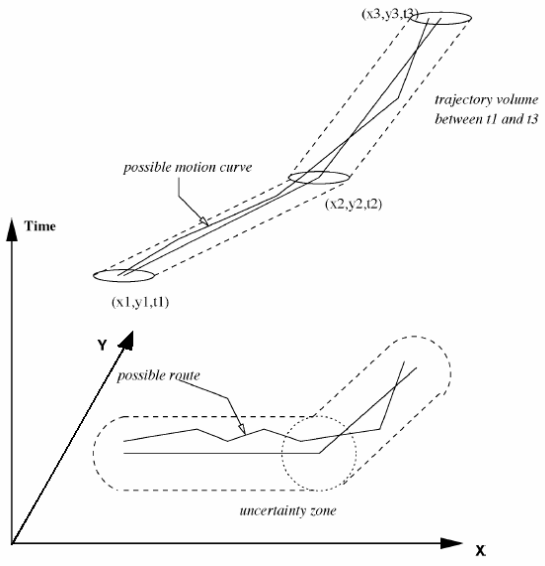


Figure 1.5: Modeling of Moving Object Uncertainty [TWHC04]

The literature on the management of the location uncertainty of spatio-temporal objects so far, apart from uncertainty representation issues [Tra03], [TWHC04], [WSCY99], also deals with probabilistic algorithms [TWHC04], [TWZC02], [CKP04] that process queries in the presence of uncertainty, estimating the probability of each trajectory to be included in the query result. On the other hand, there are cases where the user would prefer to know the influence of the measurement error in the query results, without actually executing the query. Consider for example the following real-world situation, inspired by the emerging open agoras paradigm [Ioa07]: let us assume a user who wishes to pose a query over several distributed subscribe-based data-sources containing the same spatio-temporal objects (i.e., trajectories) represented at different levels of uncertainty due to different measurement methods and, consequently, different uncertainty thresholds associated; though the criterion used to choose among them is the optimization, i.e., the minimization, of the uncertainty introduced in the final query results, the data-sources provide during the negotiation step [Ioa07] their potential customers-users with aggregate-only data. The only way thus to decide on the uncertainty of the results is the presence of a model that serves for this purpose, based on the aggregate-only information provided by the providers.

Another challenging problem, related to the one previously presented, is to determine the *maximum permitted* (im)precision of the trajectory data that will feed a MOD given the required accuracy in the results of timeslice queries. Then, users can be guided by the DBMS in the employment of the appropriate, more or less accurate - which also entails a more/less expensive - positioning method to be used for the data that will feed the system.

Both previous requirements could be fulfilled by a model that predicts the error introduced in query results based on known dataset (such as the uncertainty threshold) and query properties, without actually executing the query; moreover, such a model could be also utilized in an interactive query

builder / optmizer, informing the user about the effect of uncertainty in the query results, along with other interesting measures such as the query selectivity, estimated execution time etc.. To the best of our knowledge, a theoretical study on modeling the error introduced in spatio-temporal query results due to the uncertainty of trajectories is lacking; thus, it remains an open research problem in the domain of spatio-temporal databases.

1.3.4. Compressing Trajectories

As addressed in [MB04], it is expected that all the ubiquitous positioning devices will eventually generate an unprecedented data stream of time-stamped positions. Sooner or later, such enormous volumes of data will lead to storage, transmission, computation, and display challenges. Hence the need for compression techniques arises. However, existing work in this domain is relatively limited [CWT03], [MB04], [PPS06], [PPS06a], [PPS07], and mainly guided by advances in the field of line simplification, cartographic generalization and data series compression. According to [MB04], the objectives for trajectory data compression are:

- to obtain a lasting reduction in data size;
- to obtain a data series that still allows various computations at an acceptable (low) complexity;
- to obtain a data series with known, small margins of error, which are preferably parametrically adjustable.

As a consequence, we are interest in lossy compression techniques, which eliminate some redundant or unnecessary information under well-defined error bounds.

Especially on the subject of the error introduced on the produced data by such compression techniques, the single related work [MB04] provides a formula that estimates the mean error of the approximated trajectory in terms of distance from the original data stream. On the other hand, there are other kinds of errors that could help a user of a MOD to decide on the quality of the compressed data. For example, it is much more meaningful to provide the user with information about the mean error introduced in query results over the compressed data. Therefore, the need for an analytical model that estimates the error due to compression in the results of spatio-temporal queries arises.

Such a model could be utilized right after the compression of a trajectory dataset in order to provide the user with the average error introduced in the results of spatio-temporal queries of several sizes; it could be therefore exploited as an additional criterion for the user in order to decide whether compressed data are suitable for his/her needs, and possibly decide on different compression rates, and so on. Moreover, it could be used so as to improve the efficiency of the proposed solutions regarding trajectory compression; given that a model of this kind would expose the actual measures on which the error is depended, it could subsequently provide intuitive directions towards the employment of more sophisticated / efficient solutions. The challenge thus being present regarding trajectory compression is to provide a theoretical model that estimates the error due to compression in the results of spatio-temporal queries, and also adapt it in the context of MODs.

1.4. Thesis Contribution

This thesis presents several works being necessary for the efficient *Management of Trajectory Data*.

The uttermost goal of the conducted research is to provide effective mechanisms that allow Moving Object Databases to efficiently store and query historical trajectories; as such, the research deals with indexing, advanced query processing, supporting of uncertainty and issues on trajectory compression.

Next, we discuss the contributions of this thesis, grouped by the respective issue. Here, we have to point out, that the novelty of our approach is established in each different chapter, by appropriately presenting the respective related work. This approach is selected, instead of providing the related work in a single chapter, due to the variety of the issues so as to facilitate the reading of the thesis.

Indexing. In order to deal with the indexing requirements earlier presented, in this thesis we introduce two novel indexes, namely, the TB*-tree and the FNR-tree. The TB*-tree is an extension of the TB-tree which enables it to support *non-chronological insertions*; it is more compact, it advances its performance in terms of construction time, while, it outperforms its predecessor in the majority of the querying settings. Apart from the construction and query processing algorithms, the TB*-tree supports trajectory deletions, while its structure makes it capable of supporting trajectory compression algorithms as well, two of the properties not supported by the original TB-tree . It is essential however to clarify that the proposed TB*-tree, does not exploit the special conditions that objects have when moving on fixed networks; quite the opposite, it indexes objects moving freely in the 2D space.

On the other hand, under the network-constraint scenario this thesis provides a novel index, called Fixed Network R-tree (FNR-tree) which is an extension of the well-known R-tree [Gut84]. The general ideas on which the FNR-tree is based are roughly presented in [Fre02], nevertheless, without giving any implementation or experimental evaluation of the proposed method. The FNR-tree can be briefly portrayed as a forest of 1D (1D) R-trees on top of a 2D (2D) R-tree. The 2D R-tree is used to index the spatial data of the network graph (i.e., roads consisting of line segments), while the 1D R-trees are used to index the time interval of each object's movement on a given segment of the network. As it will be shown experimentally, the proposed FNR-tree outperforms all of its competitors in general coordinate-based queries, something that comes for the cost of lacking a mechanism which preserves trajectories.

Our results in the aforementioned topics are presented in Chapter 2. Preliminary results have been already published in [Fre03], [FT06].

Advanced Query Processing: Nearest Neighbor Search. In order to efficiently support nearest neighbor search on moving object trajectories we first propose a set of novel metrics being necessary for the ordering and pruning strategies followed by the proposed algorithms. More specifically, the definition of the minimum distance metric *MINDIST* between points and rectangles, initially proposed in [RKV95] and extended in [TPS02], is further extended in order for our algorithms to calculate the minimum distance between trajectories and rectangles efficiently. We then propose query processing algorithms to perform NN search over spatio-temporal indexes storing historical information of moving objects. Among the candidate spatio-temporal indexes, we exploit on the most commonly found indexes which are the ones supporting unconstrained movement, i.e., R-tree-like structures as the 3D R-tree [TVS96], the TB-tree [PJT00] and the TB*-tree proposed in this thesis. The description of our

algorithms for different queries depends on the type of the query object (point or trajectory) as well as on whether the query itself is continuous or not. In particular, we present efficient depth-first and best-first algorithms for historical NN queries as well as depth-first algorithms for their continuous counterparts. All the proposed algorithms are generalized to find the k nearest neighbors. Finally, we conduct a comprehensive set of experiments over large synthetic and real datasets demonstrating that the algorithms are highly scalable and efficient in terms of node accesses, execution time and pruned space.

Our results in the aforementioned topics are presented in Chapter 3. Preliminary results have been already published in [FGPT05], [FGPT07], [PFGT08].

Advanced Query Processing: Similarity Search. The issues mentioned on the subject of trajectory similarity search are addressed in this thesis, by efficiently supporting the k-MST search in MODs storing historical trajectory information, indexed by R-tree-like structures. More specifically, we support k-MST search by defining a dissimilarity metric (*DISSIM*) for the measurement of the spatio-temporal dissimilarity between two trajectories; this metric is also employed in [NP06] and can be seen as the average distance between the two trajectories in time. We subsequently propose an efficient approximation method to overcome its costly calculation, while, in the sequel, we develop a set of novel metrics along with several associated lemmas, which are employed for ordering and pruning purposes by the proposed most similar trajectory search algorithms. More specifically, using these metrics, we propose a depth-first and best-first query processing algorithm to perform k-MST search on R-tree-like structures storing historical trajectory information. We close this subject by conducting a comprehensive set of experiments over large synthetic and real datasets demonstrating that the algorithms are highly scalable and efficient in terms of node accesses, execution time and pruned space. We further demonstrate that the proposed similarity metric efficiently retrieves spatio-temporally similar trajectories in cases where related work fails. Finally, we describe how this work can be adjusted so as to support density-based trajectory clustering.

We have to point out that all the proposed algorithms do not require any dedicated index structure and can be directly applied to any member of the R-tree family used to index trajectories, such as the 3D R-tree [TVS96], the TB-tree [PJT00] and the TB*-tree proposed in this thesis. To the best of our knowledge, the proposal of this thesis is the first that provides techniques for a spatio-temporal index to support classical range, topological, nearest neighbor and similarity based queries.

Our results in the aforementioned topics are presented in Chapter 4. Preliminary results have been already published in [FGT07].

Supporting Uncertainty: The problems regarding the management of uncertainty highlighted in the previous section are initially covered by proving two lemmas that estimate the average number of false positives and false negatives when executing timeslice queries over uniformly distributed uncertain trajectories modelled via the [TWHC04], proposal; both errors depend on the radius of the cylindrical volume (i.e., the uncertainty threshold) and the perimeter of the timeslice query window, rather than its area. Then, in order to relax the location uncertainty uniformity assumption (directly derived from the

model of [TWHC04]) and to utilize the real-world adapted bivariate normal distribution [Lei95] [PTJ05], it is efficiently approximated with the uniform difference distribution. The results are close enough to the ones of the original analysis. The extension of the model towards supporting arbitrarily distributed trajectories and various distributions of the uncertainty radiuses is covered by employing novel spatio-temporal and other augmented histograms. We then perform a comprehensive set of experiments demonstrating the correctness and accuracy of the analysis. Finally, it is shown how the results of the analysis may be applied over spatial datasets: the solutions proposed are implemented on top of a commercial Spatial Database Management Systems (SDBMS), namely, the PostgreSQL [Post08b] with PostGIS spatial extension [Post08a]. Here, it is worth to note that off-the-shelf spatial histograms, already used in SDBMS for query selectivity estimation, support the proposed model without additional requirements.

Our results in the aforementioned topics are presented in Chapter 5. Preliminary results have been already published in [FGT08].

Compressing Trajectories: In order to cover the issues raised by the previous discussion regarding trajectory compression, we first describe two types of errors (namely, false negatives and false positives) when executing timeslice queries over compressed trajectories, and we prove a lemma that estimates the average number of the above error types. It is proven that the average number of the false hits of both error types depends on the Synchronous Euclidean Distance [CWT03], [MB04], [PPS06], [PPS06a] along the x- and y- axes between the original and the compressed trajectory, and the perimeter (rather than the area) of the query window. We subsequently show how the cost of evaluating the developed formula can be reduced to a small overhead over the employed compression algorithm, while we discuss how the developed analytical model helps to provide more effective compression algorithms. Finally, we conduct a comprehensive set of experiments over synthetic and real trajectory datasets demonstrating the applicability, correctness and accuracy of our analysis. It is worth to note that the most prominent application of the proposed model is based on the intruition it provides towards the development of more effective compression algorithms than the ones already present in the database literature.

Our results in the aforementioned topics are presented in Chapter 6. Preliminary results have been already published in [FT07].

In summary, the main contributions of our research are:

- The development of two novel spatio-temporal indexes, called TB*-tree and FNR-tree respectively, with the former enhancing the well known TB-tree towards the supporting of more realistic operation scenarios, and the latter exploiting the network-constraint assumption, outperforming all other compared indexes.
- The proposal of several scalable and efficient algorithms for nearest neighbor search over R-tree-like structures storing historical trajectory information.
- The development of two algorithms for Most Similar Trajectory search over R-tree-like structures storing historical trajectory information. Here, it is worth to note that using the

proposed NN and MST search algorithms, enables R-tree-like structures to support a wide range of spatio-temporal queries.

- The proposition of an analytical model that estimates the effect of uncertainty in timeslice queries over trajectory data, along with its extension to support arbitrarily distributed trajectories with the aim of histograms; the same model demonstrates great applications over stationary spatial data, while it can be directly employed in existing SDBMS.
- The development of an analytical model that estimates the effect of trajectory compression in spatio-temporal querying.

1.5. Trajectory Datasets Overview

Throughout this thesis we have experimented with a variety of real and synthetic trajectory datasets. Specifically, we have used two real trajectory datasets and also synthetic datasets generated by the GSTD data generator [TSN99], the network-based data generator of [Bri02] and a custom trajectory generator developed to fulfill specific purposes [FGT07]. The details of the employed datasets are given in Table 1.1.

Table 1.1: Summary dataset information about GSTD synthetic datasets

Dataset	*# trajectories*	*# entries*
Real Data (Trucks)	276	112K
Real Data (Buses)	145	66K
GSTD 100	100	485K
GSTD 250	250	1213K
GSTD 500	500	2426K
GSTD 1000	1000	4850K
GSTD 2000	2000	9701K
NG 200	200	106K
NG 400	400	213K
NG 800	800	417K
NG 1200	1200	626K
NG 1600	1600	831K
NG 2000	2000	1043K

1.5.1. Real trajectories

The origin of the two employed real datasets, was a fleet of trucks (dataset *Trucks*) and a fleet of school buses (dataset *Buses*), illustrated in Figure 1.6(a) and (b), respectively. The two real datasets consist of 276 (112203) and 145 (66096) trajectories (entries), respectively. Both datasets are available at http://www.rtreeportal.org.

1.5.2. Synthetic Trajectories Simulating Uncostrained Movement

In order to produce trajectories moving in the unconstrained space, we have used the GSTD data generator [TSN99]. A snapshot of the generated data using GSTD is illustrated in Figure 1.6(c). The synthetic trajectories generated by GSTD correspond to 100, 250, 500, 1000 and 2000 moving objects resulting in datasets of 500K, 1250K, 2500K, 5000K, and 10000K entries (the position of each object

was sampled approximately 5000 times), thus building indices of up to 500 Mbytes size (the case of 3D R-tree index for the GSTD 2000 dataset). Regarding the rest parameters of the GSTD generator, the initial distribution of points was Gaussian while their movement was ruled by a random distribution.

(a) a fleet of trucks (b) a fleet of school buses (c) GSTD synthetic data

Figure 1.6: Snapshots of real and synthetic spatio-temporal data

1.5.3. Synthetic Trajectories Simulating Road-network Costrained Movement

Regarding the case of network-constrained moving objects, our experiments were based upon synthetic datasets created using a network-based data generator [Bri02] and the real-world road network of San Joaquin (Figure 1.7). We produced the *NG* trajectory datasets constituting of 200, 400, 800, 1200, 1600 and 2000 moving objects, where each object's position was sampled 400 times. While the output of the generator was of the form (*id*, *t*, *x*, *y*), in our experiments we wanted to utilize those data only if (*x*, *y*) are the coordinates of a *node* of the network. Therefore, the generator was modified in order to produce records of the form (*id*, *t*, *x*, *y*) each time a moving object was passing through each node of the network. The maximum volume of line segments produced by the network-based generator was approximately 1M entries and that came up for 2000 moving objects.

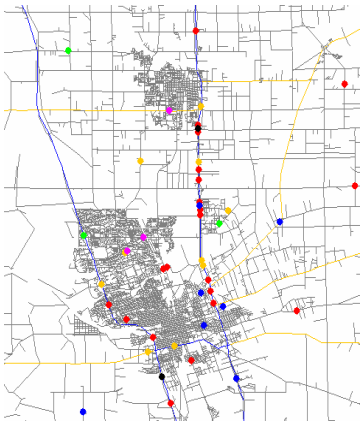


Figure 1.7: Real-world network of San Joaquin, with a snapshot of the generated data

1.6. Thesis Outline

The outline of the thesis is as follows: In Chapter 2 we propose and evaluate two novel indexes for spatio-temporal trajectories for unconstraint (the TB*-tree) and network-constraint movement (the FNR-tree), respectively. Chapters 3 and 4 propose solutions for efficient support of nearest neighbor and similarity search, respectively, over historical trajectory information. Chapters 5 and 6 propose two models, the former for the prediction of the effect of uncertainty in spatio-temporal queries, and the

latter, for the estimation of the effect of trajectory compression in spatio-temporal queries. Finally, Chapter 7 closes the thesis by summarizing the conclusions and discussing interesting open issues.

2. Trajectory Indexing

In this chapter we focus on the indexing problem regarding trajectory databases, and we present our two proposals, the TB*-tree and the FNR-tree. The outline of the chapter is as follows: Section 2.1 introduces the issues being related to the indexing of spatio-temporal trajectories while, Section 2.2 examines the related work. Section 2.3 presents the structure and the algorithms for maintaining and searching the TB*-tree, while section 2.4 stands for the structure and the algorithms of the FNR-tree. Sections 2.5 and 2.5.4 present the experimental study in unrestricted, and network-constraint space, respectively, and finally, Section 2.6.4 closes the chapter providing the conclusions.

2.1. Introduction

Like in traditional databases, querying in MODs could be very expensive due to the nature of data and the complexity of query processing algorithms. Given also that location-aware devices are almost ubiquitous nowadays, trajectory databases will, sooner or later, face enormous volumes of data. It consequently arises that performance in the presence of vast data sizes, will be a significant problem for trajectory databases. Since ordering is far from the nature of the geographic (multi-dimensional) data, traditional indexes like B-trees are not useful in spatial (and consequently in spatio-temporal) databases. In the domain of spatial databases, the R-tree proposed by Guttman [Gut84] is "almost ubiquitous", with applications ranging from Geographical Information Systems (GIS) and Computer Aided Design (CAD) to Image and Multimedia Management Systems [MNPT05].

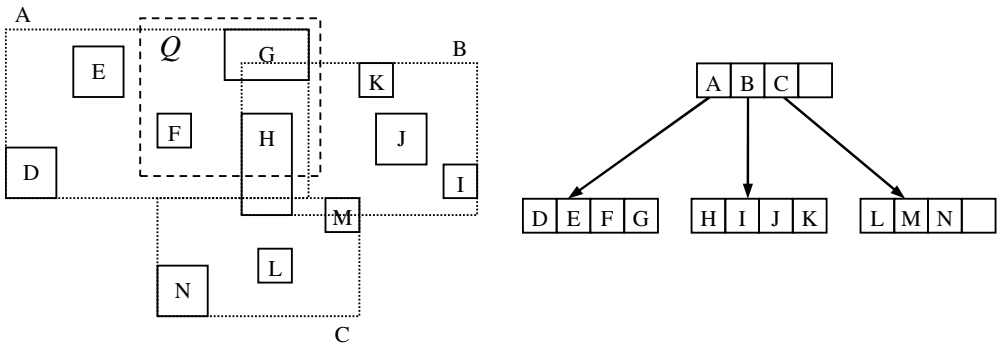


Figure 2.1: An example of spatial data, their Minimum Bounding Boxes (MBBs), a range query and the corresponding R-tree [MNPT05].

The R-tree can be considered as an extension of the B-tree in *n*-dimensional spaces. Similar to the B-tree, R-tree is a height-balanced tree with the index records in its leaf nodes containing pointers

to the actual data objects. Leaf node entries are of the form <*id*, *MBB*>, where *id* is an identifier that points to the actual object and *MBB* (Minimum Bounding Box) is an *n*-dimensional interval. Non-leaf node entries are of the form <*ptr*, *MBB*>, where *ptr* is a pointer to a child node, and *MBB* the bounding box that covers all child nodes. A node in the tree corresponds to a physical disk page (or disk block, which is the fundamental element on which the actual disk storage is organized) and contains between *m* and *M* entries (*M* is the node capacity and *m* is a tuning parameter - usually *m* is set to *M*/2 which guarantees that the space utilization is at least 50%). Contrary to the B-tree, node MBBs belonging to the same tree level are allowed to overlap. Figure 2.1 illustrates a set of spatial objects and the corresponding R-tree.

In the domain of *spatio-temporal indexing*, R-tree variations and extensions include, among others, 3D R-trees [TVS96], TB-trees and STR-trees [PJT00], Octagon-Prism trees OP-tree [ZSI02], PA-trees [NR07], MON-trees [AG05], while SETI [CEP03] is a hybrid R-tree-based and partition-based technique. We will thoroughly examine them in the next sections. Moreover, since our interest in this thesis focuses on historical MODs, we restrict our discussion to indexing techniques recording past locations. The reader interested in indexing current locations and motion vectors can find very interesting work in [SJ02], [SJLL00], [TPS03], and [XP03].

Table 2.1: Classification of spatio-temporal queries (extracted from [Pfo02])

Query Type		*Operation*
Coordinate-Based Queries		overlap, inside, nearest neighbor, etc.
Trajectory-Based Queries	Topological Queries	enter, leave, cross, bypass, etc.
	Navigational Queries	traveled distance, covered area, speed, heading, parked, etc.

2.1.1. Specifications for Trajectory Indexing

As pointed out in [PJT00], the vast majority of the proposed spatio-temporal indexes overlook the challenges posed by the nature of trajectory data, and they just index collections of line segments in the spatio-temporal space, only concerning about the processing of traditional *coordinate-based* queries (such as range and timeslice queries), ignoring at the same time other useful queries, such as topological and navigational queries, which are *trajectory-based*. In particular, queries of the form "*find all objects located within a given area during a certain time interval*" generalize the spatial range query of the form "*find all objects within a given area*" and do not take the notion of trajectory into consideration; thus, called *coordinate-based* [PJT00]. Queries of the form "*find all objects' locations within a given area at a certain time instance*", called *timeslice* queries, constitute a special type of range queries where the temporal extent is set to zero. Another straightforward extension of pure spatial queries in the domain of spatio-temporal applications includes *nearest neighbor* queries of the form "*find the nearest moving object to a query object during a certain time interval*". Moreover, in the case of spatio-temporal nearest neighbor queries, the query object could be a 2D point or another moving object trajectory, while the query would return the nearest to the query object at any time

during a time interval, or, in every time instance of the query time interval (historical continuous queries).

Furthermore, [PJT00] propose to call *trajectory-based* the queries which require the knowledge of the complete – or at least of a subset of the – object's trajectory in order to be processed. Such queries are those considering topological relations (enter, leave, etc.) and those providing derived information about an object's navigation (average speed, traveled distance etc.). Table 2.1 summarizes the above two query types.

The combination of range and topological queries produces another type of queries called *combined queries.* As an example [PJT00], consider the following query "*What were the trajectories of objects after they left Tucson street between 7 a.m. and 8 a.m. today, in the next hour*", which firstly locates the trajectories contained in an inner range query window (*Tucson street*, *between 7 a.m. and 8 a.m. today*, Q_{in} in Figure 2.2) and then retrieve those parts of objects' trajectories contained in an outer query window (*in the next hour*, Q_{out} in Figure 2.2).

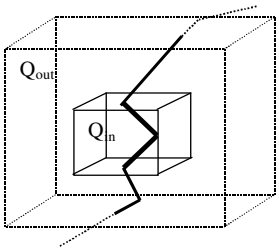


Figure 2.2: Combined search queries

In another line of research, [MB04] recently address the need for efficient trajectory compression mechanisms; according to that work, it is expected that all the ubiquitous positioning devices will eventually start to generate an unprecedented data stream of time-stamped positions. Sooner or later, such enormous volumes of data will lead to storage, transmission, computation, and display challenges. Hence the need for compression techniques arises. However, existing spatio-temporal indexes not preserving moving object trajectories and dealing with the spatio-temporal data as collection of line segments in the 2+1 dimensional space (such as SETI [CEP03] and 3D R-tree [TVS96]), overlook the need for compression, which by definition requires treating each trajectory as a single object. The same need for trajectory preservation arises when dealing with deletion operations; albeit the deletion of a line segment from trajectory database may sound meaningless, the deletion of an entire trajectory is a very useful operation which has to be supported by any real-world trajectory index.

Two index structures presented in [PJT00], namely the Spatio-Temporal R-tree (STR-tree) and the Trajectory Bundle tree (TB-tree), try to fulfill these needs and to efficiently support trajectory-based operations such as topological query processing. The outcome of this work was that the TB-tree could support non-traditional queries much more efficiently than the traditional 3D R-tree and the STR-tree. Unfortunately, in spite of its clear advantages on trajectory-based query processing, the TB-tree has a crucial drawback: because of its insertion strategy, new trajectory data are always inserted at the right 'end' of the tree, leading its performance to heavily depend by the order of data insertion. However, in real-world applications, this assumption is not guaranteed to be always true. For example, in an application where insertions occur in real-time, if the moving object enters an area where the

position transmission system does not function, its trajectory could be stored locally in the object and be transmitted to the central server – where the index operates – at a later time; meanwhile, other moving objects could have transmitted their positions, violating the above TB-tree assumption. Furthermore, the structure of the TB-tree is not suitable for supporting deletion and compression operations; a trajectory deletion would leave 'holes' in the nodes, and trajectory compression as we will discuss in the sequel, requires the index to handle data inserted in non-chronological order.

Another interesting approach regarding the indexing of spatio-temporal trajectories arises by acknowledging that trajectories are more likely to be network–constrained. As pointed out in [KGT99], the existence of restrictions in the space in which moving objects realize their movement is a condition that can be used to improve the performance of spatio-temporal indexes. Actually, this is the case in most real-world applications: planes fly in air-paths, cars and pedestrians move on road networks, while trains have fixed trajectories on railway networks. These kinds of special conditions (moving restrictions) have been the subject of research interest [KGT99], [PTKZ02]. More specifically, according to Kollios et al. [KGT99], the domain of the trajectories of objects moving on a network is not the 2+1 dimensional space, rather than, a space with 1.5 dimensions, as line segments comprising the network can be stored in a conventional index of spatial data (such as the R-tree). Then, indexing of objects moving in such a network is reduced to a one-dimensional indexing problem. In [KGT99], the problem of network-constraint trajectory indexing is studied under a more theoretical view rather than actually proposing an access method that could be used in real-world applications. On the other hand, following the directions provided by [KGT99], in the next sections, we show how the intuition of [KGT99] can be realized by developing novel access methods for indexing network-constraint trajectory data.

2.1.2. What is proposed

In order to deal with the above requirements, in this work, two novel indexes are independently proposed, namely, the TB*-tree and the FNR-tree. In particular, the TB*-tree is an extension of the TB-tree that overcomes the drawback of its predecessor, that is, the need for *trajectory preservation* and the need for *non-chronological insertions*, preserving at the same time all of its 'desired' properties. Moreover, apart from the construction and query processing algorithms, the TB*-tree supports *trajectory deletions*, while its structure makes it capable of supporting *trajectory compression* algorithms as well. The TB*-tree structure and algorithms will be demonstrated in the next sections, followed by an experimental study which reveals the positive and negative aspects of the proposed index. It is essential however to clarify that the proposed TB*-tree, does not exploit the special conditions that objects have when moving on fixed networks; quite the opposite, it indexes objects moving freely in the 2D space.

On the other hand, under the network-constraint scenario this thesis provides a novel index, called Fixed Network R-tree (FNR-tree) which is an extension of the well-known R-tree [Gut84]. The general idea that describes the FNR-tree is that of a forest of 1D (1D) R-trees on top of a 2D (2D) R-tree. The 2D R-tree is used to index the spatial data of the network graph (e.g., roads consisting of line segments), while the 1D R-trees are used to index the time interval of each object's movement on a given segment of the network. As it will be shown experimentally in the next sections, the proposed

FNR-tree outperforms the TB-, the TB*- and the 3D R-tree in general coordinate-based queries; however the demonstrated efficiency of the FNR-tree in coordinate-based queries comes for the cost of lacking a mechanism which preserves trajectories, making it therefore unable to support trajectory-based queries.

2.2. Related Work

In the sequel, the related work in the field of indexing historical trajectories of moving objects is briefly examined. It is essential to note that we do not include all these structures in our experimental study since their majority was proposed during the elaboration of this thesis; nevertheless, some of the examined related work cites, and is compared with, a preliminary version of the FNR-tree presented in [Fre03], while the others are also evaluated against the original 3D R-tree [TVS96] and TB-tree [PJT00]. We fist discus structures indexing objects moving in unconstrained space, while in the sequel, we present some of the network-constraint approaches.

2.2.1. Indexing the Trajectories of Objects Moving in Unconstrained Space

A first enhancement of the TB-tree was proposed by Zhu et al. [ZSI02], which extend the work of [PJT00] by proposing the *Octagon-Prism tree* (OP-tree); OP-trees use octagon approximations instead of MBBs. Based on the conducted experiments, OP-trees are shown to outperform the original TB-tree on both range and trajectory based queries. Here, it is important to note that the TB* modifications regarding the original TB-tree (i.e., replacement of 3D line segments by 3D points and the altered insertion strategy) may be directly applied in the context of the OP-tree by simply replacing MBB approximations with octagons.

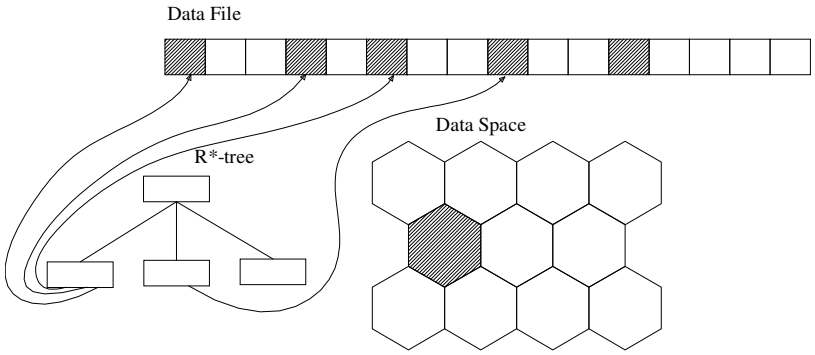


Figure 2.3: The SETI [CEP03] structure

The *Scalable and Efficient Trajectory Index* (SETI) presented in [CEP03] is a hybrid structure that indexes trajectories at two levels in order to disjoint the spatial from temporal indexing. Acknowledging that trajectory data sets continually expand the temporal dimension while the spatial boundaries remain static or at least rarely change, SETI partitions the 2D space into disjoint hexagon cells which remain static during the structure's lifetime; other adaptive spatial partitioning strategies can also be used. Each cell logically contains only those trajectory segments that are completely within the cell, while in the case of a trajectory segment that crosses the cell boundary, it is split and subsequently inserted into both cells. In physical level, trajectory segments are inserted into a data file; each page of the data file contains segments from only one cell. Then, a temporal index (i.e., a 1D R-

tree) indexing the time intervals of each particular cell in the data file, is assigned to the corresponding cell. Figure 2.3 summarizes the SETI structure.

The insertion and searching algorithms follow a multi-step approach composed of spatial filtering, temporal filtering and refinement. In particular, during each insertion, the algorithm locates the cell into which the segment has to be inserted (considering also possible splits between cells), and then inserts it in the corresponding page of the data file, updating at the same time the corresponding entry of the 1D R-tree (if this is necessary). Although as presented in the experimental study of [CEP03], SETI clearly outperforms the 3D R-tree and the TB-tree in time-interval and time-slice queries, it cannot be used to process trajectory-based queries. This is due to the fact that trajectory line segments are organized inside the index based only on their spatial and temporal relations; as such, successive line segments of the same trajectory may be placed in different disk pages. Therefore, in the worst case scenario the retrieval of a single trajectory would require to read one disk page for each trajectory line segment. Moreover, the work of [CEP03] do not provide any nearest neighbor query processing algorithm, while the development of an efficient one is not a straightforward task.

Pfoser et al. [PJ01] use the restrictions placed in the movement of objects by the existing infrastructure in order to improve the performance of spatio-temporal queries executed against a spatio-temporal index. The strategy followed does not affect the structure of the index itself. Instead, [PJ01] adopt an additional pre-processing step before the execution of each query. In particular, provided that the infrastructure is rarely updated, it can be indexed by a conventional spatial index such as the R-tree. On the other hand, a general-purpose spatio-temporal index, such as the TB-tree [PJT00] or the 3D R-tree [TVS96] can be used to index trajectories of moving objects. Then, a pre-processing step of the query, divides the initial query window in a number of smaller windows, from which the regions covered by the infrastructure have been excluded (Figure 2.4). Each one of the smaller queries is executed against the (general-purpose spatio-temporal) index returning a set of candidate objects, which are finally refined with respect to the initial query window.

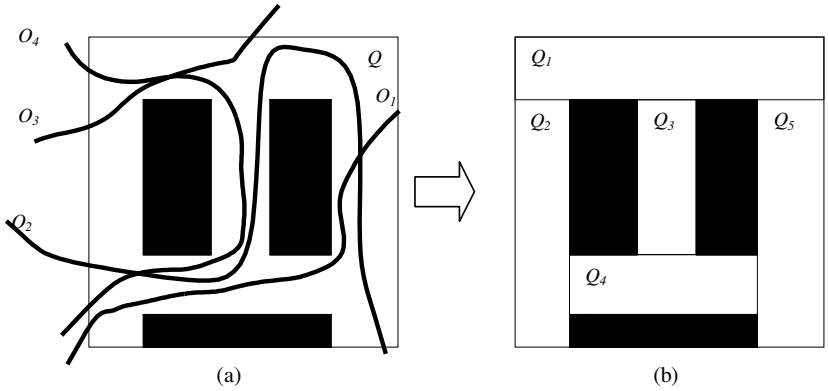


Figure 2.4: The initial query window *Q* (a) is decomposed into a number of smaller query windows *Q1*, *Q2*,.. (b) with respect to infrastructure elements (drawn in black).

In the same paper [PJ01], an algorithm is provided for the implementation of the query pre-processing step, based on the work presented in [KF93]. According to [KF93], the number of node accesses required by an R-tree-based index to answer a window query, depends not only on the

window area but also on its extent per dimension. Consequently, what concerns is not only the minimization of the area of the query window (which is achieved by removing the section containing the infrastructure from the initial window) but also the minimization of its perimeter. In the corresponding evaluation, the performance of two spatio-temporal indexes (TB- and 3D R-tree) was compared, either using the described query pre-processing step (i.e., dividing the initial window in smaller windows) or not, and it was shown that the query performance was improved for both indexes when this step was used.

Recently, work has been also done on how to optimally split trajectories for the purpose of improving range query performance [HKTG02], [HKTG06]. Hadjieleftheriou et al. [HKTG02] use a partially persistent structure, the PPR-tree, trying to confront the problem of the dead space generated by MBB approximations of moving object trajectories. Dead space is termed as the amount of space in an MBB approximation which does not actually covers any object contained inside it. [HKTG02] introduce "artificial object updates" partitioning the trajectories into smaller elements, thus reducing the dead space; they use non-linear functions to describe the moving objects' trajectories, which are initially indexed by the PPR-tree. This work is extended in [HKTG06] where a Multi-Version R-tree, such as the one proposed in [TPS03] is used instead of the PPR-tree, leading to an indexing schema with improved performance. Moreover, the proposed algorithms for handling the problem of the dead space introduced in MBBs can be used in combination with any spatio-temporal data archive, such as the R-tree and its variants.

However, the most promising approach regarding the indexing of moving object trajectories in unconstrained space is the one presented in [NR07]; according to [NR07], MBBs are not able to capture the smoothness of actual trajectory data, they propose that trajectories should be approximated as a sequence of movement functions with single continuous polynomial. They subsequently introduce the PA-tree, a parametric index that indexes the resulted polynomials; PA-trees resemble R-trees, with the main difference that entries consists of polynomial coefficients, rather than MBBs. According to the experimental study presented, PA-tree outperforms both MVR-tree [HKTG06] and SETI [CEP03] in the majority of the experimental settings.

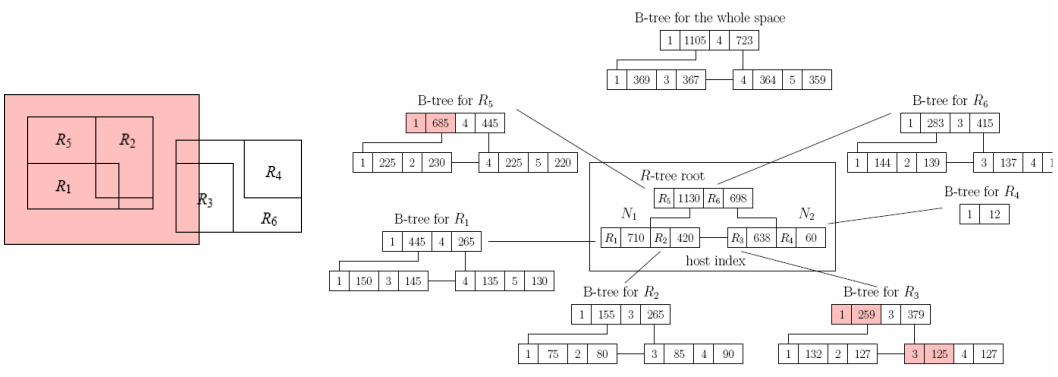
2.2.2. Indexing the Trajectories of Objects Moving in Fixed Networks

The first proposal considering network-constrained moving objects was the work by Papadias et al. in [PTKZ02] which adopted this assumption, in order to create a structure that answers spatio-temporal aggregate queries of the form "*find the total number of objects in the regions intersecting some window q_s during a time interval q_t*". Same as the FNR-tree, the proposed *aggregate R-B-tree* (aRB-tree) follows the intuition of [KGT99] and provides a combination of R- and B-trees based on the following idea: the lines of the network are stored only once and indexed by an R-tree. Then, in each internal and leaf node of the R-tree, a pointer to a B-tree is placed, which stores historical aggregate data about the particular spatial object (e.g. the MBB of the node).

In particular, this approach is based on two types of indexes: a *host index*, which manages the region extents and associates to these regions an aggregate information over all the timestamps in the base relation and some *measure indexes* (one for each entry of the host index), which are aggregate temporal structures storing the values of measures during the history, complete the proposed structure.

For a set of static regions, the authors define the aRB-tree, which adopts an R-tree with summarized information as host index, and a B-tree containing time-varying aggregate data, as measure index.

As already stated, the aRB-tree is well suited for the efficient processing of a *window aggregate query*, i.e., for the computation of the aggregated measure of the regions which intersect a given window. Indeed, for nodes that are totally enclosed within the window query, the summarized measure is already available thus avoiding descending these nodes. As a consequence, the aggregate processing is made faster. For instance, let us compute the number of phone calls inside the shaded area in Figure 2.5(a) during the time interval $[T_1,T_3]$ using the aRB-tree of Figure 2.5(b). Since R_5 is completely included in the window query there is no need to analyze R_1 and R_2 hence one accesses the B-tree for R_5. The first entry of the root of this B-tree contains the measure for the interval $[T_1, T_3]$ which is the value we are interested in. Instead, in order to obtain the sum of phone calls in the interval $[T_1, T_3]$ for R_3 one has to visit both an entry of the root of the B-tree for R_3 and also one leaf (the colored nodes). Figure 2.5 illustrates an example of the aRB-tree structure



(a) (b)

Figure 2.5: (a) Example data and (b) the corresponding aRB-tree [PTKZ02]

Exploiting the same property of a spatial network, a variation of the FNR-tree, called *Moving Objects in Networks tree* (MON-tree), has been proposed in [AG05]. Instead of using one 1D R-tree for every leaf node of the 2D R-tree, the MON-tree utilizes a 2D R-tree for every polyline of the spatial network. The MON-tree is shown to significantly outperform the 3D R-tree and the FNR-tree, in time-interval and time-slice queries, and is currently considered the state-of-the-art. However, it also shows the same disadvantage with the previously described schemes, being unable to efficiently process trajectory-based queries.

Another interesting methodology on the same subject (i.e., indexing of objects moving on networks) is presented in [PJ03]. This approach suggests the mapping of the underlying network from two to one dimension by sorting the network edges according to their Hilbert values. Hilbert values is an approach for ordering the 2D space; they are determined by applying a Hilbert curve covering the 2D space, mapping each 2D to a 1D point [WD04]. Then, the problem of indexing three (i.e., 2 spatial + 1 temporal) dimensions is reduced to the problem of indexing two (i.e., 1 spatial + 1 temporal) dimensions, which can be efficiently handled by employing any existing simple spatial index as the well known R-tree which is supported by existing DBMS. After that, each range query has to be mapped accordingly to the reduced one-dimensional space, producing thus a number of two-

dimensional (spatial and temporal) rectangles, which are subsequently posed against the R-tree. The technique also uses an R-tree to index the underlying network so as to speed up the query mapping process. The experimental study presented in [PJ03] shows that the proposed method clearly outperforms the three-dimensional approach (e.g., 3D R-tree, treating time as an extra spatial dimension) as the query size increases; the respective experimental study includes neither FNR nor MON-tree. Moreover, there is no obvious way on how this approach [PJ03] can process trajectory based queries.

2.3. Indexing the Trajectories of Objects Moving in Unconstrained Space

Before describing in detail the structure and algorithms of the TB*-tree, it is essential to briefly introduce the original TB-tree on which the former is based.

2.3.1. The TB-tree

Practically, the first index proposed to support trajectory-based queries was the Trajectory Bundle tree (TB-tree) [PJT00], which is fundamentally different from other spatio-temporal access methods mainly because of its insertion and split strategy. Similar to the original R-tree, the TB-tree is a height-balanced tree with the index records in its leaf nodes; leaf nodes contain entries of the same trajectories, and are of the form <*MBB, Orientation*>, where *MBB* is the 3D bounding box of the 3D line segment belonging to an object's trajectory (handling time as the third dimension) and *Orientation* is a flag used to reconstruct the actual 3D line segment inside the MBB among four different alternatives that exist (see Figure 2.6). Since each leaf node contains entries of the same trajectory, object *id* can be stored once in the leaf node header.

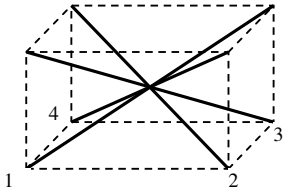


Figure 2.6: Alternative ways that a 3D line segment can be contained inside a MBB

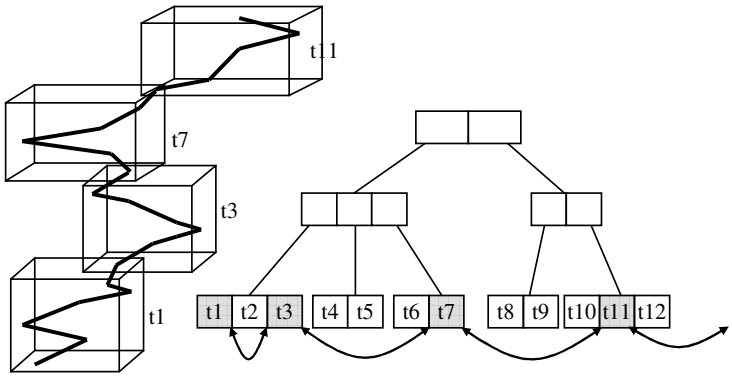


Figure 2.7: The TB-tree structure

However, contrary to the majority of the R-tree variations, its insertion algorithm is not based upon the spatial and temporal relations of moving objects but it relies only on the moving object

identifier (*id*). When new line segments are inserted, the algorithm searches for the leaf node containing the last entry of the same trajectory, and simply inserts the new entry in it, thus forming leaf nodes that contain line segments from a single trajectory. If the leaf node is full, then a new one is created and is inserted at the right-end of the tree. For each trajectory, a double linked list connects the leaf nodes that contain its portions together (Figure 2.7), resulting in a structure that can efficiently answer trajectory-based queries.

On the other hand, the TB-tree performs modestly on range queries as shown in [PJT00] because its data organization does not consider keeping together entries that lie close in 2D space. A second, perhaps more crucial, drawback is that its construction algorithm makes a consideration that positions of moving objects are most probably inserted in a chronological fashion, thus it does not favor the insertion of a position at time t_i when the latest position of any object already inserted in the index, corresponds to timestamp $t_j > t_i$. However, in real-world applications, this assumption is not guaranteed to be true. As already mentioned, if we assume that an object enters an area where the position transmission system does not function, its trajectory could be stored locally in the object and be transmitted at a later time; meanwhile other moving objects could have transmitted their positions, violating the above TB-tree assumption.

In the next section, acknowledging the basic advantages of the TB-tree on trajectory preservation, we develop a novel index, called TB*-tree, which overcomes the drawbacks of its predecessor while preserving all of its 'desired' properties.

2.3.2. The TB*-tree

The need for an index that supports insertions of object positions independently, the need for deletion support, the trajectory preservation and the efficiency for both coordinate-based and trajectory-based queries are the main requirements for the new index. In the following, we present the structure of the TB*-tree as well as algorithms for inserting, deleting, compressing, and querying object trajectories.

It is important to notice that, contrary to the original TB-tree, the TB*-tree does not care whether or not entries are inserted in chronological order. There is still an assumption on the trajectory itself (that also holds for TB-tree): entries belonging in the same trajectory are inserted in chronological order, i.e., the index does not permit the insertion of a position at time t_i when the latest position already inserted in the index *for the same object*, was at $t_j > t_i$. Even this can be easily relaxed, as will be sketched in section 2.3.2.2.1.

2.3.2.1. The TB-tree Structure*

In the original TB-tree, every time a moving object updates its position, a new 3D line segment is inserted in it using the insertion algorithm described in [PJT00]. This fact leads to storing each 3D point of the moving object's trajectory twice: once as an ending point and once as a starting point. While this would be necessary for a structure storing entries from different trajectories in its leaf nodes (e.g. the 3D R-tree [TVS96] and the STR-tree [PJT00]), it is waste of space in the TB-tree: by definition, line segments stored in the same leaf node belong to the same trajectory.

Instead of 3D line segments, TB*-tree leaf nodes store 3D points forming together a 3D polyline that represents a part of the exact trajectory of the object. Moreover, since the object *id* is stored once in

the header of the leaf node, TB*-tree leaf node entries consist of 3D points only (the *Orientation* flag is redundant). The single 3D points that appear twice are the ones at the end of a leaf node and at the start of its consecutive node (Figure 2.8). While these happen at the leaf level, the structure of non-leaf nodes remains the same as in the original TB-tree.

Formally, TB*-tree leaf nodes are of the form $\langle header, \{P_i\}\rangle$, where each $P_i = \langle t_i, x_i, y_i\rangle$ and $header = \langle id, \#entries, ptr\rangle$ (in other words, the object identifier, the number of node entries and a pointer to the parent node). On the other hand, non-leaf nodes are of the form $\langle header, \{E_i\}\rangle$, where each $E_i = \langle MBB_i, ptr_i\rangle$ with MBB_i be the enclosing 3D box of the child node pointed by ptr_i a pointer to it, and $header = \langle \#entries, ptr\rangle$ simply stores the number of node entries and a pointer to the parent node. Furthermore, similar to SETI [CEP03] and in order to support high insertion rates, the TB*-tree uses an in-memory hashed front-line structure, which maintains tuples of the form $\langle id, P_{curr}, N_{curr}\rangle$ with the object identifier *id*, its latest position $P_{curr} = \langle t_{curr}, x_{curr}, y_{curr}\rangle$ and a pointer N_{curr} to the leaf node containing P_{curr}.

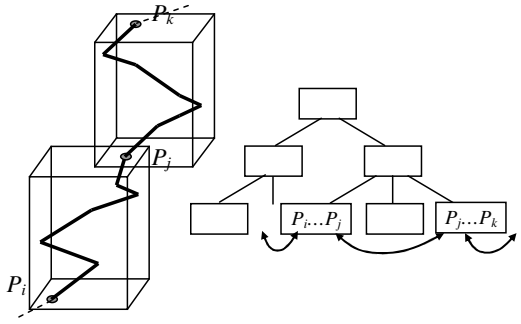


Figure 2.8: The single points appearing twice in the TB*-tree are the starting and ending ones at each leaf.

2.3.2.2. The TB-tree Algorithms*

In the sequel, we provide algorithms for maintaining the index by inserting a new position, deleting a trajectory, and compressing the index. As for query processing regarding the algorithms for range, trajectory based and combined query processing, they are identical to those presented in [PJT00] for the original TB-tree. Furthermore, the algorithms used for advanced query processing, such as nearest neighbor and most similar trajectory, will be examined in the next chapters. Nevertheless, for the sake of completeness, we include the range search algorithm in our discussion, which is essentially the `FindLeaf` algorithm originally proposed in [Gut84] for the original R-tree.

2.3.2.2.1. Inserting new Trajectory Segments

The insertion algorithm of the TB*-tree is executed every time a moving object *id* transmits its (new) position P_{curr}, thus making, with the help of the front-line structure, a new entry to be inserted in the tree rooted by *Root*. The *Insert* algorithm is illustrated in pseudo-code in Figure 2.9. The presented pseudo-code includes comments that explain each step of the algorithm. Just note that it is one entry, P_{curr}, which is inserted in the index except the case of a full node where the algorithm results to the creation of a new node with two entries, the latest already indexed, P_{prev}, and the new position, P_{curr}.

Also, by adding the front-line structure, finding the appropriate leaf node turns out to be a simple procedure (in contrast to the expensive `FindNode` algorithm for the TB-tree described in [PJT00]).

```
 1.  Algorithm Insert(node Root, int Id, 3D Point Pcurr)
 2.     // Algorithm TB*-tree Insert
 3.     // Find leaf node NN containing previous segment
 4.     NN = FrontLine(Id).LastNode
 5.     Pprev = FrontLine(Id).P_curr
 6.     // If NN exists and has space, insert Pcurr in it and propagate
 7.     // changes upwards using Guttman's AdjustTree
 8.     IF NN exists
 9.        IF NN has space
10.           Insert Pcurr in node NN
11.           AdjustTree (NN)
12.           // If, after the insertion of Pcurr, node NN becomes full,
13.           // delete and reinsert its entry in parent node using
14.           // Guttman's delete and insert algorithms
15.           IF NN is full
16.              PN = NN.Parent
17.              PE = PN.Entry_pointing_to(NN)
18.              Delete (Root, PE)
19.              Insert (Root, PE)
20.           ENDIF
21.        ELSE
22.           // Otherwise, create a new node, insert Pprev and Pcurr in
23.           // the new node and update the front-line
24.           NNode=InsertInNewNode(Root,Pprev,Pcurr)
25.           FrontLine(Id).LastNode = NNode
26.        ENDIF
27.     ELSE
28.        NNode = InsertInNewNode(Root,Pprev,Pcurr)
29.        FrontLine(Id).LastNode = NNode
30.     ENDIF
31.     FrontLine(Id).P_curr = Pcurr
```

Figure 2.9: The TB*-tree `Insert` Algorithm

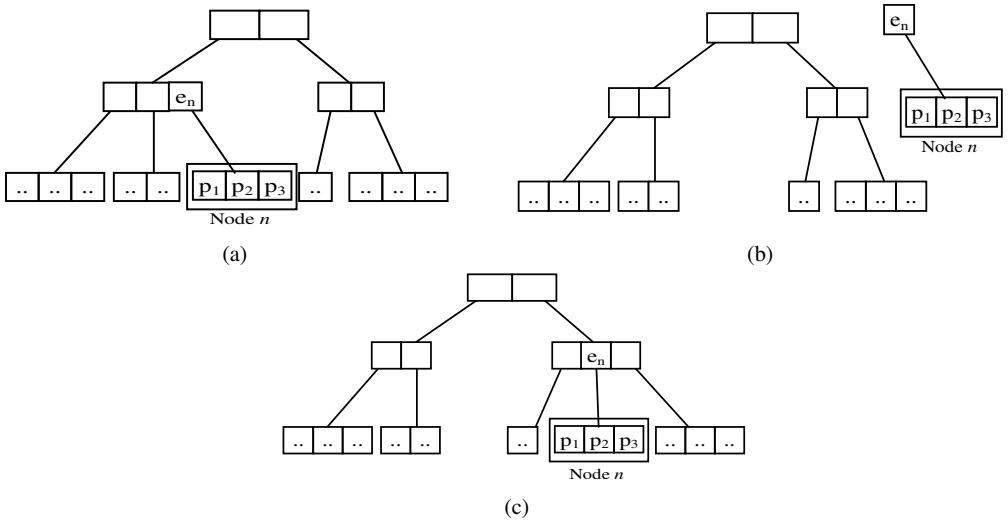


Figure 2.10: The strategy followed when a leaf node becomes full: (a) The leaf node *n* becomes full (b) Entry e_n is deleted from the tree, and (c) Entry e_n is re-inserted in the tree

A major modification in comparison with the original TB-tree takes place when a leaf node becomes full (Figure 2.10). Then, the algorithm locates the leaf node's parent entry and deletes it from the tree using Guttman's classic R-tree `Delete` algorithm [Gut84]. Then, the entry is re-inserted in the

tree, using Guttman's `Insert` algorithm, but it is placed higher in the tree (at the level above the leaf level), so that the (leaf) node that the entry brings together is located at the same level with the rest leaves – a technique also used in the original R-tree `Delete` algorithm. With this technique, when a leaf node gets full it is placed in a 'better' position, in terms of spatial neighborhood, since Guttman's `Insert` algorithm uses the least enlargement criterion in order to find the node in which to place the entry. This "delete and re-insert" technique, originally used in the R*-tree [BKSS90], is the reason for calling this novel index, TB*-tree.

Another major difference from the original TB-tree concerns the creation of new leaf nodes and the choice of the location where the new leaf nodes are placed. For this purpose, a new algorithm called `InsertInNewNode` is developed (pseudo-code in Figure 2.11), which uses Guttman's `ChooseLeaf` and `AdjustTree` algorithms [Gut84]. As already discussed, the algorithm initially places two points, P_{prev} and P_{curr}, in the new leaf (cf. Figure 2.8).

Differently from the TB-tree construction, `InsertInNewNode` algorithm of the TB*-tree finds the leaf node next to which the new leaf should be placed using the least enlargement criterion (Guttman's `ChooseLeaf` algorithm). Then, Guttman's `AdjustTree` algorithm is invoked passing both leaf nodes – the one returned by `ChooseLeaf` and the newly created one – such as it would happen if the node returned by the `ChooseLeaf` was previously split. Finally, if the procedure causes the root node to split, then the tree grows taller by creating a new root whose children are the two resulted nodes.

```
1.  Algorithm InsertInNewNode(node Root, 3D Point P_prev, 3D Point P_curr)
2.     // Algorithm TB*-tree InsertInNewNode
3.     Create New Leaf Node NNode
4.     Insert P_prev in node NNode
5.     Insert P_curr in node NNode
6.     // Find Position for the new Node using Guttman's ChooseLeaf
7.     L = ChooseLeaf (Root, (P_prev,P_curr))
8.     // Propagate changes upward
9.     AdjustTree (L, NNode)
10.    // Grow tree taller
11.    IF AdjustTree caused the Root to split
12.       Create a new Root NRoot
13.       Insert first resulted node in NRoot
14.       Insert second resulted node in NRoot
15.    ENDIF
16.    // Return the new Node
17.    RETURN NNode
```

Figure 2.11: The `InsertInNewNode` algorithm

With reference to the assumption of the TB*-tree that entries belonging to the same trajectory are inserted in chronological order, this only happens in order to keep the insertion procedure simple (a new position is inserted either in the 'current' node – as indicated by the front-line structure – or in a new node, updating accordingly the in-memory front-line). Should this assumption be relaxed, a backward search in the double-linked list of nodes is required (starting from the 'current' node), the new ('outdated') entry is to be inserted in the appropriate node and, since all nodes before the 'current' node in the list are by definition full, an entry is to be moved from each node to its next node in the chain starting from the node the entry was inserted and ending at the 'current' node.

Finally, a buffering technique can be used to optimize the insertion process in terms of touched disk pages. In particular, further than using a traditional buffering mechanism (such as LRU), the TB*-tree structure can utilize an *additional* buffer, hereafter called *Last Page (LP) Buffer*, which would hold all leaf nodes not yet filled. Since each leaf node is expected to be completely filled with leaf entries, the LP buffer can be used in order to hold those leaves not yet completed; then when each leaf node is completely filled, it is saved on the disk just once, and the next (new) leaf node of the same trajectory takes its place on the LP buffer. Therefore, the size of the LP buffer will always be equal with the number of trajectories currently indexed by the TB*-tree. As it will also be shown in the experiments, the LP buffer dramatically reduces the number of disk page accesses required for insertions.

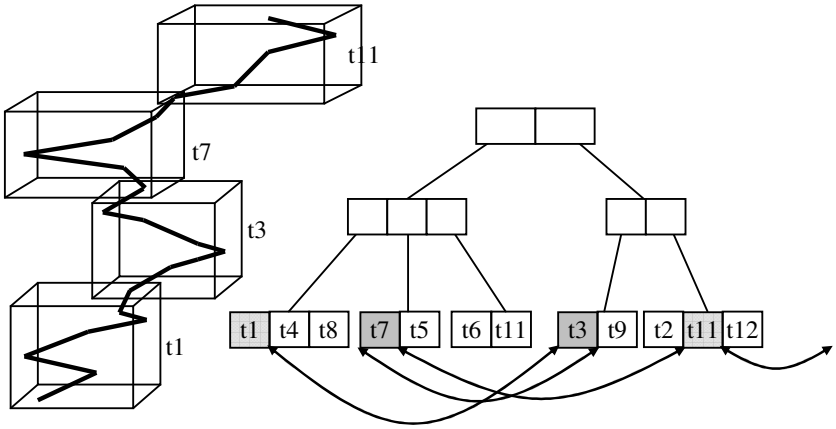


Figure 2.12: The TB*-tree structure

The general picture of the TB*-tree is illustrated in Figure 2.12. Compared with the TB-tree (cf. Figure 2.7), it is clear that leaf nodes belonging to the same trajectory are no longer placed in increasing time order (e.g. from left to right), but are placed in locations determined by the least enlargement criterion.

2.3.2.2.2. Deleting Trajectories

Deletions are often neglected when proposing indexing methods for moving object trajectories, with the main argument that deleting a 3D line segment from an object's trajectory is meaningless. Although this might be assumed to be conceptually correct (transmitted positions are recorded, thus exist), deleting an entire object's trajectory is meaningful (trajectories of objects being no more useful could be deleted from the index). Therefore, we provide an efficient algorithm to support deletions of object trajectories. The input of the algorithm is the *id* of the trajectory to be deleted.

The `DeleteTrajectory` algorithm, illustrated in Figure 2.13, can be used in the TB*-tree in order to delete a moving object's trajectory. The algorithm initially locates the 'current' leaf node *N*. Then, it removes *N*'s parent entry from its parent node executing Guttman's R-tree `Delete` algorithm [Gut84] and follows the chain backwards to nodes containing parts of the same trajectory, deleting one after the other. If necessary, according to `Delete` algorithm, nodes are rearranged, e.g. if the number of entries falls under the *m=M/2* threshold, or even the tree may be forced to condense.

```
 1.  Algorithm DeleteTrajectory(int Id)
 2.     // Algorithm TB*-tree DeleteTrajectory
 3.     // Find latest trajectory leaf node N
 4.     N = FrontLine(Id).LastNode
 5.     // Delete leaf node N's parent entry using Guttman's Delete
 6.     // Algorithm and follow the pointers to the trajectory's previous
 7.     // leaf nodes deleting also their parent entries
 8.     DO UNTIL N Is NULL
 9.        PN = N.Parent
10.        PE = PN.Entry_pointing_to(N)
11.        Delete (Root, PE)
12.        N = N.PreviousLeaf
13.     LOOP
```

Figure 2.13: The `DeleteTrajectory` Algorithm

The structure of the TB*-tree looks ideal for providing such an algorithm: having located just one line segment belonging to an object's trajectory, one could follow the double-linked lists in order to retrieve the entire trajectory and delete leaf nodes that compose it. On the other hand, the original TB-tree cannot easily support trajectory deletions: node deletions result in deletions of entries in non-leaf nodes which either require condense techniques to be handled (such as the `CondenseTree` algorithm [Gut84]) or leave holes in the nodes. In any case, the 'desired' TB-tree properties (all leaf nodes but the 'current' ones are full; a chronological order of leaf nodes exists, etc.) are not prevented.

As for other index structures (such as the 3D R-tree [TVS96], the STR-tree [PJT00], the SETI [CEP03]), they by definition lack a mechanism to efficiently retrieve an object's entire trajectory; thus, in order to support trajectory deletions they have to answer sequential range queries such as described in [PJT00] for the combined search of the 3D R-tree and the STR-tree – a very expensive approach as shown in [PJT00].

2.3.2.2.3. Compressing the Index

While the original TB-tree satisfies the trajectory preservation requirement in order to utilize the TD-TR trajectory compression algorithm [MB04], such an algorithm would have to read each indexed trajectory one-by-one, compress it, and finally feed a new TB-tree with the compressed trajectory. However, since the TB-tree places new entries always at the right 'end' of the tree, such an approach would place entire trajectories on this side of the tree without considering their temporal ordering, thus leading to a tree with high temporal overlap decreasing its performance. Therefore, in order to overcome this drawback, we would have to utilize intermediate steps processing all trajectories indexed by the TB-tree, producing the new compressed ones, sorting them according to their temporal order and finally feed the new TB-tree. Nevertheless, such a technique would require processing the entire index in the main memory, or developing specialized algorithms to handle it efficiently. Moreover, the opening window spatio-temporal algorithm presented in [MB04] would be a solution; then again, such an approach would lead to utilize a less efficient compression algorithm in terms of both quality and compression.

On the contrary, the proposed TB*-tree does not show any of these disadvantages. Its insertion algorithm supports trajectory additions in non-chronological order. As such, in Figure 2.14 we present a simple algorithm which compresses a TB*-tree by utilizing the TD-TR algorithm [MB04]. The algorithm starts by creating a new TB*-tree, and then, using the hashed structure, it accesses the last

node of every trajectory. Then, following the pointers to the previous leaves, it retrieves the entire trajectory on which the TD-TR algorithm [MB04] is applied with the given threshold. Finally, the algorithm feeds the new TB*-tree with the compressed trajectory and repeats the same procedure for the remaining trajectories until all have been accessed.

```
 1.  Algorithm CompressIndex(double Threshold, TB*-tree TB)
 2.     // Algorithm TB*-tree CompressIndex
 3.     // Create a new TB*-tree
 4.     NTB = New TB*-Tree
 5.     FOR EACH Id IN TB.Trajectories
 6.     // Find latest trajectory leaf node N
 7.        N = FrontLine(Id).LastNode
 8.        // Create a new Trajectory retrieve all of its entries
 9.        Traj = New Trajectory
10.        DO UNTIL N Is NULL
11.           Traj.Add N.Segments
12.           N = N.PreviousLeaf
13.        LOOP
14.        // Apply the top-down spatiotemporal compression algorithm
15.        // TD-TR in the Trajectory with the given threshold
16.        TD-TR (Traj, Threshold)
17.        // Insert in the new TB*-tree each point P of the compressed
18.        // trajectory
19.        FOR EACH P IN Traj
20.           Insert NTB.Root, Id, P
21.        NEXT
22.     NEXT
```

Figure 2.14: The `CompressIndex` Algorithm

2.3.2.2.4. Querying the TB-tree*

As already mentioned, since both the TB- and TB*-tree are based on the well known R-tree, the respective range search algorithms follows the `FindLeaf` algorithm originally presented in [Gut84]. This algorithm recursively visits tree nodes, rejecting node MBBs that does not overlap the query window, while following the pointers from overlapping MBBs to their respective child nodes until all candidate leaf nodes have been found. Following the example illustrated in Figure 2.1 for spatial objects, consider a range query *Q* executed against the 2D R-tree. The algorithm starts by visiting the tree root, checking whether the MBBs of the root entries overlap *Q*. If a node entry MBB overlaps *Q*, the algorithm follows the pointer to the corresponding child node (entries *A* and *B* in our example), where it repeats recursively the same task. If the algorithm reaches a leaf node, leaf entries are examined against *Q* and if their MBB overlap, the algorithm reports their ids (objects *F* and *G* when the algorithm visits leaf node *A*, and object *H* when in node *B*). The extension of the above algorithm in the spatio-temporal domain is a straightforward task, where each 2D MBB is simply replaced by the respective 3D MBB of actual objects, nodes or queries.

2.4. Indexing the Trajectories of Objects Moving in Fixed Networks

As already mentioned, following the suggestions of [KGT99], in this thesis we propose the FNR-tree, an extension of the well-known R-tree [Gut84], designed to index objects moving on fixed networks. The FNR-tree can be considered as a forest of several 1D R-trees on top of a single 2D R-tree. The 2D R-tree is used to index the spatial data of the network (i.e., roads consisting of line segments), while each one of the (temporal) 1D R-trees, hereafter called *"Children 1D R-trees"*, corresponds to a leaf

node of the 2D R-tree and indexes the time intervals during which moving objects moved on network links that fall into the Minimum Bounding Box (MBB) of the corresponding 2D R-tree leaf node. As such, the (spatial) 2D R-tree remains static during the lifetime of the FNR-tree – as long as there are no changes in the network. An additional (temporal) 1D R-tree hereafter, called *"Parent 1D R-tree"* is used to index the leaf nodes of all the children 1D R-trees with respect to their lifetime. Hence, the time interval of each 1D R-tree's leaf node is inserted along with a pointer to the actual node as a new entry in the parent 1D R-tree. The overall structure of the FNR-tree is outlined in Figure 2.15, while Figure 2.16 (b) demonstrates an example for the configuration of objects illustrated in Figure 2.16 (a).

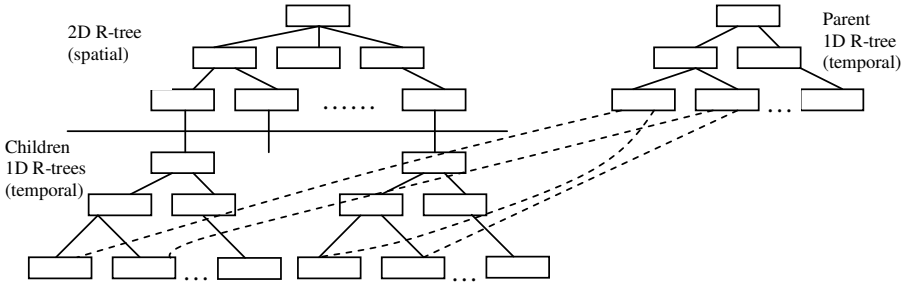


Figure 2.15: The FNR-tree structure

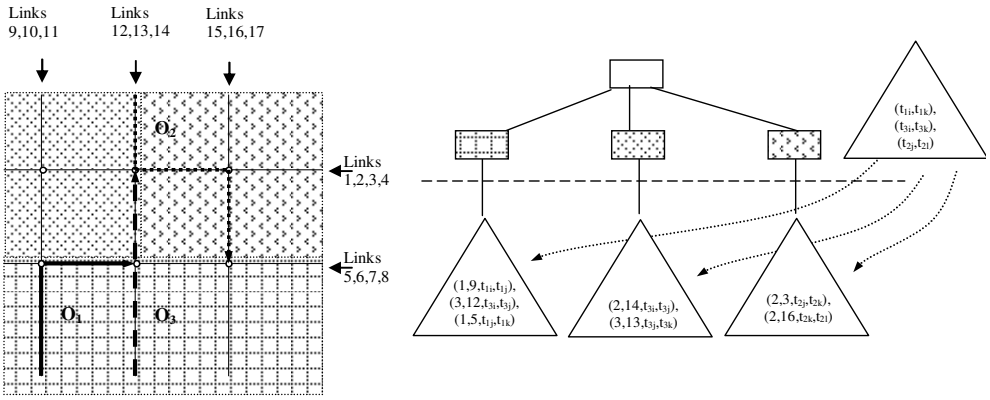
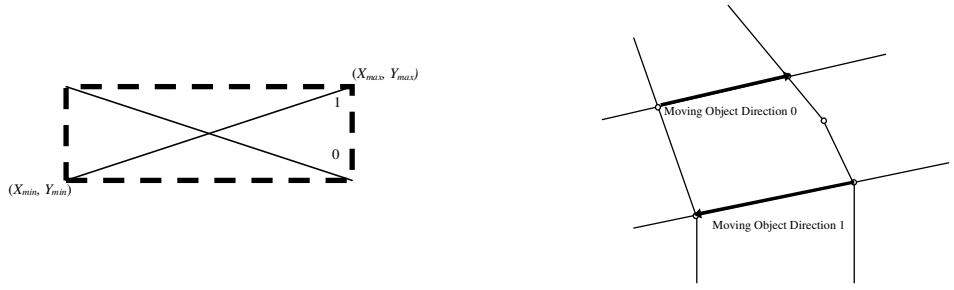


Figure 2.16: An FNR-tree example: (a) trajectories of three objects on a road network and (b) the corresponding FNR-tree components

2.4.1. The FNR-tree Structure

As already mentioned, the FNR-tree can be considerer as a 2D R-tree indexing the network line segments, along with a forest of 1D R-trees indexing time intervals. Following the standard R-tree structure, non-leaf nodes of the 2D R-tree are of the form $\langle header, \{ptr_i, MBB_i\}\rangle$, where each $MBB_i = \langle x_{min\text{-}i}, y_{min\text{-}i}, x_{max\text{-}i}, y_{max\text{-}i}\rangle$ and $header = \langle id, \#entries, ptr\rangle$. On the other hand, the structure of the 2D R-tree leaf nodes is slightly modified regarding the conventional R-tree; formally, leaf nodes are of the form $\langle header, \{link_i, MBB_i, orientation\}\rangle$ and $header = \langle id, \#entries, ptr, ptr_{child\text{-}R\text{-}tree}\rangle$. According to this form, the pointer normally located inside each leaf node entry has been replaced by an '*orientation*' flag (0/1) that describes the exact geometry of the line segment inside the MBB (Figure 2.17(a)). A

similar approach was followed in [PJT00] to represent segments of trajectories in 3D R-tree [TVS96]. Moreover, each 2D R-tree leaf node contains a pointer ($ptr_{child\text{-}R\text{-}tree}$) that points to the root of the corresponding child 1D R-tree.



(a) (b)

Figure 2.17: (a) The 'orientation' flag in 2D R-tree entries; (b) the 'direction' flag in 1D R-tree entries

Regarding the 1D R-trees, non-leaf nodes are of the form $\langle header, \{ptr_i, MBB_i\}\rangle$, while leafs are slightly different: $\langle header, \{Object\text{-}id_i, Link\text{-}id_i, MBB_i, direction\}\rangle$, and $MBB_i=\langle t_{in}, t_{out}\rangle$ is the time interval during which object with id $Object\text{-}id_i$ moved on the line segment with id $Link\text{-}id_i$, which is included in the MBB of the corresponding leaf of the 2D R-tree. *Direction* is another flag (0/1) that describes the heading of the moving object (Figure 2.17 (b)). Specifically, *direction* is set to 0 (1) when the moving object entered the line segment from the left-most (right-most) node. In the special case where the line segment is vertical, *direction* is set to 0 (1) for objects entered the line segment from the bottom-most (top-most) node. Finally, leaf node headers are of the form $header = \langle id, \#entries, ptr, ptr_{parent\text{-}R\text{-}tree\text{-}node}\rangle$ where $ptr_{parent\text{-}R\text{-}tree\text{-}node}$ stands for pointing directly from each 1D R-tree leaf node to the corresponding 2D R-tree leaf node.

Similar to the above is the structure of the parent 1D R-tree. Although the internal nodes remain identical with the former ones, leaf nodes differ to some extent: they are of the form $\langle header, \{ptr_{child\text{-}R\text{-}tree\text{-}node}, MBB_i\}\rangle$ *with* $MBB_i=\langle t_{min}, t_{max}\rangle$, and $ptr_{child\text{-}R\text{-}tree\text{-}node}$ pointing to the corresponding leaf node in the forest of the children 1D R-trees (cf. Figure 2.15).

2.4.2. The FNR-tree Algorithms

In the sequel, we provide algorithms for inserting a new entry in the FNR-tree (subsection 2.4.2.1) and searching the FNR-tree with respect to a query window (subsection 2.4.2.2).

2.4.2.1. Inserting new Trajectory Segments

The insertion algorithm of the FNR-tree is executed each time a moving object with $Object\text{-}id_i$ leaves a line segment of the network, represented by its MBB (x_{start}, y_{start}, x_{end}, y_{end}) and the *direction* flag. The list of arguments also includes the time interval (t_{in}, t_{out}) during which $Object\text{-}id_i$ was moving on the line segment. The insertion algorithm is illustrated in Figure 2.18.

In this algorithm, `R_tree_insert`, `R_tree_delete` and `R_tree_search` are Guttman's classic algorithms described in [Gut84] to maintain and search an (1D or 2D) R-tree. On the other hand, for the insertion done in line 11 (considering the children 1D R-trees of the FNR-tree), we notice that the 1D time intervals are inserted in the tree in increasing order for the reason that time is monotonic. This fact leads us to the following modification of Guttman's R-tree-insert algorithm,

hereafter called `Insert_most_recent`, also illustrated in Figure 2.19. Every new entry is simply inserted in the most recent (right-most) leaf of the 1D R-tree. In case this leaf node is full, a new node is created and the entry is inserted in it. The new leaf node is inserted in the structure as a sibling node of the (previously) most recent leaf. As such, it could cause propagates upwards using Guttman's `AdjustTree` algorithm, also described in [Gut84]. This insertion technique results to 1D R-trees with almost full leaves and very small overlapping.

```
 1.  Algorithm FNR_tree_Insert(object_id, xstart, ystart, xend,
 2.  yend, tin, tout)
 3.      // Search the line segment with Link_id in the 2D R-tree
 4.      // that object_id leaves
 5.      Link_id = 2D_R_tree_search(xstart, ystart, xend, yend)
 6.      // follow the pointer from leaf node that contains link_id
 7.      // to the corresponding 1D R-tree, RT
 8.      RT=link_id.Child_R_tree
 9.      // Insert the time interval into the 1D R-tree
10.      // Let ND be the leaf node where the input was inserted
11.      RT.Insert_most_recent(tin, tout, object_id, link_id, ND)
12.      // If necessary, update the Parent 1D R-tree by inserting or
13.      // updating the MBB of node ND
14.      IF ND is a new node caused by the insertion
15.         Parent_1D_R_tree_insert(ND.MBB, ND.ptr)
16.      ELSEIF the ND.MBB was modified
17.         Parent_1D_R_tree_delete(ND.MBB, ND.ptr)
18.         Parent_1D_R_tree_insert(ND.MBB, ND.ptr)
19.      ENDIF
```

Figure 2.18: FNR-tree Insertion Algorithm

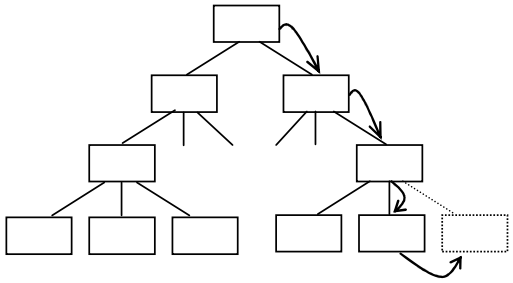


Figure 2.19: New entries are always inserted in the right-most node of each 1D R-tree when insertions are performed in chronological order

Nevertheless, given the discussion of section 2.3.2, the strategy of `Insert_most_recent` can be considered as a drawback in several application domains where new trajectory segments insertions do not necessarily follow the timeline. In order to deal with this requirement of non-chronological insertions, the FNR-tree can be also implemented employing the simple `R_tree_insert` [Gut84] algorithm in line 11, an approach which enables the index to efficiently handle line segments inserted in arbitrary time-order. Concluding, depending on the application, the FNR-tree may or may not efficiently support non-chronological insertions by simply modifying line 10. Finally, the insertion and deletion algorithms used in lines 14-18 are the conventional R-tree algorithms [Gut84] mainly due to updates required (deletions and re-insertions).

As illustrated in Figure 2.20, the insertion algorithm is executed when the moving object reaches a node (*Node j*) of the network. The first step (line 4) requires a spatial search in the 2D R-tree (with

the coordinates of Nodes *i* and *j* as arguments) in order to find link *k*, enclosed by the MBB of the 2D R-tree leaf node *N*. Next, we follow the pointer to the corresponding 1D R-tree, in which we insert a new entry (t_{in}, t_{out}, *object-id*, *link-id*). Depending on the insertion policy, the newly inserted entry is placed in the right-most 1D R-tree leaf node *M* (or in the node determined by the `R_tree_insert` algorithm). Eventual modifications in the structure due to this insertion (the MBB of that leaf node may be updated, a new leaf node may be created), are propagated upwards. Such a modification also causes updates in the Parent 1D R-tree; an updated MBB in the 1D R-tree causes a deletion and re-insertion of the corresponding entry in the Parent R-tree, while a creation of a new node in the 1D R-tree causes an insertion of a new entry in the Parent R-tree.

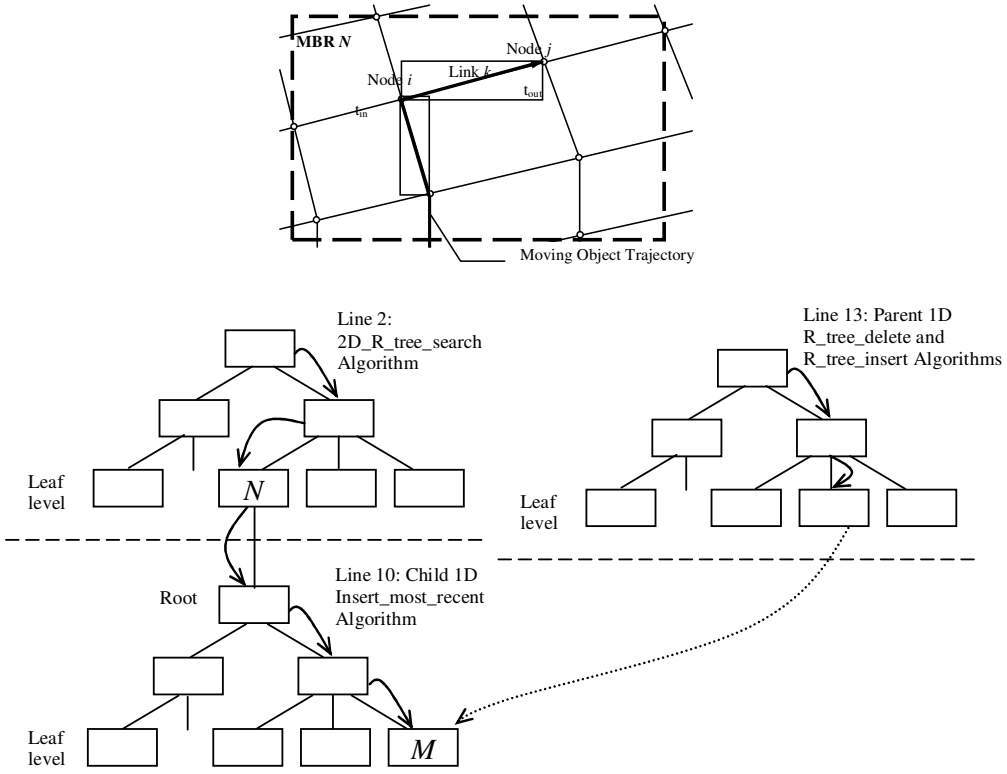


Figure 2.20: Insertion of a new entry in the FNR-tree

2.4.2.2. Querying the FNR-tree

The structure of the FNR-tree offers the flexibility to use two different search algorithms for different types of queries. The comparative advantages will be presented through examples and the performance study that will follow.

`Search-from-2D-R-tree:` The first algorithm, illustrated in Figure 2.21, starts from the 2D R-tree root, locates the 2D R-tree entries which satisfy the spatial constraints of the query and then, following the pointer(s) to the corresponding 1D R-tree(s), checks in those trees whether there are entries satisfying the query temporal constraint as well. Finally, a refinement step ensures that the algorithm returns only the entries satisfying both spatial and temporal criteria.

```
1.   Algorithm FNR_tree_Search_from_2D(x_min, x_max, y_min, y_max, t_min, t_max)
2.      // Search in the 2D R-tree with the 2D interval
3.      // (x_min, x_max, y_min, y_max) retrieving the Links contained in it
4.      Links = 2D_R_tree_search(x_min, x_max, y_min, y_max)
5.      // follow the pointers from leaf nodes ND containing the Links
6.      // to the corresponding 1D R-trees, RT
7.      FOR EACH ND containing any of Links
8.         RT=ND.Child_R_tree
9.         // Search each one of the corresponding 1D R-trees
10.        Candidates=RT.R_tree_search(tmin, tmax)
11.        // Refinement
12.        // If ND2 is completely contained inside (x_min, y_min, x_max,
13.        // y_max) all entries of ND2 are also inside
14.        IF ND2.MBB is inside (xmin, ymin, xmax, ymax)
15.           RETURN all entries in Candidates
16.        ELSE // ND2 is partially inside (x_min, y_min, x_max, y_max)
17.           FOR EACH Entry IN Candidates
18.              IF Links(Entry.Link_id).MBB is inside (x_min, x_max, y_min, y_max)
19.                 RETURN the Entry
20.              ENDIF
21.           NEXT
22.        ENDIF
23.     NEXT
```

Figure 2.21: FNR-tree `Search-from-2D-R-tree` Algorithm

`Search-from-Parent-1D-R-tree:` The second search algorithm of the FNR-tree utilizes the Parent 1D R-tree and is illustrated in Figure 2.22. It starts from the Parent 1D R-tree root and locates the entries satisfying the temporal constraints of the query. Then, following the pointers, it finds the children 1D R-tree leaf nodes containing the entries satisfying the temporal constraints of the query and the corresponding 2D R-tree leaf nodes. Finally, a refinement step guarantees that the algorithm returns only the entries satisfying both temporal and spatial criteria.

```
1.   Algorithm FNR_tree_Search_from_Parent_1D(x_min, x_max, y_min, y_max, t_min, t_max)
2.      // Search in the Parent 1D R-tree with the 1D interval
3.      // (t_min, t_max) retrieving the entries overlapping it
4.      PEntries = Parent_1D_R_tree_search(x_min, x_max, y_min, y_max)
5.      // follow the pointer to the children 1D R-tree Leaf Nodes ND1
6.      FOR EACH PEntry IN PEntries
7.         ND1=Pentry.Child_1D_R_Tree_Leaf
8.         // follow the pointer to the parent 2D R-tree Leaf Node
9.         // ND2 to get spatial extent
10.        ND2=ND1.Parent_2D_R_Tree_Leaf
11.        // Refinement
12.        IF ND2.MBB is outside (x_min, x_max, y_min, y_max)
13.           Reject ND2
14.        ELSEIF ND2.MBB is inside (x_min, x_max, y_min, y_max)
15.           RETURN all entries of ND1
16.        ELSE // ND2 is partially inside (x_min, y_min, x_max, y_max)
17.           FOR EACH Entry IN ND1
18.            IF ND2.Links(Entry.Link_id).MBB is inside(x_min, x_max, y_min, y_max)
19.                 RETURN the Entry
20.            ENDIF
21.           NEXT
22.        ENDIF
23.     NEXT
```

Figure 2.22: FNR-tree `Search-from-Parent-1D-R-tree` Algorithm

Suppose we would search the FNR-tree with a spatio-temporal query window (x_1, y_1, x_2, y_2, t_1, t_2) using the first search algorithm, `Search-from-2D-R-tree` (Figure 2.23). The first step requires a

spatial search (for (x_1, y_1, x_2, y_2)) in the 2D R-tree in order to locate the line segments and the corresponding 2D R-tree leaves (in our example, leaf node *N*) which are covered by the spatial query window. Next, a search (for (t_1, t_2)) is executed in each one of the 1D R-trees that correspond to the leaf nodes of the first step. In our example, the search directs to the 1D R-tree leaf nodes T_1 and T_2 that contain (among others) links *k*, *l* and *o*. At the final step, we retrieve from the main memory the coordinates of each link selected in the second step and – for the reason that the 2D R-tree leaf node *N* overlaps the spatial query window – we check and reject those, which are outside the spatial query window (in our example, link *o*).

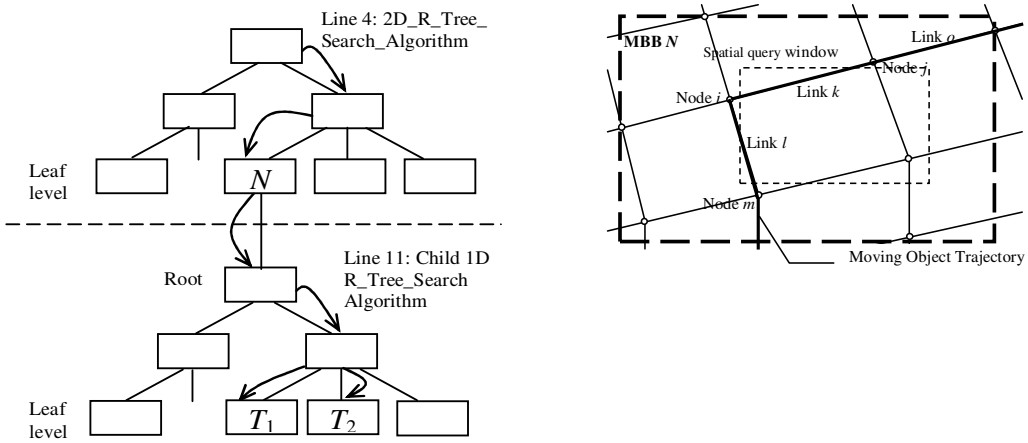


Figure 2.23: Searching the FNR-tree using Search-from-2D-R-tree Algorithm

In order to demonstrate the second search algorithm (`Search-from-Parent-1D-R-tree`), we again assume a spatio-temporal query window (x_1, y_1, x_2, y_2, t_1, t_2) (Figure 2.24). The first step of the algorithm requires a search in the parent 1D R-tree with (t_1, t_2) as argument in order to locate its leaf entries which overlap with this interval. Then, following the pointers to the children 1D R-tree leaf nodes, we locate the nodes containing the 1D R-tree entries that satisfy the temporal constraints of the query (leaf nodes T_1, T_2). These nodes belong to different children 1D R-trees, corresponding to different 2D R-tree leaf nodes, which are traced by following the pointers to them (leaf nodes N_1, N_2). Finally, we check whether the entries contained in the nodes N_1 and N_2 satisfy the spatial query constraint (x_1, y_1, x_2, y_2).

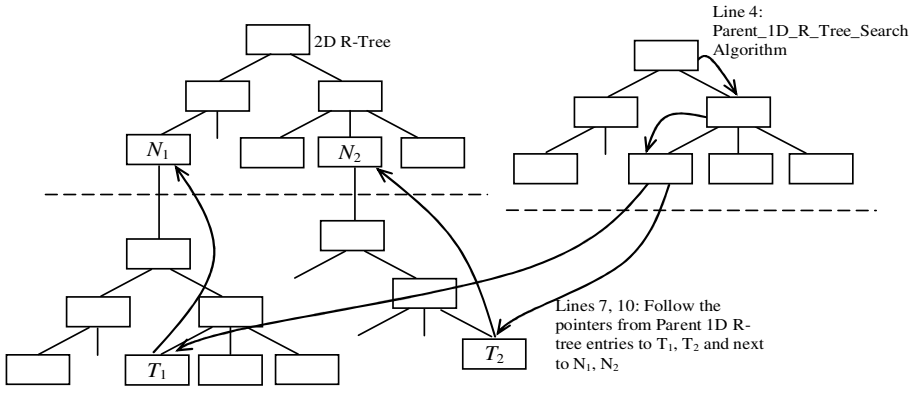


Figure 2.24: Searching the FNR-tree using Search-from-Parent-1D-R-tree Algorithm

A criticism to the second search algorithm is that it only cares about the temporal location of the data and applies a spatial filtering only at the last step. On the other hand, we expect this algorithm to be efficient in cases where only the temporal query constraints matter. We illustrate this behavior in the experimental section that follows.

At this point, it is worth to note that *the FNR-tree would also be functional without the presence of the parent 1D R-tree*. In this case, the single modification of the FRN-tree insertion algorithm would be the absence of lines 11-18 (cf. Figure 2.18). Moreover, the first search algorithm (cf. Figure 2.21) would be executed 'as-it-is'. As such, the construction and the operation of the FNR-tree would be possible without the presence of the Parent 1D R-tree.

```
 1.  Algorithm FNR_tree_Parent_1D_R_Tree_Construction
 2.     Create a new 1D R-tree
 3.     // Access the 2D R-tree. Use the 2D R-tree structure and locate
 4.     // every leaf node named ND2
 5.     FOR EACH Leaf Node ND2 IN 2D R-tree
 6.        // Follow the pointer to the child 1D R-tree RT
 7.        ND1=ND2.Child_1D_R_Tree_Leaf
 8.        // Access all the child 1D R-tree and insert the leaf
 9.        // nodes in the Parent 1D R-tree
10.        FOR EACH Leaf Node ND1 IN RT
11.           // Execute R-tree-insert algorithm in the Parent // 1D R-tree
12.           // and insert ND1 as a new entry
13.           Parent_1D_R_tree_insert(ND.MBB, ND.ptr)
14.        NEXT
15.     NEXT
```

Figure 2.25: FNR-tree `Parent-1D-R-Tree-Construction` Algorithm

However, the function of the second search algorithm (cf. Figure 2.22) requires the existence of the parent 1D R-tree. This structure can be constructed at any time instance of the FNR-tree lifetime using the construction algorithm illustrated in Figure 2.25; this algorithm accesses the complete FNR-tree structure and simply inserts the temporal extents of all children 1D R-trees leaf nodes as entries in the Parent 1D R-tree.

2.5. Experimental Study: Unrestricted Movement

In order to evaluate the performance of the TB*-tree, we implemented its structure and algorithms proposed in this chapter and made a comparison of the proposed index with the original TB-tree [PJT00], as well as the traditional 3D R-tree [TVS96].

2.5.1. Experimental Setup

We have chosen the page size for all trees to be 4 KB resulting in a fanout - maximum capacity (*M*) for the TB*-tree of 338 and 145, for the leaf and non-leaf nodes, respectively. Further from the LP buffer introduced, we used a (variable size) LRU buffer fitting the 10% of the index size, with a maximum capacity of 1000 pages. The experiments were performed in a PC running Microsoft Windows XP with AMD Athlon 64 3GHz processor, 1 GB RAM and several GB of disk space.

In order to achieve scalability in the cardinality of the datasets and study the behavior of the index structures under several settings we have employed the synthetic GSTD datasets introduced in section 1.5.2 (the real datasets of section 0 are not suitable for our study, since have constant, rather limitted size). Furthermore, we used two different strategies to insert the datasets into the three

structures. The first strategy requires the dataset to be ordered by time. This is the usual case in online spatio-temporal applications, where, due to time monotonicity, we expect the trajectory data to be collected and inserted in the index in ascending order of time (hence, 'time' organization in the experiments to follow). The second strategy does not make this assumption, provided that the trajectory data of a single moving object are inserted in chronological order. This is the case where moving objects record their position and do not maintain online communication with the central server that maintains the index; on the contrary, objects send their location(s) as soon as it is possible (e.g. when they are in range of the device used for the transmission), or at scheduled timestamps. This is also the case where the index is built after the compression of another existing index or any other file containing trajectory information. Thus, in order to simulate both previous situations, trajectory data are inserted into the index in ascending moving object id / time order (hence, 'id/time' organization in the experiments to follow).

Table 2.2: Results on tree size (GSTD synthetic datasets)

Dataset	*index size in pages (of 4 KB each)*		
	3D R-tree	*TB-tree*	*TB*-tree*
GSTD 100	6253	3054	1522
GSTD 250	15471	7649	3808
GSTD 500	30937	15301	7597
GSTD 1000	61864	30588	15156
GSTD 2000	122703	61170	30557

2.5.2. Results on Tree Size and Insertion Cost

The sizes of the built index structures are illustrated in Table 2.2. It is clear that the size of the TB*-tree is almost half of the size of the TB-tree, and almost the 15% of the size of the standard 3D R-tree. Moreover, the space utilization for both the TB- and the TB*-tree is as high as expected: about 99% and 96%, respectively, whereas the respective value of the 3D R-tree is 56%, which is a typical value for R-trees. It is therefore proven that the TB*-tree is a very compact index structure, outperforming both of its competitors.

Table 2.3: Index size, space utilization and node accesses per insertion on the *GSTD2000* dataset

	3D R-tree	*TB-tree*	*TB*-tree*
Index size (KB per object)	99.6	30.6	15.2
Space utilization	56%	99%	96%
Node Accesses per Insertion (average)	2.3	4.0 (1.2)	1.4

The results on node accesses per insertion for all datasets are illustrated in Table 2.3. Each insertion of a new trajectory line segment in the TB*-tree requires an average of 1.4 node accesses. The reason for this first-rate performance are the usage of the in-memory front-line structure, which points directly to the node wherein the new line entry must be inserted, and the presence of the LP buffer. On the contrary, the TB-tree and the 3D R-tree require larger number of nodes per insertion, 4.0 and 2.3, respectively; this is due to the TB-tree *FindNode* algorithm, which follows a multi-way path (and not a

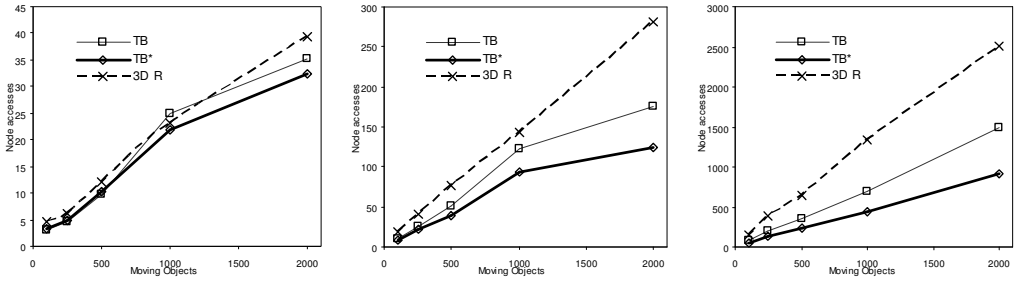
one-way as the *ChooseLeaf* algorithm of the R-tree does) to find the appropriate node where to place the new entry.

Here, we have to point out, that the original TB-tree insertion algorithm can be modified using the front-line structure introduced in the TB*-tree and simply replace the *FindNode* algorithm step by following the pointer to the moving object's 'current' leaf node. The original TB-tree can also employ the LP buffer which, on the contrary, cannot be used in the case of other non-trajectory oriented indexes (such as the 3D R-tree), because such a strategy would require the LP buffer to hold all the leaf nodes. Thus, in order to demonstrate the influence of those improvements (front-line and LP buffer) in the behavior of the simple TB-tree we employed them in the conducted experiments, resulting in the second number (1.2) in Table 2.3. As it can be seen, these techniques drastically improve the insertion performance, making therefore the simple TB-tree also able to support high insertion rates.

2.5.3. Results on Search Cost

Range, timeslice, and combined queries were used in order to evaluate the performance of the TB*-tree. In particular, we used the following set of five queries ($Q_1 - Q_5$):

- Q_1–Q_3: three sets of 500 cubic query windows with a range of 0.01%, 0.1% and 1% of the total space, respectively, over the synthetic data increasing the number of moving objects (GSTD100 – GSTD2000 datasets).
- Q_4: one set of 500 timeslice query windows with the 100% of the extent in the spatial dimensions and zero temporal extent, over the synthetic data increasing the number of moving objects (GSTD100 – GSTD2000 datasets).
- Q_5: one set of 500 combined queries with inner window 0.01% and outer 1% of the total space, over the synthetic data increasing the number of moving objects (GSTD100 – GSTD2000 datasets).



(a) Q_1 (b) Q_2 (c) Q_3

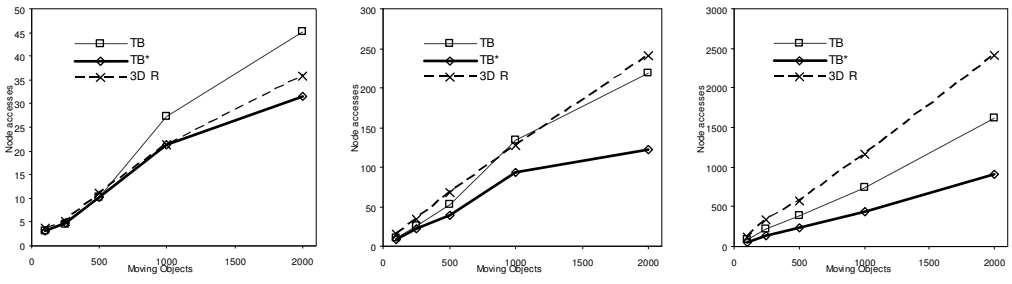
Figure 2.26: Queries $Q_1 - Q_3$ with the synthetic data inserted organized by time

2.5.3.1. Results on Range Queries

Figure 2.26 illustrates the average number of node accesses per query for various ranges and datasets. In particular, Figure 2.26 shows the average number of node accesses for range queries with a cubic window of 0.01%, 0.1% and 1% of the total space over the synthetic data inserted in the structures organized by chronological order irrespective of id ('*time*'), while Figure 2.27 shows the average number of node accesses for the same range queries over the same data organized by id and then chronological order ('*id/time*'). It is clear that the TB*-tree has superior range query performance over both its competitors regarding the query with sizes of 0.1% (Q_2) and 1% (Q_3) of the total space for both

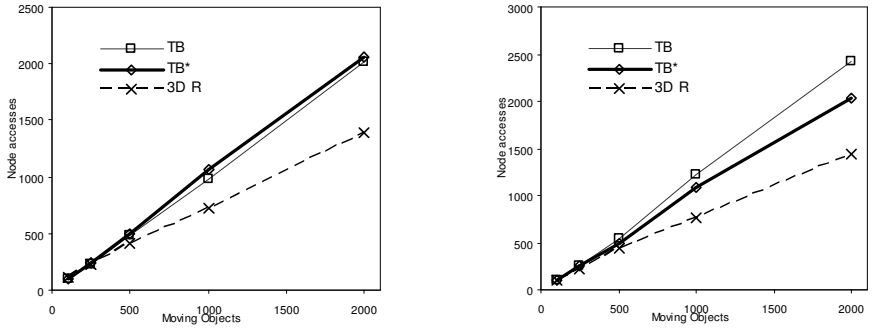
different insertion organizations. Regarding the queries with smaller size (0.01% of the total space, Q_1) and the trajectory data organized by chronological order, the TB*-tree performance is only marginally better than the original TB-tree, a difference which is more clear as the dataset cardinality grows.

Another conclusion that arises from the comparison between Figure 2.26 and Figure 2.27 is that, while the TB*-tree and the 3D R-tree show approximately the same behavior following the two different insertion strategies (the slope of the TB*-tree and the 3D R-tree lines is approximately the same in Figure 2.26 and Figure 2.27), the behavior of the original TB-tree is significantly affected, resulting in the second case in a tree with drastically decreased performance. This behavior of the TB-tree is expectable since the basic assumption which the efficiency of the tree in range queries is based on, i.e. the insertion of new entries in chronological order, does not hold any more.



(a) Q_1 (b) Q_2 (c) Q_3

Figure 2.27: Queries Q_1 – Q_3 with the synthetic data inserted organized by id/time



(a) (b)

Figure 2.28: Queries Q_4 with the synthetic data organized by (a) time, (b) id/time

2.5.3.2. Results on Timeslice Queries

Figure 2.28(a) presents the average number of node accesses for timeslice queries with 100% in each spatial dimension (e.g. find all objects in a certain timestamp) when inserting the trajectory data organized in purely chronological order. As shown, in the first case, the original TB-tree only marginally outperforms the TB*-tree, which is an expected behavior, since the original TB-tree takes full advantage of the monotonicity of time and stores object's trajectories considering only the order in which they are inserted in the index. However, this turns out to be a drawback when data are not inserted in purely chronological order (Figure 2.28(b)); in this case, the TB*-tree shows better performance, a behavior that is similar to the one showed by this structure when data were inserted in strictly chronological order.

As a conclusion, the TB*-tree performance in timeslice queries is reduced compared with its performance in range queries, although it still outperforms the TB-tree when the trajectory data inserted in the structures are organized by 'id/time'.

2.5.3.3. Results on Combined Queries

Figure 2.29 shows the average number of node accesses required by the TB- the TB*- and the 3D R-tree in order to answer combined queries. In both figures the TB*-tree outperforms both TB- and 3D R-tree regardless of the dataset cardinality. Moreover, in accordance with all the previous experiments, the difference in the performance between the TB- and the TB*-tree increases in favor of the TB*-tree with the dataset cardinality. Furthermore, in the second case (Figure 2.29(b)) where that data inserted in the trees with 'id/time' organization, the TB*-tree shows even better performance over its predecessor.

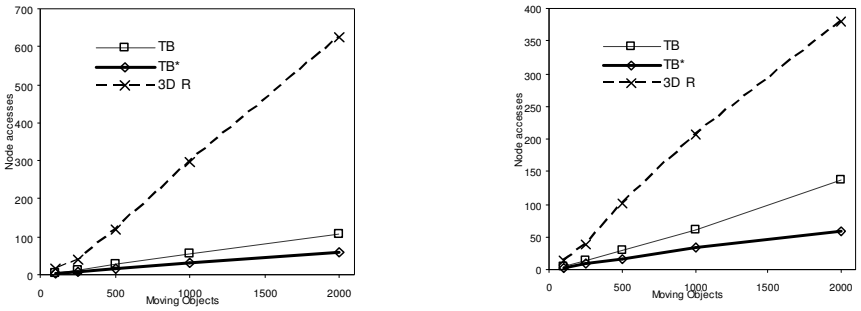


Figure 2.29: Combined queries, (Q_5) with the synthetic data organized by (a) time, (b) id/time

2.5.4. Summary of the Experiments

The experiments conducted in order to evaluate the performance of the proposed TB*-tree against the original TB-tree and the 3D R-tree showed that the proposed index supports range queries efficiently. More specifically, when dealing with relative large query extents (0.1% and 1% of the entire space) the TB*-tree always outperforms the TB-tree and the 3D R-tree, while in smaller query extents (0.01% of the total space) it is marginally better than its competitors, a benefit which becomes clearer as the dataset cardinality increases. On the contrary, in timeslice queries the TB*-tree appears to have reduced performance, requiring a few more page accesses in order to process a timeslice query.; in this case the "winner" is the 3D R-tree. Finally, regarding the combined queries the new index shows superior performance over the original TB-tree in all settings. Moreover, the superiority of the proposed index is established in the general case where the indexes are built inserting the trajectory data not in purely chronological order ('id/time' organization), a case which is expected in real-world applications and when the index is built after a dataset compression. Under such conditions, the TB*-tree is always much more efficient than the TB-tree. On the subject of the size of the TB*-tree, its space utilization, like the original TB-tree, reaches up to 96%, and the average size per moving object is the half than that of the TB-tree. Moreover, the TB*-tree supports high insertion rates since its insertion algorithm is proved to be very fast, is more compact than its competitors, behaves well in non-chronological trajectory insertions that appear in real-world environments, and supports trajectory deletions and trajectory compression efficiently.

2.6. Experimental Study: Network-Constrained Movement

In order to determine the conditions under which the FNR-tree is efficient, we compared it with other spatio-temporal access methods, namely the 3D R-, the TB- and the TB*-tree.

2.6.1. Experimental Setup

In order to evaluate the performance of the FNR-tree, we implemented its structure in main memory, simulating its behaviour. We have chosen the page size for all trees to be 4096 bytes, acquiring the following fanout settings for the FNR-tree: (a) for the 2D R-tree a fanout of 193 and 202, for leaf and non-leaf nodes, respectively; (b) for the children 1D R-trees a fanout of 290 and 339, for leaf and non-leaf nodes, respectively; (c) for the Parent 1D R-tree, a fanout of 339 for both leaf and non-leaf nodes. Compared with the fanout of its competitors, the FNR-tree is as compact as the TB*-tree (outperforming the other two index structures). We also used a (variable size) LRU buffer fitting the 10% of the index size, with a maximum capacity of 1000 pages. The experiments were performed in a PC running Microsoft Windows XP with AMD Athlon 64 3GHz processor and 1 GB RAM. In order to achieve scalability in the cardinality of the datasets and study the behavior of the index structures under several settings we have employed the synthetic NG datasets introduced in section 1.5.3 (again, the real datasets of section 0 are not suitable for our study since they have constant, rather limited size).

Table 2.4: Results on tree size (NG synthetic datasets)

Dataset	*index size in pages (of 4 KB each)*			
	FNR-tree	*3D R-tree*	*TB-tree*	*TB*-tree*
NG 200	769	1204	770	424
NG 400	1139	2397	1533	848
NG 800	1850	4603	3040	1669
NG 1200	2575	7001	4499	2495
NG 1600	3281	9234	5972	3310
NG 2000	4030	11636	7455	4158

2.6.2. Results on Tree Size and Insertion Cost

The size of the built index structures is illustrated in Table 2.4. As listed there, the FNR-tree is much smaller than the 3D R-tree and the TB-tree. The ratio between the index size of FNR and 3D R-tree varies between 0.30 and 0.45 for large number of moving objects. For example, the size of the FNR-tree for 2000 moving objects is about 16 MB, while the size of the respective 3D R-tree is 48 MB. It is only the TB*-tree that appears to have a comparable size with the FNR-tree.

Table 2.5: Index size, space utilization and node accesses per insertion on the *NG 2000* dataset

	FNR-tree	*3D R-tree*	*TB-tree*	*TB*-tree*
Index size (KB per object)	8.1	24.0	14.9	8.3
Space utilization	92%	64%	86%	75%
Node Accesses per Insertion (average)	2.0	2.1	4.0	1.04

Similar results are exposed regarding the space utilization. As shown in Table 2.5, the space utilization of the 3D R-tree is about the typical 65%, remaining steady regardless of the number of

moving objects. Likewise, the space utilization for the TB- and TB*-tree is respectively about 86% and 75%, respectively, percentages that are not affected by the number of moving objects. Nevertheless, we have to point that the space utilization of TB- and TB*-tree is mainly affected by the number of each trajectory's time-stamped positions, which is rather low, around 500 vertices per trajectory; bearing in mind that each leaf node contains approximately 300 entries, it becomes clear that each trajectory should occupy 2 leaf nodes, the first being completely filled, and the other being half-full. On the contrary, the space utilization of the FNR-tree grows with the dataset cardinality. Thus, the space utilization of the FNR-tree with 200 moving objects is 65 %, while, for 1200 and over it reaches 92%.

Regarding the results on node accesses per insertion, each insertion of a 3D line segment in the FNR-tree requires an average cost of 2.0 node accesses, while an insertion in the 3D R-tree requires an average of 2.1 node accesses. It is clear, that the pre-searching in the 2D R-tree in order to find the 1D R-tree to place the newly inserted line segment does not add significant additional overhead, mainly due to the effect of the employed LRU buffer. Regarding the TB- and TB*-tree, they still demonstrate the same behavior observed in the previous experiments in unrestricted space, being able to support high insertion rates.

2.6.3. Results on Search Cost

Range and timeslice queries were used in order to evaluate the performance of the FNR-tree. Both were executed against the FNR-, the 3D R-, the TB- and the TB*-tree indexing the NG datasets. In particular, we used sets of 500 queries with the following query windows:

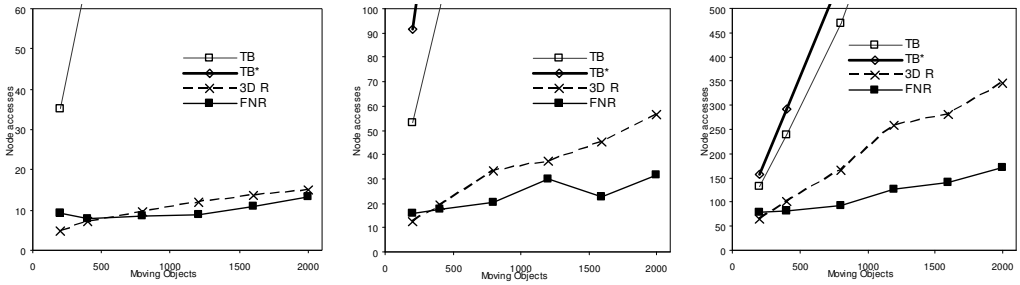
- Q_1–Q_3: three sets of 500 cubic query windows with a range of 0.01%, 0.1% and 1% of the total space, respectively, increasing the number of moving objects.
- Q_4–Q_6: three timeslice query windows with a range of 1%, 10% and 100% extent in each spatial dimension and zero temporal extent.

We used the `Search-from-2D-R-tree` (cf. Figure 2.21) FNR-tree algorithm against all the above queries and additionally, we tested `Search-from-Parent-1D-R-tree` (cf. Figure 2.22) against Q_6 i.e., those with 100% extent in each spatial dimension.

2.6.3.1. Results on Range Queries

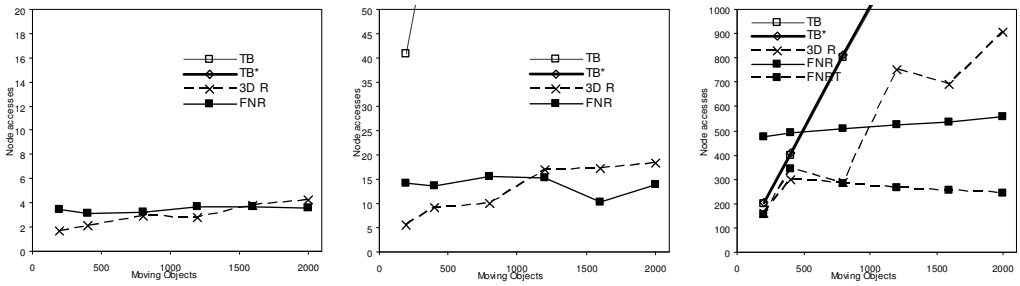
Figure 2.30 illustrates the average number of node accesses per query for various ranges and datasets. In particular, Figure 2.30 (a), (b) and (c), show the average number of node accesses for range queries with a window of 0.01%, 0.1% and 1% of the total space. As it is clearly illustrated, the FNR-tree has superior range query performance over all of its competitors for dataset cardinality above a threshold, in all query sizes. The break-even point after which the FNR-tree outperforms the rest depends on the query size. Specifically the break-even point is at about 400 moving objects for small query sizes (0.01%), while greater query sizes result in smaller break-even point, abound 200 moving objects. Regarding the rest structures, the 3D R-tree performs always much better than the TB- and the TB*-tree two when using the data moving on the networks; note that the diagrams may not contain all the curves of the four indexes, due to the fact that they are not contained inside the given display scale (i.e., they have values greater than the ones displayed in the diagram's *y*- axis). We have to point however, that the observation regarding the performance of 3D R-, TB- and TB*-tree can not be generalized.

Specifically, the poor performance that TB- and TB*-tree demonstrate in the experiments, is mainly due to the small number of time-stamped positions in each trajectory, which forces each trajectory to be divided between two tree nodes only. It is expected (as also the previous experiments showed) that as the temporal extent of trajectories grows and more time-stamped positions are added into each individual trajectory, the performance of TB- and TB*-tree will tend to have more "normal" values as the ones demonstrated in the experiments on unrestricted space. Nevertheless, the tool provided by [Bri02] and used to produce the synthetic trajectories on netowrks, can not generate longer trajectories; therefore we can not employ larger (elongated in the temporal dimension) datasets.



(a) Q_1 (b) Q_2 (c) Q_3

Figure 2.30: Queries Q_1 **– Q_3**



(a) (b) (c)

Figure 2.31: Queries Q_4 – Q_6

2.6.3.2. Results on Timeslice Queries

The FNR-tree performance in timeslice queries is reduced compared with its performance in range queries, although it still outperforms its competitors in most cases. Figure 2.30 shows the average number of node accesses for timeslice queries with several datasets and spatial extents. In particular, the FNR-tree shows better performance over its competitors for a number of moving objects and above; the break-even point depends on the query size and is about 1600 for 1%, 1200 for 10% and 1000 for 100% query size in each spatial dimension. The other two competitors (TB- and TB*-tree) still show the same disadvantages that demonstrated in the previous experiment regarding general range queries.

Furthermore, the line marked as FNRT in Figure 2.31(c) represents the performance of the FNR-tree using the second search algorithm, which in this special type of queries outperforms the first. Specifically, using the second algorithm shifts the break-even point from which the FNR-tree is better than the 3D R-tree from 1000 to 800 moving objects. For a direct comparison between the two FNR-tree search algorithms, we also present Figure 2.32. There, it is clearly illustrated that the average

number of node accesses of the second search algorithm – for 2000 objects – remains stable regardless of the query spatial extent, while the cost of the first search algorithm grows sublinearly with the spatial extent.

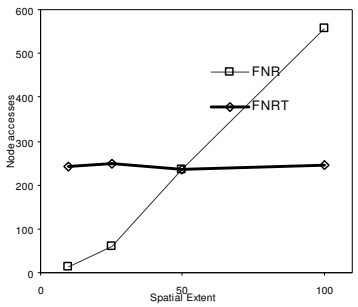


Figure 2.32: Timeslice queries with incremental spatial extent in the FNR-tree with 2000 moving objects

2.6.4. Summary of the Experiments

The experiments that we conducted in order to evaluate the performance of the FNR-tree showed that it supports range and timeslice queries much more efficiently than its three competitors. Especially for the latter case of timeslice queries, the FNR-tree only conditionaly outperforms the 3D R-tree; this happens after the cardinality of the dataset exceeds 1000 trajectories. Besides, we establish the conditions under which the second search algorithm is more efficient than the first; it is shown that timeslice queries with spatial extent greater than the 50% of the total spatial space are more efficiently supported by the `Search-from-Parent-1D-R-tree` algorithm. On the subject of the size of the FNR-tree, its space utilization may reach 92%; the average size per moving object is comparable with the one of the TB*-tree, and it may become 3 times smaller than the respective size of the 3D R-tree. Finaly, the average node accesses per insertion in the FNR-tree is better then the one of the original TB-tree and in the same order of magnitude with the simple 3D R-tree.

2.7. Conclusions

The domain of indexing spatio-temporal data has been very active during the last decade. While the vast majority of real-world spatio-temporal applications concerns objects producing trajectory data, a great part of the developed indexes overlook the challenges posed by the nature of the these data, and they just index collections of line segments in the spatio-temporal space, handling only traditional coordinate-based queries. Moreover, since a great number of such applications assumes that the space on which objects move is network-constrained (fleet management systems, and so on), spatio-temporal indexes should exploit this property in order to become more efficient as suggested in [KGT99].

The first in the literature index proposed to efficiently support trajectory-based queries, the TB-tree [PJT00], was fundamentally different from other spatio-temporal access methods since it proposed grouping of line segments in the same leaf nodes, based not on their spatial or temporal proximity but on the trajectory in which they belong. However, the TB-tree turns out to have some drawbacks with the major one being its dependence on the order in which trajectory data are inserted into the index. Moreover, while motion restrictions have been a subject of research [KGT99], [PTKZ02], [Pfo02],

[PJ01], until recently, there was no proposal for a spatio-temporal access method suitable for objects moving on fixed networks.

In this chapter, state-of-the-art is advanced towards two independent directions:

- In the first case, where objects move freely in the space, acknowledging the basic advantages of the TB-tree, we propose an extension of it, called TB*-tree. The proposed index overcomes the main disadvantages of its predecessor while at the same time preserving all of its 'desired' properties. In particular, it supports trajectory insertions, deletions and compression, while querying is performed by employing the same algorithms provided in [PJT00].
- In the second case of network-constrained objects, a novel indexing technique, called Fixed Network R-tree (FNR-tree) is proposed. The general idea that describes the FNR-tree is that of a forest of several 1D R-trees [Gut84] on top of a single 2D R-tree. The 2D R-tree is used to index the spatial data of the network (i.e., roads consisting of line segments), while the 1D R-trees are used to index the time interval of each object's movement on a given segment of the network. Additionally, the leaf nodes of all the 1D R-trees are indexed by another 1D R-tree used to answer queries with no spatial extent.

The experiments conducted in order to evaluate the performance of the proposed TB*-tree against the original TB-tree and the 3D R-tree showed that the proposed index supports range and combined queries efficiently. Its superiority is established in the general case where the indexes are built inserting the trajectory data not in purely chronological order ('id/time' organization), a case which is expected in real-world applications and when the index is built after a dataset compression. The TB*-tree, is more compact than the original TB-tree, it supports high insertion rates, behaves well in non-chronological trajectory insertions that appear in real-world environments, and supports trajectory deletions and trajectory compression efficiently.

We also experimentally compared the FNR-tree with the TB*-tree and the traditional 3D R-tree [TVS96] and TB-tree [PJT00]. Under various datasets and range queries, the FNR-tree was shown to outperform all its competitors in the vast majority of settings. The FNR-tree has high space utilization, smaller size per moving object and supports range queries much more efficiently. In general, we argue that the FNR-tree is an access method ideal for fleet management applications; however, it may only be used under the network-constrained scenario.

3. Advanced Trajectory Query Processing: Nearest Neighbor Search

In this chapter we provide a set of algorithms for performing nearest neighbor search on moving object trajectories, by employing R-tree-like structures storing historical trajectory information. The chapter is organized as follows. Section 3.1 provides an introduction to the core subject of this chapter. Related work is discussed in Section 3.2, while Section 3.3 introduces, at an abstract level, the set of *k*-NN algorithms over moving object trajectories, as well as the metrics that support our search ordering and pruning strategies. Sections 3.4 and 3.5 constitute the core of the chapter describing in detail the query processing algorithms to perform NN search over historical trajectory information together with their continuous counterparts; the algorithms presented are based on the depth-first and best-first paradigm, on R-tree-like structures. Section 3.6 presents the results of our experimental study and Section 3.6.5 provides our conclusions.

3.1. Introduction

Research in the field of advanced query processing in historical spatio-temporal trajectory databases is guided by related work performed in the domain of (stationary) spatial databases. For example, queries of the form "*find all objects located within a given area during a certain time interval*" generalize the respective spatial *range* query of the form "*find all objects within a given area*". Such queries are considered as basic ones, since the proposed indexes by definition should support them; for that reason, they are not further examined hereafter. On the other hand, other spatial operators, such as *nearest neighbor* [RKV95] and *distance join* [HS99], are considered as advanced, since they require more sophisticated query processing techniques in order to be efficiently processed. Moreover, such advanced techniques may or may not consider the presence of a spatio-temporal index. Then again, in MODs we typically have to deal with huge volumes of historical data which correspond to a large number of mobile and stationary objects. As a consequence, querying functionality embedded in an extensible DBMS that supports moving objects has to present robust behavior in the above mentioned issues. Hence, in this thesis, we restrict our discussion on advanced query processing techniques under the perspective of the former case, that is, assuming the operation of some kind of spatio-temporal index.

An important class of queries that is definitely useful for MOD processing is the so-called *k* nearest neighbor (*k*-NN) queries, where one is interested in finding the *k* closest trajectories to a

predefined query object Q. To our knowledge, in the literature such queries primarily deal with either static ([RKV95], [CF98], [HS99]) or continuously moving query points ([SR01], [TPS02]) over stationary datasets, or queries about the future or current positions of a set of continuously moving points ([BJKS02], [TP02], [ISS03], [YPK05], [XMA05], [MHP05]). Apparently, these types of queries do not cover NN search on historical trajectories.

Thus, one of the challenges accepted in this thesis is to describe diverse mechanisms to perform k-NN search on MODs exploiting spatio-temporal indexes storing historical information. To illustrate the problem, consider an application tracking the positions of rare species of wild animals. Such an application is composed of a MOD storing the location dependent data, together with a spatio-temporal index for searching and answering k-NN queries in an efficient manner. Experts in the field would be advantaged if they could pose a query like "*find the nearest trajectories of animals to some stationary point (lab, source of food or other non-emigrational species) from which this species passed during March*". Now imagine that the expert's wish is to pose the same query with the difference that the query object Q is not a stationary point but a moving animal moving from location P_1 to P_2 during a period of time. This query gives us rise to deduce a more generic query where the expert may wish to set another trajectory of the same or relative class of species as the query object Q. It is self-evident that by these types of queries an expert may figure out motion habits and patterns of wild species or deviations from natural emigration, which could be interrelated with environmental and/or ecological changes or destructions. Having in mind that MOD users are usually interested in continuous types of queries, the two previously discussed queries are extended to their continuous counterparts. In their continuous variation, each query returns a time-varying number (denoting the nearest distance, which depends on time) along with a collection of trajectory ids and the appropriate time intervals for which each moving object is valid $\{O_1[t_1, t_2), O_2[t_2, t_3), \ldots\}$.

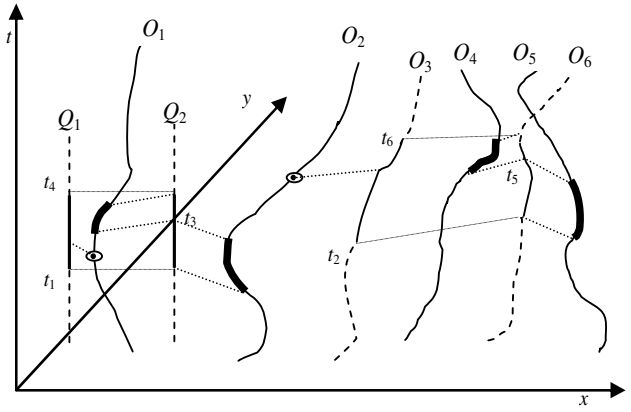


Figure 3.1: NN queries over moving objects trajectories

Posing the problem in a more human-centric context, consider an application analyzing the dynamics of urban and regional systems. The intention here is to assist the development of spatio-temporal decision support systems (STDSS) aimed at the planning profession. Such a case requires similar methodologies for comprehending, in space and time, the interrelations of the life courses of individuals. The life courses of most individuals are built around two interlocking successions of events: a residential trajectory and an occupational career. These patterns of events became more

complex during last decades, creating new challenges for urban and regional planners. We believe that an expert may take advantage of the features provided by our nearest neighbor query processing algorithms and utilize them for analyzing human life courses.

To make the previous examples more intelligible, consider Figure 3.1 illustrating the trajectories of six moving animals $\{O_1, O_2, O_3, O_4, O_5, O_6\}$ along with two stationary points (Q_1 and Q_2) representing two sources of food. Now, consider the following queries also demonstrated in Figure 3.1 (Queries 2 and 4 are the continuous counterparts of Queries 1 and 3, respectively):

- Query 1. "*Find which animal was nearest to the stationary food source Q_1 during the time period $[t_1,t_4]$*", resulting to animal O_1.
- Query 2. *"Find which animal was nearest to the stationary food source Q_2 at any time instance of the time period $[t_1,t_4]$"*, resulting to a list of objects: O_2 for the interval $[t_1,t_3)$; O_1 for the interval $[t_3,t_4]$.
- Query 3. *"Find which animal was nearest to animal O_3 during the time period $[t_2,t_6]$"*, resulting to O_2.
- Query 4. *"Find which animal was nearest to animal O_6 at any time instance of the time period $[t_2,t_6]$"*, resulting to a list of objects: O_5 for the interval $[t_2,t_5)$; O_4 for the interval $[t_5,t_6]$.

Unlike traditional databases, MODs have the characteristic that several spatio-temporal queries are by nature continuous. In contrast to snapshot queries, which are invoked only once, continuous queries require continuous evaluation as the query result becomes invalid after a short period of time. Putting the previous discussion under the perspective of historical trajectories, although queries 2 and 4 are continuous in nature (*at any time instance*) they cannot be characterized as pure continuous queries; with respect to the database engine, a continuous query is one that is submitted to the database only once and remains active, continuously updating the query result with the evolution of time, until its completion is declared by either a user's message or a predetermined query lifetime [BW01], [HXL05], [MXA04]. In this sense, queries 2 and 4 are snapshot queries. However, in order to differentiate them from queries 1 and 3 and also from pure continuous queries, hereafter we will call them *Historical Continuous NN queries (HCNN)*.

Summarizing the previous discussion, the main contributions of the current chapter are outlined as follows:

- We propose novel metrics to support our search ordering and pruning strategies. More specifically, the definition of the minimum distance metric *MINDIST* between points and rectangles, initially proposed in [RKV95] and extended in [TPS02], is further extended in order for our algorithms to calculate the minimum distance between trajectories and rectangles efficiently.
- We propose query processing algorithms to perform NN search on spatio-temporal indexes storing historical information of moving objects. We exploit on the most commonly founded spatio-temporal indexes, supporting unconstrained movement, i.e., R-tree-like structures such as the 3D R-tree [TVS96], the TB-tree [PJT00] and the TB*-tree proposed in this thesis. The description of our algorithms for different queries depends on the type of the query object (point or trajectory) as well as on whether the query itself is continuous or not. In particular,

we present efficient depth-first and best-first (incremental) algorithms for historical NN queries as well as depth-first algorithms for their continuous counterparts. All the proposed algorithms are generalized to find the *k* nearest neighbors.

- We conduct a comprehensive set of experiments over large synthetic and real datasets demonstrating that the algorithms are highly scalable and efficient in terms of node accesses, execution time and pruned space.

We have to point out that the proposed algorithms do not require any dedicated index structure and can be directly applied to any member of the R-tree family used to index trajectories, such as the ones presented in the previous chapter.

3.2. Related Work

In the last decade, NN queries have motivated the spatial and spatio-temporal database community with a series of interesting noteworthy research issues. An affluence of methods for the efficient processing of NN queries for static query points already exist, the most influential probably being the branch-and-bound R-tree traversal algorithm proposed by Roussopoulos et al. [RKV95] for finding the nearest neighbor of a single stationary point. The algorithm utilizes two metrics, *MINDIST* and *MINMAXDIST*, in order to implement tree pruning and ordering. Specifically, starting from the root of the tree, the algorithm identifies the entry with the minimum distance from the query point (with the use of the above metrics). The process is recursively repeated until the leaf level is reached, where the first candidate nearest neighbor is found. Returning from this recursion, only the entries with a minimum distance less than the distance of the nearest neighbor already found are visited. The above process was generalized to support *k*-NN queries. Later, Cheung and Fu [CF98] proved that, given the *MINDIST*-based ordering, the pruning obtained by [RKV95] can be preserved without the use of *MINMAXDIST* metric (the calculation of which is computationally expensive).

Hjaltason and Samet [HS99] presented a general incremental NN algorithm, which employs a best-first traversal of the R-tree structure. When deciding what node of the tree to traverse next, the proposed algorithm picks the node with the least distance in the set of all nodes that have yet to be visited. In order to achieve this, the algorithm utilizes a priority queue where the tree nodes are stored in increasing order of their distance from the query object. This best-first algorithm outperforms Roussopoulos et al. algorithm in terms of pruned space. Additionally, once the nearest neighbor has been found, the *k*-NN can be retrieved with virtually no additional work, since the algorithm is incremental. The basic drawback of this best-first algorithm is that its performance depends on the size of the priority queue. In case the priority queue becomes very large, the execution time of the algorithm increases rapidly.

The first algorithm for *k*-NN search over a moving query point was proposed in [SR01]. The algorithm assumes that sites (landmark points) are static and their locations (known in advance) are stored in an R-tree-like structure. A discrete time dimension is assumed, thus a periodical sampling technique is applied on the trace of the moving query point. The location of the query point that lies between two consecutive sampled locations is estimated using linear or polynomial splines. Executing a Point Nearest Neighbor (PNN) query for every sample point of the query trace is highly inefficient,

so the proposed algorithm adopts a progressive approach, based on the observation that when two query points are close, the results of the *k*-NN search at these locations have to be related. Therefore, when computing the result set for a sample location, the algorithm tries to exploit information provided by the result sets of the previous samples. The basic drawback of this approach is that the accuracy of the results depends on the sampling rate. Moreover, there is a significant computational overhead.

A technique that avoids the drawbacks of sampling relies on the concept of time-parameterized (TP) queries [TP02]. TP queries retrieve the current result at the time the query is issued, the validity period of the result and the change (i.e. the set of objects) that causes the expiration of the result. Given the current result and the set of objects that affect its validity, the next result can be incrementally computed. The significance of TP queries is two-fold: i) as stand-alone methods, they are suitable for applications involving dynamic environments, where any result is valid for a certain period of time, and ii) they lie at the core of more complex query mechanisms, such as the Continuous NN (CNN) queries. The main disadvantage of using TP queries for the processing of a CNN query is that several NN queries are required to be performed. Thus, the cost of the method is prohibitive for large datasets.

Using the TPR-tree (Time Parameterized Tree) structure [SJLL00], Benetis et al. [BJKS02] presented efficient solutions for NN and RNN (Reverse Nearest Neighbor) queries for moving objects. (An RNN query returns all the objects that the query object is the nearest neighbor of.) The proposed algorithm was the first to address continuous RNN queries, since previous existing RNN algorithms were developed under the assumption that the query point is stationary. The algorithms for both NN and RNN queries in [BJKS02] refer to future (estimated) locations of the query and data points, which are assumed to be continuously moving on the plane. In the same paper, an algorithm for answering CNN queries is also proposed.

Tao et al. [TPS02] also studied CNN queries and proposed an R-tree based algorithm (for moving query points and static data points) that avoids the pitfalls of previous ones (false misses and high processing cost). The proposed tree pruning heuristics exploit the *MINDIST* metric presented in [RKV95]. At each leaf entry, the algorithm focuses on the accurate calculation of the split points (the points of the query segment that demonstrate a change of neighborhood). A theoretical analysis of the optimal performance for CNN algorithms was presented and cost models for node accesses were proposed. Furthermore, the CNN algorithm was extended for the case of *k* neighbors and trajectory inputs.

Based on the TP queries presented in [TP02], Iwerks et al. [ISS03] described a technique that focuses on the maintenance of CNN queries (for future predicted locations) in the presence of updates on moving points, where the motion of the points is represented as a function of time. A new approach was also presented, which filters the number of objects to be taken into account when maintaining a future CNN query.

Recently, under the same field, Xiong et al. [XMA05], proposed a method for scalable processing of CNN queries in spatio-temporal databases. They propose a general framework for processing large numbers of simultaneous k-CNN queries with static or moving queries over static or (currently) moving datasets without making any assumptions about the object trajectories. Unlike other proposals, their solution in order to support high update rates is not based on the R-tree but on a simple

grid structure maintained on the disk. A similar method was also proposed by Yu et al. [YPK05] for monitoring k-CNN queries over (currently) moving objects without making any assumptions about the object trajectories. The method also uses (main memory) grid indices indexing moving objects and queries and is shown to outperform R-tree-based solutions. Mouratidis et al. [MHP05] also relax the assumption that moving object's trajectories are fully predictable by their motion parameters, and propose a comprehensive technique for the efficient monitoring of continuous NN queries. The proposed method, named conceptual partitioning monitoring method (CPM), uses also a grid structure and achieves low running time by handling moving object's location updates only from objects falling in the vicinity of some query. The experimental results presented in [MHP05] show that the CPM method outperforms the techniques presented in [XMA05] and [YPK05].

Shahabi et al. [SKS03] presented the first algorithm for processing the *k*-NN queries for moving objects in road networks. Their proposed algorithm, which utilizes the network distance between two locations instead of the Euclidean, is based on transforming the road network into a higher dimensional space, in which simpler distance functions can be applied. Using this embedding space, efficient techniques are proposed for finding the shortest path between two points in the road network. The above procedure, which is utilized in the case of static query points, is slightly modified in order to support the case of moving query points.

Acknowledging the advantages of the above fundamental techniques, in this thesis we present the first complete treatment of historical NN queries over moving object trajectories indexed by specialized and general-purpose indexes, handling both stationary and moving query objects.

3.3. Problem Statement and Metrics for Nearest Neighbor Search

We first define the NN queries that are considered in this thesis. Subsequently, we present the heuristics utilized by our algorithms and the metrics required to formulate our ordering and pruning strategy. Finally, an analytical method is provided in order to formulate the function of distance with time between two objects moving synchronously with constant speed and direction, i.e., between two consecutively sampled points, as well as its minimum value; both analysis outcomes are essential for the algorithms provided in the next section. The notation used is summarized in Table 3.1.

Table 3.1: Table of notations

Notation	*Description*
D	a trajectory database
O_i	A moving object identifier
T_i	an indexed trajectory
$T_{i,k}$	the *k*-th line segment of T_i
$x_{i,k}$, $y_{i,k}$, $t_{i,k}$	the coordinates of trajectory T_i a timestamp t_k
Q_p, Q_T, Q_{per}	A query point, a query trajectory and a query period $[t_{start}, t_{end}]$

3.3.1. Problem Statement

Let D be a database of N moving objects with objects ids $\{O_1, O_2, \ldots, O_N\}$ and trajectories $\{T_1, T_2, \ldots, T_N\}$. We have already stated that NN queries search for the closest trajectories to a query object Q. In our case, we distinguish two types of query objects: Q_p, a point (x, y) that remains stationary during the

time period of the query $Q_{per}[t_{start}, t_{end}]$, and Q_T, a moving object with trajectory T. Having in mind the previous discussion, we define the following two types of NN queries:

- *NN_Q_p* (D, Q_p, Q_{per}) query searches database D for the NN over a point Q_p that remains stationary during a time period Q_{per}, and returns the closest to Q_p point p_c from which a moving object O_i passed during the time period Q_{per}, as well as the implied minimum distance.
- *NN_Q_T* (D, Q_T, Q_{per}) query is similar to the previous with the difference being upon the query object Q which in the current case is a moving object with trajectory T.

The extensions of the above queries to their historical continuous counterparts vary in the output of the algorithms. In the continuous case, each query returns a time-varying real number, as the nearest distance depends on time. We introduce the following two types of historical CNN queries:

- *HCNN_Q_p* (D, Q_p, Q_{per}) query over a point Q_p that remains stationary during a time period Q_{per} returns a list of triplets consisting of the time-varying real value R_i along with a moving object O_i (belonging in database D) and the corresponding time period $[t_{i\text{-}start}, t_{i\text{-}end})$ for which the nearest distance between Q_p and O_i stands. These time-varying real values R_i are, at any time instance of their lifetime, smaller or equal to the distance between any moving object O_j in D and the query point Q_p. The time periods $[t_{i\text{-}start}, t_{i\text{-}end})$ are mutually disjoint and their union forms Q_{per}.
- Similarly, *HCNN_Q_T* (D, Q_T, Q_{per}) differs, compared to the previous, upon the query object Q which in the current case is a moving object with trajectory T. The corresponding time-varying real values R_i are, at any time instance of their lifetime, smaller or equal to the distance between any moving object O_j and the query trajectory Q_T. The respective time periods $[t_{i\text{-}start}, t_{i\text{-}end})$ are mutually disjoint and their union forms Q_{per}.

The above four queries are generalized to produce the corresponding k-NN queries. The generalization of the first two queries is straightforward by simply requesting the 1st, 2nd, …, k-th nearest point – with respect to a query point or a query trajectory – from which a moving object O_i passed during the time period Q_{per}, excluding at the same time points belonging to a moving object already marked as the j-th nearest ($1 \leq j < k$). The historical continuous queries are generalized to produce k-HCNN requesting to provide with k lists of $\{R_i, [t_{i\text{-}start}, t_{i\text{-}end}), O_i\}$ triplets. Then, for any time during the time period Q_{per}, the i-th list ($1 \leq i \leq k$) will contain the i-order NN moving object (with respect to the query point or the query trajectory) at this time instance.

To exemplify the proposed k-NN extensions, let us recall Figure 3.1. Searching for the 2-NN versions of the four queries (Query 1, 2, 3 and 4) presented in Section 3.1, we will have the following results:

- Query 1 (historical non-continuous): O_1 (1st NN) and O_2 (2nd NN)
- Query 2 (historical continuous): 1-NN list includes O_2 for the interval $[t_1,t_3)$ and O_1 for the interval $[t_3,t_4]$; 2-NN list includes O_1 for the interval $[t_1,t_3)$ and O_2 for the interval $[t_3,t_4]$
- Query 3 (historical non-continuous): O_2 (1st NN) and O_4 (2nd NN)
- Query 4 (historical continuous): 1-NN list includes O_5 for the interval $[t_2,t_5)$ and O_4 for the interval $[t_5,t_6]$; 2-NN list includes O_4 for the interval $[t_2,t_5)$ and O_5 for the interval $[t_5,t_6]$.

3.3.2. Metrics

We exploit on the definition of the minimum distance metric (*MINDIST*) presented in [RKV95] between points and rectangles, in order to calculate the minimum distance between line segments and rectangles and the minimum distance between trajectories and rectangles, both of which are needed to implement the above discussed algorithms.

Initially, in [RKV95], Roussopoulos et al. defined the Minimum Distance (*MINDIST*) between a point *P* and a rectangle *R* in the n-dimensional space as the square of the Euclidean distance between *P* and the nearest edge of *R*, if *P* is outside *R* (or zero, if *P* is inside *R*). Then, Tao et al. [TPS02] proposed a method to calculate the *MINDIST* between a 2D line segment *L* and a rectangle M (Figure 3.2).

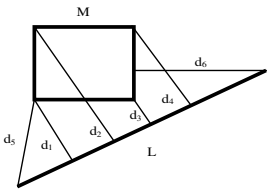
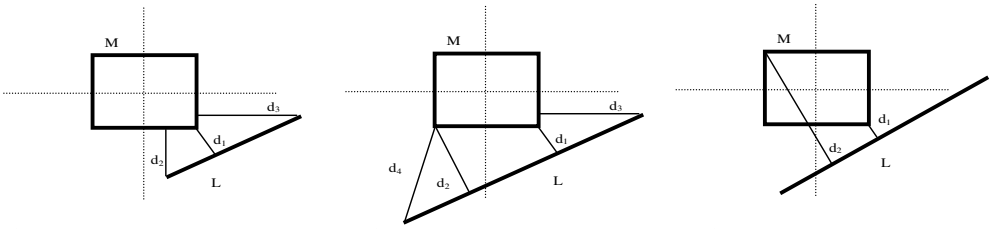


Figure 3.2: Calculating *MINDIST* between a line segment and a rectangle [TPS02]

MINDIST calculation method in [TPS02] initially determines whether *L* intersects *M*; if so, *MINDIST* is set to zero. Otherwise, the shortest among six distances is chosen, namely the four distances between each corner point of *M* and *L* (d_1, d_2, d_3, d_4) and the two minimum distances from the start and end point of *L* to *M* (d_5, d_6). Therefore, the calculation of *MINDIST* between a line segment and a rectangle involves an intersection check, four segment-to-point *MINDIST* calculations and two point-to-rectangle *MINDIST* calculations.



Case 1: *L.start* and *L.end* belong to the same quadrant

Case 2: *L.start* and *L.end* belong to adjacent quadrants

Case 3: *L.start* and *L.end* belong to non adjacent quadrants

Figure 3.3: The proposed calculation method of *MINDIST* between a line segment and a rectangle

In this thesis, we propose a more efficient method to calculate *MINDIST* between a line segment *L* and a rectangle *M* (Figure 3.3). As before, if *L* intersects *M*, then *MINDIST* is obviously zero. Otherwise, we decompose the space in four quadrants using the two axes passing through the center of *M* and we determine the quadrants Q_s and Q_e in which the start (*L.start*) and the end (*L.end*) point of *L* lie in, respectively.

Then, *MINDIST* is the minimum among:

- **Case 1** (the two end points of the line segment belong to the same quadrant (Q_s)): (i) *MINDIST* between the corner of *M* in Q_s and *L*, (ii) *MINDIST* between *L.start* and *M*, and (iii) *MINDIST* between *L.end* and *M*.

- **Case 2** (*L.start* and *L.end* belong to adjacent quadrants Q_s and Q_e, respectively): (i) *MINDIST* between the corner of *M* in Q_s and *L*, (ii) *MINDIST* between the corner of *M* in Q_e and *L*, (iii) *MINDIST* between *L.start* and *M*, and (iv) *MINDIST* between *L.end* and *M*.
- **Case 3** (*L.start* and *L.end* belong to non adjacent quadrants Q_s and Q_e, respectively): two *MINDIST* between the two corners of *M*, that do not belong in either Q_s or Q_e, and *L*.

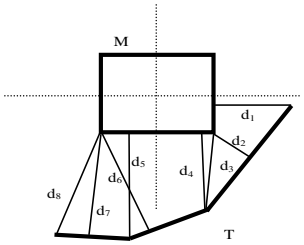


Figure 3.4: The proposed calculation method of *MINDIST* between a route (projection of a trajectory on the plane) and a rectangle

This method utilizes a smaller number of (point-to-segment and point-to-rectangle) distance calculations compared to the corresponding algorithm in [TPS02]. The worst-case scenario of the proposed *MINDIST* calculation includes the determination of the quadrant in which the starting and ending points of the line segment belong, and two point-to-segment and two point-to-rectangle distance calculations, while the corresponding algorithm of [TPS02] employs four point-to-segment and two point-to-rectangle calculations. Therefore, the proposed *MINDIST* calculation, in its worst case, determines the quadrant of the starting and ending point instead of performing two additional point-to-segment distance calculations.The efficiency of the proposed improvement over the *MINDIST* computation for line segments and trajectories will be shown in the experimental section.

Finally, we extend the above algorithm in order to calculate *MINDIST* metric between the projection of a trajectory *T* on the plane (usually called *route*) and a rectangle *M* (Figure 3.4), and provide the *MINDIST_Trajectory_Rectangle* metric. Since a route can be viewed as a collection of 2D line segments, the *MINDIST_Trajectory_Rectangle* between a route of a trajectory and a rectangle can be computed as the minimum of all *MINDIST* between the rectangle and each line segment composing the route. The efficiency of this calculation can be enhanced by simply not computing twice, with respect to the query rectangle, the quadrant and the *MINDIST* of the end and the start of adjacent line segments.

3.3.3. Determining the Function of Distance between two Synchronously Moving Trajectories

Before proceeding into the core of this chapter describing the respective algorithms, it is essential to point out that any algorithm that queries a trajectory database in order to retrieve the nearest to a query trajectory (i.e., the $NN\_Q_T(D, Q_T, Q_{per})$ case following the notation of section 0), should calculate the minimum distance between two concurrently moving trajectories; moreover, given that the historical continuous nearest neighbor search $HCNN\_Q_T(D, Q_T, Q_{per})$ retrieves time-varying real values R_i describing the distance between the query trajectory and the nearest database trajectories at any time instance of the query period Q_{per}, it naturally results that these time varying real values should be

functions of distance with time between the corresponding trajectories. Then again, given that moving object trajectories are modeled as strings of consecutive 3D line segments (i.e., 3D polylines), this minimum distance may be translated to the minimum distance between two 3D line segments.

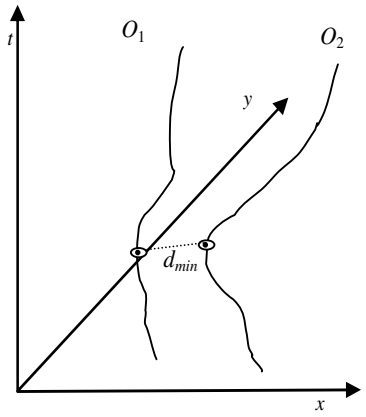


Figure 3.5: Minimum Synchronous Euclidean distance (i.e., "horizontal") between two trajectories

The function of the *Synchronous Euclidean distance* (i.e., "horizontal", illustrated in Figure 3.1) between two 3D line segments, $P\left(\left(P_{1x},P_{1y},t_1\right),\left(P_{2x},P_{2y},t_2\right)\right)$ and $Q\left(\left(Q_{1x},Q_{1y},t_1\right),\left(Q_{2x},Q_{2y},t_2\right)\right)$ is:

$$Dist(t)=\sqrt{\left(Q_x(t)-P_x(t)\right)^2+\left(Q_y(t)-P_y(t)\right)^2} \tag{3.1}$$

Replacing $Q_x(t)=Q_{1x}+\left(Q_{2x}-Q_{1x}\right)\cdot\Delta t$, $Q_y(t)=Q_{1y}+\left(Q_{2y}-Q_{1y}\right)\cdot\Delta t$, $P_x(t)=P_{1x}+\left(P_{2x}-P_{1x}\right)\cdot\Delta t$, $P_y(t)=P_{1y}+\left(P_{2y}-P_{1y}\right)\cdot\Delta t$, in (3.1), we get

$$Dist(t)=\sqrt{\left(Q_{1x}+\left(Q_{2x}-Q_{1x}\right)\cdot\Delta t-P_{1x}-\left(P_{2x}-P_{1x}\right)\cdot\Delta t\right)^2+\left(Q_{1y}+\left(Q_{2y}-Q_{1y}\right)\cdot\Delta t-P_{1y}-\left(P_{2y}-P_{1y}\right)\cdot\Delta t\right)^2}$$

In the sequel, we use the square of the Euclidean distance for sake of readiness.

$$Dist^2(t)=\left(Q_{1x}+\left(Q_{2x}-Q_{1x}\right)\cdot\Delta t-P_{1x}-\left(P_{2x}-P_{1x}\right)\cdot\Delta t\right)^2+\left(Q_{1y}+\left(Q_{2y}-Q_{1y}\right)\cdot\Delta t-P_{1y}-\left(P_{2y}-P_{1y}\right)\cdot\Delta t\right)^2=$$

$$=\left(\left(Q_{2x}-Q_{1x}-P_{2x}+P_{1x}\right)^2+\left(Q_{2y}-Q_{1y}-P_{2y}+P_{1y}\right)^2\right)\cdot\Delta t^2+$$

$$+2\left(\left(Q_{2x}-Q_{1x}-P_{2x}+P_{1x}\right)\left(Q_{1x}-P_{1x}\right)+\left(Q_{2y}-Q_{1y}-P_{2y}+P_{1y}\right)\left(Q_{1y}-P_{1y}\right)\right)\cdot\Delta t+\left(Q_{1x}-P_{1x}\right)^2+\left(Q_{1y}-P_{1y}\right)^2$$

Setting

$$A=\left(Q_{2x}-Q_{1x}-P_{2x}+P_{1x}\right)^2+\left(Q_{2y}-Q_{1y}-P_{2y}+P_{1y}\right)^2 \tag{3.2}$$

$$B=2\left(\left(Q_{2x}-Q_{1x}-P_{2x}+P_{1x}\right)\left(Q_{1x}-P_{1x}\right)+\left(Q_{2y}-Q_{1y}-P_{2y}+P_{1y}\right)\left(Q_{1y}-P_{1y}\right)\right) \tag{3.3}$$

$$C=\left(Q_{1x}-P_{1x}\right)^2+\left(Q_{1y}-P_{1y}\right)^2 \tag{3.4}$$

and replacing Δt according to the following formula $\Delta t=\frac{t-t_1}{t_2-t_1}$, the Synchronous Euclidean "horizontal" distance function of two 3D line segments is computed as follows:

$$Dist^2(t)=\frac{A}{(t_2-t_1)^2}t^2+\left(\frac{B}{t_2-t_1}-\frac{2A\cdot t_1}{(t_2-t_1)^2}\right)t+\frac{A\cdot t_1^2}{(t_2-t_1)^2}-\frac{B\cdot t_1}{t_2-t_1}+C, \tag{3.5}$$

where *A*, *B*, *C* are defined by formulas (3.2), (3.3), and (3.4), respectively.

According to equation (3.5), the square of the Synchronous Euclidean "horizontal" distance function between two 3D line segments has the quadratic form $Dist(t) = a \cdot t^2 + b \cdot t + c$, which is minimized at $Dist_{\min} = c - \frac{b^2}{4a}$ for $t = -\frac{b}{2a}$. Thus, in our case

$$Dist^2_{\min} = \frac{A \cdot t_1^2}{(t_2 - t_1)^2} - \frac{B \cdot t_1}{t_2 - t_1} + C - \frac{\left(\frac{B}{t_2 - t_1} - \frac{2A \cdot t_1}{(t_2 - t_1)^2}\right)^2}{\frac{4A}{(t_2 - t_1)^2}} \tag{3.6}$$

for

$$t = \frac{\frac{2A \cdot t_1}{(t_2 - t_1)^2} - \frac{B}{t_2 - t_1}}{\frac{2A}{(t_2 - t_1)^2}} \tag{3.7}$$

where A, B, C are defined by formulas (3.2), (3.3), and (3.4), respectively.

We have to note that formula (3.6) can be used in case where t calculated by formula (3.7) is inside the query time period $Q_{per}[t_{start}, t_{end}]$. Otherwise, we distinguish between the following two cases:

- if $t \leq t_{start}$, then the minimum synchronous Euclidean "horizontal" distance is provided by formula (3.5) by setting $t = t_{start}$
- if $t \geq t_{end}$, then the minimum synchronous Euclidean "horizontal" distance is provided by formula (3.5) by setting $t = t_{end}$.

3.4. Algorithms for Nearest Neighbor Queries over Trajectories

In this section several algorithms, answering the first two (historical non-continuous) types of NN queries presented in Section 0 are thoroughly introduced and, then, generalized in order to support the respective *k*-NN queries. Both approaches traditionally used to process nearest neighbor queries over spatial data, are followed, namely the Depth-First [RKV95] and the Best-First [HS99] approach. As such, depth-first algorithms are first presented, followed by their best-first counterparts.

3.4.1. Non-incremental (Depth-First) NN Algorithms over Trajectories

Hereafter are presented non-incremental algorithms answering the first two (historical non-continuous) types of NN queries presented in Section 0, also generalized in order to support the respective *k*-NN queries.

3.4.1.1. Non-incremental NN Algorithm for Stationary Query Objects

The non-incremental NN algorithm for stationary query objects (`PointNNSearch` algorithm illustrated in Figure 3.6), provides the ability to answer NN queries for a static query object Q_p, during a certain query time period $Q_{per}[t_{start}, t_{end}]$. The algorithm uses the same heuristics as in [RKV95] and [CF98], pruning the search space according to Q_{per}.

The algorithm accesses the tree structure (which indexes the trajectories of the moving objects) in a depth-first way pruning the tree nodes according to Q_{per} rejecting those being fully outside it. At the leaf level, the algorithm iterates through the leaf entries checking whether the lifetime of an entry overlaps Q_{per} (Line 7); if the temporal component of the entry is fully inside Q_{per}, the algorithm calculates the actual Euclidean distance between Q and the (spatial component of the) entry; otherwise,

if the temporal component of the entry is only partially inside Q_{per}, a linear interpolation is applied so as to compute the entry's portion being inside Q_{per} (Line 9) and calculate the Euclidean distance between Q and the portion of that entry. When a candidate *nearest* is selected, the algorithm, backtracking to the upper level, prunes the nodes in the active branch list (Line 27) applying the *MINDIST* heuristic [RKV95] [CF98].

```
 1. Algorithm PointNNSearch(node N, point Q, period Q_per, struct Nearest)
 2.    IF N Is Leaf
 3.    // Iterate through leaf entries computing Euclidean
 4.    //distance from point Q
 5.       FOR EACH Entry E IN N
 6.          // If entry is (fully or partially) inside the period
 7.          IF Q_per Overlaps (E.T_S, E.T_E)
 8.             // Compute entry's spatial extent inside the period
 9.             nE = Interpolate(E, Max(Q_per.T_S, E.T_S), Min(Q_per.T_E, E.T_E))
10.             // Compute Entry's actual distance from Q.
11.             // Update Nearest if necessary
12.             Dist = Euclidean_Dist_2D(Q, nE)
13.             IF Dist < Nearest.Dist Update Nearest with nE, Dist
14.          ENDIF
15.       NEXT
16.    ELSE
17.       // Generate Node's branch list with entries overlapping
18.       // the query period
19.       BranchList = GenBranchList(Q, N, Q_per)
20.       // Sort active branch List by MinDist
21.       SortBranchList(BranchList)
22.       // Iterate through active branch List
23.       FOR EACH Entry E IN BranchList
24.          // Visit Child Nodes
25.          PointNNSearch(E.ChildNode, Q, Q_per, Nearest)
26.          // Apply MinDist heuristic to do pruning
27.          PruneBranchList(BranchList)
28.       NEXT
29.    ENDIF
```

Figure 3.6: Historical NN search algorithm for stationary query points

3.4.1.2. Non-incremental NN Algorithm for Moving Query Objects

`PointNNSearch` algorithm can be modified in order to support the second type of NN query where the query object is a trajectory of a moving point (`TrajectoryNNSearch` algorithm, illustrated in Figure 3.7). At the leaf level, the algorithm calculates the minimum Euclidean distance between each leaf entry and each query trajectory segment by using the `Min_Horizontal_Dist` function (Line 10), which computes the minimum Synchronous Euclidean distance between two 3D line segments, applying equations (3.6) or (3.5), according to the corresponding discussion. In addition, for each query trajectory segment *QE* and before calculating its distance from the current leaf entry we first interpolate in order to produce a tuple of entry - query segment with identical temporal extent (Lines 8, 9). In order to decrease the number of temporal overlap evaluations between leaf entries and trajectory segments, our algorithm utilizes a plane sweep method, which scans leaf entries and trajectory segments in their temporal dimension in a single pass (Lines 5, 6, 7). This requires that the leaf entries are previously sorted according to their temporal extent (Line 4), unless the underlying tree structure (such as the TB-tree) stores them in temporal order anyway.

```
 1.  Algorithm TrajectoryNNSearch(node N, trajectory Q, period Q_per,
                                  struct Nearest)
 2.     Q = Interpolate(Q, Max(Q.T_S, Q_per.T_S), Min(Q.T_E, Q_per.T_E))
 3.     IF N Is Leaf
 4.        Sort(N, T_S) // Sort A-Z Entries in Node N by their Tstart
 5.        FOR EACH Entry E IN N
 6.           Find next query trajectory entry QS with QS.T_e<N.T_S; QE=QS
 7.           DO UNTIL QE.T_s > E.T_e
 8.              nE = Interpolate(E, Max(QE.T_S, E.T_S), Min(QE.T_E, E.T_E))
 9.              nQE = Interpolate(QE, Max(QE.T_S, E.T_S), Min(QE.T_E, E.T_E))
10.              Dist = Min_Horizontal_Dist(nQE, nE)
11.              IF Dist < Nearest.Dist Update Nearest with nE, Dist
12.           NEXT query entry QE
13.           Return QE in the query entry QS
14.        NEXT
15.     ELSE
16.        BranchList = GenTrajectoryBranchList(Q, N)
17.        SortBranchList(BranchList)
18.        FOR EACH Entry E IN BranchList
19.           TrajectoryNNSearch(E.ChildNode, E.Trajectory, Nearest)
20.           PruneBranchList(BranchList)
21.        NEXT
22.     ENDIF
```

Figure 3.7: Historical NN search algorithm for moving query points

At the non-leaf levels, the algorithm utilizes the `GenTrajectoryBranchList` function (pseudo-code in Figure 3.8) instead of `GenBranchList`. The `GenTrajectoryBranchList` function utilizes the *MinDist_Trajectory_Rectangle* metric introduced in Section 3.3.2 in order to calculate *MINDIST* between the query trajectory and the rectangle of each entry of node *N*. Here, we have to point out that we do not need to calculate *MinDist_Trajectory_Rectangle* against the actual query trajectory *Q*, but only against the part of *Q* being inside the temporal extent of the bounding rectangle of *N*, and in order to do this (if it is necessary) we interpolate to produce the new query trajectory *nQ* (Line 6). The interpolated trajectory *nQ* is also stored inside the *Branchlist* along with the respective node entry and the calculated distance (Line 8). Since all the nodes in the sub-tree of *N* are spatially and temporally contained inside *N*, the interpolated trajectory *nQ* can be used as the query trajectory for the nodes of the next level inside the sub-tree, allowing us to avoid unnecessary calculations.

```
 1.  Algorithm genTrajectoryBranchList(node N, trajectory Q)
 2.     FOR EACH Entry E IN N
 3.        // If entry is partially inside the trajectory lifetime
 4.        IF (Q.T_S, Q.T_E) Overlaps (E.T_S, E.T_E)
 5.           // Compute trajectory's spatial extent inside E's lifetime
 6.           nQ = Interpolate(Q, Max(Q.T_S, E.T_S), Min(Q.T_E, E.T_E))
 7.           // Compute MinDist between the trajectory and the rectangle
 8.           Dist=MinDist_Trajectory_Rectangle(nQ, E)
 9.           // Add the rectangle along with its calculated distance and
10.           // the interpolated trajectory in the list
11.           List.Add(E, Dist, nQ)
12.        ENDIF
13.     NEXT
14.     RETURN List
```

Figure 3.8: Generating Branch List of Node *N* against Trajectory *Q*

3.4.1.3. Extending to non-incremental k-NN Algorithms

In the same fashion as in [RKV95], we generalize the above two algorithms to searching the *k*-nearest neighbors by considering the following:

- Using a buffer of at most *k* (current) nearest objects sorted by their actual distance from the query object (point or trajectory)
- Pruning according to the distance of the (currently) furthest nearest object in the buffer.
- Updating the distance of each moving object inside the buffer when visiting a node that contains an entry of the same object closer to the query object.

3.4.2. Incremental (Best-First) NN Algorithms over Trajectories

Following from the previous section, we now present the best-first counterparts of the previously presented algorithms and, then, we generalize them in order to support the respective *k*-NN queries.

3.4.2.1. Incremental NN Algorithm for Stationary Query Objects

The proposed algorithm, which is based on the NN algorithm for static objects presented in [HS99], traverses the tree structure in a best-first way. The algorithm uses a priority queue, in which the entries of the tree nodes are stored in increasing order of their distance from the query object.

```
 1.  Algorithm IncPointNNSearch(R-tree R, 2D point Q, time period Q_per)
 2.     EnQueue Queue, R.RootNode, 0
 3.     DO WHILE Queue.Count > 0
 4.        Element = DeQueue(Queue)
 5.        IF Element Is MovingObjectEntry
 6.           RETURN Element as the next nearest object
 7.        ELSEIF Element Is Leaf
 8.           // Iterate through leaf entries computing Euclidean
 9.           // distance from Q
10.           FOR EACH Entry E IN leaf node Element
11.              // If entry is (fully or partially) inside the period
12.              IF Q_per Overlaps (E.T_S, E.T_E)
13.                 // Compute entry's spatial extent inside the period
14.                 nE = Interpolate(E,Max(Q_per.T_S E.T_S),Min(Q_per.T_E,E.T_E))
15.                 // Compute Entry's actual distance from Q.
16.                 Dist = Euclidean_Dist_2D(Q, nE)
17.                 EnQueue Queue, nE, Dist
18.              ENDIF
19.           NEXT
20.        ELSE // Element is a non leaf node
21.           // Iterate through node entries computing their
22.           // minimum distance from Q
23.           FOR EACH Entry E IN node Element
24.              // If entry is (fully or partially) inside the period
25.              IF Q_per Overlaps (E.T_S, E.T_E)
26.                 // Compute Entry's MinDist from Q.
27.                 Dist = MinDist(Q, E)
28.                 EnQueue Queue, E, Dist
29.              ENDIF
30.           NEXT
31.        ENDIF
32.     LOOP
```

Figure 3.9: Historical Incremental NN search algorithm for stationary query points

Figure 3.9 illustrates the `IncPointNNSearch` algorithm. In Line 1, the priority queue is initialized. In Line 6, the next nearest object is reported. As in the respective depth-first algorithm described in Section 3.4.1.1, at the leaf level the algorithm iterates through the leaf entries checking

whether the lifetime of an entry overlaps the time period of the query Q_{per} (Line 10); if the temporal component of the entry is fully inside Q_{per}, the algorithm calculates the actual Euclidean distance between Q and the (spatial component of the) entry; otherwise, if the temporal component of the entry is only partially inside Q_{per}, a linear interpolation is applied so as to compute the entry's portion being inside Q_{per} (Line 14) and calculate the Euclidean distance between Q and the portion of that entry (Line 16). In Line 17, the leaf entry is enqueued along with its real distance from the query object. At the non leaf levels (Lines 23-30), the algorithm simply calculates *MINDIST* between the query object and each node's entry overlapping the query period Q_{per}, and in the sequel enqueues this entry along with its *MINDIST* value.

3.4.2.2. Incremental NN Algorithm for Moving Query Objects

The `IncPointNNSearch` algorithm proposed above can be slightly modified in order to support the second type of NN query where the query object is a trajectory of a moving point, thus resulting in `IncTrajectoryNNSearch` algorithm, illustrated in Figure 3.10. The changes to be made are the following three: firstly, as in the respective depth-first algorithm (Section 3.4.1.1), at the leaf level, the algorithm calculates the minimum "horizontal" Euclidean distance between each leaf entry and each segment of the query trajectory Q, using the `Min_Horizontal_Dist` function (Line 15) exploiting equation (3.6). We also utilize the same plane sweep algorithm, so as to determine which leaf entries and segments of Q overlap in their temporal dimension, and then we calculate the distance between those who do overlap (Lines 10-12).

```
 1.  Algorithm IncTrajectoryNNSearch(R-tree R,trajectory Q, period Q_per)
 2.     Q = Interpolate(Q, Max(Q.T_S, Q_per.T_S), Min(Q.T_E, Q_per.T_E))
 3.     EnQueue Queue, R.RootNode, Q, 0
 4.     DO WHILE Queue.Count > 0
 5.        DeQueue(Queue, Element, Q)
 6.        IF Element Is MovingObjectEntry
 7.           RETURN Element as the next nearest object
 8.        ELSEIF Element Is Leaf
 9.           Sort(Element, T_S)// Sort A-Z Entries in Node by their T_start
10.           FOR EACH Entry E IN leaf node Element
11.              Find next query trajectory entry QS with QS.T_e<N.T_S; QE=QS
12.              DO UNTIL QE.T_s > E.T_e
13.                 nE = Interpolate(E,Max(QE.T_S, E.T_S),Min(QE.T_E, E.T_E))
14.                 nQE = Interpolate(QE,Max(QE.T_S, E.T_S),Min(QE.T_E, E.T_E))
15.                 Dist = Min_Horizontal_Dist(nQE, nE)
16.                 EnQueue Queue, nE, Dist
17.              NEXT query entry QE
18.              Return QE in the query entry QS
19.           NEXT
20.        ELSE
21.           FOR EACH Entry E IN node Element
22.              IF (Q.T_S, Q.T_E) Overlaps (E.T_S, E.T_E)
23.                 nQ = Interpolate(Q, Max(Q.T_S, E.T_S), Min(Q.T_E, E.T_E))
24.                 Dist = MinDist_Trajectory_Rectangle(nQ, E)
25.                 EnQueue Queue, E, Dist, nQ
26.              ENDIF
27.           NEXT
28.        ENDIF
29.     LOOP
```

Figure 3.10: Historical Incremental NN search algorithm for moving query points

At the non-leaf levels, the algorithm utilizes the *MinDist_Trajectory_Rectangle* metric in order to calculate the *MINDIST* between the query trajectory and the rectangle of each entry of the node (Line 24). Just like `TrajectoryNNSearch` algorithm, if necessary, we interpolate in order to produce *nQ,* which is the part of *Q* being inside the temporal extent of the bounding rectangle of each node's entry (Line 23), and then we store it inside the *Queue* along with the respective node entry and the calculated distance (Line 25). Since all the nodes in the *N*'s sub-tree are spatially and temporally contained inside *N*, then, the interpolated trajectory *nQ* can be further used as the query trajectory for the nodes of the next level inside the sub-tree, allowing us to avoid unnecessary calculations.

3.4.2.3. Extending to Incremental k-NN Algorithms

The algorithms described in Sections 3.4.2.1 and 3.4.2.2 are incremental in the sense that the *k*-th NN can be obtained with very little additional work once the (*k*-1)-th NN has been found. Recall for example `IncTrajectoryNNSearch` illustrated in Figure 3.10; after having found the 1st NN, the next time the condition of Line 4 is true, the 2nd NN will have been found, and so on.

Here, we have to point out that the two different strategies used for the historical non-continuous NN algorithms appear to have both advantages and drawbacks. As already mentioned, while the best-first approach results always in fewer actually visited nodes, and fewer distance evaluations, its performance heavily depends on the size of the priority queue; as it will be clearly shown in the experiments, this drawback can cause the incremental algorithms to perform worse than the depth-first algorithms in terms of execution time, even though they require fewer nodes to be visited and less distances to be evaluated. On the other hand, the incremental algorithms have a serious advantage over the depth-first ones, which is the ability of retrieving each of the *k* nearest neighbors incrementally, while the depth-first approach requires the prior knowledge of the parameter *k*.

3.5. Algorithms for Historical Continuous Nearest Neighbor Queries over Trajectories

In this section we describe the historical continuous counterparts of the algorithms of Section 3.4. In particular, we will address the third type of NN query (searching for NN with respect to a stationary query point at any time during a given time period) and the fourth type of NN query (where the query object is the trajectory of a moving point) and then we will extend them towards k-NN search.

3.5.1. HCNN Algorithm for Stationary Query Objects

We begin the description of the algorithms with the third type of NN query, which searches for the nearest moving objects to a stationary query point at any time during a given time period, The `HContPointNNSearch` algorithm proposed for this type of query is illustrated in Figure 3.11.

All the historical continuous algorithms use a *MovingDist* structure (Figure 3.11, Line 6), storing the parameters of the distance function (calculated using the coefficients of Eq.(3.5)), along with the entry's temporal extent and the associated minimum and maximum of the function during its lifetime. We also store the actual entry inside the structure in order to be able to return it as the query result. The `ConstructMovingDistance` function simply calculates this structure (e.g. the parameters of the

distance function *a*, *b*, *c*, and the minimum D_{min} and maximum D_{max} of the function inside the lifetime of the entry, also applying the discussion of Section 3.3.3).

```
 1.  Algorithm HContPointNNSearch(node N, 2D point Q, Period Q_per, List
                                  Nearests, Roof)
 2.     IF N Is Leaf
 3.        FOR EACH Entry E IN N
 4.           IF Q_per Overlaps (E.T_S, E.T_E)
 5.              nE = Interpolate(E, Max(Q_per.T_S, E.T_S), Min(Q_per.T_E,E.T_E))
 6.              MovingDist = ConstructMovingDistance(nE, Q)
 7.              IF MovingDist.D_min < Roof
 8.                 UpdateNearests(Nearests,MovingDist,Roof)
 9.              ENDIF
10.           ENDIF
11.        NEXT
12.     ELSE
13.        BranchList = GenBranchList(Q, N, Q_per)
14.        SortBranchList(BranchList)
15.        PruneHContBranchList(BranchList, Nearests, Roof)
16.        FOR EACH Entry E IN BranchList
17.           HContPointNNSearch(E.ChildNode, Q, Q_per, Nearests, Roof)
18.           PruneHContBranchList(BranchList, Nearests, Roof)
19.        NEXT
20.     ENDIF
```

Figure 3.11: Historical CNN search algorithm for stationary query points

An interesting point of the algorithm is exposed in Line 6, where the *Nearests* structure is introduced. *Nearests* is a list of adjacent "*Moving Distances*" temporally covering the period Q_{per}. `Roof` is the maximum of all moving distances stored inside the *Nearests* list and is used as a threshold to quickly reject those entries (and prune those branches at the non-leaf level) having their minimum distance greater than *Roof* (consequently, greater than all moving distances stored inside the *Nearests* list). Section 3.5.3, is comprehensively presents the maintenance of the *Nearests* list.

When at non-leaf levels, the `HContPointNNSearch` algorithm in its backtracking applies the pruning algorithm `PruneHContBranchList` (Line 18), which prunes the branch list using the *MINDIST* heuristic: First, it compares the *MINDIST* of each entry with `Roof` and then it calculates the maximum distance inside the `Nearests` list during the entry's lifetime. Then, it prunes all entries having *MINDIST* greater than the one calculated.

3.5.2. HCNN Algorithm for Moving Query Objects

The fourth type of NN query is the historical continuous version of the NN query where the query object is the trajectory of a moving point. The `HContTrajNNSearch` algorithm, used to process this type of query is illustrated in Figure 3.12.

`HContTrajNNSearch` differs from `HContPointNNSearch` at only two points: The first is that, at the leaf level, the `ConstructMovingDistance` function calculates the "moving distance" between two moving points, instead of one moving and one stationary (Line 10). Secondly, at the non-leaf levels, `GenBranchList` is replaced by the `GenTrajectoryBranchList` function introduced in the description of the `TrajectoryNNSearch` algorithm (Line 18). Moreover, as in `TrajectoryNNSearch`, for each query trajectory segment *QE* and before calculating the moving distance from the current leaf entry we first interpolate in order to produce a tuple of entry - query segment with identical temporal extent (Lines 8, 9). We also use the same plane sweep method, in

order to reduce the number of distance calculations between the segments of Q and the leaf entries (Lines 5-7).

```
 1.  Algorithm HContTrajNNSearch (node N, Trajectory Q, period Q_per, List
                                  Nearests, Roof)
 2.      Q = Interpolate(Q, Max(Q.T_S, Q_per.T_S), Min(Q.T_E, Q_per.T_E))
 3.      IF N Is Leaf
 4.         Sort(N, T_S)
 5.         FOR EACH Entry E IN N
 6.            FIND next query trajectory entry QS with QS.T_e<N.T_S; QE=QS
 7.            DO UNTIL QE.T_s > E.T_e
 8.               nE = Interpolate(E, Max(QE.T_S, E.T_S), Min(QE.T_E, E.T_E))
 9.               nQE = Interpolate(QE, Max(QE.T_S, E.T_S), Min(QE.T_E, E.T_E))
10.               MovingDist = ConstructMovingDistance(nE, nQE)
11.               IF MovingDist.D_min<Roof
12.                  UpdateNearests(Nearests,MovingDist,Roof)
13.               ENDIF
14.            NEXT query entry QE
15.            Return QE in the query entry QS
16.         NEXT
17.      ELSE
18.         BranchList = GenTrajectoryBranchList(Q, N)
19.         SortBranchList(BranchList)
20.         PruneHContBranchList(BranchList, Nearests, Roof)
21.         FOR EACH Entry E IN BranchList
22.            HContTrajNNSearch(E.ChildNode,E.Trajectory,Nearests, Roof)
23.            PruneHContBranchList(BranchList, Nearests, Roof)
24.         NEXT
25.      ENDIF
```

Figure 3.12: Historical CNN search algorithm for moving query points

3.5.3. Maintaining the *Nearests* List

The pseudo-code of the `UpdateNearests` function, which is responsible for the maintenance of the *Nearests* list, is presented in Figure 3.13. In particular, the algorithm iterates through the elements of the active *Nearests* list searching for those elements temporally overlapping the checked entry (*CM*). When such an element is found, the algorithm applies linear interpolation in both entries (the checked and the one already on the list) producing two new entries having the same temporal extent (*M* and *T*). Then, it compares the two distance functions in order to determine whether the entry already on the list is to be replaced or not.

Figure 3.14 graphically explains all the possible comparisons between the parabolas of two "*Moving Distance*" functions. In particular, Figure 3.14(a) corresponds to line 6 of the algorithm presented in Figure 3.13, where the maximum distance of *M* is smaller than the minimum of *T*, leading to the replacement of *T* with *M*. Otherwise, after computing the discriminant of the difference between the distance functions of *M* and *T*, we have to distinguish among three different cases:

- **Case 1:** The discriminant is less than zero, meaning that the two functions *M* and *T* are asymptotic and they do not intersect (Line 10); we only have to check their minimum in order to determine which is the global minimum (see Figure 3.14(b))
- **Case 2:** The discriminant is equal to zero, meaning that the two functions osculate in their common minimum (Line 12); we only have to check their maximum in order to determine the global minimum (see Figure 3.14(c))

```
 1.  Algorithm UpdateNearests (List Nearests, struct CM, Roof)
 2.     FOR EACH T IN Nearests
 3.        IF (T.TS, T.TE)Overlaps(CM.TS, CM.TE)
 4.           M=Interpolate(CM, Max(CM.TS, T.TS), Min(CM.TE, T.TE))
 5.           T=Interpolate(T, Max(CM.TS, T.TS), Min(CM.TE, T.TE))
 6.           IF M.DMax < T.DMin
 7.              Nearests.Replace T with M
 8.           ELSEIF M.DMax < T.DMax
 9.              D = Discriminant(M-T)
10.              IF D < 0
11.                 IF T.DMin > M.DMin THEN Nearests.Replace T with M
12.              ELSEIF D=0
13.                 IF T.DMax > M.DMax THEN Nearests.Replace T with M
14.              ELSE
15.                 RR1=Solution1(T - M); RR2=Solution2(T - M)
16.                 R1=Min(RR1,RR2); R2=Max(RR1,RR2)
17.                 IF R2<T.TS OR R1>T.TE
18.                    IF T.DMax > M.DMax THEN Nearests.Replace T with M
19.                 ELSEIF R2<T.TE AND R1>T.TS
20.                    IF M.Dmin < T.Dmin
21.                       M1=Part(M,,R1); M2=Part(M,R2); T1=Part(T,R1,R2)
22.                       Nearests.Replace T with (M1,T1,M2)
23.                    ELSE
24.                       T1=Part(T,,R1); T2=Part(T,R2); M1=Part(M,R1,R2)
25.                       Nearests.Replace T with (T1,T2,M1)
26.                    ENDIF
27.                 ELSE
28.                    IF M(R1 - 1)<T(R1 - 1)
29.                       M1=Part(M,,R1); T1=Part(T,R1)
30.                       Nearests.Replace T with (M1,T1)
31.                    ELSE
32.                       T1=Part(T,,R1); M1=Part(M,R1)
33.                       Nearests.Replace T with (T1,M1)
34.                    ENDIF
35.                 ENDIF
36.              ENDIF
37.           ENDIF
38.        ENDIF
39.        Roof=max(Roof,T.Dmax)
40.     NEXT
```

Figure 3.13: `UpdateNearests` Algorithm

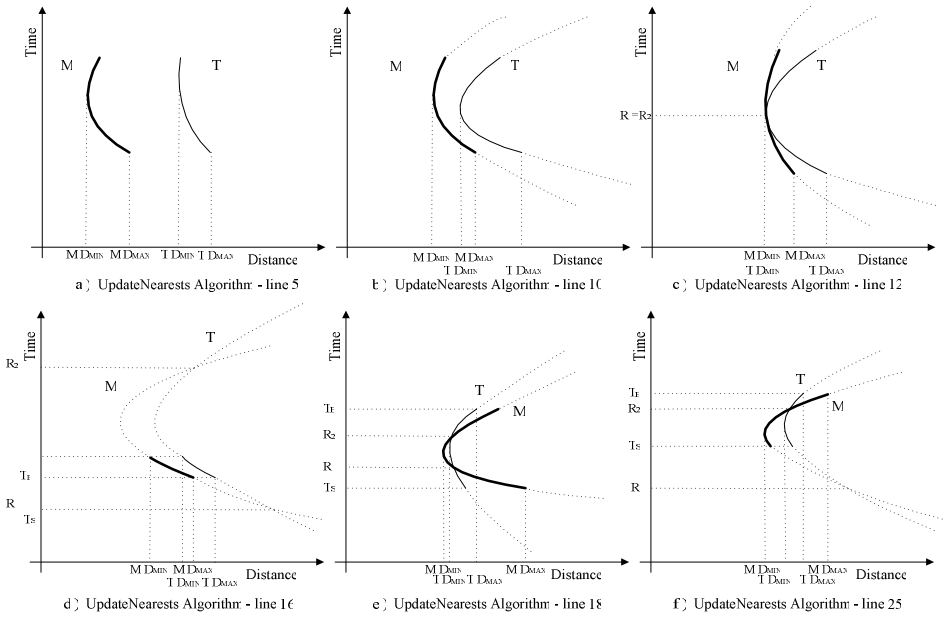


Figure 3.14: Graphical illustration of `UpdateNearests` Algorithm Comparisons

- **Case 3:** The discriminant is greater than zero, meaning that the two functions intersect in two points (Line 14). In this case, we have to determine whether these time instances are inside the entry's lifetime. Hence, we further distinguish among three sub-cases:
    - **Case 3a:** Both solutions are outside the temporal extent of *M* (and *T*) (Line 17). We only have to check their maximum in order to determine which is the globally minimum inside the current temporal interval (see Figure 3.14(d))
    - **Case 3b:** Both solutions are inside the temporal extent of *M* (and *T*) (Line 19). We must break apart the entry into 3 different entries (see Figure 3.14(e)) and determine the part of *T* to be replaced by *M*.
    - **Case 3c:** Only one solution is found inside the temporal extent of *M* (Line 27). We must break apart the entry into two different entries (see Figure 3.14(f)) and determine the part of *T* to be replaced by *M*.

3.5.4. Extending to *k*-HCNN algorithms

The two historical continuous algorithms proposed above can be also generalized to searching the *k*-nearest neighbors by considering the following:

- Using a buffer of at most *k* current *Nearests* lists;
- Pruning according to the distance of the furthest *Nearests* lists in the buffer – therefore *Roof* is calculated as the maximum distance of the furthest *Nearests* list;
- Processing each entry against the *i*-th list (with *i* increasing, from 1 to *k*) checking whether it qualifies to be in a list;
- When a moving distance is replaced by a new entry in the *i*-th list, testing it against the (*i*+1)-th list to find whether it qualifies to be in that list.

3.6. Experimental Study

The above illustrated algorithms can be implemented in any R-tree-like structure storing historical moving object information such as the 3D R-tree, the STR-tree [PJT00] the TB-tree [PJT00] and the TB*-tree.

3.6.1. Experimental Setup

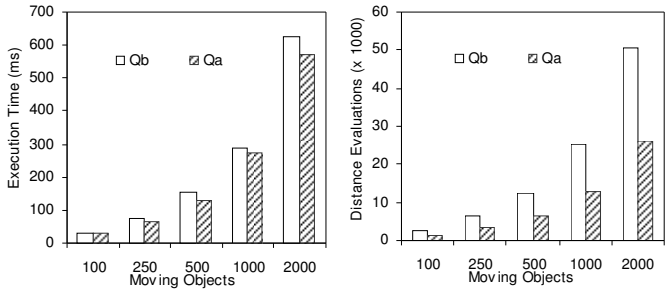
All algorithms were implemented on top of our implementation of R-tree-like structures used in the previous chapter, employing the development environment of Microsoft Visual Basic. The experiments were performed in a PC running Microsoft Windows XP with AMD Athlon 64 3GHz processor, 512 MB RAM and several GB of disk space, a page size of 4 KB and a (variable size) buffer fitting the 10% of the index size, with a maximum capacity of 1000 pages. Finally, in our experimentation we employed the real and synthetic trajectory datasets introduced in sections 0 and 1.5.2, respectively.

3.6.2. Results on the Calculation of the *MINDIST* Metric

In order to demonstrate the efficiency of the proposed *MINDIST* calculation over the one presented in [TPS02], we conducted a set of experiments executing 500 queries over the GSTD datasets indexed by the TB-tree using the `TrajectoryNNSearch` algorithm; nevertheless, similar results gathered when employing the other two alternatives, namely, the 3D R- and the TB*-tree. The queries were initially

executed with the proposed *MINDIST* calculation, forming the Q_a query set, and then with the *MINDIST* calculation proposed in [TPS02], forming the Q_b query set. The set of 500 query objects (trajectories) were produced using GSTD also employing a Gaussian initial distribution and a random movement distribution. Then, a random 1% part of each trajectory was used as the query trajectory. Each query performance was measured in terms of execution time and actual distance evaluations between point and point, point and line, and point and MBB.

Figure 3.15(a) illustrates the average execution time for query sets Q_a and Q_b. Clearly, the `TrajectoryNNSearch` algorithm with the proposed improvement over the *MINDIST* computation is always superior over the corresponding computation as proposed in [TPS02], in all datasets. The improvement over the execution time varies between 8% (in the GSTD 100 dataset) and 17% (in the GSTD 250 dataset). The efficiency of the proposed improvement over the *MINDIST* computation can be further established by Figure 3.15(b), illustrating the actual distance evaluations made from each alternative computation; Figure 3.15(b) shows that the proposed *MINDIST* computation requires in all settings almost half of the distance evaluations made by the analogous computation proposed in [TPS02].



(a) (b)

Figure 3.15: (a) Execution Time and (b) actual Distance Evaluations for query sets Q_a and Q_b increasing the number of moving objects

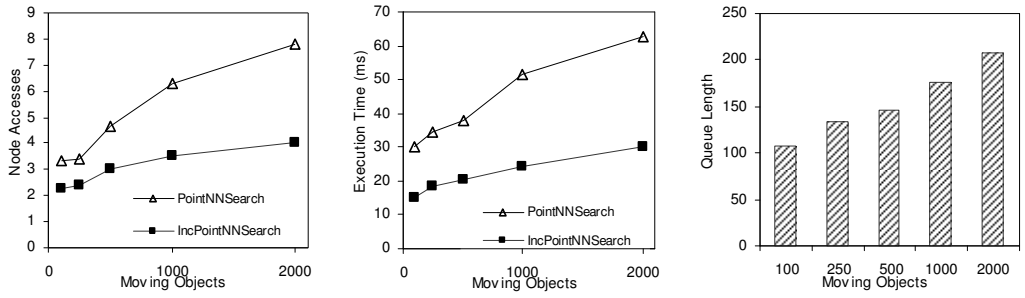
3.6.3. Results on the Search Cost of the Historical Non-continuous Algorithms

The performance of the proposed algorithms was measured in terms of node accesses and execution time. Several queries were used in order to evaluate the performance of the proposed algorithms over the synthetic and real data sets. In particular, we have used the following query sets:

- Q_1: the `PointNNSearch` and the `IncPointNNSearch` algorithms were evaluated with one set of 500 NN queries increasing the number of moving objects over the GSTD datasets indexed by TB-, TB*- and 3D R-tree. The queries used a random point in the 2D space and a time period of 1% of the temporal dimension for Q_1.
- Q_2: the `TrajectoryNNSearch` and the `IncPointNNSearch` algorithms were evaluated with one set of 500 NN queries increasing the number of moving objects over the GSTD datasets indexed by TB-, TB*- and 3D R-tree. The set of 500 query objects (trajectories) was produced using GSTD employing also a Gaussian initial distribution and a random movement distribution. Then, in Q_2 we used a random 1% part of each trajectory as the query trajectory.

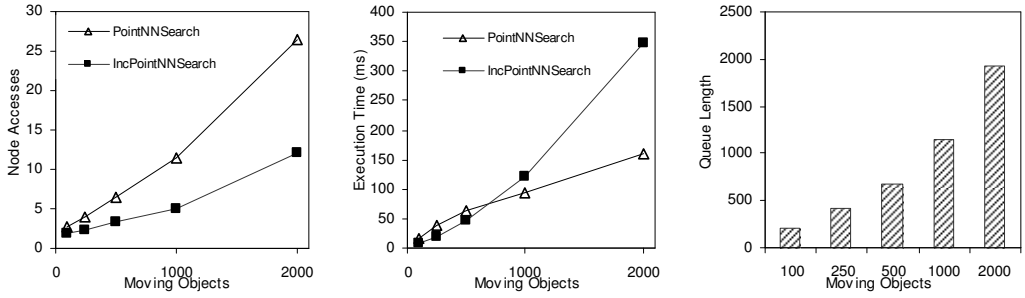
- Q_3, Q_4: two sets of 500 k-NN queries over the real Trucks dataset increasing the number of k with fixed time and increasing the size of the time interval (with fixed k=1) respectively. For the `PointNNSearch` algorithm we used a random point in the 2D space with a 1% of time as query period, while for `TrajectoryNNSearch` algorithm we used a random part of a random trajectory belonging to the Buses dataset, temporally covering 1% of the time.

Figure 3.16 illustrates the results for the Q_1 query set evaluating `PointNNSearch` and `IncPointNNSearch` algorithms over the 3D R-tree, in terms of (a) average number of node accesses and (b) average execution time per query. As it is clearly illustrated, the performance of both algorithms depends sub-linearly on the dataset cardinality, downgrading (more pages are accessed) as the cardinality grows. Another conclusion drawn from the same charts is that `IncPointNNSearch` algorithm outperforms the `PointNNSearch` algorithm in all datasets, in terms of both node accesses and execution time. Figure 3.16(c) illustrates the average length (in nodes) of the queue utilized by the `IncPointNNSearch` in order to answer the queries, increasing linearly with the cardinality of the dataset.



(a) (b) (c)

Figure 3.16: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q_1 executing point NN search over the 3D R-tree indexing the GSTD datasets

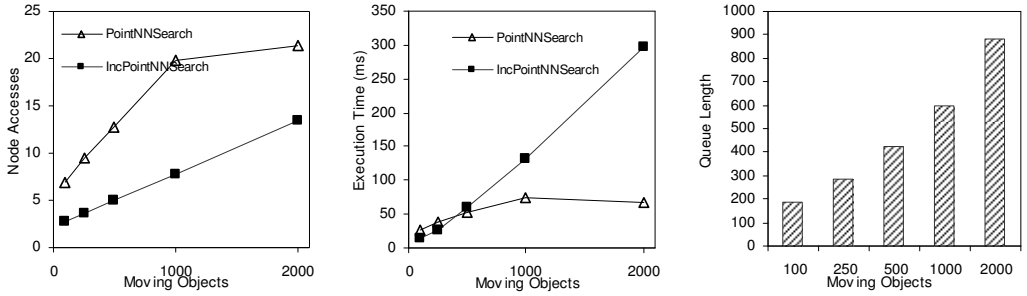


(a) (b) (c)

Figure 3.17: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q_1 executing point NN search over the TB-tree indexing the GSTD datasets

The Q_1 query set evaluating `PointNNSearch` and `IncPointNNSearch` was also executed against the TB-tree and the TB*-tree, leading to the results presented in Figure 3.17 and Figure 3.18, respectively. Although, just as reported for the 3D R-tree, the `IncPointNNSearch` outperforms `PointNNSearch` in terms of average node accesses per query in all datasets (Figure 3.17(a) and

Figure 3.18(a)), the actual average time required for each query execution (Figure 3.17(b) and Figure 3.18(b)) by the `IncPointNNSearch`, increases faster than the respective execution time of the `PointNNSearch`, leading to a superiority of the non-incremental algorithm as the cardinality of the dataset grows.

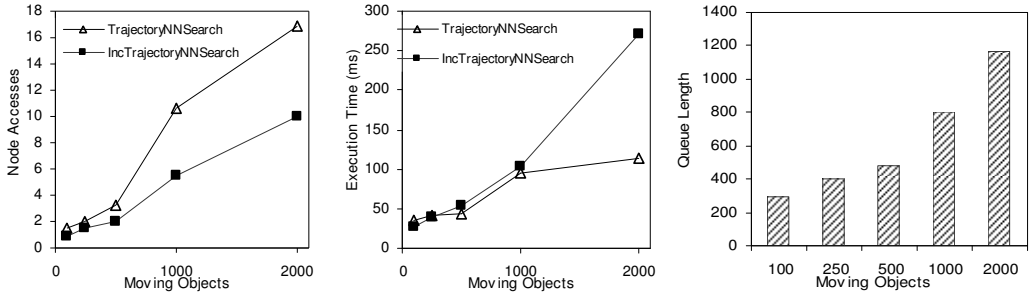


(a) (b) (c)

Figure 3.18: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q_1 executing point NN search over the TB*-tree indexing the GSTD datasets

Exactly the same trend as the one presented for the execution time of the `IncPointNNSearch` is presented in Figure 3.17(c) and Figure 3.18(c) illustrating the length of the queue utilized by the respective algorithm. More specifically, `PointNNSearch` outperforms its incremental counterpart when the average length of the respective queue exceeds a certain number of nodes (approximately 400 nodes in the GSTD 500 dataset). The above conclusion can be also verified from the results of the 3D R-tree, where the length of the queue is always less than 400, leading to a superiority of the incremental algorithm. Regarding the comparison between the performance of the TB, the TB* and the 3D R-tree, the latter outperforms the other two as the dataset cardinality grows, like what was reported in [PJT00] regarding simple range queries of small extent; then again, the original TB-tree seems to marginally outperform the developed in this thesis TB*-tree.

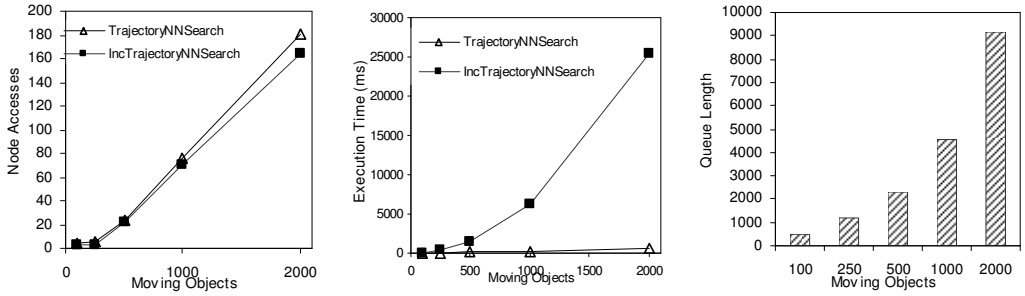
Figure 3.19 illustrates the results for the Q_2 query set evaluating `TrajectoryNNSearch` and `IncTrajectoryNNSearch` algorithms over the 3D R-tree, in terms of average number of node accesses (a) and average execution time per query (b). The performance of both algorithms depends linearly on the dataset cardinality, downgrading as the dataset cardinality grows. Although `IncTrajectoryNNSearch` outperforms `TrajectoryNNSearch` in all datasets in terms of node accesses, the average execution time of the incremental algorithm becomes greater than the respective time of the non-incremental one, as the dataset cardinality grows. The average queue length utilized by the `IncTrajectoryNNSearch`, is also illustrated in Figure 3.19(c); following the results for the execution time of the incremental algorithm, the queue length increases linearly with the cardinality of the dataset. This enlargement of the queue length is also responsible for the behavior showed regarding the comparison of the execution time between the `TrajectoryNNSearch` and the `IncTrajectoryNNSearch` algorithm; as the queue length increases, each update becomes a more expensive operation leading to the downgrade of the performance of the respective algorithm.



(a) (b) (c)

Figure 3.19: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q_2 executing trajectory NN search over the 3D R-tree indexing the GSTD datasets

Regarding a comparison of the performance of the incremental algorithms illustrated in Figure 3.16 and Figure 3.19 leads to the observation that while in the first case, fewer node accesses lead to smaller execution time (than the non-incremental one), in the second case the execution time of the incremental algorithm becomes higher than the respective of its non incremental counterpart. This fact can be explained by observing the respective queue lengths: in the first case the queue length in not more than 200 objects (i.e., less than a typical *BranchList*), while in the second case, the queue length includes 1000's of objects resulting in a decrease of the algorithm's performance.

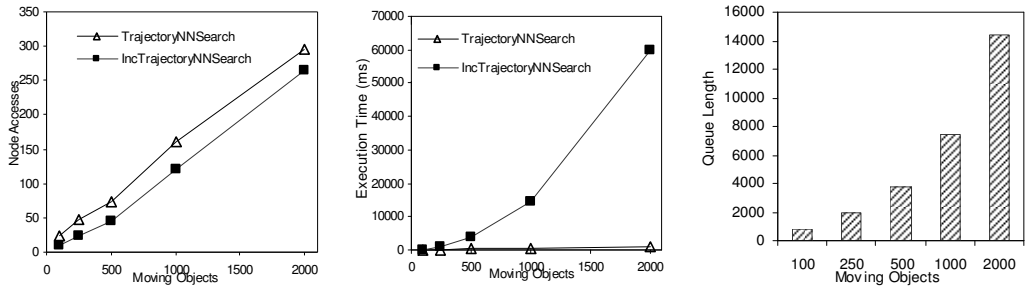


(a) (b) (c)

Figure 3.20: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q_2 executing trajectory NN search over the TB-tree indexing the GSTD datasets

The results of the Q_2 query set over the TB-tree and the TB*-tree are presented in Figure 3.20 and Figure 3.21, respectively. While `IncTrajectoryNNSearch` always outperforms `TrajectoryNNSearch` in terms of average node accesses (Figure 3.20(a) and Figure 3.21(a)), their disparity is not as significant as it was reported for the 3D R-tree. Moreover, the actual execution time of the incremental algorithm (Figure 3.20(b) and Figure 3.21(b)) is always by far longer than the respective execution time of the non-incremental one. These results can be explained by two reasons. The first is that the actual execution time of the incremental algorithm depends heavily on the respective queue length which, as shown in Figure 3.20(c) and Figure 3.21(c), exceeds 1000 nodes for the GSTD 250 dataset reaching 9000 nodes in the GSTD 2000 dataset indexed by the TB-tree, while in the case of the TB*-tree the queue cardinality grows to even higher values (14000). The second is that TB-tree and TB*-tree group entries belonging to the same trajectory together, exploiting only the

temporal order in which the entry insertion occurs ignoring at the same time any spatial proximity. This insertion strategy leads to nodes with high spatial (and low temporal) overlap, meaning that internal nodes will often cross the query trajectory, and the respective *MINDIST* will be equal to zero. Then, the internal nodes need to be visited since their *MINDIST* equals to zero and they are leading inside the queue, resulting to the loss of the advantage of the incremental algorithm.



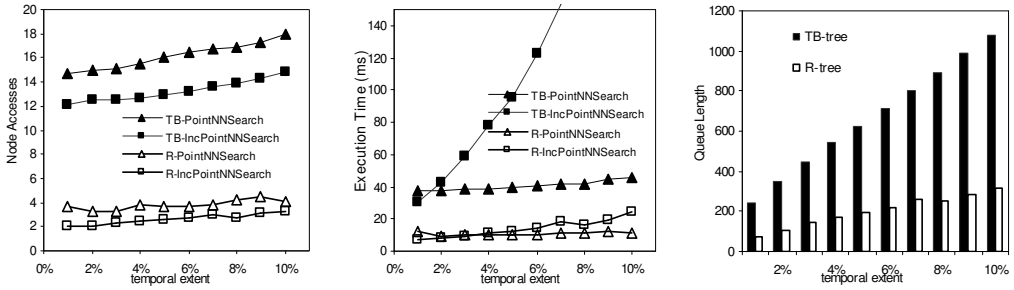
(a) (b) (c)

Figure 3.21: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q_2 executing trajectory NN search over the TB*-tree indexing the GSTD datasets

The same reasons also affect the comparison of the performance between the TB-, the TB*- and the 3D R-tree, where the latter outperforms the other two as the dataset cardinality grows. Moreover, the advantage of the original TB-tree against the TB*-tree that has been revealed in point NN queries, becomes clearer here, where the latter always perform worse than the former. It becomes therefore obvious that the structure of the TB*-tree is not suitable for NN queries. This is mainly due to the fact that the TB*-tree contains wider MBBs (since its leaf capacity is almost the double of the original TB-tree), leading to higher node overlap and lower spatial discrimination; the same tendency has been also detected in the original work of [PJT00] regarding the TB-tree, in the case of range queries of small extent (1% along each dimension, i.e., 0.0001% of the total space), where the high space utilization of TB-tree becomes a drawback that affects its performance. This similarity between small range and nearest neighbor queries can be actually justified considering the work of [TZPM04], where the cost of executing NN queries over multidimensional data is estimated by approximating the vicinity circle $C(q, R)$, i.e., the circle inside which the search is performed with center the query point q and radius R its distance from the k-th nearest neighbor, with a vicinity rectangle of equal area. As such, the more objects in the index, the smaller the radius of the k-th NN, and the smaller the respective vicinity rectangle; finally, the equivalent of a NN query turns to be a range query with small extent (and total area equal to the area of $C(q, R)$). Due to the aforementioned reasons, as well as for the sake of the clarity of the presentation, the TB*-tree will not be further included in the experimental study on historical non-continuous NN queries. Nevertheless, the rest of the conducted experiments verify the observed trend so far, and show that the TB*-tree performs always worse than its other two competitors.

The performance of the historical non-continuous point NN algorithms increasing the query temporal extent, in terms of average node access and average execution time per query, is shown in Figure 3.22 against the 3D R-tree and the TB-tree, both indexing the Trucks dataset. Clearly, under both indexes, the number of node accesses needed for the processing of a NN query, increases linearly

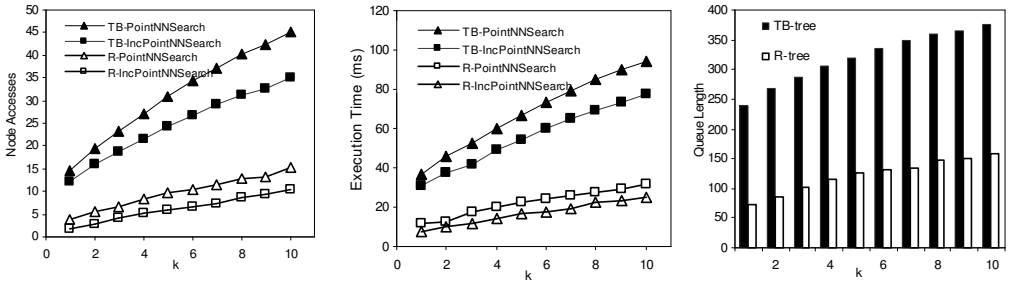
with the query temporal extent, with the `IncPointNNSearch` being always below the `PointNNSearch`. In terms of execution time, both indexes show the same behavior having a breakeven point where the superlinearly increasing execution time of the `IncPointNNSearch` (a consequence of the increasing queue length illustrated in Figure 3.22 (c)) becomes even with the linearly increasing execution time of the `PointNNSearch` algorithm. Regarding the TB-tree, the breakeven point is around the 1.5% of the temporal extent while in the 3D R-tree increases around 3.5%.



(a) (b) (c)

Figure 3.22: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q_3 executing point NN search over the 3D R- and the TB-tree indexing the Trucks dataset

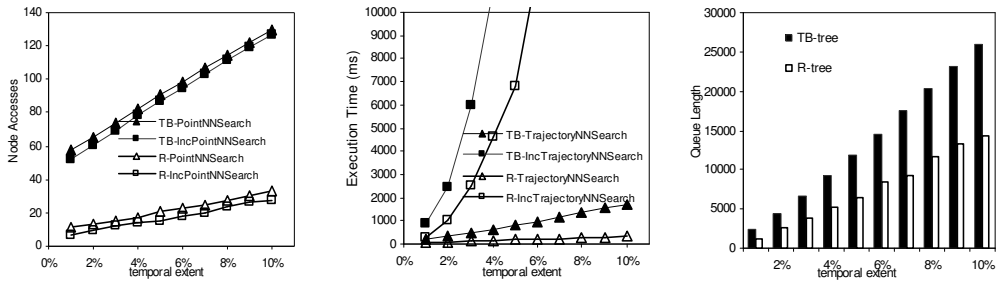
Figure 3.23 illustrates the average number of node accesses and execution time per historical non-continuous point query increasing the number of *k* against the Trucks dataset indexed by the 3D R-tree and TB-tree. Under both indexes it is clear that the incremental algorithm outperforms the `PointNNSearch` in terms of both average node accesses and execution time. Using the 3D R-tree, the performance of both algorithms decreases linearly with the number of *k*, whereas when using the TB-tree the reduction is sub-linear.



(a) (b) (c)

Figure 3.23: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q_3 executing point NN search over the 3D R- and the TB-tree indexing the Trucks dataset

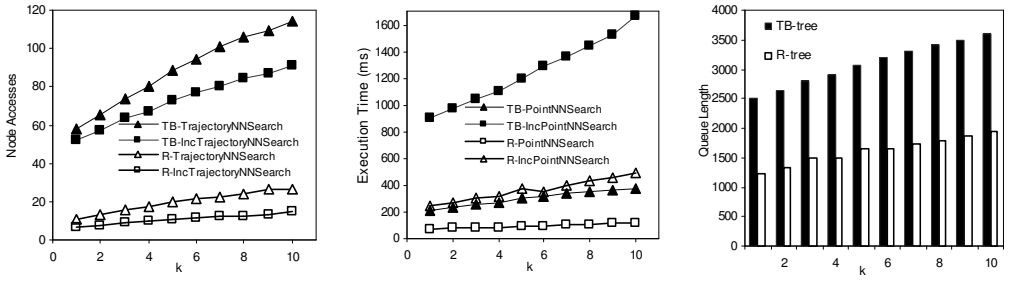
The results for the historical non-continuous trajectory NN algorithms increasing the query temporal extent against the 3D R-tree and TB-tree indexing the Trucks dataset are illustrated in Figure 3.24. Once again, the number of node accesses required for the processing of a NN query with both algorithms under both indexes, increases linearly with the query temporal extent. However, regarding the execution time, the performance of the incremental algorithm grows superlinearly with the temporal extent as a consequence of the excessive queue length (Figure 3.24(c)).



(a) (b) (c)

Figure 3.24: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q_4 executing trajectory NN search over the 3D R- and the TB-tree indexing the Trucks dataset

The performance of the historical non-continuous trajectory query increasing the number of k against the Trucks dataset is shown in Figure 3.25 where the `TrajectoryNNSearch` algorithm outperforms its incremental counterpart in terms of execution time, with the respective queue containing in any case more than 1000 nodes.



(a) (b) (c)

Figure 3.25: (a) Node Accesses, (b) Execution Time and (c) Queue Length in queries Q_4 executing trajectory NN search over the 3D R-tree indexing the Trucks dataset

3.6.4. Results on the Search Cost of the Historical Continuous Algorithms

In coincidence with the experiments conducted for the historical non-continuous algorithms, the historical continuous NN search algorithms were evaluated, also in terms of node accesses and execution time, with the following query sets:

- Q_5: the `HContPointNNSearch` algorithm was evaluated with one set of 500 NN queries increasing the number of moving objects over the GSTD datasets indexed by TB-, TB* and 3D R-tree like what was done for query set Q_1.
- Q_6: the `HContTrajectoryNNSearch` algorithm was evaluated with one set of 500 NN queries increasing the number of moving objects over the GSTD datasets indexed by TB-, TB* and 3D R-tree like what was done for query set Q_2.
- Q_7, Q_8: two sets of 500 k-NN queries over the real Buses dataset increasing the number of k with fixed time and increasing the size of the time interval (with fixed k = 1), respectively. For the `HContPointNNSearch` algorithm we used a random point in the 2D space with a 1% of time as query period, while for `HContTrajectoryNNSearch` algorithm we used a

random part of a random trajectory belonging to the Trucks dataset, temporally covering 1% of the time.

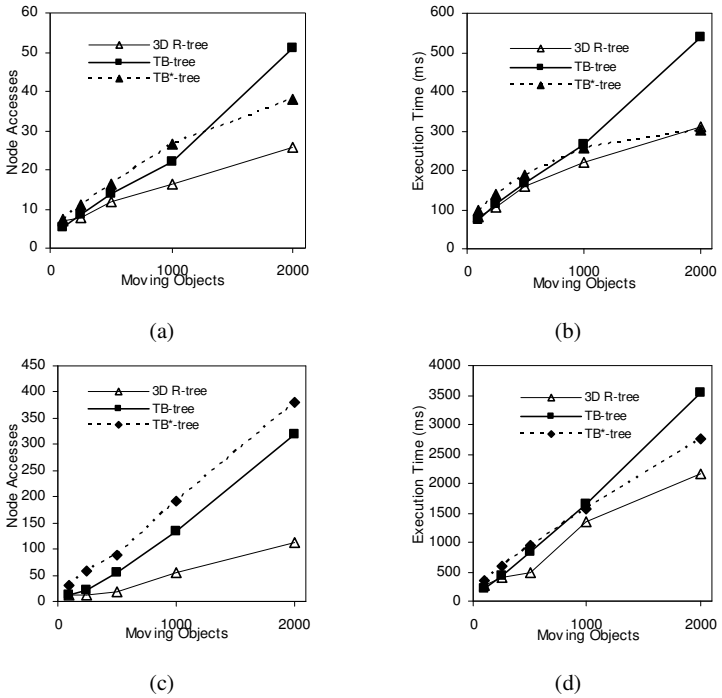


Figure 3.26: Node Accesses and Execution Time in queries Q_5 (a, b) and Q_6 (c, d) over the 3D R-tree, the TB-tree and the TB*-tree increasing the number of moving objects

Figure 3.26(a) and (b) illustrates the results of the `HContPointNNSearch` algorithm over the GSTD datasets by increasing the number of moving objects in terms of (a) average node accesses and (b) average execution time per query. As in its historical non-continuous counterpart, the performance of the algorithm depends linearly on the dataset cardinality downgrading as the cardinality grows, while the average execution time for both indexes follows the same trend as the average number of visited nodes. Another result gathered is that, as the cardinality grows, the 3D R-tree outperforms the TB-tree and the TB*-tree, following the same trend illustrated in [PJT00] for simple range queries of small extent. Similar results are illustrated in Figure 3.26(c) and (d) where the `HContTrajectoryNNSearch` algorithm is executed against the GSTD datasets.

A comparison between the historical non-continuous NN algorithms with their continuous counterpart (i.e., Figure 3.16 and Figure 3.17 vs. Figure 3.26(a) and (b), and Figure 3.19 and Figure 3.20 vs. Figure 3.26(c) and (d)), shows that the historical continuous algorithms are much more expensive than the non-continuous ones. This conclusion was expected since the historical continuous algorithms do not utilize a single distance to prune the search space; instead they use a list of moving distances, which in general stores greater distances than the minimum. Actually, the historical non-continuous algorithms prune the search space with the minimum possible distance stored inside the *Nearests* list, therefore performing pruning much more efficiently than their continuous counterpart.

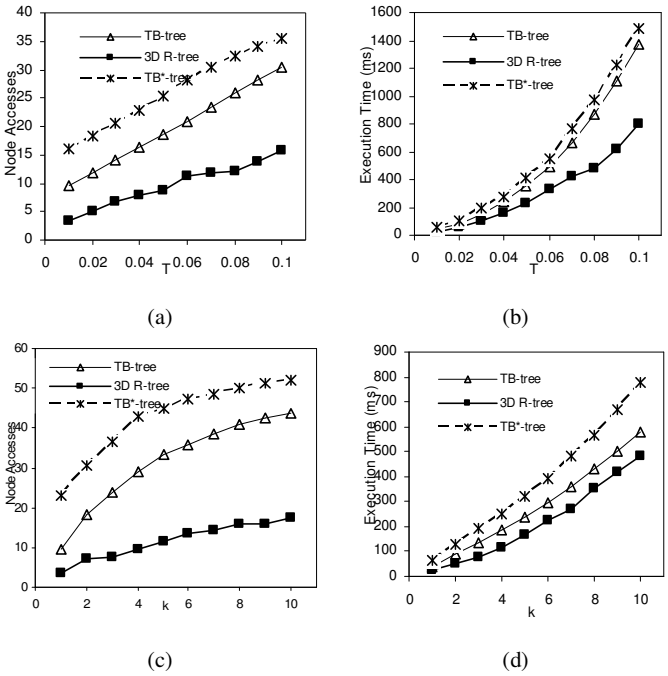


Figure 3.27: Node Accesses and Execution Time in queries *Q*7 (a, b) and *Q*8 (c, d) over the 3D R-tree, the TB-tree and the TB*-tree indexes increasing the query temporal extent

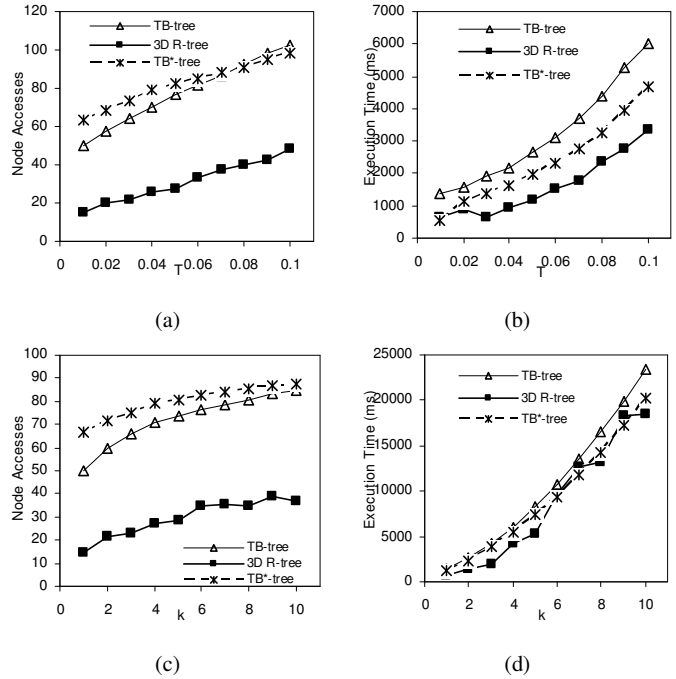


Figure 3.28: Node Accesses and Execution Time in queries *Q*7 (a, b) and *Q*8 (c, d) over the 3D R-tree, the TB-tree and the TB*-tree indexes increasing the number of *k*

The scaling of the historical continuous algorithms with the query temporal extent is presented in Figure 3.27. Both algorithms (`HContPointNNSearch` and `HContTrajectoryNNSearch`) were executed over the real Buses dataset indexed by the TB-, the TB*- and the 3D R-tree. From Figure

3.27(a) and (c) it is clear that the performance of both algorithms in terms of node accesses is sub-linear with respect to the query temporal extent. Nevertheless, the actual execution time needed by each query increases superlinearly with the query extent, as a consequence of the increasing length of the query output (the *Nearests* list). The performance of the historical continuous NN algorithms increasing the number of *k* against the Buses dataset indexed by the TB, the TB*- and the 3D R-tree is illustrated in Figure 3.28. As drawn from Figure 3.28(a) and (c), the average number of node accesses required for the processing of a *k*-HCNN point or trajectory query increases sub-linearly with *k*. However, the actual execution time presented in Figure 3.28(b) and (d) increases superlinearly with the *k*, similarly with the temporal extent, as a consequence of the increasing size of the query output (the *k* *Nearests* lists).

3.6.5. Summary of the Experiments

Most of the presented algorithms, in terms of node accesses, are linear or sub-linear with the main parameters of our experimental study: the dataset cardinality, the query temporal extent and the number of *k*. However, the execution time of the `IncPointNNSearch` and `IncTrajectoryNNSearch` algorithms seems to grow super-linearly with the query temporal extent as a result of the increasing queue length, similarly with the execution time of `HContPointNNSearch` and `HContTrajectoryNNSearch`, which have the same trend with respect to the temporal extend and the number of *k*, as a consequence of the increasing *Nearests* list length.

Table 3.2: Actual indexed space accessed by each NN algorithm for the GSTD 2000 dataset

Algorithm	*3D R-tree*	*TB-tree*	*TB*-tree*
PointNNSearch	0.006%	0.022%	0.070%
IncPointNNSearch	0.003%	0.010%	0.044%
TrajectoryNNSearch	0.014%	0.148%	0.963%
IncTrajectoryNNSearch	0.008%	0.134%	0.868%
HContPointNNSearch	0.016%	0.042%	0.124%
HContTrajectoryNNSearch	0.053%	0.259%	1.248%

Table 3.2 summarizes the pruning power of our algorithms presenting the percentage of the indexed space accessed in order to execute all the proposed algorithms with k=1 and temporal extent the 1% of the indexed time. As it can be concluded our algorithms show high pruning ability, well bounding the space to be searched in order to answer NN and HCNN queries, except of the case of the TB*-tree which, overall, seems that is not a good choice when dealing with NN queries.

3.7. Conclusions

NN queries have been in the core of spatial and spatio-temporal database research during the last decade. The majority of the algorithms processing such queries so far mainly deals with either stationary or moving query points over static datasets or future (predicted) locations over a set of continuously moving points. In this work, acknowledging the contribution of related work, we presented the first complete treatment of historical NN queries over moving object trajectories stored on R-tree-like structures.

Based on our proposed novel metrics (i.e., *MINDIST_Trajectory_Rectangle*), which support our ordering and pruning strategies, we presented algorithms answering the NN and HCNN queries for stationary query points or trajectories and generalized them to search for the *k* nearest neighbors. The algorithms are applicable to R-tree variations for trajectory data, among which for our performance study we used the 3D R-tree, the TB-tree and the TB*-tree. Appart from the implementation of the proposed algorithms over R-tree-like structures used during the experimental study, the `IncPointNNSearch` and `IncTrajectoryNNSearch`, algorithms have been implemented in the ORACLE Object – Relational DBMS and integrated into the HERMES engine [PFGT08], which has been also extended so as to include the TB-tree [PJT00].

In order to measure the performance of our algorithms we conducted an extensive experimental study based on synthetic and real datasets. At first, we demonstrated that our improvement over the *MINDIST* computation can sufficiently increase the performance of the proposed algorithms. Regarding the historical non-continuous algorithms, it has been shown that while the incremental (best-first) approach is always less expensive than the non-incremental (depth-first) in terms of node accesses, its actual execution time heavily depends on the used queue length. In general, the best-first approach outperforms its competitor only for point NN queries under small temporal extent (less than 2-4% depending on the index used and under any *k*), while in all other cases the depth first approach takes less time to be executed. This drawback of the incremental algorithms is mainly due to the queue length which may become huge, especially in the case of the TB-tree ant the TB*-tree. Regarding a comparison between the used indexes, the 3D R-tree outperforms the TB-tree in terms of both node accesses and execution time, while the TB*-tree proposed in Chapter 3 is shown that it is not a suitable choice when dealing with NN queries.

4. Advanced Trajectory Query Processing: Similarity Search

The purpose of this chapter is to demonstrate the algorithms for similarity search on R-tree-like structures storing historical trajectories of moving objects. Its structure is as follows: Section 4.1 motivates the chapter and provides the initial ideas. Related work is discussed in Section 4.2. Section 4.3 formally introduces the main purpose of this chapter and thoroughly examines the metrics employed for Most Similar Trajectory (MST) search, as well as the ones used to support our search ordering and pruning strategies. Section and 4.4 constitutes the core of the chapter describing in detail the query processing algorithms to perform MST search over historical trajectory information; the algorithms presented are based on the depth-first and best-first paradigm, employing R-tree-like index structures. Section 4.5 presents the results of our experimental while our conclusions are presented in Section 4.6.

4.1. Introduction

Another interesting query type that is useful in MOD search is derived from the so-called *trajectory similarity* problem, which aims to find 'similar' trajectories of moving objects. To illustrate the problem, consider the following example. Suppose that the metro network of a city has been recently extended, initiating a new transportation line, in view of providing transport services to a major part of the residents of the city suburbs. This metro network extension requires the re-designing of the existing transportation network (buses, tram, trolley-buses, etc.). Experts in the field would be assisted if they could pose queries about the similarity between the trajectories of the existing transport means and the new metro line. As such, they would be able, for example, to change the timetable of a bus line, if it matches in a certain day with the timetable of the new metro line, or even abort it. To handle such queries efficiently, MOD systems should include methods for answering the so-called *Most-Similar-Trajectory* (MST) search also discussed in [The03].

Trajectory similarity search is a relatively new topic in the literature; the majority of the methods proposed so far are based on either the context of time series analysis and the Longest Common Subsequence (LCSS) model [VKG02], or the recently proposed Edit Distance on Real Sequence (EDR) [COO05]. However, all these methods have the main drawback that they either ignore the time dimension of the movement, therefore calculating the spatial (and not the spatio-temporal) similarity between the trajectories, or assume that the trajectories are of the same length and have the same

sampling rate. To exemplify the problem derived when different sampling rates are present, recall Figure 1.4 presenting two trajectories T and Q with their position being sampled in different rates. While Q and T sample their position 4 and 32 times respectively, they have approximately the same length traversing through the same area. Though the two trajectories are obviously similar, methods based on the LCSS or the EDR model cannot detect this kind of similarity since they try to match trajectory sampled positions one by one, which clearly does not happen in the above (real world) example. Moreover, the majority of the proposed approaches exploit specialized index structures in order to prune the search space and retrieve the most similar to a query trajectory.

The challenge thus accepted in this thesis, is to efficiently support the k-MST search in MODs storing historical trajectory information, indexed by R-tree-like structures. The main contributions of this chapter are outlined as follows:

- A dissimilarity metric (*DISSIM*) for the measurement of the spatio-temporal dissimilarity between two trajectories is defined; this metric which can be seen as the average distance between the two trajectories in time is also independently presented and employed in [NP06]. We subsequently propose an efficient approximation method to overcome its costly calculation.
- A set of novel metrics (*MINDISSIM*, *PESDISSIM*, *OPTDISSIM*) along with several associated lemmas are proposed, and subsequently used for pruning purposes by two most similar trajectory search algorithms. Specifically, using these metrics, we propose a depth-first and a best-first query processing algorithm to perform k-MST search on R-tree-like structures storing historical trajectory information.
- We conduct a comprehensive set of experiments over large synthetic and real datasets demonstrating that the algorithms are highly scalable and efficient in terms of node accesses, execution time and pruned space. We further demonstrate that the proposed similarity metric (*DISSIM*) efficiently retrieves spatio-temporally similar trajectories in cases where related work fails.
- Finally, we show how the proposed metrics and heuristics can be employed in the context of density-based trajectory clustering [NP06].

Again, we have to point out that all the proposed algorithms do not require any dedicated index structure and can be directly applied to any member of the R-tree family used to index trajectories, such as the 3D R-tree, the TB-tree [PJT00] and the TB*-tree proposed in this thesis. To the best of our knowledge, the proposal of this thesis is the first that provides techniques for a spatio-temporal index to support traditional range, as well as topological and similarity based queries.

4.2. Related Work

Similarity search has been extensivelly studied in the time series analysis domain. As a measure of approximate matching, Agrawal et al. [AFS93] proposed the utilization of the Discrete Fourier Transformation (DFT). An alternative time series matching technique through dimension reduction was proposed by Chan and Fu [CF99], using the Discrete Wavelet Transformation (DWT). In order to compare sequences with different lengths, Berndt and Clifford [BC96] used the Dynamic Time

Warping (DTW) technique that allowed sequences to be stretched along the time axis to minimize the distance between sequences. Although DTW incurred a heavy computation cost, it was more robust against noise.

In [YAS03] an indexing method for processing shape-based similarity queries for trajectory databases was presented. The proposed method was based on Euclidean Distance. However it could be applied only on trajectories with same lengths being valid during the same time interval. Cai and Ng [CN04] proposed the utilization of Chebyshev polynomials for approximating and indexing trajectories for similarity matching purposes. Still, this method suffered from the requirement that the trajectories should be of the same length (in terms of the number of spatio-temporal points that are composed of).

Vlachos et al. [VGD04] presented a distance measure that allowed to find similar trajectories under translation, scaling and rotational transformations. The first step of their method was the mapping of each trajectory to a trajectory in a rotation invariant space. For the calculation of the distance between two trajectories in the new rotation invariant space, the DTW technique was utilized.

Sakurai et al. [SYF05] proposed an improved version of DTW, the Fast search method for Dynamic Time Warping (FTW), based on a new lower bounding measure for the approximation of the time warping distance. They proved that FTW could prune a significant portion of the search space, leading to a significant reduction of the search cost. Recently, Lin and Su [LS05] have studied the time independent similarity search problem of moving object trajectories. The "one way distance" (OWD) function is introduced for comparing the spatial shapes of trajectories along with appropriate algorithms for computing OWD. Their experimental study shows that the adoption of OWD function outperforms DTW algorithm in terms of precision and performance.

Several approaches are based on the Longest Common Sub Sequence (LCSS) similarity measure. LCSS measure matches two sequences by allowing them to stretch, without rearranging, the sequence of the elements, but allowing some elements to be unmatched (which is the main advantage of the LCSS measure compared with Euclidean Distance and DTW). Therefore, LCSS can efficiently handle outliers and different scaling factors. Vlachos et al. [VKG02] adopted the utilization of the LCSS method. Introducing two similarity measures allowing time stretching and translations respectively, the authors proposed non-metric similarity functions, which were very robust to the presence of noise and provided an intuitive notion of similarity between trajectories by giving more weight to the similar portions of the trajectories. Moreover, an efficient index structure (based on hierarchical clustering) for similarity queries was presented. However, as will be shown in the experimental study, the proposed method suffers when trajectories have different sampling rates.

In [COO05] a distance function, called Edit Distance on Real Sequences (EDR), was introduced. This distance function, based on edit distance, was shown to be more robust than DTW and LCSS over trajectories with noise. The efficiency of this distance function was improved by the application of three pruning strategies, which reduced the respective computational cost in terms of computations between the query and data trajectories without introducing false dismissals. On the other hand, same as LCSS, EDR determines spatial similarity only, ignoring time, while trajectories with different sampling rates cannot be handled efficiently, as it will be shown in the experimental study. Moreover,

both [VKG02] and [COO05] propose the employment of dedicated indexes to prune the search space so as to efficiently support *k*-MST search.

Keogh et al. [KWX+06] presented an algorithm (based on the *LB_Keogh* function introduced in [Keo02]), which dramatically reduced the time complexity of the calculation of the Euclidean Distance measure. This speed up was further achieved by allowing indexing. However, the above algorithm, which was generalized to other distance measures, such as DTW and LCSS, could be applied only to 2D shapes.

Recently, Pelekis et al. [PKM+07], consider the problem of trajectory similarity search through a different perspective. Contrary to other works which make use of generic similarity metrics that virtually ignore the temporal dimension, [PKM+07] introduce a framework consisting of a set of distance operators based on primitive (space and time) as well as derived parameters of trajectories (speed and direction); as a consequence, they define different distance measures for each kind of similarity between trajectories: spatial, temporal, spatio-temporal, speed-pattern and directional similarity. The novelty of the approach is not only to provide qualitatively different means to query for similar trajectories, but also to support trajectory clustering and classification mining tasks, which definitely imply a way to quantify the distance between two trajectories. For each of the proposed distance operators highly parametric algorithms are devised, the efficiency of which is evaluated through an extensive experimental study.

Acknowledging the contributions of the above proposals, in the sequel we propose novel metrics and algorithms for trajectory similarity search on R-tree-like structures.

Table 4.1: Table of notations

Notation	*Description*
D	a trajectory database
O_i	A moving object identifier
T, Q	an indexed and a query trajectory
T_k, Q_k	the k-th line segment of T or Q
x_k, y_k, t_k	the coordinates of trajectory T a timestamp t_k
$Dist_{Q,T}(t)$	function with time of the synchronous Euclidean distance between trajectories Q and T
a, b, c	factors of the $Dist_{Q,T}(t)$ trinomial
$E_{Q,T}$	calculation error of the dissimilarity between trajectories
$Dist$	distance between trajectories
V	relative speed between moving objects
N	R-tree node
$MINDIST(Q,N)$	minimum distance between Q and N
V_{max}	the sum of the maximum speed of indexed trajectories plus the maximum speed of the query trajectory
S_R	the set of line segments already retrieved from the index
S_C	the set of trajectories with line segments already retrieved from the index but not yet fully completed inside the given time period.

4.3. Problem Statement and Metrics for Most Similar Trajectory Search

In this section the notion of Most Similar Trajectory (MST) queries w.r.t. a dissimilarity metric is defined, and then, the notion of spatio-temporal dissimilarity used in the approach of this thesis is formally introduced. Finally, a series of metrics and heuristics for MST Search used in the algorithms presented in this thesis is established. Table 4.1 presents the notations used in the rest of this chapter.

4.3.1. Problem Statement

Let D be a database of N moving objects with objects ids $\{O_1, O_2, \ldots, O_N\}$ assuming linear interpolation between sampled points. The trajectory T of a moving object O_i consists of n-1 3D-line segments $\{T_1, T_2, .. T_{n-1}\}$. Each 3D line segment T_k is of the form $((x_k, y_k, t_k), (x_{k+1}, y_{k+1}, t_{k+1}))$, where $t_0 \leq t_k < t_{k+1} \leq now$. Bearing in mind that many similarity metrics have been proposed in the literature, as discussed in the previous section, the definition of an MST query should be as general as possible. Therefore, we formally define MST search to be independent of the underlying similarity metric:

Definition 4.1: *Given a query trajectory Q, a trajectory database D and a metric DSIM measuring the dissimilarity between two trajectories, a most similar trajectory query is a query*

$$MST(D,Q) = \left(T, DSIM(Q,T)\right) : DSIM(Q,T) \leq DSIM(Q,T') \forall T' \in D \quad (4.1)$$

that searches database D for the trajectory T having the minimum dissimilarity with the query trajectory Q among all trajectories in D, as well as the implied value of dissimilarity.

However, regarding the underlying similarity metric, the majority of existing work in the domain of trajectory similarity search, either ignores the time dimension of the movement, as such calculating the spatial similarity between trajectories or assumes that trajectories have the same lengths (in terms of the number of spatio-temporal points that are composed of) and the same sampling rate. In order to overcome these obstacles, we may generalize the well known Euclidean Distance metric and provide the notion of spatio-temporal *dissimilarity* between two trajectories T and Q both being valid during a definite time interval $[t_1, t_n]$, by integrating their Euclidean distance in time.

Definition 4.2*: The Dissimilarity DISSIM(Q, T) between trajectories Q and T being valid during the period $[t_1, t_n]$ is defined as the definite integral of the function of time of the Euclidean distance between the two trajectories during the same period:*

$$DISSIM(Q,T) = \int_{t_1}^{t_n} Dist_{Q,T}(t)\,dt, \quad (4.2)$$

where DistQ,T(t) is the function of the Euclidean distance between trajectories Q and T with time.

However, since each trajectory is represented by a collection of discrete points where linear interpolation is applied in between, the definition of dissimilarity is transformed to:

$$DISSIM(Q,T) = \sum_{k=1}^{n-1} \int_{t_k}^{t_{k+1}} Dist_{Q,T}(t)\,dt, \quad (4.3)$$

where t_k are the timestamps that objects T and Q recorded their position. Obviously, in real world applications, the sampling rates of trajectories may vary, resulting in trajectories with positions sampled at different timestamps; however, considering two trajectories with this characteristic, the position of the first object at the time instance when the second recorded its position can be approximated by applying linear interpolation.

The Euclidean distance between two points moving with linear functions of time between consecutive timestamps, was determined in Eq.(3.5) and is the square root of a trinomial:

$$Dist_{Q,T}(t) = \sqrt{at^2 + bt + c}\ , \tag{4.4}$$

where a, b, c are the factors of this trinomial (real numbers, $a \geq 0$).

In order to calculate the integral of $Dist_{Q,T}(t)$, we distinguish between the following two cases for the value of the non-negative factor a:

- $a = 0$. As shown in [MB04], it implies that $b = 0$. Hence,

$$\int_{t_k}^{t_{k+1}} Dist_{Q,T}(t)\,dt = \frac{\sqrt{c}}{t_{k+1} - t_k} \tag{4.5}$$

- $a > 0$. According to [MB04]:

$$\int_{t_k}^{t_{k+1}} Dist_{Q,T}(t)\,dt = \left| \frac{2at+b}{4a}\sqrt{at^2+bt+c} - \frac{b^2-4ac}{8a\sqrt{a}} \operatorname{arcsinh}\left(\frac{2at+b}{\sqrt{4ac-b^2}} \right) \right|_{t_i}^{t_{i+1}} \tag{4.6}$$

In order to avoid such a computationally expensive operation, we adopt the utilization of the Trapezoid Rule for the computation of the integral, resulting in the following Lemma.

Lemma 4.1*: The dissimilarity value between two points moving linearly with time can be approximated by the following expression:*

$$DISSIM(Q,T) \approx DISSIM_{approx}(Q,T) = \frac{1}{2}\sum_{k=1}^{n-1}\left(\left(Dist_{Q,T}(t_k) + Dist_{Q,T}(t_{k+1}) \right) \cdot (t_{k+1} - t_k) \right) \tag{4.7}$$

with the error of the approximation, which depends on t_k, t_{k+1} values, being bounded by:

$$E_{Q,T} \leq \sum_{k=1}^{n-1} \begin{cases} \frac{(t_{k+1}-t_k)^3}{12} \left| Dist_{Q,T}^{(2)}\left(-b/2a \right) \right| & ,if\ t_k \leq -b/2a \leq t_{k+1} \\ \frac{(t_{k+1}-t_k)^3}{12} \left| Dist_{Q,T}^{(2)}(t_{k+1}) \right| & ,if\ t_k < t_{k+1} < -b/2a \\ \frac{(t_{k+1}-t_k)^3}{12} \left| Dist_{Q,T}^{(2)}(t_k) \right| & ,if\ -b/2a < t_k < t_{k+1} \end{cases} \tag{4.8}$$

Proof: The Trapezoid approximation $T_n(f)$ of $\int_{x_0}^{x_n} f(x)\,dx$ associated with the partition $x_0 < x_1 < \ldots < x_n$ is given by:

$$T_n(f) = \frac{1}{2}(x_n - x_0) \cdot \left[f(x_0) + 2f(x_1) + \ldots + 2f(x_{n-1}) + f(x_n) \right] \tag{4.9}$$

If $f^{(2)}(x)$ is continuous in $[x_0, x_n]$, then the error $E_n(f)$ in the trapezoid rule is bounded as follows:

$$E_n(f) \leq \frac{(x_n - x_0)^3}{12n^2} \left| f^{(2)}(M) \right|, \tag{4.10}$$

where $f^{(2)}(M)$ is the maximum value of $f^{(2)}(x)$ in $[x_0, x_n]$, i.e.,

$$\left| f^{(2)}(M) \right| \geq \left| f^{(2)}(x) \right| \forall\, x \in [x_0, x_n] \tag{4.11}$$

In our case, by setting $n = 1$, we finally calculate:

$$\int_{t_k}^{t_{k+1}} Dist_{Q,T}(t)\,dt \approx \frac{1}{2}\left(Dist_{Q,T}(t_k) + Dist_{Q,T}(t_{k+1}) \right) \cdot (t_{k+1} - t_k) \tag{4.12}$$

with the error of our approximation being bounded by:

$$E_{Q_k,T_k} \le \frac{(t_{k+1}-t_k)^3}{12}\left|Dist^{(2)}_{Q,T}(M)\right|, \quad (4.13)$$

where $Dist^{(2)}_{Q,T}(M)$ is the maximum value of $Dist^{(2)}_{Q,T}(t)$ in $[t_k, t_{k+1}]$. Therefore, we determine the maximum value of $Dist^{(2)}_{Q,T}(t) = \frac{4ac-b^2}{4(at^2+bt+c)^{3/2}}$ in $[t_k, t_{k+1}]$. Since the first derivative of $Dist^{(2)}_{Q,T}(t)$, $Dist^{(3)}_{Q,T}(t)$ zeroes at $t = -b/2a$ and $D^{(4)}_{Q,T}(-b/2a) = \frac{-3a(4a)^{5/2}}{4(4ac-b^2)^{3/2}} \le 0$ (since $a \ge 0$), the largest value of $Dist^{(2)}_{Q,T}(t)$ in is $Dist^{(2)}_{Q,T}(-b/2a)$. Finally, we distinguish between three cases:

- $t_k \le -b/2a \le t_{k+1}$. In this case, $Dist^{(2)}_{Q,T}(M) = Dist^{(2)}_{Q,T}(-b/2a)$ and the error is $E_{Q_k,T_k} \le \frac{(t_{k+1}-t_k)^3}{12}\left|Dist^{(2)}_{Q,T}(-b/2a)\right|$;
- $t_k < t_{k+1} < b/2a$. In this case, $Dist^{(2)}_{Q,T}(M) = Dist^{(2)}_{Q,T}(t_{k+1})$ and the error is $E_{Q_k,T_k} \le \frac{(t_{k+1}-t_k)^3}{12}\left|Dist^{(2)}_{Q,T}(t_{k+1})\right|$;
- $-b/2a < t_k < t_{k+1}$. In this case, $Dist^{(2)}_{Q,T}(M) = Dist^{(2)}_{Q,T}(t_k)$ and the error is $E_{Q_k,T_k} \le \frac{(t_{k+1}-t_k)^3}{12}\left|Dist^{(2)}_{Q,T}(t)\right|$.

Summing the *n*-1 equations of the dissimilarity error calculation by sides, it implies that the approximation error $E_{Q,T}$ is computed as presented in Lemma 1. ■

Figure 4.1 demonstrates the trapezoid approximation illustrating the approximation error *E* in the three above cases: the value of $-b/2a$ is the flex of $Dist^{(2)}_{Q,T}$; E_k is calculated based on the value of $Dist^{(2)}_{Q,T}(t_{k+1})$ (case b), E_{k+1} is calculated based on the value of $Dist^{(2)}_{Q,T}(-b/2a)$ (case a) and E_{k+2} is calculated based on $Dist^{(2)}_{Q,T}(t_{k+2})$ (case c).

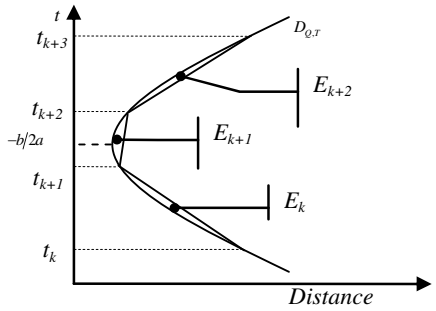


Figure 4.1: Trapezoid approximation

So far we have defined the dissimilarity between two trajectories (Definition 4.2) and have approximated this measure with a less expensive computation and a bounded error. As already mentioned, the location of non-recorded timestamps is approximated by linear interpolation between

consecutive recorder points (Support of non-linear e.g. arc, movement is left as an open issue). In the sequel, we will provide a series of metrics that will be used in our MST search algorithms.

4.3.2. Speed-Dependent Metrics

In this section we define two metrics, namely *OPTDISSIM* and *PESDISSIM*, and provide several lemmas to be used for pruning purposes during MST Search. Before proceeding into the core of the section, we define the *Linearly Depended Dissimilarity* (*LDD*) which is used in the definition of our metrics:

Definition 4.3: *The Linearly Depended Dissimilarity (LDD) between two moving objects with initial distance D moving collinearly with relative speed V during the period* $\Delta t = [t_1, t_n]$*, is given by the following expression:*

$$LDD(D,V,\Delta t) = \begin{cases} \Delta t \cdot \left(D + \frac{V \cdot \Delta t}{2} \right) & ,if\ D + V \cdot \Delta t \geq 0 \\ \frac{D^2}{2|V|} & ,otherwise \end{cases} \quad (4.14)$$

The relative speed *V* is a negative (positive) number when the distance between the two objects decreases (increases, respectively). To illustrate this definition, consider Figure 4.2 where *LDD* is described as the shaded area encompassed by the inclined line representing a distance function between two objects moving towards each other with relative speed *V*, with the horizontal lines t_1 and t_n defining Δt. The two cases of *LDD* definition are illustrated in Figure 4.2(a) and Figure 4.2(b), respectively.

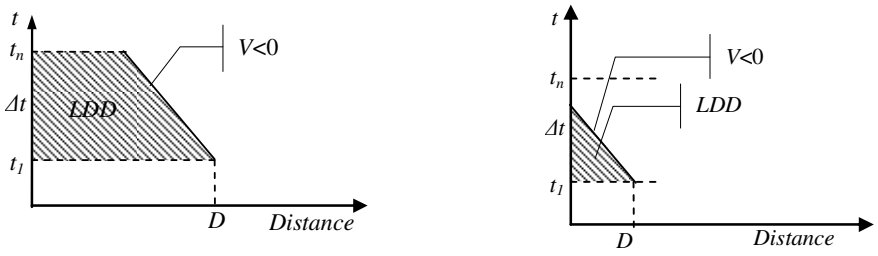


Figure 4.2: *LDD* definition

Having defined *LDD*, we can continue with the definition of the first metric used in our ordering and pruning strategies:

Definition 4.4: *The minimum DISSIM (MINDISSIM) during a period* $\Delta t = [t_1, t_n]$ *between a trajectory indexed by an R-tree-like structure with a line segment lying inside an index node N and a query trajectory Q, is defined as:*

$$MINDISSIM(Q,N,\Delta t) = LDD\left(MINDIST(Q,N), V_{\max}, \Delta t\right) \quad (4.15)$$

where V_{max} *is the sum of (a) the maximum speed of indexed trajectories and (b) the maximum speed of the query trajectory.*

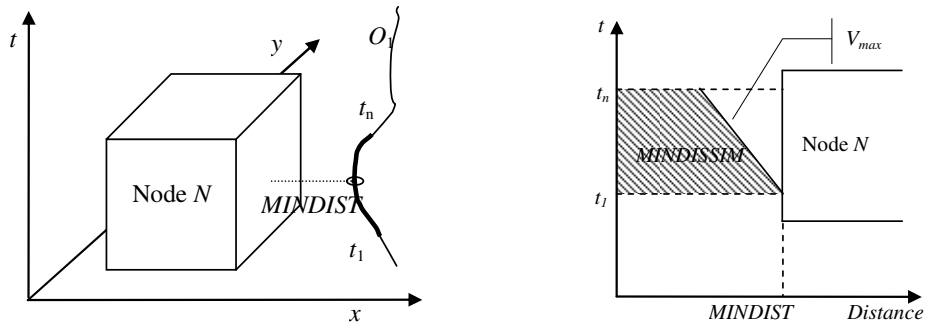


Figure 4.3: *MINDISSIM* definition

This metric can be used for ordering and pruning purposes due to the lemma that follows.

Lemma 4.2: *The DISSIM between a trajectory indexed by an R-tree-like structure partially contained inside an index node N and a query trajectory Q during a period* $[t_1,t_n]$ *cannot have DISSIM smaller than the respective MINDISSIM of the node.*

Proof: According to Definition 4.4, *MINDISSIM* corresponds to the *DISSIM* of a moving object located inside *N* for *a single* time instance and then moved towards the query trajectory with the maximum possible speed (the area of the shaded region in Figure 4.3). Obviously, any other object with at least one line segment contained inside *N* will approach the query trajectory with speed lower or equal than V_{max}, increasing therefore the shaded trapezoid area of Figure 4.3 (i.e. the slope of the inclined line in Figure 4.3(b) will be greater). Furthermore, if the object remains inside *N* for more than one time instance, the inclined line would cover a part of the query interval (and not the whole), leading to a region with greater area. ■

Depending on the presence or absence of index, any algorithm used for MST or similarity range search will have to calculate the dissimilarity between a query trajectory and several (indexed or not) trajectories; obviously, at any time instance such an algorithm will have retrieved several parts of candidate MSTs.

Although we cannot calculate the exact *DISSIM* of these partially retrieved trajectories from the query trajectory, we can safely estimate a lower bound for it, called *OPTDISSIM*. Consider, for example, Figure 4.4 that illustrates *OPTDISSIM* of a partially retrieved candidate trajectory *T* from the query trajectory *Q*. *OPTDISSIM* partially consists of the dissimilarity of the entries already retrieved from the index (the shaded area during the time intervals $[t_1,t_2]$ and $[t_3,t_4]$). Regarding the period $[t_4,t_5]$, the smallest possible dissimilarity is given assuming that the moving object started from its position at t_4 approaching the query object with the maximum possible speed (the inclined line between t_4 and t_5). Finally, when dealing with intermediate time intervals such as $[t_2,t_3]$, one has to calculate the time instance t_2^o in which the object stopped its movement towards the query trajectory (the inclined line between t_2 and t_2^o) and then returned to its known position at the time instance t_3 (the inclined line between t_2^o and t_3). Now we can proceed with the formal definition of *OPTDISSIM*:

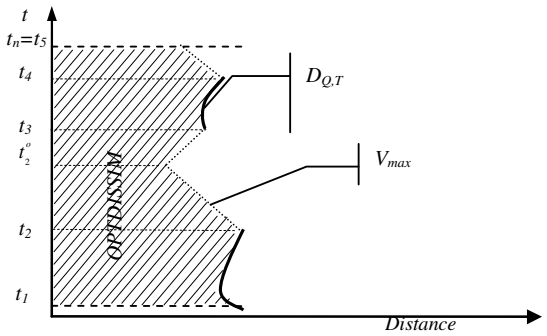


Figure 4.4: *OPTDISSIM* definition

Definition 4.5: *The most optimistic DISSIM (OPTDISSIM) between a query trajectory Q and an indexed trajectory T with line segments partially retrieved from the index, during a period* $[t_1, t_n]$*, is defined as:*

$$OPTDISSIM(Q,T,t_1,t_n) = \sum_{k=1}^{n-1} \begin{cases} DISSIM(Q_k,T_k) & ,if\ T_k \in S_R; \\ LDD\left(Dist_{Q,T}(t_{k+1}), -V_{\max}, (t_{k+1}-t_k)\right) & ,if\ T_k \notin S_R, k=1; \\ LDD\left(Dist_{Q,T}(t_k), -V_{\max}, (t_{k+1}-t_k)\right) & ,if\ T_k \notin S_R, k=n-1; \\ LDD\left(Dist_{Q,T}(t_k), -V_{\max}, (t_k^o-t_k)\right) + & \\ LDD\left(Dist_{Q,T}(t_{k+1}), V_{\max}, (t_{k+1}-t_k^o)\right) & ,otherwise \end{cases} \quad (4.16)$$

where $Dist_{Q,T}$ *is the function of distance with time between trajectories Q and T,* S_R *is the set of line segments already retrieved from the index,* V_{max} *is the sum of the maximum speed of indexed trajectories plus the maximum speed of the query trajectory, and* t_k^o *is given by the following expression:*

$$t_k^o = \frac{1}{2}\left(t_k + t_{k+1} + \frac{\left(D_{Q,T}(t_{k+1}) - D_{Q,T}(t_k)\right)}{V_{\max}}\right) \quad (4.17)$$

Recalling Figure 4.4, the value of t_k^o is straightforward utilizing the fact that the slope of the two inclined lines between $[t_2, t_2^o]$ and $[t_2^o, t_3]$ is the same and equal to V_{max}. Having defined *OPTDISSIM*, we can provide the following lemma, which will also turn out to be useful for pruning purposes:

Lemma 4.3: *A trajectory indexed by an R-tree-like structure with line segments partially retrieved from the index cannot have smaller DISSIM from a query trajectory Q during a period* $[t_1, t_n]$ *than its respective OPTDISSIM.*

Proof: According to the previous definition, *OPTDISSIM* is the sum of the *DISSIM* of the trajectory entries already retrieved from the index (belonging to set S_R), a value which is fixed, plus the *DISSIM* of an object which approached the query trajectory with the maximum possible speed (V_{max}) during the time intervals not already retrieved from the index, with the additional constraint that the object has to be found at given positions at the start and/or the end of the interval. Therefore, since the two objects approach each other with the maximum possible speed during those periods, the distance between them is minimized; hence minimizing the corresponding integral and consequently their dissimilarity. ■

Likewise, by adopting the same scenario where an MST algorithm has only partially retrieved trajectories, one can estimate an upper bound, for the *DISSIM* between the query and a partially

retrieved trajectory, named *PESDISSIM*. As illustrated in Figure 4.5, *PESDISSIM* works in a fashion similar to *OPTDISSIM* with the difference that during time intervals where the movement of the object is not known, the object is assumed to diverge (and not to approach) the query trajectory with the maximum possible speed V_{max}. In the same way, we formally define *PESDISSIM*:

Definition 4.6: *The most pessimistic DISSIM (PESDISSIM) between a query trajectory Q and an indexed trajectory T with line segments partially retrieved from the index, during a period* $[t_1, t_n]$*, is defined as:*

$$PESDISSIM(Q,T,t_1,t_n) = \sum_{k=1}^{n-1} \begin{cases} DISSIM(Q_k, T_k) & ,if\ T_k \in S_R; \\ LDD\left(D_{Q,T}(t_{k+1}), V_{\max}, (t_{k+1} - t_k)\right) & ,if\ T_k \notin S_R, k = 1; \\ LDD\left(D_{Q,T}(t_k), V_{\max}, (t_{k+1} - t_k)\right) & ,if\ T_k \notin S_R, k = n-1; \\ LDD\left(D_{Q,T}(t_k), V_{\max}, (t_k^p - t_k)\right) + & \\ LDD\left(D_{Q,T}(t_{k+1}), -V_{\max}, (t_{k+1} - t_k^p)\right) & ,otherwise \end{cases} \quad (4.18)$$

where $D_{Q,T}$*,* S_R *and* V_{max} *are as defined in previous definitions, and* t_k^p *is given by the following expression:*

$$t_k^p = \frac{1}{2}\left(t_k + t_{k+1} + \frac{\left(D_{Q,T}(t_k) - D_{Q,T}(t_{k+1})\right)}{V_{\max}}\right) \quad (4.19)$$

The following lemma is directly derived by the definition of PESDISSIM.

Lemma 4.4: *A trajectory indexed by an R-tree-like structure with line segments partially retrieved from the index cannot have DISSIM from a query trajectory Q during a period* $[t_1, t_n]$ *greater than its respective PESDISSIM.*

Proof: According to the previous definition, *PESDISSIM* is the sum of the *DISSIM* of the trajectory entries already retrieved from the index (belonging to set S_R), a value which is fixed, plus the *DISSIM* of an object which diverged the query trajectory with the maximum possible speed (V_{max}) during the time intervals not already retrieved from the index, with the additional constraint that the object has to be found in given positions at the start and/or the end of the interval. Therefore, the distance between the two trajectories during those periods is maximized, hence maximizing their dissimilarity. ■

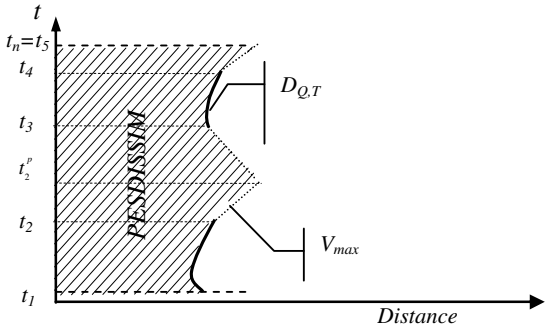


Figure 4.5: *PESDISSIM* definition

4.3.3. Speed-Independent Metrics

The utilization of the previously defined metrics in an MST search algorithm can significantly enhance its performance by pruning several candidate trajectories. However, these metrics are relatively loose,

since they are based on the maximum speed V_{max} which, theoretically speaking, could be orders of magnitude higher than the mean object speed. Therefore, we need to define other metrics not influenced by V_{max}, supporting our speed-independent MST search algorithms. These metrics can be developed when an MST algorithm reports index nodes in incremental order of their *MINDIST* from the query trajectory. Obviously, this is a reasonable assumption considering R-tree like structures where a best-first strategy like the one proposed in [HS99] can be utilized.

Consider, for example, Figure 4.6 that describes the *DISSIM* of a partially retrieved candidate trajectory T from the query trajectory Q; According to our previous discussion, the *DISSIM* between $[t_1,t_2]$ and $[t_3,t_4]$ is accurately defined. In this case however, we can utilize the fact that index nodes are accessed in incremental order of their *MINDIST* from the query trajectory. Consequently, any line segment not yet retrieved from the index, cannot be closer to Q than *MINDIST*(Q,N) where N is the next index node in the queue, and the lower bound of *DISSIM* turns into the shaded area of Figure 6.

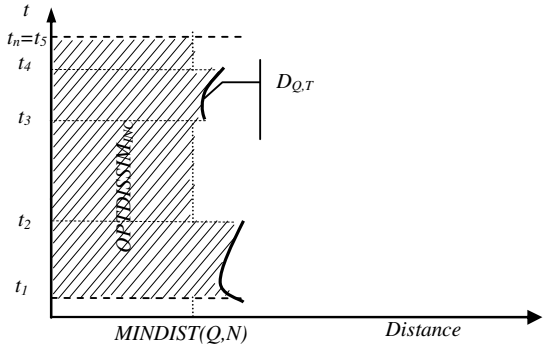


Figure 4.6: $OPTDISSIM_{INC}$ definition

More formally, we define $OPTDISSIM_{INC}$ as follows:

Definition 4.7: *Assuming that index nodes are reported in incremental order of their MINDIST from the query trajectory, the most optimistic DISSIM between a query trajectory Q and an indexed trajectory T during a period $[t_1,t_n]$ having a line segment inside a tree node N, is given by the following expression:*

$$OPTDISSIM_{INC}(Q,T,N,t_1,t_n) = \sum_{k=1}^{n-1} \begin{cases} DISSIM(Q_k,T_k) & , \text{ if } T_k \in S_R; \\ MINDIST(N,T)\cdot(t_{k+1}-t_k) & , \text{ otherwise} \end{cases} \quad (4.20)$$

where S_R is the set of line segments already retrieved from the index.

Using the above definition of $OPTDISSIM_{INC}$, we can also define the minimum *DISSIM* of an index node N:

Definition 4.8: *Assuming that index nodes are reported in incremental order of their MINDIST from the query trajectory, the minimum DISSIM between a trajectory T, indexed by an R-tree-like structure having a line segment inside a node N, and a query trajectory Q during a period $[t_1,t_n]$, is defined as:*

$$MINDISSIM_{INC}(Q,N,t_1,t_n) = \min \begin{cases} MINDIST(Q,N)\cdot(t_n-t_1) \\ OPTDISSIM_{INC}(Q,T,N,t_1,t_n), \quad \forall T \in S_C \end{cases} \quad (4.21)$$

where S_C, is the set of the trajectories with line segments already retrieved from the index but not yet fully completed inside the period $[t_1,t_n]$.

Lemma 4.5: *Assuming that index nodes are reported in incremental order of their MINDIST from a query trajectory Q, a trajectory that is partially stored inside a tree node N cannot have smaller DISSIM from Q during the time period* $[t_1,t_n]$ *than the node's respective* $MINDISSIM_{INC}$.

Proof: Any line segment inside *N* resides in a trajectory that either belongs to S_c or not. In the former case, considering that nodes are reported in incremental order, trajectory entries not yet retrieved cannot be closer to the query object than the *MINDIST* of the node in which they belong. So, the minimum dissimilarity of an object of S_c is the sum of the dissimilarity of its entries already retrieved from the index, plus the dissimilarity of an object being as close as *MINDIST* to the query trajectory during the rest of the query time period - a sum which corresponds to $OPTDISSIM_{INC}$ definition. In the latter case, where the trajectory does not belong to S_c, the line segment cannot belong to an object fully retrieved from the index because this would lead to duplicate line segments in the index. Hence the line segment belongs to a moving object with no segments retrieved from the previously accessed nodes and it cannot be closer to the query trajectory than *MINDIST*. Thus, in the best case, its distance from the query object during the query period is equal to *MINDIST* and its *DISSIM* is equal to $MINDIST(Q,N)\cdot\Delta t$. ■

4.3.4. Heuristics

The lemmas provided in previous sections support the following heuristics directly used in the MST Search algorithms that will be presented in the following Sections.

- **Heuristic 1**: Every trajectory line segment contained in an R-tree-like node with *MINDISSIM* greater than the current most similar (i.e. the one with the smallest calculated *DISSIM* - or *PESDISSIM* if there is not a fully calculated *DISSIM*) belongs to a moving object which cannot be more similar to the query trajectory than the current most similar; as such it can be pruned from the candidates list.
- **Heuristic 2**: Every trajectory with *OPTDISSIM* less than the current most similar cannot be more similar to the query trajectory than the current most similar; as such it can be pruned from the candidates list.
- **Heuristic 3**: When leaf and internal nodes are reported in incremental order of their *MINDIST* from the query trajectory, every trajectory line segment contained in a node with $MINDISSIM_{INC}$ greater than the current most similar belongs to a moving object which cannot be more similar to the query trajectory, hence, the node can be pruned from the candidates list. Moreover, since any node reported after the one processed will have *MINDIST* greater or equal to *MINDIST* of the current node, according to Definition 4.8 the same will hold for the respective values of $MINDISSIM_{INC}$. As a result, all these nodes will have $MINDISSIM_{INC}$ greater than the current most similar, and the algorithm can be terminated since all the remaining nodes can be pruned.

4.4. Algorithms for *k*-Most Similar Trajectory Search

In this section we describe in detail the algorithms answering MST queries using the three heuristics described in the previous section and, then, we generalize them in order to support *k*-MST queries. We

provide two alternatives: one depth-first and one best-first where the second assumes that index nodes are reported in incremental order of their *MINDIST*.

4.4.1. Depth-First MST Search Algorithm

The first algorithm (`DFMSTSearch`, illustrated in Figure 4.7) accesses the tree structure in a depth-first mode, pruning tree nodes not fulfilling the temporal constraint of the query trajectory, as also shown in subsection 3.4.1.1 regarding trajectory nearest queries. The algorithm starts by interpolating to produce the part of the query trajectory being entirely inside the query temporal extent. Generally speaking, the algorithm uses the *DISSIM* between trajectories as distance metric and not the Euclidean Distance; at higher levels the *MINDISSIM* metric is used to sort the branch list and prune it using heuristic 1 during the algorithm's backtracking (lines 31-36). At leaf level, the algorithm uses three hashed in-memory structures: One with the completed trajectories (*Completed*), one with the partially completed trajectories (*Valid*) and one with the partially completed nevertheless already rejected (*Rejected*) trajectories. The *Completed* and *Valid* in-memory structures store *lists* each one of them containing the moving object's time intervals along with their starting and ending distances, its (partial) *DISSIM* the respective calculation error and the *OPTDISSIM* and *PESDISSIM* values. The *Rejected* in-memory structure contains only trajectory *id*s.

When a leaf entry is processed, the algorithm checks whether it belongs to a *Rejected* moving object (by simply using its id) and rejects it if it does (line 6). In the sequel it checks whether the entry belongs to a *Valid* moving object and, if so, retrieves its list *L*; otherwise, it creates a new list and adds it to *Valid* (lines 7-11). The algorithm uses a plane sweep method which scans leaf entries and trajectory segments in their temporal dimension in a single pass. This requires that the leaf entries are previously sorted according to their temporal order (line 4), unless the underlying tree structure (such as the TB-tree) stores them in temporal organization anyway.

When a leaf entry and the query trajectory overlap in the temporal dimension the algorithm adds the period to the list *L* (lines 14-15), and calculates *DISSIM*, *OPTDISSIM* and *PESDISSIM,* together with the respective calculation error (line 16). If the list *L* is completed, it is removed from *Valid* and added to *Completed*, its *DISSIM* is checked against the current most similar and, if smaller, takes its position in *MSim* (lines 18-19). In the case where *L* is not yet completed, its *PESDISSIM* is checked against the current most similar and, if smaller, takes its position in *MSim* (lines 21-24); its *OPTDISSIM* is also compared with the current most similar and, if greater, the list is moved from *Valid* to *Rejected* applying heuristic 2 (lines 24-26).

```
 1.  Algorithm DFMSTSearch(node N, trajectory Q, time period Q_per, struct
                           MSim, Hash Valid, Hash Completed, Hash Rejected)
 2.      Q = Interpolate(Q, Max(Q.T_S, Q_per.T_S), Min(Q.T_E, Q_per.T_E))
 3.      IF N is leaf
 4.         Sort(N,TS)
 5.         FOR EACH leaf entry E IN leaf node N
 6.            IF Rejected not contains E.Id
 7.               IF Valid contains E.Id
 8.                  retrieve list L
 9.               ELSE
10.                  create list L; Add L in Valid
11.               ENDIF
12.               Find next query entry QS with QS.T_e<N.T_S; QE=QS
13.               DO UNTIL QE.T_s > E.T_e
14.                  Interpolate to produce nE, nQE in period (T1,T2)
15.                  Add (T1,T2) in L
16.                  Calc(DISSIM,PESDISSIM,OPTDISSIM,ERR)
17.                  IF L is completed
18.                     Move L from Valid to Completed
19.                     IF DISSIM<MSim.DISSIM Update MSim with nE.Id,DISSIM
20.                  ELSE
21.                     IF PESDISSIM<MSim.DISSIM
22.                        Update MSim with nE.Id,PESDISSIM
23.                     ENDIF
24.                     IF OPTDISSIM>MSim.DISSIM
25.                        Move L from Valid to Rejected
26.                     ENDIF
27.                  ENDIF
28.            NEXT query entry QE
29.            Return QE in the query entry QS
30.         NEXT
31.      ELSE
32.         BranchList=GenTrajectoryBranchList(Q,N)
33.         SortBranchList(BranchList)
34.         FOR EACH entry E IN BranchList
35.            DFMSTSearch E.ChNode, E.Trajectory, MSim
36.            PruneBranchList(BranchList)
37.         NEXT
38.      ENDIF
```

Figure 4.7: Depth-first most similar trajectory search algorithm (DFMSTSearch algorithm)

4.4.2. Best-First MST Search Algorithm

The second algorithm (BFMSTSearch, illustrated in Figure 4.8) accesses the tree structure in a best-first mode, calculating the appropriate *MINDIST*s between the query trajectory and the tree nodes, thus reporting leaf and internal tree nodes in incremental order of their *MINDIST* from the query trajectory.

Again, the algorithm starts by interpolating to produce the part of the query trajectory being entirely inside the query temporal extent. In the sequel, when an internal node is processed (lines 35-39), the algorithm calculates the *MINDIST* between the node and the part of the query trajectory *Q* being inside the temporal extent of the node utilizing the *MinDist_Trajectory_Rectangle* metric (also employed in our trajectory nearest neighbor algorithms) and then is enqueued. When a leaf is processed (lines 9-30), the algorithm processes entries with exactly the same way as the DFMSTSearch algorithm does. In both cases where a node (leaf or internal) is processed, the algorithm first checks whether its $MINDISSIM_{INC}$ is greater than the current most similar and if so, the algorithm terminates applying heuristic 3, and returns the current most similar as the query reply (lines 5-7). Note that in order to avoid calculating all the $OPTDISSIM_{INC}$ values involving in the $MINDISSIM_{INC}$ definition (i.e., $T \in S_C$ in definition 6), we first check whether the $MINDIST(Q,N)\cdot(t_n - t_1)$ value of the node is less than the current most similar. In such a case, the calculation of the $OPTDISSIM_{INC}$ values is omitted,

since the value of $MINDISSIM_{INC}$ will be less than the current most similar regardless of the $OPTDISSIM_{INC}$ values.

```
 1.  Algorithm BFMSTSearch (R-tree R, trajectory Q, period Q_per)
 2.     Q = Interpolate(Q, Max(Q.T_S, Q_per.T_S), Min(Q.T_E, Q_per.T_E))
 3.     EnQueue Queue, R.RootNode, 0, Q
 4.     DO WHILE Queue.Count > 0
 5.        Element = DeQueue(Queue); N=Element.Node; Q=Element.QueryTrajectory
 6.        IF Completed.Count>0
 7.           IF MINDISSIM_INC(Q,N)>MSim.DISSIM RETURN MSim
 8.        ELSE
 9.           IF N is leaf node
10.              Sort(N, TS)
11.              FOR EACH leaf entry E IN leaf node N
12.                 IF Rejected not contains E.Id
13.                    IF Valid contains E.Id
14.                       retrieve list L
15.                    ELSE
16.                       create list L; Add L in Valid
17.                    ENDIF
18.                    Find next query entry QS in Q with QS.T_e<N.T_S; QE=QS
19.                    DO UNTIL QE.T_S > E.T_e
20.                       Interpolate to produce nE, nQE period (T1,T2)
21.                       Add (T1,T2) in L
22.                       Calc DISSIM, PESDISSIM, OPTDISSIM, ERR
23.                       IF L is completed
24.                          Move L from Valid to Completed
25.                       ENDIF
26.                       IF DISSIM<MSim.DISSIM
27.                          Update Msim with nE, DISSIM
28.                       ELSE
29.                          IF PESDISSIM<MSim.DISSIM
30.                             Update MSim with nE, PESDISSIM
31.                          ENDIF
32.                          IF OPTDISSIM>MSim.DISSIM
33.                             Move L from Valid to Rejected
34.                          ENDIF
35.                       ENDIF
36.                    NEXT query entry QE
37.                 ENDIF
38.                 Return QE in the query entry QS
39.              NEXT
40.           ELSE
41.              FOR EACH entry E IN the node Element
42.                 IF (Q.T_S,Q.T_E) Overlaps (E.T_S,E.T_E)
43.                    Interpolate to produce nQE in period (T1,T2)
44.                    Dist = MinDist_Trajectory_Rectangle(nQ, E)
45.                    EnQueue Queue, E, Dist, nQ
46.                 ENDIF
47.              NEXT
48.           ENDIF
49.        ENDIF
50.     LOOP
```

Figure 4.8. Best-first most similar trajectory search algorithm (`BFMSTSearch` algorithm)

4.4.3. Extending to *k*-MST algorithms

In the same fashion as in [RKV95] and our work in nearest neighbor queries presented in the previous chapter, we generalize the above two algorithms to support the *k*-most similar trajectory search by considering the following:

- using a buffer of at most *k* (current) most similar trajectories sorted by their actual dissimilarity from the query trajectory;
- either pruning according to the dissimilarity of the more dissimilar object in the buffer, when extending the `DFMSTSearch` algorithm, or

- terminating the algorithm execution when processing a node with $MINDISSIM_{INC}$ greater than the dissimilarity of the more dissimilar object in the buffer, when extending the `BFMSTSearch` algorithm.

4.4.4. Error Management

Both algorithms calculate the dissimilarities between query and indexed trajectories using the approximation introduced in Lemma 4.1 computing at the same time the appropriate approximation error (denoted as *ERR* in both Figure 4.7 and Figure 4.8). However, apart from its computation, the usage of the error is fundamental in order to compute exact and correct results, a task which is not explicitly described in the two algorithms for sake of clarity. Actually, three modifications must be introduced in both algorithms so as to incorporate the role of the approximation error:

- A candidate most similar trajectory, not already completed, is compared against the current *k*-th most similar by using the value of *PESDISSIM-ERR*.
- A completed candidate most similar trajectory is compared against the current *k*-th most similar using the value *DISSIM-ERR*.
- Instead of using one *k*-th most similar, it is required to utilize a buffer of the candidate *k*-th most similar trajectories. These will be all the trajectories with *DISSIM* greater than the *k*-th most similar and *DISSIM-ERR* less than it.

According to the previous discussion, both algorithms may end up with $m \geq k$ candidate most similar trajectories; in such cases, a post processing step is required after their execution in order to determine the definite *k* most similar trajectories by calculating the actual dissimilarity of each candidate trajectory against the query trajectory. Although, this is a computational expensive operation, it only happens when the error buffer contains more than one trajectory, or when the order in which the trajectories are reported from the *k*-buffer can be affected by the calculation error of each trajectory's similarity.

4.5. Experimental Study

The two previously presented `DFMSTSearch` and `BFMSTSearch` algorithms can be implemented in any R-tree-like structure storing historical moving object information such as the 3D R-tree, the STR-tree [PJT00], the TB-tree [PJT00] and the TB*-tree proposed in Chapter 2. Among them, we implemented the algorithms using the 3D R-tree, the TB-tree and TB*-tree.

4.5.1. Experimental Setup

During the experiments, we used a page size of 4KB and a (variable size) buffer fitting the 10% of the index size, with a maximum capacity of 1000 pages. The experiments were performed in a PC running Microsoft Windows XP with AMD Athlon 64 3GHz processor, 512 MB RAM and several GB of disk space.

Regarding the datasets that were employed for the purpose of this study, the real datasets used by related work on trajectory similarity ([COO05], [VKG02]), are not suitable for our objectives due to the fact that they are composed by 2D projections of trajectories without any information about the sampled timestamps; a reasonable fact, bearing in mind that the similarity measured in those papers

only depends on the spatial and not the spatio-temporal trajectory similarity. For this reason we employed the *Trucks* dataset (cf. Section 0) using it so as to evaluate the quality of the proposed similarity measure (section 4.5.2). However, since this dataset is relatively small (273 trajectories and 112203 line segments), it could not expose the actual performance of the algorithms; therefore, the performance study (section 4.5.3) was conducted using synthetic datasets generated by a custom generator based on the GSTD data generator [TSN99]. The main purpose of using a custom data generator and not the widely used GSTD, is that a fundamental parameter influencing the performance of the proposed algorithms is the relation between the mean and the maximum speed of the moving objects indexed by the tree, which cannot be controlled by GSTD.

Table 4.2: Summary dataset information

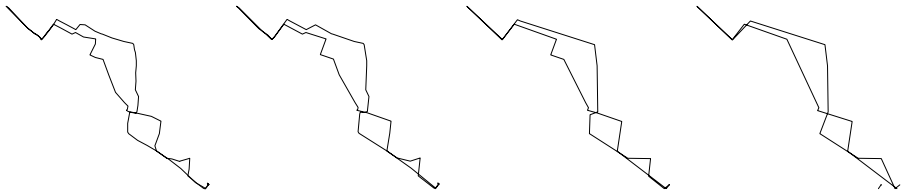
Dataset	*# Objects*	*MMS*	*# Entries (x1K)*	*Speed Distribution*			*Index Size (MB)*		
				Type	*μ*	*σ*	*3D R- tree*	*TB- tree*	*TB*- tree*
Trucks	*276*	*16*	*112*	*Real data*			*3.2*	*1.8*	*1.0*
$S_{0100,10}$	*100*	*10*	*200*	*Lognormal*	*1*	*0.6*	*10.7*	*5.2*	*2.4*
$S_{0250,10}$	*250*	*10*	*500*	*Lognormal*	*1*	*0.6*	*25.8*	*13.1*	*6.1*
$S_{0500,10}$	*500*	*10*	*1000*	*Lognormal*	*1*	*0.6*	*51.0*	*26.2*	*12.2*
$S_{1000,10}$	*1000*	*10*	*2000*	*Lognormal*	*1*	*0.6*	*99.1*	*52.4*	*24.5*
$S_{0500,2}$	*500*	*2*	*1000*	*Normal*	*3*	*1.0*	*51.4*	*26.2*	*12.2*
$S_{0500,5}$	*500*	*5*	*1000*	*Lognormal*	*1*	*0.4*	*51.1*	*26.2*	*12.2*
$S_{0500,20}$	*500*	*20*	*1000*	*Lognormal*	*1*	*0.8*	*50.3*	*26.2*	*12.2*

In order to achieve scalability in the volumes of the datasets, we generated synthetic trajectories of 100, 250, 500 and 1000 moving objects resulting in datasets of 200K, 500K, 1000K, and 2000K entries, respectively (the position of each object was sampled approximately 2000 times), thus building indices of up to 100 MB size. The max / mean speed ratio (denoted in the rest of the chapter as *MMS*) of those datasets was set to a default of 10, which is a reasonable value considering real world applications where trajectories represent walking humans or moving vehicles. Nevertheless, in order to investigate the sensitivity of the algorithms regarding this parameter, we generated 3 additional sets of 500 moving objects setting *MMS* to 2, 5 and 20, respectively. Regarding the rest parameters of the generator, the initial distribution and the heading of objects in all cases was random, while their speed was ruled by a normal or lognormal distribution depending on the desired *MMS*. Table 4.2 illustrates summary information about the real and the generated datasets and the corresponding indexes. Note that each synthetic dataset is denoted by its cardinality and its *MMS* (e.g. the $S_{0100,10}$ constitutes from 100 trajectories with *MMS* equal to 10).

4.5.2. Experiments on the Quality

In order to evaluate the quality of the proposed similarity measure we conducted an extensive set of experiments using the real *Trucks* dataset. All trajectories of the dataset were compressed using the TD-TR algorithm described in [MB04] thus producing artificial trajectories, which were similar (but not identical) to the ones of the original dataset. Then, we used each compressed trajectory to query the

original dataset, expecting the algorithm to return the corresponding original trajectory as most similar. We run one set of queries setting k=1 and we counted the number of times the query failed to return the original trajectory as the most similar. We also scaled the value of the TD-TR parameter p from 0.1% to 10% of the length of each trajectory, in order to achieve different values of similarity since an increasing TD-TR parameter produces a compressed trajectory with fewer sampled points and greater dissimilarity regarding the original trajectory. As an example, Figure 4.9 illustrates (a) an original trajectory and the trajectories produced using the TD-TR algorithm with (b, c, d) different values of p. A major observation derived from Figure 4.9 is that while the general sketch of the trajectory remains unaffected with the evolution of p, the number of vertices outlining the trajectory decreases and the local details are vanished.



p=0 *(168 Vertices)* (a) | *p=0.1 %* *(65 Vertices)* (b) | *p=1 %* *(29 Vertices)* (c) | *p=2 %* *(22 Vertices)* (d)

Figure 4.9: Different degree of compression on a trajectory

Among the related work we have chosen to run the same experiments using the LCSS [VKG02] and EDR [COO05] similarity measures. We did not include DTW [BC96] in our experimental study, since both LCSS and EDR were shown to outperform it [VKG02], [COO05]. We set the value of the parameter ε for these two measures to be a quarter of the maximum standard deviation of trajectories, which leads to the best clustering results, according to [COO05]. We also normalized the trajectory dataset as suggested in the same paper. Furthermore, for a fair comparison, we made an obvious improvement over LCSS and EDR, by manually adding samples in the under-sampled (query) trajectory with linear interpolation at the timestamps the checked dataset trajectory was sampled. We called these improved versions LCSS-I and EDR-I respectively.

The results of the experiments evaluating the quality of the proposed similarity metric are illustrated in Figure 4.10. Clearly, the proposed dissimilarity measure (*DISSIM*) outperforms both its competitors in all settings, regarding also their improved versions. Actually, in the largest part of the experiments, *DISSIM* correctly identifies the original trajectory from which the query one has been produced. On the other hand, it produces false responses only when the value of p exceeds 5%, verifying that it is a very robust similarity metric. LCSS (and LCSS-I) also achieves good quality classifying correctly the query trajectory in the majority of the experimental settings; nevertheless, it is always less accurate than *DISSIM*. Regarding EDR and EDR-I, it turns out that for p values greater than 1% they completely fail to describe the similarity between trajectories, since the false responses exceed 60%.

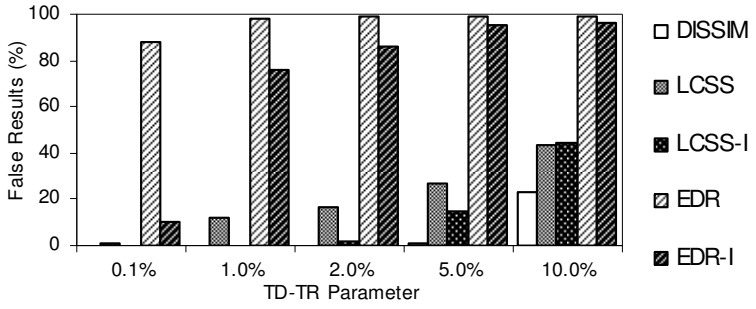


Figure 4.10: False results increasing the value the TD-TR parameter

The reason for the poor performance of EDR similarity measure demonstrated in these experiments can be explained considering its definition: EDR is the number of insert, delete, or replace operations that are needed to convert trajectory A into B [COO05]. Thus, supposing that n is the number of vertices in A and m is the number of vertices in (the compressed) A_c, $EDR(A, A_c) \geq n - m$ since at least n-m vertices are needed to be added into A_c so as to convert it to A. For an arbitrary dataset trajectory T with k vertices being spatially away from A, it can be easily shown that EDR between T and A_c is at most max(m, k). Therefore, if a dataset contains a trajectory T with k vertices and $\max(m,k) \leq n - m$, e.g. a trajectory composed by a small number of vertices, then it also holds that $EDR(T, A_c) \leq EDR(A, A_c)$.

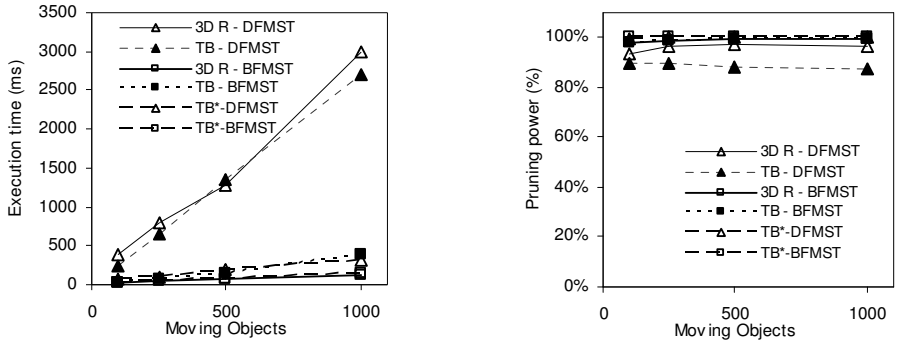
4.5.3. Experiments on the Performance

Both algorithms were evaluated with four sets of 500 queries according to the settings presented in Table 4.3. As such the effects of cardinality (Q_1), *MMS* (Q_2), query length (Q_3) and k (Q_4) were evaluated using the 3D R-, the TB- and the TB*-tree.

Table 4.3: Query Settings

Query Set	*Datasets*	*Query Trajectory (as part of a random data trajectory)*	*k*
Q_1	$S_{0100,10} \ldots S_{1000,10}$	*5%*	*1*
Q_2	$S_{0500,02} \ldots S_{0500,20}$	*5%*	*1*
Q_3	$S_{0500,10}$	*1% … 100%*	*1*
Q_4	$S_{0500,10}$	*5%*	*1..10*

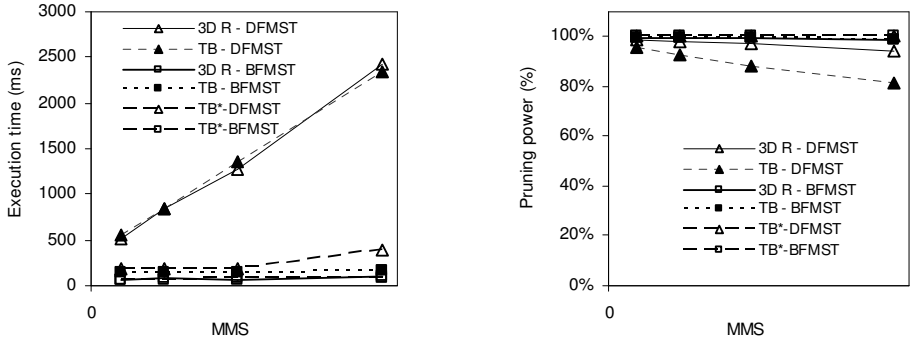
Figure 4.11 illustrates the execution time and the achieved pruned space for the query set Q_1 (scaling with the dataset cardinality) evaluating the `DFMSTSearch` and `BFMSTSearch` algorithms. It is clear that, `BFMSTSearch` outperforms `DFMSTSearch` algorithm in terms of execution time, while both demonstrate a very good pruning power (over 80% in all the experiments). The reason for the slightly worse pruning power of `DFMSTSearch` algorithm is due to the influence of the parameter V_{max} in the definition of *MINDISSIM*, *OPTDISSIM* and *PESDISSIM* metrics, which leads to relatively "loose" heuristics. As the number of moving objects increases, V_{max} becomes several times greater than the speed of the majority of moving objects, reducing the efficiency of the above metrics and consequently the performance of the MST algorithm.



(a) (b)

Figure 4.11: Scaling with the dataset cardinality (Q1)

Another observation obtained from Figure 4.11 is that while the `DFMSTSearch` algorithm shows good pruning power, its execution time does not follow a similar behavior. This can be explained by the fact that the pruning in the MST algorithm is mainly due to the *OPTDISSIM* heuristic, which requires the algorithm to read leaf entries and reject them without processing them (if their *ids* belong to a *Rejected* moving object). On the other hand, `BFMSTSearch` algorithm mainly prunes by the $MINDISSIM_{INC}$ heuristic which directly rejects all tree nodes not yet processed by the time it realizes.



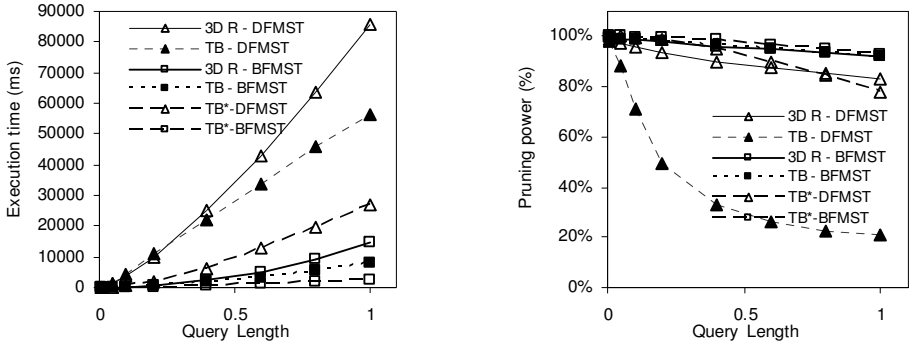
(a) (b)

Figure 4.12: Scaling with the *MMS* (Q2)

The influence of the V_{max} parameter is highlighted in the second set of experiments with the query set Q_2, scaling with the value of *MMS* (Figure 4.12). As it can be observed, the execution time of `DFMSTSearch` algorithm increases linearly with the value of *MMS*. The execution time of `BFMSTSearch` algorithm remains constant, since it does not utilize objects speed. Again, both algorithms achieve a very good pruning of the searched space.

Similar conclusions as the above are drawn from for the query set Q_3, scaling with the query length (Figure 4.13). In this case it is worth to point out the "bad" behavior of the TB-tree regarding its pruning power, as the query length increases. This observation can be explained bearing in mind the insertion algorithm of TB-tree, which stores in each leaf node segments belonging to the same trajectory. This has the main drawback that spatially close segments from different trajectories are stored in different nodes. As such, the TB-tree preserves the temporal order of the positions of the

moving objects, while ignores their spatial allocation. However, as the query temporal extent increases, the pruning power of the TB-tree deteriorates, since the drawback of the inadequate spatial allocation of the positions of the moving objects overcomes the gain of having them stored according to their temporal order. Here, we have to mention the advantage of TB*-tree, which seems to be the overall winner in all experimental settings. It turns out that the "delete and re-insert" strategy adopted in the TB*-tree in construunction with the increased fanout of its nodes is adequately effective in the case of similarity search.

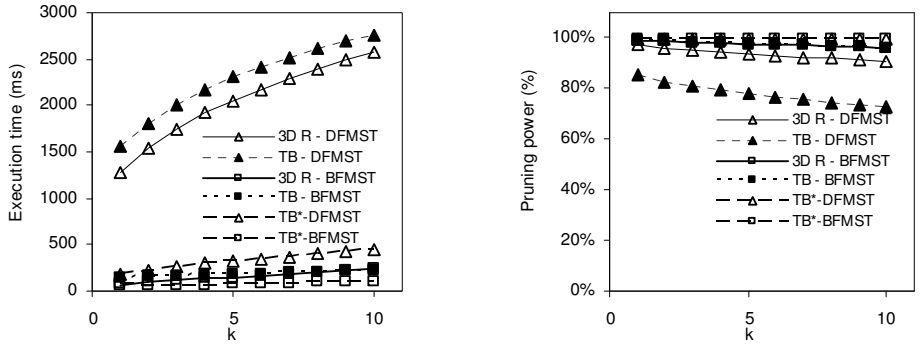


(a) (b)

Figure 4.13: Scaling with the query length (Q3)

Finally, Figure 4.14 illustrates the behavior of the proposed algorithms regarding the number of most similar trajectories requested. Again `BFMSTSearch` algorithm achieves high pruning power and small execution time decreasing its performance with *k* with a relatively small ratio. On the other hand, although `DFMSTSearch` algorithm shows good pruning power – but always worse than that of `BFMSTSearch` – its execution time is several times higher the respective execution time of its competitor, having the same explanation as the previous ones.

Summarizing the results of our experimental study, both algorithms show high pruning power while the `BFMSTSearch` always outperforms `DFMSTSearch` by several orders of magnitude. The pruning power of the `DFMSTSearch` depends on the dataset's *MMS*, the query length and the number of *k*, while that of `BFMSTSearch` depends only on the query length and (in smaller degree) in the number of *k*. Moreover, `BFMSTSearch` always achieves pruning power above 90%. Regarding execution time, `BFMSTSearch` always outperforms `DFMSTSearch` due to the utilization of the $MINDISSIM_{INC}$ heuristic which directly rejects all tree nodes not yet processed by the time it realizes.



(a) (b)

Figure 4.14: Scaling with number of *k* (Q4)

4.6. Conclusions

Related work on similarity query processing for trajectories is based on the context of time series analysis or the Longest Common Subsequence (LCSS) model [VKG02] and the recently proposed Edit Distance on Real Sequence (EDR) [COO05]. However, all these methods have the main drawback that they either ignore the temporal dimension of the movement therefore calculating the spatial (not the spatio-temporal) similarity between the trajectories, or assume that the trajectories are of the same length and have the same sampling rate. What is more, the majority of the proposed approaches exploit specialized index structures in order to prune the search space and retrieve the most similar to a query trajectory.

In this thesis we relaxed these assumptions by defining a novel metric, called *DISSIM*, and then we presented a complete treatment of historical MST queries over moving object trajectories stored on R-tree-like structures avoiding the drawbacks of the existing methods. We proposed a set of metrics, based on simple notions of trajectories, such as the dataset maximum speed, each one followed by a lemma that support our ordering and pruning strategies. Then, we presented two MST algorithms over trajectories indexed by R-tree-like structures following the depth-first [RKV95] and best-first [HS99] paradigm.

Under various synthetic and real trajectory datasets, we illustrated the superiority of the proposed *DISSIM* metric against related work [VKG02], [COO05], in terms of quality, while our algorithms show high pruning ability when processing MST queries, also verified in the case of *k*-MST queries. Among the algorithms proposed, the `BFMSTSearch` following the best-first paradigm [HS99] seems more promising showing better performance over its depth-first competitor `DFMSTSearch`; in particular, it demonstrates linear behavior in terms of execution time and node accesses, while its pruning power remains above 90% in all settings tested during the experimental study (whereas the pruning power of `DFMSTSearch` degrades to very small values as the query length increases).

The proposed algorithms do not require any dedicated index structure and can be directly applied to any member of the R-tree family used to index trajectories, such as the 3D R-tree, the TB-tree and the TB*-tree used in our implementation.

5. Managing the Effect of Location Uncertainty in Trajectory Databases

In this chapter we provide our theoretical model for estimating the effect of uncertainty in spatio-temporal querying. The chapter is structured as follows. Section 5.1 motivates the work in this chapter. Section 5.2 discusses related work, while Section 5.3 describes the theoretical analysis on the effect of uncertainty under several uniformity assumptions. In Section 5.4 the proposed model is extended in order to support non-uniform distributions over the problem parameters. Section 5.5 evaluates the accuracy of the model through an extensive experimental study over synthetic and real datasets, while Section 5.6 discusses the employment of the proposed model in the context of spatial databases. Finally, Section 5.7 provides the conclusions of the chapter.

5.1. Introduction

A common assumption adopted in spatial and spatio-temporal databases is that the position of objects is precisely known. However, a variety of reasons, such as GPS and sampling errors, may influence the accuracy of the recorded locations of trajectories, since location data obtained from measuring devices are inherently imprecise. Moreover, several recent works [BS03], [CZBP06], [GL05] suggest that the location privacy of mobile users should be protected by adding a controlled degree of noise in each object's measured position. Consequently, all these errors introduce an uncertainty factor into the answers of traditional queries.

The literature on the management of the location uncertainty of spatio-temporal objects so far, deals with either uncertainty representation issues [Tra03], [TWHC04], [WSCY99] or probabilistic algorithms [CKP04] that process queries in the presence of uncertainty, estimating the probability of each trajectory to be included in the query result. On the other hand, in this thesis we argue that there are cases where the user would prefer to know the influence of the measurement error in the query results, without actually executing the query. The challenge thus accepted in this chapter, is to *provide a theoretical framework that estimates the error introduced due to the uncertainty of moving objects' locations in the results of spatio-temporal queries*. Among the spatio-temporal query types, our interest is focused on *timeslice queries*, which can be used to retrieve the positions of moving objects at a given time point in the past and can be seen as a special case of spatio-temporal range queries, with their temporal extent set to zero [PJT00]. This type of query can also be seen as the combination of a spatial (i.e., query window W) and a temporal (i.e., timestamp t) component. As it will be discussed in Chapter

7, the extension of the model provided by this work in order to support range queries with non-zero temporal extent is by no means trivial and is left as future work. To the best of our knowledge, our work is the first that tackles this problem.

Towards this goal, the model proposed in [TWHC04], [TWZC02] regarding the uncertainty of trajectory data is initially adopted. In particular [TWHC04] propose that the trajectories of moving objects should be modeled as 3D cylindrical volumes around the tracked trajectory (recall Figure 1.5); as such, when executing a timeslice query over the trajectory database, the original trajectory is transformed to a single point and the cylindrical volume to a disk (Figure 5.1) which is called *uncertainty disk* and the actual location of the moving object at this particular timestamp is assumed to follow a uniform distribution within this disk. Although the model proposed in [TWHC04] (and consequently, the uniform statistical distribution) may be assumed when artificially injecting uncertainty into data objects as proposed by [BS03], [CZBP06], [GL05], it is rather unrealistic to describe the actual measurement and sampling error introduced by various devices (GPS-equipped smartphones, etc.) and interpolation methods being employed to calculate the position of the moving object between consecutive time-stamped positions. Therefore, in the sequel, we employ other statistical distributions [Lei95], [PTJ05] and augmented histograms in order to support more realistic scenarios of uncertainty distribution.

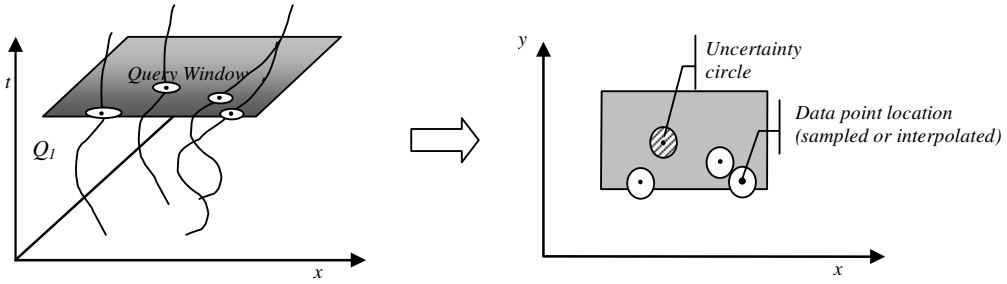


Figure 5.1: Problem Setting

The model described in this thesis can be used in MODs so as to estimate the average number of false hits in query results due to location uncertainty introduced in spatio-temporal data; thus, it could be utilized in an interactive graphical query builder/analyzer, providing online an approximation of the percentage of the false hits due to location uncertainty along with other estimations, such as selectivity, execution time, etc. Moreover, the proposed methodology can be directly employed in existing Spatial Database Management Systems (SDBMS) in order to cover the same needs; actually, the majority of the techniques developed in this chapter may be straightforwardly used in the context of traditional Spatial Databases, since the timeslice of a spatio-temporal database actually produces a snapshot of a set of static spatial objects (Figure 5.1).

A more vivid example demonstrating the applicability of the proposed model can be obtained considering the following real-world situation, inspired by the emerging open agoras paradigm [Ioa07]: let us assume a user who wishes to pose a timeslice query over several distributed subscribe-based data-sources containing the same trajectories represented at different levels of uncertainty due to the different measurement methods and, consequently, different errors; though the criterion used to choose among them is the optimization, i.e., the minimization, of the uncertainty introduced in the final query

results, provided by the data-sources during the negotiation step with their potential customers/users [Ioa07]. Under such circumstances, only the model proposed in this thesis may provide the user-side query optimizer with the error introduced in the results of the query for each different data-source.

Additionally, the proposed model can be utilized in order to determine the *maximum permitted* (im)precision of the trajectory data that will feed a MOD (and consequently, an SDBMS) given the required accuracy in the results of timeslice (respectively, range) queries. Then, users can be guided by the DBMS in the employment of the appropriate, more or less accurate - which also entails a more/less expensive - positioning method to be used for the data that will feed the system.

Perharps the most prominent application of the developed model is over summary data, which contain aggregate-only information instead of actual data objects, e.g., the number of distinct trajectories inside a given spatial region and timestamp (or the number of spatial objects inside a given spatial region, in the case of simple spatial data). Consider, for example, the case of a Trajectory Data Warehouse (TDW) [MFN+08], where aggregation may exhibit partial containment relationships instead of the total containment relationships normally assumed in conventional data warehouses; that is, a spatial cell may be contained in city A by 30% and in city B by 70%. Given that pre-aggregated information is only stored at the lowest level of the data warehouse location dimension hierarchy, i.e., the cells or base cuboids, a roll-up operation at the *city* level at a given timestamp, would, among others, aggregate over the number of partially contained cells. The above situation is illustrated in Figure 5.2, which presents the bounds between four cities, A, B, C and D, along with a snapshot of a set of uncertain trajectories (transformed to data points along with their uncertainty disks), and a regular grid standing for representing the cells containing the pre-aggregated information.

Under this setting, the option of performing probabilistic queries cannot be applied, since they require the presence of the actual data along with the distribution of their uncertainty. On the other hand, the model developed in this chapter can still be directly applied utilizing aggregate information, i.e., the number of objects and the radius of the uncertainty disk or standard deviation of the normal distribution, producing finally an approximation of the error introduced in the aggregation results. In particular, given that our model is capable of determining the effect of the location uncertainty in the *Minimum Bounding Rectangle* (MBB) of city *A* considering it as a range query, it can approximate the effect in the actual spatial object *A*, involving only the cardinality of each cell, the MBB and the uncertainty radius.

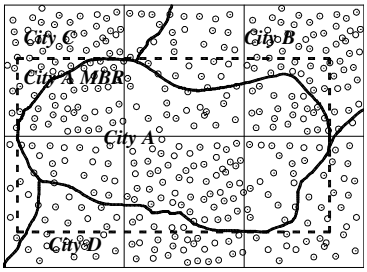


Figure 5.2: Partial containment in Trajectory Data Warehouses

To the best of our knowledge, a theoretical study on modeling the error introduced in spatio-temporal (or spatial) query results in terms of false hits due to the uncertainty of trajectories (or spatial

objects) is lacking. Outlining the major issues that will be addressed, the main contributions of this chapter are as follows:

- Two lemmas that estimate the average number of false positives and false negatives when executing timeslice queries over uniformly distributed uncertain trajectories modelled via the [TWHC04] proposal, are proved; both errors depend on the radius of the cylindrical volume and the perimeter of the timeslice query window, rather than its area.
- In order to relax the location uncertainty uniformity assumption (directly derived from the model of [TWHC04]), we utilize the real-world adapted bivariate normal distribution [Lei95], [PTJ05], which is efficiently approximated by the uniform difference distribution. The results are close enough to the ones of the original analysis.
- Novel spatio-temporal and other augmented histograms are employed in order to estimate the average number of false hits when the uniformity assumption of objects' distribution in the data space is relaxed, as well as to support various distributions of the uncertainty radius. The same methodology is also employed in other forms of summary data, e.g., data warehouses, in order to describe the effect of uncertainty.
- A comprehensive set of experiments is performed demonstrating the correctness and accuracy of the analysis.
- Finally, it is shown how the results of the analysis may be applied over spatial datasets: the solutions proposed in this chapter are implemented on top of a commercial SDBMS, namely, the PostgreSQL [Post08a] with PostGIS spatial extension [Post08b]. It is worth to note that off-the-shelf spatial histograms, already used in SDBMS for query selectivity estimation, support the proposed model without additional requirements.

5.2. Related Work

Wolfson et al. [WSCY99] address the problem of the location imprecision of moving objects by proposing a set of updating policies of the database that stores the object locations. The basic idea is that the database is updated whenever the distance between the actual location of an object and the stored in the database value exceeds a threshold. In this way, an uncertainty factor of every object's location is introduced, since objects are within distance of 1 Km from the last recorded locations. Adopting the utilization of *pdf*s, they describe an algorithm that processes a probabilistic spatial range query applied in the above database. The output of this type of query, which returns the set of objects that are within a region R, consists of pairs of the form (O_i, P_i), where P_i is the probability that object O_i intersects query region R. Cheng et al. [CKP04] adopt the definition of the probabilistic query introduced in [WSCY99] and extend it in the case of nearest neighbor (NN) queries.

Pfoser and Jensen [PJ99] propose a representation of location uncertainty due to measurement and sampling errors. There, the spatial projection of the trajectory of an object is modeled as a 2D elliptical area, defined by two consecutive tracked positions. They also present the influence of the location uncertainty in the processing of probabilistic range queries and propose a filter-and-refinement method to answer them.

Location uncertainty of moving objects is also discussed by Trajcevski et al. [Tra03], [TWHC04], where a trajectory of an object is modeled as a 3D cylindrical volume around the tracked trajectory. Furthermore, two categories of operators for querying trajectories with uncertainty are introduced, concerning spatio-temporal point and range queries, respectively, and efficient algorithms are presented for their implementation.

Ni et al. [NRB03] propose a probabilistic spatial data model for the positional accuracy of polygon data. According to this model, each polygon is partitioned into disjoint independent chunks. Each chunk is a contiguous series of vertices with fully correlated locational uncertainties. Based on the above model, a probabilistic spatial join algorithm is described, in which the object pairs of the result are associated with the intersection probability between each pair. A variation of the R-tree, called probabilistic R-tree, is introduced for the support of the probabilistic filtering of the join algorithm, in which each MBB approximation is augmented with the probability distribution of MBB's boundary.

Cheng et al. [CXP+04] investigated the problem of indexing uncertain data in order to efficiently answer probabilistic threshold queries, in which the appearance probability of each data point in the result of the query exceeds a given threshold. Two index structures are proposed. The pruning power of the first index is based on the utilization of uncertain information augmented to the internal nodes of the index, while in the second index data points with similar degrees of uncertainty are clustered together. Recently, Tao et al. [TCX+05] studied a similar type of query, the probabilistic range query, which retrieves the objects that appear in a rectangular area with probabilities of at least a pre-defined value. They introduced a fully dynamic index structure on uncertain data. This structure, called U-tree, maintains "auxiliary information" at all of its levels for the respective indexed objects that can be used to validate the presence of an object in the results of a probabilistic range query, without calculating its computationally expensive appearance probability.

Dai et al. [DYM+05] have studied the problem of evaluating spatial queries for existentially uncertain data; in this case, uncertainty does not refer to the locations of the objects but to their existence. The authors define two probabilistic query types: the so-called thresholding and ranking queries in which the output is controlled by either thresholding the results of low probability to occur or ranking them and selecting the ones with the highest probability respectively. In the sequel, probabilistic variants of spatial range and NN queries are presented for objects indexed by a 2D index, such as the R-tree. Finally, in order to improve the efficiency of their proposed algorithms, they propose an extension of the R-tree, in which the non-leaf entries are augmented with the maximum existential probability of the objects indexed under them.

Perhaps the most relevant to our work is the study by Yu and Mehrotra [YM03], where the effect of uncertainty in probabilistic spatial queries, similar to the work presented in [NRB03], is discussed. By performing a theoretical analysis, they provide a novel technique which can be used in order to set the data precision in the data collection process, so that a probabilistic guarantee on the uncertainty in answers of spatial queries can be provided. The first outcome of the analysis are the cardinalities of the three subsets of a range query result, namely the *MUST*, *MAY* and *ANS* sets: *MUST* is the set of objects that "must" be located within the query range, *MAY* is the set of objects that "may" be located within

the query range, and *ANS* is the approximate answer set of objects whose recorded locations are in the query region. The second outcome is a method for determining the largest possible imprecision, i.e., the *uncertainty radius* of our analysis, given that the answer to a random *COUNT* query should include an uncertainty $\delta \leq \delta_0$, i.e., the cardinality of the *MAY* set be less than a value, with a probability $P \geq P_0$.

Comparing the proposed in this thesis model with [YM03], the first remark is that the numbers E_N and E_P of false hits that we estimate is actually a *refinement*, i.e., a subset, of the *MAY* set estimated by [YM03], and it is not straightforward to remove the overestimation provided by [YM03] unless our model is used; this overestimation was clearly shown in the experimental results presented in Section 5.6.2.1. A second remark is that the model presented in [YM03] is based on the uniformity assumption, whereas our study addresses more realistic requirements.

5.3. Modeling Error due to Location Uncertainty

Consider a dataset P consisting of N trajectories T_i, $i = 1, \ldots, N$, distributed in the unit spatio-temporal space $S = [0,1] \times [0,1] \times [0,1]$, that is, all dimensions are normalized in the interval between 0 and 1. We initially give the notion of *uniformly distributed trajectories*: *a set of trajectories is uniformly distributer iff the positions of moving objects obtained by a snapshot of P at an arbitrary timestamp t_k, producing a set of points $T_{i,k}$, $i = 1, \ldots, N$, and k=1…now, are uniformly distributed.* Moreover, the product of this snapshot on S is the space $S_k = [0,1] \times [0,1]$.

According to [TWHC04], moving object trajectories should by modelled as cylindrical volumes of constant radius d around the actual sampled positions of moving objects and the corresponding interpolated trajectory. As such, a snapshot of a trajectory T_i on timestamp t_k produces an *uncertainty disk* with center $T_{i,k}$ and radius d, inside which the *actual* position $T_{i,k}^{\dagger}$ of trajectory T_i on timestamp t_k, is uniformly distributed. Let also R be the set of all timeslice queries posed over dataset P, R_k the subset of R timestamp t_k, and $R_{k,a\times b}$ the subset of R_k containing all timeslice queries having sides of length $2a$ and $2b$ along the x- and y- axis, respectively.

Two error types are introduced when executing a timeslice query $W_k \in R_{k,a\times b}$ over the dataset P:

- E_N is the set of *false negatives*, i.e., trajectories qualifying the query window but not retrieved; formally, $E_N = \left\{ T_i \in P : T_{i,k} \notin W_j \mid T_{i,k}^{\dagger} \in W_j \right\}$, and
- E_P is the set of *false positives*, i.e., trajectories retrieved while not qualifying the query window; formally, $E_P = \left\{ T_i \in P : T_{i,k} \in W_j \mid T_{i,k}^{\dagger} \notin W_j \right\}$.

The problem is to make an as accurate as possible estimation of false negatives and false positives for a random W_j at timestamp t_k, based only on known dataset and query parameters. From the above problem definition, it is clear that we initially make four main assumptions:

- A_I - *uncertainty uniformity assumption*: the actual position $T_{i,k}^{\dagger}$ of trajectory T_i at timestamp t_k is uniformly distributed inside the uncertainty disk $C(T_{i,k},d)$,
- A_{II} - *data uniformity assumption*: the trajectories T_i (and consequently, points $T_{i,k}$ at timestamp t_k), are uniformly distributed in the data space,

- A_{III} - *constant uncertainty radius assumption*: the radius d of the cylindrical volume (and consequently, uncertainty disk) is constant,

and, not directly extracted from the problem definition,

- A_{IV} - *uncertainty size assumption*: radius d is always less than the half of the length of the smallest side of query window W_j.

Table 5.1: Table of notations

Notation	*Description*
S, P, N	the unit spatio-temporal data space $[0,1]\times[0,1]\times[0,1]$ the trajectory dataset, and its cardinality (also, density)
t_k, S_k	a timestamp and the snapshot of S at timestamp t_k
T_i, $T_{i,k}$, $T_{i,k}^{\dagger}$, d	a trajectory, the (recorded or interpolated) location of T_i at timestamp t_k, its actual location, and the radius of the uncertainty disk
W_j, $W_{j,c1}$ - $W_{j,c4}$	the window of a timeslice query, and its four corners (clockwise, starting from the lower-left)
$W_j^{x,L}, W_j^{x,U}, W_j^{y,L}, W_j^{y,U}$	the minimum and maximum coordinates of the timeslice query window W_j along the x- and y- axis.
R, R_k, $R_{k,a\times b}$	the set of all timeslice queries over P, its subset invoked at timestamp t_k, and its subset with half-sides a and b along the x- and y- axis, respectively
$C(T_{i,k},d)$, $A_{i,j}$	the uncertainty disk of the (recorded or interpolated) location of T_i at timestamp t_k with radius d and the portion of its area that lies inside (in the case of false negatives) or outside (in the case of false positives) W_j
$Dist\left(T_{i,k},W_j\right)$	the minimum Euclidean distance between the (recorded or interpolated) location of T_i at timestamp t_k and the boundary of W_j
r_x, r_y	the distance of the closest to $T_{i,k}$ point of the boundary of W_j along the x- and y- axis, respectively.
$A_{1x}(r_x,r_y)$, $A_{1y}(r_x,r_y)$	the area encompassed by a chord perpendicular to the x- (or y-) axis with r_x (or r_y, respectively distance from $T_{i,k}$ and the respective arc of its uncertainty disk
$A_2(r_x,r_y)$	the overlapping area between the uncertainty disk of $T_{i,k}$ and a query window corner being inside the disk, with r_x and r_y coordinates relatively to $T_{i,k}$.
$V_{i,j}$, $V_{1x}(r_x,r_y)$, $V_{1y}(r_x,r_y)$, $V_2(r_x,r_y)$	the volumes of the conical segments, equivalent to areas $A_{i,j}$, $A_{1x}(r_x,r_y)$, $A_{1y}(r_x,r_y)$, $A_2(r_x,r_y)$ when following the uncertainty uniformity difference assumption.
$AvgP_{i,P}(R_{k,a\times b})$, $AvgP_{i,N}(R_{k,a\times b})$	the average probability of a single trajectory T_i to be false positive (or false negative) with respect to all query windows $W_j \in R_{k,a\times b}$
$E_P\left(R_{k,a\times b}\right)$, $E_N\left(R_{k,a\times b}\right)$	the average number of false positives (or false negatives) in the results of a timeslice query $W_j \in R_{k,a\times b}$

Regarding the first three assumptions (A_I - A_{III}), they will be relaxed in the model extension to be presented in Section 5.4. Regarding assumption A_{IV}, we argue that it is a reasonable property of the involved spatial objects, since typical sizes of query window W_j are usually orders of magnitude larger than d; for example, trajectories sampled with GPS devices usually introduce an error of a few meters (usually less than 10m), while query windows in real applications are expected to be at least hundreds of square meters.

Having described the framework of our work, in the next two sections we prove two lemmas which are fundamental for our model. Table 5.1 summarizes the notations used in the rest of the chapter.

5.3.1. Estimating the Number of False Negatives

In this section we prove a lemma which undertakes the calculation of the average number of false negatives.

Lemma 5.1*: The average number $E_N(R_{k,a\times b})$ of false negatives in the results of a timslice query $W_j \in R_{k,a\times b}$ with half-sides of length a and b at timestamp t_k over a trajectory dataset that follows the data uniformity and uncertainty uniformity assumptions is given by the formula:*

$$E_N\left(R_{k,a\times b}\right) = N \cdot \left(\frac{8d}{3\pi}(a+b) - \frac{d^2}{2\pi}\right) \qquad (5.1)$$

where d is the radius of the uncertainty disk.

Proof: The average number $E_N(R_{k,a\times b})$ of trajectories being false negatives in the results of a timeslice query $W_j \in R_{k,a\times b}$, i.e., $T_{i,k} \notin W_j \mid T_{i,k}^{\dagger} \in W_j$, can be obtained by the average probability $AvgP_{i,N}(R_{k,a\times b})$ of an arbitrary trajectory T_i to be false negative regarding an arbitrary query window $W_j \in R_{k,a\times b}$, multiplied by the total number N of trajectories:

$$E_N\left(R_{k,a\times b}\right) = N \cdot AvgP_{i,N}\left(R_{k,a\times b}\right) \qquad (5.2)$$

Obviously, our target is to determine $AvgP_{i,N}(R_{k,a\times b})$. Towards this goal, we formulate the probability that $T_{i,k} \notin W_j \mid T_{i,k}^{\dagger} \in W_j$. This probability is given by the ratio or the area $A_{i,j}$ of the portion of the uncertainty disk $C(T_{i,k},d)$ included inside the query window, over the total area of $C(T_{i,k},d)$. However, $A_{i,j}$ is zero in cases where $C(T_{i,k},d)$ does not overlap the query boundary.

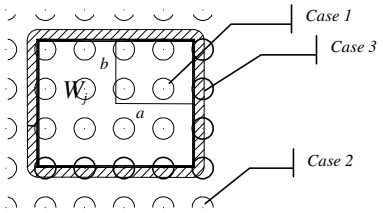


Figure 5.3: Snapshot of trajectories contributing to the number of false negatives

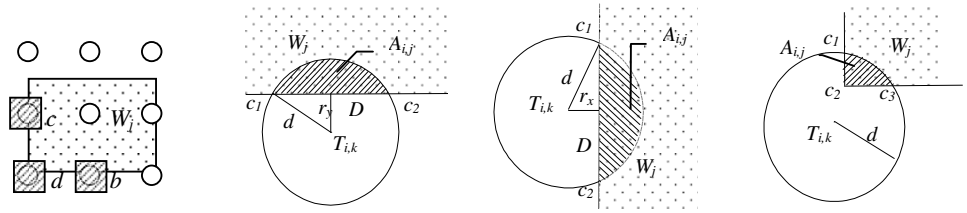
Figure 5.3 illustrates a timeslice query window W_j extended by a buffer of width d, over a subset of uniformly distributed points, corresponding to the snapshot of P near W_j at timestamp t_k: trajectories represented as points with uncertainty disks being inside the query window, i.e., those labeled as "case 1", cannot incur false negatives because they will be actually retrieved by the query. The same is also true for points with uncertainty disks located outside the buffer zone, illustrated as "case 2" in Figure 5.3. The single case where $T_{i,k}$ is not retrieved by the query while $T_{i,k}^{\dagger}$ may be found inside W_j is when $T_{i,k}$ is located inside the buffer zone that surrounds W_j, which is illustrated as "case 3" in Figure 5.3.

The above discussion expresses the fact that a trajectory T_i is a candidate to be false negative if and only if $T_{i,k}$ is located outside the query window, while the corresponding uncertainty disk $C(T_{i,k},d)$ intersects the query boundary. Alternatively, $T_{i,k}$ should be located inside the *Minkowski region* of W_j with radius d in order to be candidate to be false negative; this region can be determined by extending

W_j with distance d on all directions [TZPM04]. Minkowski regions are directly derived from the concept of *Minkowski sum* [AFH02] between the query window W_j and a disk of radius d, which, in our case, is composed by a set of line segments and circular arcs, illustrated as the boundary exterior of W_j in Figure 5.3. Now, the probability of a trajectory T_i to be false negative, regarding a query window W_j, is:

$$P\left(T_{i,k} \notin W_j \mid T_{i,k}^{\dagger} \in W_j\right) = \begin{cases} \dfrac{A_{i,j}}{\pi d^2}, \text{ if } T_{i,k} \notin W_j \text{ and } Dist\left(T_{i,k}, W_j\right) \le d \\ 0, \qquad \text{otherwise} \end{cases} \tag{5.3}$$

The area $A_{i,j}$, which is illustrated in Figure 5.4, is determined by taking into account the uncertainty size assumption by distinguishing between three cases illustrated in Figure 5.4(b) – (d): In the first two cases, where the distance between $T_{i,k}$ and each of the four corners of W_j is larger than d, $A_{i,j}$ is the portion of the uncertainty disk enclosed by (a) the chord c_1c_2 formed by the query side and the uncertainty disk and (b) the respective arc $\overset{\frown}{c_1c_2}$. Thus, it can be computed as the integral of the function of the chord length D, given as an expression of its distance, r_y or r_x (depending on which query side is regarded) from the disk center.



(a) (b) (c) (d)

Figure 5.4: The unit space (a) and three details of it (b, c, d)

Let the chord c_1c_2 be parallel to x axis (Figure 4(b)), it holds that $D(r_x, r_y) = 2\sqrt{d^2 - r_y^2}$ and $A_{i,j} = A_{1y}(r_x, r_y) = \int_{r_y}^{d} D(r_x, r_y)dr_y = 2\int_{r_y}^{d} \sqrt{d^2 - r_y^2}\, dr_y$, resulting in[1]:

$$A_{i,j} = A_{1y}(r_x, r_y) = d^2 \arctan\left[\sqrt{\left(\frac{d}{r_y}\right)^2 - 1}\right] - r_y\sqrt{d^2 - r_y^2} \tag{5.4}$$

Equivalently, let the chord c_1c_2 be parallel to y axis (Figure 4(c)), the area $A_{i,j} = A_{1x}(r_x, r_y)$ is calculated by substituting r_y with r_x in Eq.(5.4). In the third case, where the distance between $T_{i,k}$ and one of the four corners of W_j is less than d (Figure 4(d)), $A_{i,j}$ can be determined in a similar way resulting in:

$$A_{i,j} = A_2(r_x, r_y) = \frac{1}{2}\left[d^2 \operatorname{arccot}\left[\sqrt{\frac{r_y}{R^2 - r_y^2}}\right] - d^2 \arctan\left[\sqrt{\frac{r_x}{R^2 - r_x^2}}\right] - r_y\sqrt{d^2 - r_y^2} - r_x\sqrt{d^2 - r_x^2} + 2r_xr_y\right] \tag{5.5}$$

The average, with respect to any query window in $R_{k,axb}$, probability of a trajectory T_i to be false negative is calculated by integrating Eq.(5.3) over all possible query windows:

$$AvgP_{i,N}\left(R_{k,a\times b}\right) = \int_{W_j \in R_{k,a\times b}} P\left(T_{i,k} \notin W_j \mid T_{i,k}^{\dagger} \in W_j\right) dW = \iint_{S_k} P\left(T_{i,k} \notin W_j \mid T_{i,k}^{\dagger} \in W_j\right) dxdy \tag{5.6}$$

[1] All advanced calculations in this chapter were performed using the *Mathematica* software [28].

In order to compute the above integral, it is necessary to determine the main zones inside which the area $A_{i,j}$ can be expressed as a single function. To facilitate discussion, Figure 5.5(a) illustrates the fact that the area determined by $Dist(T_{i,k},W_j) \leq d$ can be divided into three sets of zones inside which point $T_{i,k}$ can be found regarding the position of the query window: the first drawn with vertical stripes, the second drawn with horizontal stripes, and the shaded one, called Z_1, Z_2 and Z_3, respectively. Z_1 regions contain the data points such that the area resulted by the intersection of their uncertainty area with W_j forms a complete circular segment; alternatively, Z_1 regions is the locus of the points in the space such that they are outside W_j, their distance from W_j is smaller than d and their distance from the four corners of W_j is greater than d. Z_2 regions is the locus of the points in the space such that points are outside W_j, their distance from W_j is smaller than d and their x or y coordinates are inside the projection of W_j along the x- or y- axis, respectively; similarly, Z_3 regions differ only on that the x and y coordinates of their points are outside the projection of W_j along the x- or y- axis.

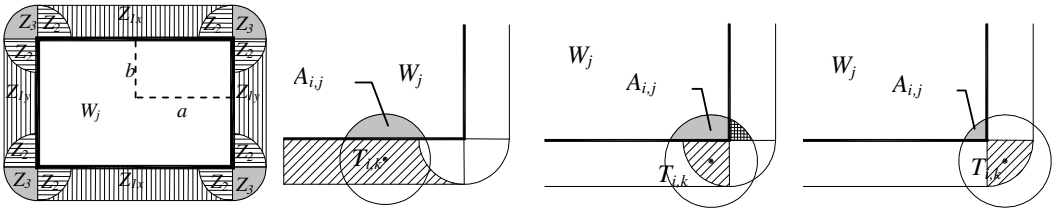


Figure 5.5: Zones where area $A_{i,j}$ contributing in false negatives is expressed as a single function

Zones $Z_{1,j}$, $Z_{2,j}$ and $Z_{3,j}$ associated with query window W_j are formally defined by the following expressions:

$$Z_{1,j} = \left\{ T_i \in P : T_{i,k} \notin W_j \wedge Dist\left(T_{i,k}, W_j\right) \leq d \wedge Dist\left(T_{i,k}, W_{j,c_i}\right) \geq d, i = 1..4 \right\} \tag{5.7}$$

$$Z_{2,j} = \left\{ \begin{array}{c} T_i \in P : T_{i,k} \notin W_j \wedge Dist\left(T_{i,k}, W_j\right) \leq d \wedge Dist\left(T_{i,k}, W_{j,c_i}\right) \leq d, i = 1..4 \wedge \\ \left(T_{i,k}^x \in \left[W_j^{x,L}, W_j^{x,U}\right] \vee T_{i,k}^y \in \left[W_j^{y,L}, W_j^{y,U}\right]\right) \end{array} \right\} \tag{5.8}$$

$$Z_{3,j} = \left\{ \begin{array}{c} T_i \in P : T_{i,k} \notin W_j \wedge Dist\left(T_{i,k}, W_j\right) \leq d \wedge Dist\left(T_{i,k}, W_{j,c_i}\right) \leq d, i = 1..4 \wedge \\ \left(T_{i,k}^x \notin \left[W_j^{x,L}, W_j^{x,U}\right] \wedge T_{i,k}^y \notin \left[W_j^{y,L}, W_j^{y,U}\right]\right) \end{array} \right\} \tag{5.9}$$

Regarding zones of type Z_1, i.e., those labeled Z_{1x} and those labeled Z_{1y} in Figure 5.5(b), area $A_{i,j}$ can be computed using Eq.(5.4). When the relative positions of $T_{i,k}$ and W_j constrain it to be inside a zone of type Z_2, $A_{i,j}$ can be computed by subtracting the small area at the upper-right corner of the uncertainty disk (Figure 5.5(c)), which is given by Eq.(5.5), from the overall uncertainty disk area being above the lower query side (Eq.(5.4)). Finally, for points inside zones of type Z_3, as illustrated in Figure 5.5(d), $A_{i,j}$ can be computed using Eq.(5.5). Summarizing, $T_{i,k}$ may be found inside:

- one out of two zones Z_{1x} (top and bottom in Figure 5.5(a)), and two zones Z_{1y} (left and right in Figure 5.5(a)); in these cases, $A_{i,j}$ is calculated by A_{1x} and A_{1y}, respectively,
- one out of four zones Z_3, one for each query window corner; in these cases, $A_{i,j}=A_2$,
- one out of four zones Z_2, for each query window corner along the x- and another four along the y- axis; in these cases, $A_{i,j} = \left(A_{1x} - A_2\right)$ and $A_{i,j} = \left(A_{1y} - A_2\right)$, respectively,
- elsewhere; in this case, $A_{i,j}$ is zeroed.

Bearing in mind that (a) Eq.(5.6) integrates $P\left(T_{i,k} \notin W_j \mid T_{i,k}^{\dagger} \in W_j\right) = A_{i,j} / \pi d^2$ over the whole space S_k, and (b) the value of $A_{i,j}$ is equal to zero in any other place, expect of the zones Z_1, Z_2, Z_3 where $A_{i,j}$ is provided in terms of the relative position between $T_{i,k}$ and W_j, i.e., r_x and r_y, Eq.(5.6) can be rewritten as follows:

$$AvgP_{i,N}\left(R_{k,a\times b}\right) = \frac{1}{\pi d^2}\left(\begin{array}{c} 2\iint_{Z_{1x}} A_{1x}(r_x,r_y)dr_y dr_x + 2\iint_{Z_{1y}} A_{1y}(r_x,r_y)dr_y dr_x + 4\iint_{Z_3} A_2(r_x,r_y)dr_y dr_x \\ 4\iint_{Z_2}\left(A_{1x}(r_x,r_y) - A_2(r_x,r_y)\right)dr_y dr_x + 4\iint_{Z_2}\left(A_{1y}(r_x,r_y) - A_2(r_x,r_y)\right)dr_y dr_x \end{array}\right) \Rightarrow$$

$$AvgP_{i,N}\left(R_{k,a\times b}\right) =$$

$$\frac{1}{\pi d^2}\left(2\iint_{Z_{1x}+2Z_2} A_{1x}(r_x,r_y)dr_y dr_x + 2\iint_{Z_{1y}+2Z_2} A_{1y}(r_x,r_y)dr_y dr_x - 4\iint_{Z_3} A_2(r_x,r_y)dr_y dr_x\right) \quad (5.10)$$

The two $Z_{1x} + 2Z_2$ areas involved in the above integrals may be regarded as the top and down rectangles of Figure 5.5(a) formed by the Z_{1x} and the two Z_2 areas surrounding it, and their size along the x- and y-axis is $2a$ and d, respectively. The same also holds regarding the two $Z_{1y} + 2Z_2$ areas, also having extents d and $2b$ along the x- and y-axis, respectively. According to this discussion, the above formula can be rewritten as follows:

$$AvgP_{i,N}\left(R_{k,a\times b}\right) =$$

$$\frac{1}{\pi d^2}\cdot\left(2\int_0^d\int_0^{2a} A_{1x}\left(r_x,r_y\right)dr_x dr_y + 2\int_0^{2b}\int_0^d A_{1y}\left(r_x,r_y\right)dr_x dr_y - 4\int_0^d\int_0^{\sqrt{d^2-x^2}} A_2\left(r_x,r_y\right)dr_x dr_y\right) \quad (5.11)$$

Substituting $\int_0^d A_{1y}\left(r_x,r_y\right)dr_y = \int_0^d A_{1x}\left(r_x,r_y\right)dr_x$ with $\frac{2}{3}d^3$, and $\int_0^d\int_0^{\sqrt{d^2-x^2}} A_2\left(r_x,r_y\right)dr_y dr$ with $\frac{1}{8}d^4$ in the above long expression, we result in the simple formula:

$$AvgP_{i,N}\left(R_{k,a\times b}\right) = \frac{8d}{3\pi}\left(a+b\right) - \frac{d^2}{2\pi} \quad (5.12)$$

Substituting Eq.(5.12) into Eq.(5.2) we have proven Lemma 5.1. ■

5.3.2. Estimating the Number of False Positives

In the sequel, we prove a similar lemma regarding the average number of false positives:

Lemma 5.2*: The average number $E_P(R_{k,a\times b})$ of false positives in the results of a timeslice query $W_j \in R_{k,a\times b}$ with half-sides of length a and b at timestamp t_k over a trajectory dataset that follows the data uniformity and uncertainty uniformity assumptions is given by the formula:*

$$E_P\left(R_{k,a\times b}\right) = N\cdot\left(\frac{8d}{3\pi}\left(a+b\right) - \frac{d^2}{2\pi}\right) \quad (5.13)$$

where d is the radius of the uncertainty disk.

Proof: The average number $E_P(R_{k,a\times b})$ of trajectories being false positives in the results of a timeslice query $W_j \in R_{k,a\times b}$, i.e., $T_{i,k} \in W_j \mid T_{i,k}^{\dagger} \notin W_j$, can be obtained by the average probability $AvgP_{i,P}(R_{k,a\times b})$ of an arbitrary trajectory T_i to be false positive regarding an arbitrary query window $W_j \in R_{k,a\times b}$, multiplied by the total number N of trajectories in the data space:

$$E_P\left(R_{k,a\times b}\right) = N\cdot AvgP_{i,P}\left(R_{k,a\times b}\right) \quad (5.14)$$

Then, following a methodology similar to that followed in the proof of Lemma 1, it is proven that the probability that $T_{i,k} \in W_j \mid T_{i,k}^{\dagger} \notin W_j$ is:

$$P\left(T_{i,k} \in W_j \mid T_{i,k}^{\dagger} \notin W_j\right) = \begin{cases} \dfrac{A_{i,j}}{\pi d^2}, \text{ if } T_{i,k} \in W_j \text{ and } Dist\left(T_{i,k}, W_j\right) \le d \\ 0, \qquad \text{otherwise} \end{cases} \tag{5.15}$$

and the average, with respect to any query window in $R_{k,axb}$, probability of a trajectory T_i to be false positive is:

$$AvgP_{i,P}\left(R_{a\times b}\right) = \int_{W_j \in R_{k,axb}} P\left(T_{i,k} \in W_j \mid T_{i,k}^{\dagger} \notin W_j\right) dW = \int_{S_k} P\left(T_{i,k} \in W_j \mid T_{i,k}^{\dagger} \notin W_j\right) dxdy \tag{5.16}$$

The above integral is again computed by determining the zones inside which the area $A_{i,j}$ is expressed as a single function. These zones are found within the region formed by the original query window W_j and the *Minkowski difference* of W_j with a disk of radius d [TWHC04]. The Minkowski difference, also called *offsetting*, is a complementary operation to the Minkowski sum [TWHC04]; it is extensively studied in the field of computer graphics, while its calculation for convex polygons is a straightforward operation requiring linear time [TWHC04]. Figure 6(a) illustrates the three sets of such zones, namely Z_1, Z_2 and Z_3, which can be defined in a way similar to the ones of the false negatives computation. Formally:

$$Z_{1,j} = \left\{ \begin{array}{c} T_i \in P : T_{i,k} \in W_j \wedge Dist\left(T_{i,k}, W_j\right) \le d \wedge \\ \left(T_{i,k}^{x} \in \left[W_j^{x,L} + d, W_j^{x,U} - d\right] \vee T_{i,k}^{y} \in \left[W_j^{y,L} + d, W_j^{y,U} - d\right]\right) \end{array} \right\} \tag{5.17}$$

$$Z_{2,j} = \left\{ T_i \in P : T_{i,k} \in W_j \wedge T_{i,k} \notin Z_{1,j} \wedge Dist\left(T_{i,k}, W_j\right) \le d \wedge Dist\left(T_{i,k}, W_{j,c_i}\right) \ge d, i = 1..4 \right\} \tag{5.18}$$

$$Z_{3,j} = \left\{ T_i \in P : T_{i,k} \in W_j \wedge Dist\left(T_{i,k}, W_{j,c_i}\right) \le d, i = 1..4 \right\} \tag{5.19}$$

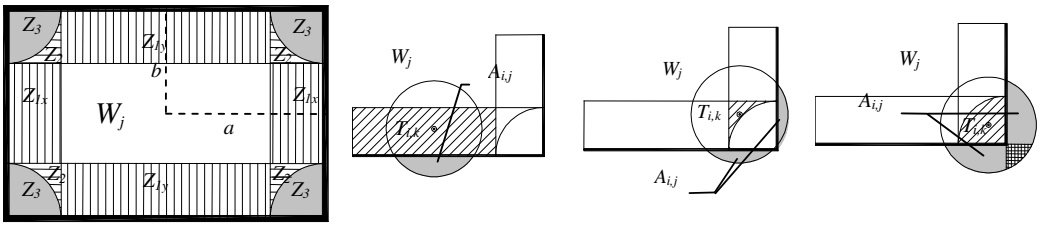


Figure 5.6: Zones where area $A_{i,j}$ contributing in false positives is expressed as a single function

Regarding zones Z_{1x} and Z_{1y}, the area $A_{i,j}$ is computed using Eq.(5.4) (Figure 5.6(b)). When in zone Z_2, $A_{i,j}$ is determined by summing up Eq.(5.4) along the x and y axes (Figure 5.6(c)). Finally, points representing trajectories inside Z_3 are also computed by summing up Eq.(5.4) along the x and y axes and subtracting the small area in the lower-right corner of the uncertainty disk (Figure 6(d)), which is given by Eq.(5.5). Summarizing, $T_{i,j}$ may be found inside:

- one out of two zones Z_{1x} (top and bottom in Figure 5.6(a)); in these cases, $A_{i,j}$ is calculated by A_{1x},
- one out of two zones Z_{1y} (left and right in Figure 5.6(a)); in these cases, $A_{i,j}$ is calculated by A_{1y},
- one out of four zones Z_3, one for each query window corner; in these cases, $A_{i,j}=A_{1x}+A_{1y}$,
- one out of four zones Z_2, for each query window corner; in these cases, $A_{i,j}=A_{1x}+A_{1y}-A_2$,

and Eq.(5.16) is reformulated as follows:

$$AvgP_{i,P}\left(R_{k,a\times b}\right)=$$

$$\frac{1}{\pi d^2}\cdot\begin{pmatrix}2\iint_{Z_{1y}}A_{1y}(r_x,r_y)dr_ydr_x+2\iint_{Z_{1x}}A_{1x}(r_x,r_y)dr_ydr_x+4\iint_{Z_2}\left(A_{1x}\left(r_x,r_y\right)+A_{1y}\left(r_x,r_y\right)\right)dr_ydr_x\\ 4\iint_{Z_3}\left(A_{1x}\left(r_x,r_y\right)+A_{1y}\left(r_x,r_y\right)-A_2\left(r_x,r_y\right)\right)dr_ydr_x\end{pmatrix} \quad (5.20)$$

which, after the necessary calculations, results in:

$$AvgP_{i,P}\left(R_{k,a\times b}\right)=\frac{8d}{3\pi}\left(a+b\right)-\frac{d^2}{2\pi} \quad (5.21)$$

Substituting Eq.(5.21) into Eq.(5.14) we have proven Lemma 5.2. ■

5.3.3. Discussion

Summarizing, the analytical model for the prediction of the number of false positives and false negatives when executing a timeslice query over uniformly distributed trajectory data, consists of Lemma 5.1 and Lemma 5.2 proved in the previous subsections. It turns out that the average number of false negatives and false positives of an arbitrary timeslice query at timestamp t_k with known size $2a$ and $2b$ along the x- and y- axes respectively, is a function of a, b, the uncertainty radius d and the cardinality N of the dataset. Another result is the corollary that, theoretically, the average number of false negatives is equal to the average number of false positives:

$$E_N\left(R_{k,a\times b}\right)=E_P\left(R_{k,a\times b}\right) \quad (5.22)$$

While such a result sounds strange at a first thought, it turns out to be reasonable when we take into consideration that, on the one hand, the number of trajectories contributing to the number of false negatives, represented by the shaded area in Figure 5.5(a), is greater than the respective one for false positives (Figure 5.6(a)) and, on the other hand, the area $A_{i,j}$ of the uncertainty disk of each trajectory contributing to the number of false negatives (Figure 5.5(d)) is smaller than the respective one for false positives (Figure 5.6(d)). Our analytical calculation of $E_N(R_{a\times b})$ and $E_P(R_{a\times b})$ proves that the above two complementary factors finally result into two equal values for the number of false positives and false negatives, thus resulting in Eq.(5.22).

Moreover, it notably arises from Eq.(5.1) and Eq.(5.13) that the average number of false negatives and false positives of a timeslice query depends on the query perimeter (a+b), rather than the query area ($a\cdot b$). A last observation is that, when our model is utilized to determine the maximum permitted (im)precision of the data that will feed a MOD, Eq.(5.1) and Eq.(5.13) can be solved for the value of the uncertainty radius d, given the values of the required accuracy in terms of false hits and the query's typical extent.

Intuitively, the two parts of the multiplier of N in Eq.(5.1) and Eq.(5.13), i.e., $\frac{8d}{3\pi}\left(a+b\right)$ and $\frac{d^2}{2\pi}$, stand for representing the contribution in the total number of false hits, of the length of the query perimeter, and the four corners of the query window, respectively. This detail will turn out to be very useful in the next section when we will relax the data uniformity assumption with the aim of histograms.

Finally, it must be pointed again, that the above developed formulas, as well as the majority of the ones presented hereafter, can be straightforwardly applied to simple spatial data without the need

for any modification; this due to the fact that a timeslice query over a set of trajectories can be seen as a range window query over a snapshot of the trajectories at the timestamp that is determined by the timeslice query. As such, the average number $E_N(R_{a\times b})$ and $E_P(R_{a\times b})$ of false negatives and false positives in the results of range window queries with half-sides a and b over simple spatial data, under the four assumptions stated in the beginning of this section is:

$$E_N\left(R_{a\times b}\right) = E_P\left(R_{a\times b}\right) = N\cdot\left(\frac{8d}{3\pi}\left(a+b\right)-\frac{d^2}{2\pi}\right) \tag{5.23}$$

The same argument (and the same result) also stands for all the formulas developed in the next sections, when relaxing the uniformity assumptions. It will be further, shown in the experimental study that the applicability of the developed model is also large over commercial SDBMS.

5.4. Relaxing the Uniformity Assumptions

In this section we relax the three assumptions, $A_I - A_{III}$, made in the problem definition in Section 5.3. This will be done in a gradually increasing order. We first show how to support real-world, non-uniform uncertainty distributions thus relaxing A_I (Section 5.4.1), we then employ spatio-temporal histograms in order to relax A_{II} (Section 5.4.2), and, finally, show how such histograms can be augmented to relax A_{III} (Section 5.4.3).

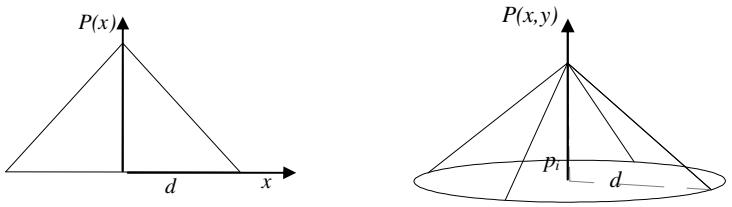
5.4.1. Relaxing the Uncertainty Uniformity Assumption

The analysis made in Section 5.3 was based on the uncertainty uniformity assumption, meaning that the actual position of each trajectory point at a given timestamp is uniformly distributed inside an uncertainty disk with the point representing the trajectory in the center and a known radius. Nevertheless, in this section we extend the proposed model towards non-uniform distributions of location uncertainty. The rationale behind this extension is that if the actual point $T_{i,k}^{\dagger}$ is located inside a circular neighbourhood of $T_{i,k}$, it is more likely that the probability of a location being the actual location of $T_{i,k}^{\dagger}$ decreases as its distance from $T_{i,k}$ increases. Even more, it is well-known that the error associated with GPS-tracked positions is normally distributed, i.e., following the *bivariate normal distribution* with uncorrelated variables x and y, which is the extension of the normal distribution in 2D space [Lei95]; given also that the usage of GPS allows for high sampling rates, the total error in such cases is dominated by the error introduced from the tracking device. As such, the argument that the uncertainty in real spatio-temporal (and stationary spatial) data tends to be normally distributed is well established [CC07], [NRB03], [PTJ05].

According to the previous discussion, the goal of this section is to relax the uniformity assumption in location uncertainty of moving objects and make the proposed model to support the bivariate normal distribution. The respective probability density function (*pdf*), when variables x, y are uncorrelated, is given by the following expression [PTJ05]:

$$P_{BN}(x,y) = \frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{5.24}$$

where σ^2 is the variance, along the x- and y- axis; then σ is the corresponding standard deviation. However, the computation of the respective formulas as done in Section 2 is a hard task since it involves the integration of several exponential functions.



(a) (b)

Figure 5.7: Uniform difference distribution *pdf*s in (a) 1D and (b) 2D space

On the other hand, the density function of the bivariate normal distribution can be efficiently approximated by the *two-dimensional uniform difference distribution* (2*d-UDD*), which is the extension of the *uniform difference distribution* in 2D space, i.e., the distribution of the difference between two uniformly distributed variables in [0, *d*]. The *pdf* of 2d-UDD is:

$$P_{2d-UDD}(x,y) = \frac{3}{\pi d^2} \cdot \begin{cases} 1 - \frac{1}{d}\sqrt{x^2+y^2} & if \ \sqrt{x^2+y^2} \le d \\ 0 & otherwise \end{cases} \tag{5.25}$$

which is the extension in 2D space of the uniform difference distribution with the following *pdf* :

$$P_{UDD}(x) = \frac{1}{d} \cdot \begin{cases} 1 - \frac{|x|}{d} & if \ |x| \le d \\ 0 & otherwise \end{cases} \tag{5.26}$$

Both distributions are illustrated in Figure 5.7 which exposes the silhouette of their *pdf*s; in fact, $P_{2d\text{-}UDD}$ forms a conical surface with base radius *d* and unit volume as illustrated in Figure 7(b).

Towards the reformulation of the proposed model, the uncertainty uniformity assumption must be replaced by the following *uncertainty uniformity difference assumption*: the actual position $T_{i,k}$ of each trajectory T_i at timestamp t_k is handled by $P_{2d\text{-}UDD}$ distribution described above. Based on this assumption, the following lemma is provided:

Lemma 5.3*: The average numbers $E_N(R_{k,a\times b})$ and $E_P(R_{k,a\times b})$ of false negatives and false positives, respectively, in the results of a timeslice query $W_j \in R_{k,a\times b}$ with half-sides of length a and b at timestamp t_k over a trajectory dataset that follows the data uniformity and uncertainty uniformity difference assumptions are given by the formula:*

$$E_N(R_{k,a\times b}) = E_P(R_{k,a\times b}) = N \cdot \left(\frac{2d}{\pi}(a+b) - \frac{3d^2}{10\pi} \right) \tag{5.27}$$

where d is the radius of the uncertainty disk.

Proof: $E_N(R_{k,a\times b})$ and $E_P(R_{k,a\times b})$ can be obtained from the average probabilities $AvgP_{i,N}(R_{k,a\times b})$ and $AvgP_{i,P}(R_{k,a\times b})$, respectively, multiplied by the total number *N* of objects in the data space. The probability of a trajectory T_i to be false negative or false positive, with respect to a query window W_j, at timestamp t_k is:

$$P\left(T_{i,k} \notin W_j \mid T_{i,k}^{\dagger} \in W_j\right) = \begin{cases} V_{i,j}, & if \ T_{i,k} \notin W_j \ and \ Dist\left(T_{i,k}, W_j\right) \le d \\ 0, & otherwise \end{cases} \tag{5.28}$$

respectively

$$P\left(T_{i,k} \in W_j \mid T_{i,k}^{\dagger} \notin W_j\right) = \begin{cases} V_{i,j}, & if \ T_{i,k} \in W_j \ and \ Dist\left(T_{i,k}, W_j\right) \le d \\ 0, & otherwise \end{cases} \tag{5.29}$$

where $V_{i,j}$ is the volume of the *2d-UDD pdf* $P_{2d\text{-}UDD}$, contained fully inside or outside W_j, respectively. The volume $V_{i,j}$, of the $P_{2d\text{-}UDD}$ being inside (outside, respectively) the query window is determined following the same methodology as in the proof of Lemma 5.1 (Lemma 5.2, respectively), taking also into account the uncertainty size assumption. In particular, bearing in mind that Figure 5.4(b) – (d) illustrate also the projection of $P_{2d\text{-}UDD}$ in the *x-y* plane, we can employ them in our discussion: in the two first cases (Figure 5.4(b) and 4(c)) where the distance between $T_{i,k}$ and each of the four corners of W_j is more than d, $V_{i,j}$ is equal to $V_{1x}(r_x,r_y)$ (or $V_{1y}(r_x,r_y)$) which is the portion of the $P_{2d\text{-}UDD}$ being above (or right of, respectively) the vertical plane passing from c_1c_2. In the third case, where the distance between $T_{i,k}$ and one of the four corners of W_j is less than d (Figure 5.4(d)), $V_{i,j}$ is equal to $V_2(r_x,r_y)$, which is the portion of the $P_{2d\text{-}UDD}$ being right of the vertical plane passing from c_1c_2, and above the one passing from c_2c_3.

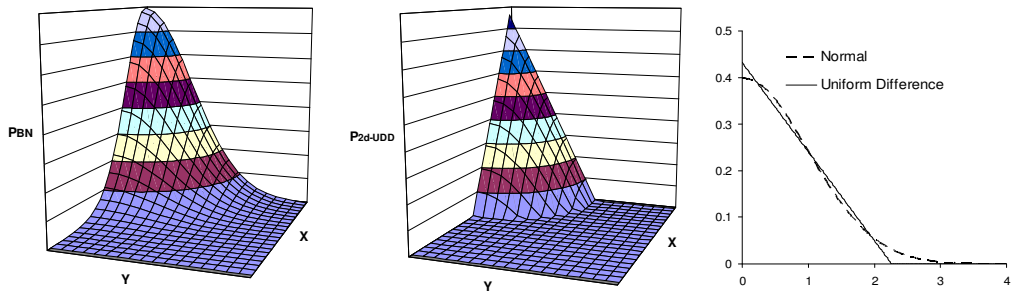
The average, with respect to any query window in $R_{k,axb}$, probability of a trajectory T_i to be false negative (false positive, respectively) at timestamp t_k is calculated by integrating Eq.(5.28) (Eq.(5.29), respectively) over all query positions as in Eq.(5.6) (Eq.(5.16), respectively). The corresponding integral is computed in the same way as the one followed in the proof of Lemma 5.1 (Lemma 5.2, respectively) by replacing the values of $A_{1x}(r_x,r_y)$, $A_{1y}(r_x,r_y)$ and $A_2(r_x,r_y)$ with $V_{1x}(r_x,r_y)$, $V_{1y}(r_x,r_y)$ and $V_2(r_x,r_y)$ into Eq.(5.10) (Eq.(5.20), respectively). Then, by substituting $\int_0^d V_{1y}\left(r_x,r_y\right)dr_y = \int_0^d V_{1x}\left(r_x,r_y\right)dr_x = \frac{d}{2\pi}$, and $\int_0^d \int_0^{\sqrt{d^2-x^2}} V_2\left(r_x,r_y\right)dr_y dr_x = \frac{3d^2}{40\pi}$ into the respective formulas and performing the necessary calculations we result in:

$$AvgP_{i,N}\left(R_{k,a\times b}\right) = \frac{2d}{\pi}\left(a+b\right) - \frac{3d^2}{10\pi} \tag{5.30}$$

and

$$AvgP_{i,N}\left(R_{k,a\times b}\right) = \frac{2d}{\pi}\left(a+b\right) - \frac{3d^2}{10\pi} \tag{5.31}$$

By multiplying the above formulas with N we have proven Lemma 5.3. ■



(a) (b) (c)

Figure 5.8: (a) Two-Dimensional UDD, (b) bivariate normal distribution and, (c) best fitting in a single dimension (c)

Up to this point, given that the distribution of the actual data point follows the uncertainty uniformity difference assumption, our model constitutes of Eq.(5.27), which is much alike the ones in Section 5.2 under the uncertainty uniformity assumption. In particular, when Eq.(5.27) is compared with Eq.(5.1) and Eq.(5.13), the formulas differ only in the weights of the function variables $d(a+b)$ and d^2. Although the model described above does not directly consider the bivariate normal

distribution, it can be used to efficiently approximate it. Consider, for example, Figure 5.8 that illustrates the probability function of the bivariate normal distribution with uncorrelated variables (Figure 5.8(a)), the probability function of the 2d-UDD (Figure 5.8(b)), and the silhouette of the two distributions in 1D space (Figure 5.8(c)); the two probability functions turn out to be close to each other. Hence, we can utilize least squares and estimate the radius of the cone which fits best in the Gaussian "bell" of the bivariate normal distribution.

Formally, the following lemma is provided:

Lemma 5.4: *The 2D uniform difference distribution which best approximates the bivariate normal distribution with uncorrelated variables, is taken when the radius d of the uncertainty disk is:*

$$d \approx 2.36533 \times \sigma \tag{5.32}$$

where σ is the standard deviation of the bivariate normal distribution along the x- and y- axis.

Proof: According to the Least Squares Theory, the best approximation of a function f by another function g in the same domain D is given by minimizing the integral $\iint_D \left(f(x)-g(x)\right)^2 dx$ of the square of their difference along D. Subsequently, in order to prove our Lemma, we have to determine the value of d that minimizes $\iint_{\square^2} \left(P_{2d-UDD}(x,y)-P_{BN}(x,y)\right)^2 dxdy$. Towards this goal, it holds that:

$$\iint_{\square^2} \left(P_{2d-UDD}(x,y)-P_{BN}(x,y)\right)^2 dxdy = \iint_{C(0,d)} \left(P_{2d-UDD}(x,y)-P_{BN}(x,y)\right)^2 dxdy + \iint_{\square^2-C(0,d)} \left(P_{2d-UDD}(x,y)-P_{BN}(x,y)\right)^2 dxdy \tag{5.33}$$

where $C(0,d)$ is the disk with center (0,0) and radius d. Applying Eq.(5.25) and Eq.(5.24) into Eq.(5.33) , we get:

$$\iint_{\square^2} (P_{2d-UDD}(x,y)-P_{BN}(x,y))^2 dxdy = \iint_{C(0,d)} \left(\frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} - \frac{3}{\pi d^2}\left(1-\frac{\sqrt{x^2+y^2}}{d}\right)\right)^2 dxdy + \iint_{\square^2-C(0,d)} \left(\frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} - 0\right)^2 dxdy \tag{5.34}$$

At this point, we utilize the Cartesian-to-Polar transformation, which transforms (x, y) to (ρ, θ) according to the following formula:

$$\iint f(x,y)\,dxdy = \iint f(\rho\cos\theta, \rho\sin\theta)\,\rho d\rho d\theta \tag{5.35}$$

Applying the above transformation to Eq.(5.34), we get:

$$\iint_{\square^2} \left(P_{2d-UDD}(x,y)-P_{BN}(x,y)\right)^2 dxdy = \int_0^{2\pi}\int_0^d \left(\frac{1}{2\pi\sigma^2} e^{-\frac{\rho^2}{2\sigma^2}} - \frac{3}{\pi d^2}\left(1-\frac{\rho}{d}\right)\right)^2 \rho\, d\rho d\theta + \int_0^{2\pi}\int_d^{\infty} \left(\frac{\rho}{2\pi\sigma^2} e^{-\frac{\rho^2}{2\sigma^2}}\right)^2 \rho d\rho d\theta$$

This results in

$$\iint_{\square^2} \left(P_{2d-UDD}(x,y)-P_{BN}(x,y)\right)^2 dxdy = \frac{d^3 - 18d\sigma^2 + 12\sqrt{2\pi}\sigma^3 \,\mathrm{Erf}\left[\frac{d}{\sqrt{2}\sigma}\right]}{4d^3\pi\sigma^2} \tag{5.36}$$

where Erf[x] is the error function encountered in integrating the normal distribution. In the sequel, we calculate the first derivative of Eq.(5.36) with respect to d:

$$\frac{\partial \iint_{\square^2} \left(P_{2d-UDD}(x,y)-P_{BN}(x,y)\right)^2 dxdy}{\partial d} = -\frac{9d + 6de^{-\frac{d^2}{2\sigma^2}} - 9\sqrt{2\pi}\,\mathrm{Erf}\left[\frac{d}{\sqrt{2}\sigma}\right]}{d^4\pi} \tag{5.37}$$

and by substituting d/σ with a variable a ($a \neq 0$), we result in the following expression:

$$\frac{\partial \iint_{\square^2} \left(P_{2d-UDD}(x,y) - P_{BN}(x,y)\right)^2 dxdy}{\partial d} = \frac{9a + 6ae^{-\frac{a^2}{2}} - 9\sqrt{2\pi}\,\mathrm{Erf}\left[\frac{a}{\sqrt{2}}\right]}{ad^3\pi} \tag{5.38}$$

which is zeroed when the numerator becomes zero. Hence, the first derivative of Eq.(5.36) is zeroed when

$$9a + 6ae^{-\frac{a^2}{2}} - 9\sqrt{2\pi}\,\mathrm{Erf}\left[\frac{a}{\sqrt{2}}\right] = 0 \tag{5.39}$$

After numerically evaluating Eq.(5.39) it turns out that

$$a \approx 2.36533 \tag{5.40}$$

Recalling that $a = d/\sigma$ we have proven Lemma 5.4 ■

Concluding, the proposed model for normally distributed uncertainty constitutes of Eq.(5.27) and Eq.(5.32); the value of d provided by Eq.(5.32) can be directly used as input in Eq.(5.27) in order to approximate the normal distribution quite effectively, as it will be shown later in the experimental study.

5.4.2. Relaxing the Data Uniformity Assumption

Sections 5.3 and 5.4.1 assumed that trajectories, and consequently, snapshot data points, are uniformly distributed in the data space. In this section, we relax the data uniformity assumption and apply the proposed approach to arbitrarily distributed data with the employment of *histograms* [Ioa93], [IP95]. Histograms have been widely used in query optimization issues, such as spatial and spatio-temporal selectivity estimation [APR99], [TS96], [CC02], [HKT03], [TSP03], in order to overcome similar assumptions made when estimating the number of disk page accesses required to answer a query. The background idea is that when data are included in a small space, they may be considered as uniform even though the distribution of the entire dataset may differ significantly. The goal therefore when using histograms, is to break down the space into small regions, called *buckets*, which can be assumed to contain uniform data. Among the schemes proposed, we adopt the concept of [APR99], since it can be modified in a simple way in order to apply it in our requirements.

In particular, Acharya et al. [APR99] present several space partitioning techniques for the construction of spatial histograms utilized in selectivity estimation of range queries. Among them, the *MinSkew* technique has been shown to provide the most accurate selectivity estimates for spatial queries. *MinSkew* is a binary space partitioning (BSP) technique employing the *spatial skew* definition, also provided in [APR99]. More specifically, *the spatial skew of a bucket is the statistical variance of the spatial densities of all points grouped within this bucket, and the spatial skew of the entire set is the weighted sum of spatial skews of all buckets*. The proposed technique initially uses a uniform grid of regions and their spatial densities as input; as such it produces a compact approximation on the input data in place of the original in order to build the histogram (grouping) in memory. Then, the construction algorithm repeatedly partitions the given set of regions such that the spatial-skew is minimized at each step until no more buckets are available for the histogram. Since it always partitions an existing region into two, the result is a BSP partitioning. As a result, the constructed histogram H is the set of n buckets $H = \{B_i : \bigcup(B_i) = S \wedge \bigcap(B_i) = \varnothing\}$ and $B_i = \{[x_{i,L}, x_{i,U}], [y_{i,L}, y_{i,U}]\}$ .The main

advantage of this technique is that the initial cells grouped together within the same bucket have small spatial skew, i.e., variance. It is therefore expected that the cells contained inside each bucket should enclose approximately the same number of data points; as a result, it is usually assumed that the data distribution inside each bucket B_i is uniform. Actually, this assumption, as demonstrated both in [APR99] and in our experiments, is rather reasonable even in the presence of totally skewed spatial data, such as the *Charminar dataset* [APR99].

The main usage of spatial histograms is to provide estimates for the *local density* of the dataset, given a spatial region. Towards this goal, the buckets that overlap the spatial query are initially determined, and then, the local density is calculated by producing the weighted average of the overlapped buckets densities N_i. This happens by weighting density N_i of each bucket B_i with the corresponding area A_i that partially covers the given region, normalized by the total area:

$$N' = \frac{1}{4ab} \sum_{i=1..n} (N_i \cdot A_i) \tag{5.41}$$

In the followings, it is showed how to appropriate modify the MinSkew histogram structure so as to support spatio-temporal timeslice queries.

5.4.2.1. Spatio-temporal Histograms for Time-slice Queries

The first step towards the construction of a spatio-temporal histogram that supports the estimation of the selectivity of timeslice queries, is to augment the spatial data space initially used by [APR99] in order include the temporal dimension. As such, the proposed histogram is $H = \{B_i : \bigcup(B_i) = S \wedge \bigcap(B_i) = \varnothing\}$ and $B_i = \{[x_{i,L}, x_{i,U}], [y_{i,L}, y_{i,U}], [t_{i,L}, t_{i,U}]\}$. The basic idea that allows us to use MinSkew partitioning for our purposes is summarized as following: apply a uniform grid of n intervals on each spatial dimension that forms n^2 spatial regions G_i, repeat it at several uniformly distributed timestamps t_k (k = 1...*now* – 1), and compute the number of trajectories $m_{i,k}$ found inside each G_i on every t_k, . If a quite large number of t_k is used, then the number of trajectories found inside region G_i at any timestamp during the period $[t_k, t_{k+1}]$ can be considered to be equal with $m_{i,k}$; in other words, trajectories can be considered stationary between t_k and t_{k+1} if the length of period $[t_k, t_{k+1}]$ is small enough. Then, we can straightforwardly use the construction algorithm of [APR99] applying the same set of heuristics in order to minimize the spatial skew of the constructed buckets: the goal again is to group together several grid regions G_i at several timestamps t_k having similar values of $m_{i,k}$, thus resulting in a grouping with as smaller spatial skew as possible.

Finally, the local density N' of a timeslice query invoked at timestamp t_k is calculated by producing the weighted average of the overlapped buckets densities N_i being at the same time valid at this particular timestamp:

$$N' = \frac{1}{4ab} \sum_{i:t_{i,L} \le t_k < t_{i,U}} (N_i \cdot A_i) \tag{5.42}$$

Eq.(5.42) expresses the fact that the local density N' is calculated by weighting density N_i of each bucket B_i with the corresponding area A_i that partially covers the given region, normalized by the total area.

5.4.2.2. Estimating the Effect of Uncertainty Using Spatio-temporal Histograms

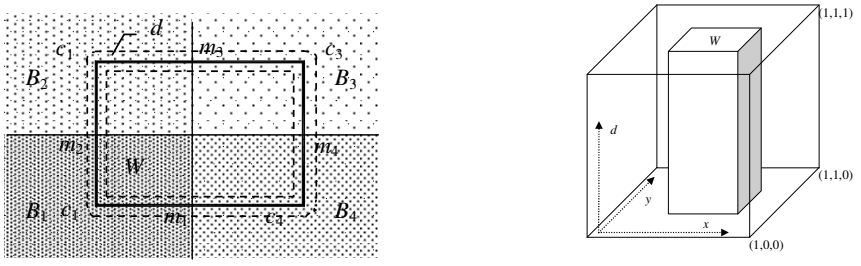
Moving into our core problem, *MinSkew* spatio-temporal histograms can be utilized in order to apply our analysis in non-uniform data and estimate the error introduced in the query results without actually executing the query. Specifically, two alternative approaches are proposed for estimating $E_P(R_{k,axb})$ and $E_N(R_{k,axb})$. The first one is to simply use the estimate of the local density produced by the spatio-temporal histogram in place of the total density employed in the proposed model; this can be achieved by evaluating Eq.(5.1), Eq.(5.13) or Eq.(5.27) using the local density N', derived from Eq.(5.42), instead of the overall space density N.

As an alternative approach, instead of computing a global local density N' for the total timeslice query window, we may consider the different contributions of the query window sides and query window corners in the total number of false hits, as discussed in Section 5.3.3. Therefore, given a spatio-temporal histogram containing n disjoint spatio-temporal buckets B_i, the estimation of the number of false positives and false negatives in the results of a timeslice query invoked at timestamp t_k under the uncertainty uniformity assumption, can be determined using the formula:

$$E_P\left(R_{k,a\times b}\right) = E_N\left(R_{k,a\times b}\right) = \sum_{i:t_{i,L}\le t_k<t_{i,U}} \left(N_i \cdot \left(\frac{2d}{3\pi} L_i - \frac{d^2}{8\pi} \cdot s_i \right) \right), \tag{5.43}$$

where L_i is the length of the part of the query perimeter that spatially overlaps B_i and s_i is the number of timeslice query window corners being inside B_i.

Eq. (5.43) formulates the fact that the total number of false negatives or positives is the summation of the contributions of the different query components as discussed in Section 2.3. More specifically, the $\frac{2d}{3\pi}L_i$ part of Eq.(5.43) is derived from the $\frac{8d}{3\pi}(a+b)$ of Eq.(5.1) and Eq.(5.13), multiplied by the length of the query perimeter L_i that overlaps bucket B_i and divided by the total query length $4(a+b)$; in the same manner, the $\frac{d^2}{8\pi}s_i$ part of Eq.(5.43) is the transformation of the $\frac{d^2}{2\pi}$ part of Eq.(5.1) and Eq.(5.13), multiplied by the actual number of query window corners s_i spatially inside bucket B_i, divided by their total number, i.e., 4.



(a) (b)

Figure 5.9: (a) A timeslice query window over of a snapshot of a spatio-temporal histogram (b) A timeslice query window over a snapshot of the augmented 4-D space.

Consider, for example, Figure 5.9(a) that illustrates the snapshot of a timeslice query window W at timestamp t_k, overlapping at this particular timestamp four histogram buckets (B_1 … B_4). Since false hits may only be found close to the boundary of W, the number of false positives or negatives on

bucket B_1 depends on the length of the query perimeter that overlaps it, that is, the length of lines $|m_1c_1|+|c_1m_2|$ and the number or corners s_1=1. It is also worth to note that using the above procedure, the query window is not dissected across the histogram buckets' boundaries, as such an approach would increase the total perimeter and consequently decrease the accuracy of the model. Moreover, in the *2d-UDD* uncertainty distribution case, the formula for estimating the number of false positives and false negatives is:

$$E_P\left(R_{k,a\times b}\right)=E_N\left(R_{k,a\times b}\right)=\sum_{i:t_{i,L}\le t_k<t_{i,U}}\left(N_i\cdot\left(\frac{d}{2\pi}L_i-\frac{3d^2}{40\pi}\cdot s_i\right)\right), \tag{5.44}$$

The above formula is derived counting the different contributions of the query sides and corners of Eq.(5.27) in a way similar with the above. In particular, the $\frac{d}{2\pi}L_i$ part of Eq.(5.44) is computed by multiplying the $\frac{2d}{\pi}(a+b)$ of Eq.(5.27) by the part of the query perimeter L_i that spatially overlaps bucket B_i, divided by the total query length 4(a+b), while, the $\frac{3d^2}{40\pi}s_i$ part of Eq.(5.44) is obtained by multiplying the $\frac{3d^2}{10\pi}$ part of Eq. (5.27) by the actual number of query window corners s_i spatially inside bucket B_i, divided by their total number, i.e., 4.

The same methodology can be applied to any bucket–based data storage scheme containing summary information, such as data cubes in trajectory data warehouses (TDW). Since a trajectory data cube consists of disjoint spatio-temporal buckets, i.e., the base cuboids, along with summary information, Eq.(5.43) and Eq.(5.44), depending on the type of uncertainty distribution, can be applied in OLAP operations and produce an estimation for the total number of false positives or false negatives. For example, when aggregating from the *cell* to the *city* level as discussed in the introduction, i.e., performing a roll-up operation, the MBB of a city can be considered as a query window and be used to estimate the false hits introduced in the aggregation. Given, however, that the density between the boundary of the actual city and its MBB can be much different, the N_i involved in Eq.(5.43) or Eq.(5.44) should be determined by using the actual perimeter of the city polygon in place of its MBB, and the L_i lengths should be weighted accordingly using the MBB and the polygon perimeter. This approach will be tested in the following sections regarding simple spatial data, and it will be shown to produce very good estimations.

5.4.3. Relaxing the Constant Uncertainty Radius Assumption

The third extension of the model presented in this thesis in order to support real-world application scenarios, is to deal with datasets of trajectories having different values of uncertainty radius or standard deviation for each one of them. Consider, for example, m sets P_j containing N_j trajectories each, obtained by using different positioning technologies, such as GPS, Wi-Fi positioning, etc. Then, the union of all sets $P=\bigcup_{i=1\ldots m}\{P_j\}$ contains trajectories having several uncertainty radiuses depending on each trajectory's original data source. A straightforward approach in order to determine

the error E_P or E_N introduced in the results of a timeslice query over P, is to calculate the specific errors $E_{P,j}$ or $E_{N,j}$ for each one P_j separately and then summarize the resulted errors. More formally,

$$E_P\left(R_{a\times b}\right)=\sum_{j=1..m}E_{P,j}\left(R_{a\times b}\right)\ \text{and}\ E_N\left(R_{a\times b}\right)=\sum_{j=1..m}E_{N,j}\left(R_{a\times b}\right) \tag{5.45}$$

Such an approach would reasonably be successful when dealing with uniformly distributed data. However, when dealing with real-world, usually skewed data, the methodology provided in the previous section should be applied, meaning that we would have to maintain m different histograms, one for each different possible value of uncertainty radius. Nevertheless, in this thesis a more sophisticated solution is provided to the above challenge. Specifically, we may further augment the spatio-temporal histogram proposed in section 5.4.2.1, with the uncertainty radius considered as the fourth dimension. In other words, we propose to use the MinSkew histogram in the normalized 4D space formed by the two spatial dimensions, the temporal one, and the length of the uncertainty radius d.

More formally, the proposed histogram is $H=\left\{B_i:\bigcup\left(B_i\right)=S\times[0,1]\wedge\bigcap\left(B_i\right)=\varnothing\right\}$ and $B_i=\left\{\left[x_{i,L},x_{i,U}\right],\left[y_{i,L},y_{i,U}\right],\left[t_{i,L},t_{i,U}\right],\left[d_{i,L},d_{i,U}\right]\right\}$. It is built by applying a uniform grid in $S\times[0,1]$ and counting the number of data points found inside each cell in the 4D space, and then, recursively partitioning the data space, minimizing the total spatial skew at each step. Following the respective discussion of the previous section regarding simple spatial histograms, it is assumed that the data distribution inside each 4D bucket B_i is uniform. Then, the estimation of the number of false hits can be easily calculated in the case of the uncertainty uniformity assumption as follows:

$$E_P\left(R_{a\times b}\right)=E_N\left(R_{a\times b}\right)=\sum_{i:t_{i,L}\le t_k<t_{i,U}}\left[\frac{N_i}{d_{i,U}-d_{i,L}}\cdot\int_{d_{i,L}}^{d_{i,U}}\left[\left(\frac{2d}{3\pi}L_i-\frac{d^2}{8\pi}s_i\right)dd\right]\right], \tag{5.46}$$

where L_i is the length of the query perimeter that overlaps bucket B_i in the two spatial dimensions, s_i is the number of query window corners being inside bucket B_i, and $d_{i,L}$, $d_{i,U}$ are the lower and upper values of the third dimension d in B_i, respectively. Eq.(5.46) is directly derived when integrating Eq.(5.43) over all possible values of d in the data space, bearing also in mind that the actual number of objects found at each slice of the third dimension is $N_i/\left(d_{i,U}-d_{i,L}\right)$ and $\left(d_{i,U}-d_{i,L}\right)$ is the bucket's extent along this dimension. Intuitively, the above two formulas express the fact that the total error is the summation of the errors encountered on each histogram bucket the query window boundary overlaps; moreover, in this case, the spatial component of the query window W is also augmented in the dimension of d, forming a box entirely covering this dimension, as illustrated in Figure 5.9(b). Finally, Eq.(5.46), after the necessary calculations turns into:

$$E_P\left(R_{k,a\times b}\right)=E_N\left(R_{k,a\times b}\right)=\sum_{i:t_{i,L}\le t_k<t_{i,U}}\left[N_i\cdot\left(\frac{d_{i,U}+d_{i,L}}{3\pi}L_i-\frac{d_{i,U}^2+d_{i,L}^2+d_{i,L}d_{i,U}}{24\pi}s_i\right)\right], \tag{5.47}$$

Following a similar approach, the estimation of the number of false hits in the case of the uncertainty uniformity difference assumption is calculated as:

$$E_P\left(R_{k,a\times b}\right)=E_N\left(R_{k,a\times b}\right)=\sum_{i:t_{i,L}\le t_k<t_{i,U}}\left[N_i\cdot\left(\frac{d_{i,U}+d_{i,L}}{4\pi}L_i-\frac{d_{i,U}^2+d_{i,L}^2+d_{i,L}d_{i,U}}{40\pi}s_i\right)\right]. \quad (5.48)$$

The proposed approach has two basic advantages regarding the alternative of maintaining different histograms for the *m* sets of trajectories; the first is that the space requirements are sufficiently reduced, especially in the case where the number of different uncertainty radiuses increases significantly. However, the most important advantage of this proposal is revealed bearing in mind that data belonging to the same class may have different accuracy; for example the uncertainty due to GPS depends on a large number of parameters, such as the number of visible satellites, the frequency interference, and the satellite signal reflection in large glass surfaces inside urban areas, resulting in a different uncertainty radius for each individual sampled point of each trajectory; the naïve approach could not fulfil such requirements since we would have to maintain a separate histogram for each possible value of uncertainty radius. On the other hand, our proposal can absorb these necessities and handle an unrestrained number of different radiuses without increasing the memory space requirement of the constructed histogram, producing at the same time a very good estimation.

5.5. Experimental Study: Spatio-temporal Data

In this section several experiments are presented in order to demonstrate the correctness and accuracy of the previous analysis using synthetic trajectory datasets. In the experimental study that follows we demonstrate the accuracy of the analytical model under uniform distribution of uncertainty with the aid of spatio-temporal histograms (Eq.(5.43)), as well as its sensitivity with respect to the involved parameters, i.e., the uncertainty radius and the length of the query perimeter.

Here it is worth to note that for typical query and uncertainty sizes (e.g., queries of 0.05×0.05 to 0.30×0.30 in the unit space, and uncertainty radius set to 0.01), the formulas of the proposed model produce values of false negatives / positives between 0.0004×*N* and 0.0025×*N*, meaning that for 2000 trajectories we expect between 0.8 and 5.0 trajectories as false positives / negatives per query. As such, it becomes clear that for typical query sizes and uncertainty radiuses, the dataset population should be quite large in order to produce a significant number of false hits, so as to be counted and compared against the results of the proposed model. However, since the cardinality of trajectory datasets is usually small (on the other hand, their actual size may become huge as time evolves), the details of the developed model using all possible setting combination, will be tested using synthetic and real spatial datasets in the next section.

5.5.1. Experimental Setup

The experimental study over spatio-temporal data is based on the NG synthetic datasets (section 1.5.3). Each trajectory was modelled as a cilyndrical volume [TWHC04], following therefore the uniform distribution of uncertainty assumption. During each experiment the dataset was queried with 1000 randomly distributed square, i.e., with *a*=*b*, timeslice queries. Each query initially retrieved the interpolated position of each trajectory in the dataset at the timestamp determined by it, and then, the assumption of [TWHC04] was used in order to reveal the actual position of each moving object at this particular timestamp. As such, a number of false negatives and false positives were generated, since the query results gathered by the first step were different than the ones determined after the second step.

We used for the estimation of the same number our analytical model expressed by Eq.(5.43), that is, with the aid of spatio-temporal histograms, as presented in the previous section (thus, relaxing assumption A_{II}); the *MinSkew* partitioning of the dataset under consideration was created using a uniform grid of original grid size set to 0.005×0.005×0.005, as discussed in [APR99]. The radius of the cilyndrical volume (uncertainty radius) was scaled between 0.0005 and 0.02, while each square query side's length was scaled between 0.06 and 0.36; elongated query windows reported similar behavior. We conducted our experiments on a Windows XP workstation with AMD Athlon 64 3GHz processor CPU, 1 GB of main memory and several GB of disk space.

5.5.2. Experimental Results

Two statistical measures were used so as to demonstrate the behavior of our model. The *average number of false negatives and false positives*, $\overline{E_N}$ and $\overline{E_P}$, respectively, and the *average absolute error in the estimation of false negatives and false positives in each individual query*, $\overline{ES_N}$ and $\overline{ES_P}$, respectively. Formally, these measures are defined as:

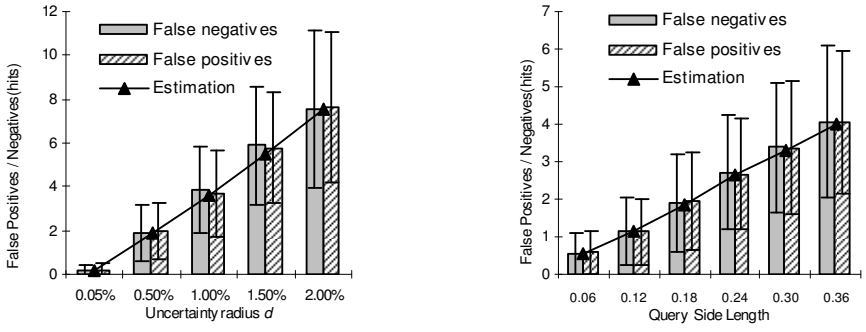
$$\overline{E_N} = \frac{1}{n}\sum_{i=1..n} E_{N,i}\ ,\ \overline{E_P} = \frac{1}{n}\sum_{i=1..n} E_{P,i} \tag{5.49}$$

and,

$$\overline{ES_N} = \frac{1}{n}\sum_{i=1..n} \left|E_{N,i} - E_N\left(R_{k,a\times b}\right)\right|,\ \overline{ES_P} = \frac{1}{n}\sum_{i=1..n} \left|E_{P,i} - E_P\left(R_{k,a\times b}\right)\right| \tag{5.50}$$

where n is the number of executed queries and $E_{P,i}$ ($E_{N,i}$) the actual number of false positives (false negatives, respectively) in the i-th query. We distinguish between, e.g. $\overline{E_P}$ and $\overline{ES_P}$, in order to uncover the details of the behavior of our model, as will be shown in the following experiments.

In the first series of experiments the synthetic dataset is utilized in order to demonstrate the accuracy and the behavior of the presented analytical model scaling the two influencing factors: the radius d of the uncertainty disk and the size (a, b) of the query window. Note that in all figures the query size is exposed in terms of its side length $2a = 2b$, e.g., for query side length 0.30, the size of the query window is equal to $0.30 \times 0.30 = 0.09$ of the unit space.



(a) (b)

Figure 5.10: Average false negatives / positives and their estimations scaling with (a) d and (b) the query size (synthetic data – uniform distribution of uncertainty).

In particular, in the first experiment the value of d is scaled between 0.05% and 2% of the space extent along the x- and y- axis, querying the synthetic dataset, with fixed side length 0.18 (i.e., $a = b =$

0.09 resulting in a query window sized 3.24% of the data space). The results of this experiment are illustrated in Figure 5.10(a); as a first result, the number of false positives and false negatives turn out to be almost equal, verifying the correctness of the corollary in Eq.(5.22). Moreover, the estimations $E_P(R_{k,a\times b})$ and $E_N(R_{k,\times b})$ are very accurate with respect to $\overline{E_P}$ and $\overline{E_N}$, with the error being always below 6%, whereas the error bars in each graph column, illustrating $\overline{ES_P}$ and $\overline{ES_N}$, demonstrate low to medium values. Specifically, the average error in individual queries is around 40% in the vast majority of the experimental settings while it increases significantly only in the extreme case where the uncertainty radius d is set to its minimum (d = 0.05%).

Similar results are exposed in the second experiment, illustrated in Figure 5.10(b), where the query size is scaled. In particular, the uncertainty radius is set to 0.5%, and the length of the query side is scaled between 0.06 and 0.36, resulting in query sizes covering between 0.36% and 12.96% of the data space. When comparing the estimation of the number of false negatives and false positives, and the respective average values $\overline{E_P}$ and $\overline{E_N}$, the reported estimation error is again below 6%, regardless of the query size, while the error bars in each graph column (i.e., $\overline{ES_P}$ and $\overline{ES_N}$) show the same trend as previous being around 40%; again the only case where they reach high values, occurs when both σ and the query size were set to their minimum values.

While at a first thought these values of $\overline{ES_P}$ and $\overline{ES_N}$ may be considered as high ones, it has to be pointed out that the error of the estimation is lowered significantly as the cardinality of the dataset increases, a fact that will be demonstrated over simple spatial data in the next section. Finally, in order to justify the accuracy of the estimations, we have to indicate that the values of $\overline{ES_P}$ and $\overline{ES_N}$ never exceed 2 false hits in absolute values (e.g., actual vs. estimated false negatives : 6 vs. 8).

5.6. Experimental Study: Spatial Data

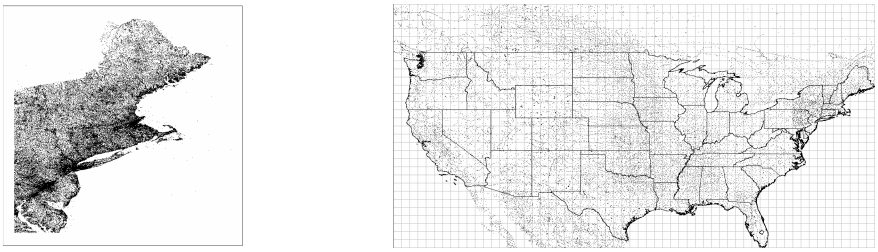
In this section, following the previous experiments on spatio-temporal trajectory data, we present a series of experiments using synthetic and real spatial data so as reveal all the details of the proposed model, over datasets with medium cardinality (nevertheless, significantly greater than the one of the experimental study of spatio-temporal data), as well as the efficiency of the proposed solutions. Concisely, the objectives of the experimental study that follows are to:

- demonstrate the accuracy of the simple analytical model (Eq.(5.1) and Eq.(5.13)), as well as its sensitivity with respect to the involved parameters, i.e., the uncertainty radius, or standard deviation, and the length of the query perimeter.
- show the quality of the approximation of normally distributed location uncertainty by *2d-UDD* utilizing the model supported by Eq.(5.27) and Eq.(5.32)
- present the accuracy of the estimation provided by the analytical models – Eq.(5.43), Eq.(5.44), Eq.(5.47), and Eq.(5.48) – over real spatial data utilizing histograms and also demonstrate their advantage to the alternative of utilizing the histogram as a local density estimator using Eq.(5.41),

- show how the proposal of this chapter can be used in the context of spatial data warehouses, and,
- reveal the efficiency of the provided solutions implemented on top of a commercial SDBMS.

5.6.1. Experimental Setup

The experimental study of this section over spatial data is based on both synthetic and real point datasets. Specifically, the employed datasets are as follows: a synthetic dataset (Rnd_0) of 100K 2D points randomly distributed in the unit data space as well as two real datasets, namely, the North East (*NE*) and the Digital Chart of the World (*DCW*) datasets, illustrated in Figure 5.11(a) and (b), respectively.



(a) (b)

Figure 5.11: Real datasets: (a) North East and (b) Digital Chart of the World

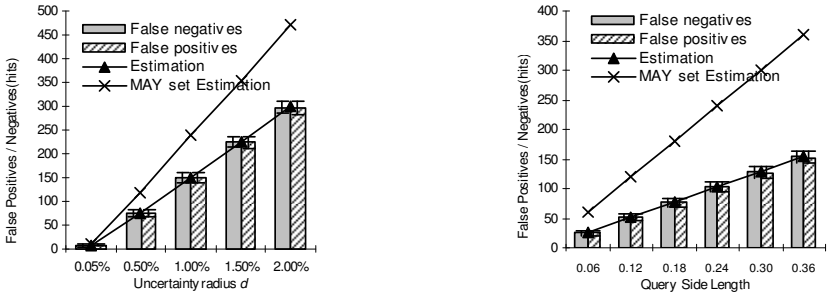
Then, as suggested by [BS03], [CZBP06], [GL05], in each dataset point is added noise in a controlled way. In particular, the location of each point in all three datasets is modified by adding noise, either uniformly distributed inside an uncertainty disk of radius d , producing the respective *U-d* dataset, or following a bivariate normal distribution with standard deviation σ, producing the respective *N-σ* dataset; for each *U-d* and *N-σ* dataset, we produced five different datasets that is $Rnd_{U\text{-}d\text{-}1}$, to $Rnd_{U\text{-}d\text{-}5}$, $NE_{U\text{-}d\text{-}1}$ to $NE_{U\text{-}d\text{-}5}$, and $DCW_{U\text{-}d\text{-}1}$ to $DCW_{U\text{-}d\text{-}5}$, and also the same five datasets for each one of the $Rnd_{N\text{-}\sigma}$, $NE_{N\text{-}\sigma}$, $DCW_{N\text{-}\sigma}$ cases. In order also to test the accuracy of our estimations under the settings of Section 3.3, we produced the $NE_{N\text{-}v0.02}$ dataset on which we have added noise following the bivariate normal distribution with σ varying between 0 and 0.02. Unless otherwise indicated, all experimentations involving spatial queries were performed by running 1000 randomly distributed square, i.e., with $a=b$, queries over all five datasets of the respective case; elongated query windows reported similar behavior. All experiments are conducted on a Windows XP workstation with AMD Athlon 64 3GHz processor CPU, 1 GB of main memory and several GB of disk space; all evaluated methods were implemented on both VB.NET and PostgreSQL 8.2 [Post08a] with the PostGIS 1.2.1 [Post08b] extension using the PL/PgSQL language.

5.6.2. Experiments on the Quality

Following from the experimental study over spatio-temporal data, in this section we also utilize the average number of false negatives and false positives, $\overline{E_N}$ and $\overline{E_P}$ (Eq.(5.49)), as well as the average absolute error in the estimation of false negatives and false positives in each individual query, $\overline{ES_N}$ and $\overline{ES_P}$ (Eq.(5.50)).

5.6.2.1. *Experiments over Synthetic Data Following all three Original Assumptions A_I, A_{II}, A_{III}*

In the first series of experiments the synthetic datasets are utilized in order to demonstrate the accuracy and the behavior of the analytical model scaling the two influencing factors as already done in the previous section regarding spatio-temporal data. In the first experiment the value of d was scaled between 0.05% and 2% of the space extent along the x- and y- axis, querying both Rnd_0 and the respective $Rnd_{u\text{-}d}$ dataset, with fixed side length 0.18. The results of this experiment are illustrated in Figure 5.12(a); as a first result, the estimations $E_P(R_{a\times b})$ and $E_N(R_{a\times b})$ are extremely accurate with respect to $\overline{E_P}$ and $\overline{E_N}$, with the error being always below 3%, whereas the error bars in each graph column, illustrating $\overline{ES_P}$ and $\overline{ES_N}$, are shown to be relatively low. Specifically, the average error in individual queries is below 10% in the vast majority of the experimental settings and is up to 29% in a single extreme case where the uncertainty radius d is set to its minimum (d = 0.05%). It is therefore confirmed the initial intention of the experimental study over spatial datasets, that is, to demonstrate that the estimations produced by the proposed analytical model over spatial datasets of medium cardinality are much better than the ones produced over datasets of small cardinality (as the ones used in the previous section).



(a) (b)

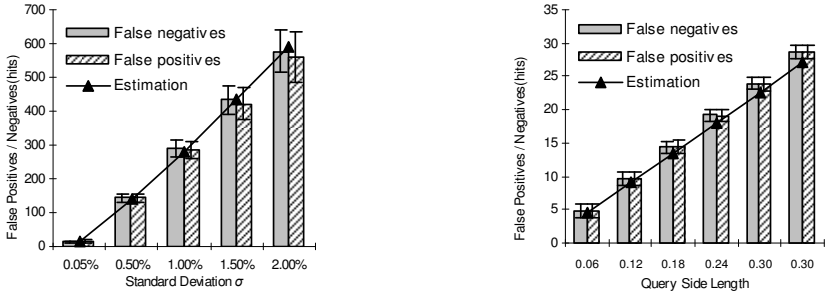
Figure 5.12: Average false negatives / positives and their estimations scaling with (a) d and (b) the query size (synthetic data – uniform distribution of uncertainty).

In the same experiments the methodology provided by [YM03], which estimates the cardinality of the *MAY* set, was also included. As already stated, the *MAY* set is actual a superset containing, among others, the false hits calculated by our analysis; nevertheless, we evaluate the assumption that 50% of the *MAY* set are false hits, that is, an object in the *MAY* set maybe either true or false hit with the same probability. However, as illustrated in Figure 5.12 by the *MAY set estimation* curve, the above assumption does not result in correct estimations. It is worth to note, however, that the goal of the analysis presented in [YM03] is not to provide the number of false hits the way our analysis does. Our assumption regarding the portion of the *MAY* set encountering false hits, i.e., the 50%, is used due to the lack of any other suggestions on this subject included in [YM03]. Moreover, Figure 5.12(a) could also lead to the presumption that a simple multiplier on the *MAY* set estimation, i.e., lowering the corresponding curve of Figure 5.12, could force it to produce better results. Still, in order to determine this multiplier, it is the methodology provided in the presented analysis that should be followed.

Similar results are exposed in the second experiment, illustrated in Figure 5.12(b), where the query size is scaled. In particular, the uncertainty radius is set to 0.5%, and the length of the query side is scaled between 0.06 and 0.36, resulting in query sizes covering between 0.36% and 12.96% of the data space. When comparing the estimation of the number of false negatives and false positives, and the respective average values $\overline{E_P}$ and $\overline{E_N}$, the reported estimation error is below 1%, regardless of the query size. Furthermore, the estimation based on the *MAY* set cardinality, once again could not yield on comparable results; as such, based on the observation that this estimation systematically overestimates $\overline{E_P}$ and $\overline{E_N}$, it will be excluded from the rest of the experimental study. Regarding the error bars in each graph column, illustrating the respective $\overline{ES_P}$ and $\overline{ES_N}$, they are relatively small in the majority of the experiments being below 16%; the only case where it reached higher values, i.e., 35%, occurred when both σ and the query size were set to their minimum values.

5.6.2.2. Experiments over Synthetic Data Relaxing Assumption A_1

In order to evaluate the accuracy of the estimation of the number of false positives and false negatives calculated by Eq.(5.27), and Eq.(5.32), a similar experimentation was performed with the $Rnd_{N\text{-}\sigma}$ datasets where σ and the query size were scaled. The results of these experiments are illustrated in Figure 5.13 and it is clear that the estimation error regarding $\overline{E_P}$ and $\overline{E_N}$ is always below 5%. Moreover, the respective error bars, illustrating $\overline{ES_P}$ and $\overline{ES_N}$, are shown to be relatively small, being usually below 12 %, while reaching 36% only in the case where both *d* and the length of the query side were set to their minimum values.

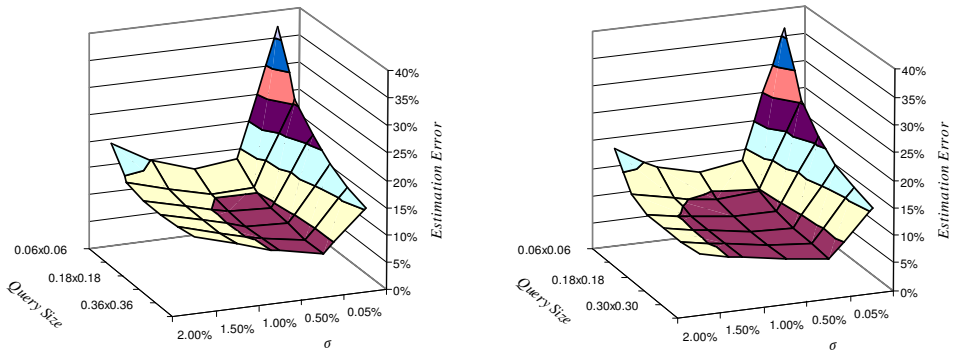


(a) (b)

Figure 5.13: Average false negatives, positives and estimation scaling (a) with σ and (b) with the query size (synthetic data - normal distribution of uncertainty).

A more detailed presentation of the average estimation error in each individual query $\overline{ES_P}$ and $\overline{ES_N}$ is illustrated in Figure 5.14(a) and (b), as a percentage of the number of false positives and false negatives, respectively. Both figures illustrate that $\overline{ES_P}$ and $\overline{ES_N}$ vary from small values, i.e., less than 10% for high values of σ, to higher ones for very small values of σ. They also depend on the query size, increasing as the size decreases. In general, it appears that $\overline{ES_P}$ and $\overline{ES_N}$ are essentially ruled by the standard deviation σ and, at a smaller extent, on the query size. Furthermore, for small values of σ and

small query sizes, while the estimation is still accurate regarding $\overline{E_P}$ and $\overline{E_R}$ (Figure 5.13(a) and (b), respectively), $\overline{ES_P}$ and $\overline{ES_N}$ increase significantly up to 40%.



(a) (b)

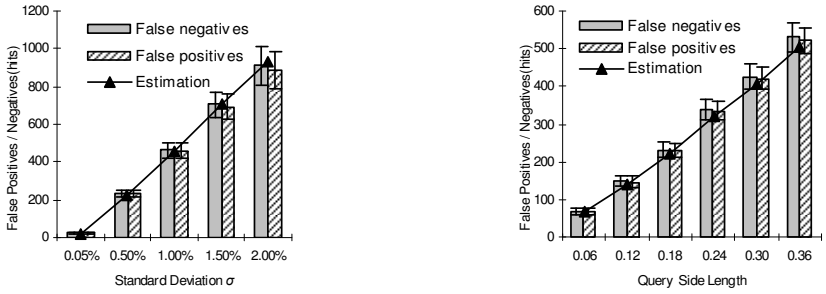
Figure 5.14: Average estimation error of (a) false positives $\overline{ES_P}$ and (b) false negatives $\overline{ES_N}$, in each query, scaling with *d* and the query size (synthetic data – normal distribution of uncertainty).

5.6.2.3. Experiments over Real Data Relaxing Assumption A_{II}

In order to support real, arbitrarily distributed spatial data by employing histograms, the *NE* dataset along with the respective $NE_{N\text{-}\sigma}$ datasets were employed. Subsequently, the *MinSkew* partitioning of each modified dataset was created using a uniform grid of original grid size set to 0.001×0.001, as discussed in [APR99]. The experiments over the $NE_{U\text{-}d}$ datasets, i.e., with uniform uncertainty distribution, reported similar behavior and thus are omitted. In particular, in order to evaluate the accuracy of the analysis of section 5.4.2, i.e., the estimation of false negatives and false positives using Eq.(5.44), the *NE* and $NE_{N\text{-}\sigma}$ datasets were used for experimentation, first scaling σ with query size fixed to 0.18×0.18, and then, scaling the query size with σ fixed to 0.5%.

Figure 5.15 illustrates the actual and estimated values of false negatives and false positives using the above experimental settings. Clearly, the estimations are accurate with the reported error being always lower than 6%. Additionally, the average absolute error of the estimation in each individual query $\overline{ES_P}$ and $\overline{ES_N}$, which is illustrated in the error bars of Figure 5.15 and, in more detail, in Figure 5.16(a) and (b), respectively, is considerably small being below 12% in the majority of the experimental settings. It is also clear that as the query size increases, $\overline{ES_P}$ and $\overline{ES_N}$ decrease to values lower than 11%. On the other hand, small query sizes lead to increased $\overline{ES_P}$ and $\overline{ES_N}$ values, between 12% and 24% regarding query sizes of 0.06×0.06, nevertheless with smaller error peak than the ones reported for random data without the usage of histograms, e.g., the reported 36% in Figure 5.14 vs. 24% in Figure 5.16. The above observation can be explained by the fact that histograms provide a locally more accurate value of the estimated error, than the global formula does over synthetic data, since they help the model absorb the local density changes of real, arbitrarily distributed, spatial data. Here, is worth to note that the settings of this particular experiment make it directly comparable with the one of Section 5.5, i.e., both use histograms in order to achieve better estimations; it becomes

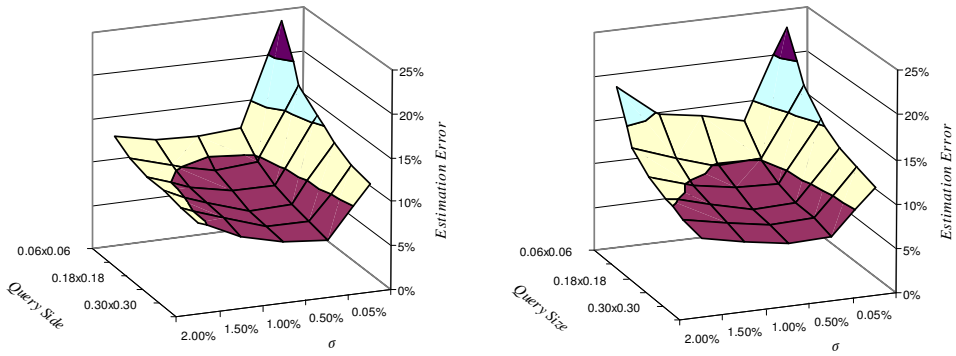
therefore clearer that the functionality of the proposed analytical model over datasets of (at least) medium cardinality is much better than the ones produced over datasets of small cardinality.



(a) (b)

Figure 5.15: Average false negatives / positives and their estimations scaling with (a) σ and (b) the query size (real data – bivariate normal distribution of uncertainty).

The impact of the analysis in real datasets with the aid of histograms is demonstrated by performing a set of experiments over the *NE* and $NE_{N\text{-}\sigma}$ datasets, computing our model by two different approaches: (a) producing the local density via Eq.(5.42) and then using it in Eq.(5.27), and, (b) directly utilizing Eq.(5.44). In this experiment is set σ = 0.5% and the side of the query window is scaled from 0.06 to 0.36. The respective results are illustrated in Figure 5.17(a), which demonstrates that although approach (a), labeled as *Estimation – Area* in Figure 5.17(a), provides an accurate average estimation, the obtained values for $\overline{ES_P}$ and $\overline{ES_N}$ are higher than those produced by approach (b), labeled as *Estimation* in Figure 5.17(a). This confirms that the appropriate use of histograms in our model is according to the analysis in Section 5.4.2 by directly employing Eq.(5.44).



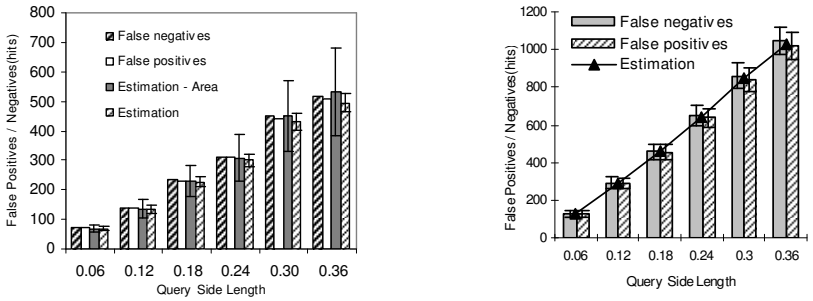
(a) (b)

Figure 5.16: Average estimation error of (a) false positives $\overline{ES_P}$ and (b) false negatives $\overline{ES_N}$, in each query, scaling with σ and the query size (real data – bivariate normal distribution of uncertainty).

5.6.2.4. Experiments over Real Data Relaxing Assumption A_{III}

In order to demonstrate the high quality estimations obtained when using the augmented histogram methodology of Section 5.4.3 (adapted to support spatial datasets), an experiment is performed by employing the *NE* and $NE_{N\text{-}v0.02}$ datasets; as already mentioned, the $NE_{N\text{-}v0.02}$ contain data with *variable* known size of standard deviation σ varying between 0 and 0.02. We then scaled the side of the query

window from 0.06 to 0.36. The respective results, illustrated in Figure 5.17(b), show that there is no significant difference between this case and the one where σ was set to a constant value (Figure 5.15(b)) and the estimations of $\overline{E_P}$, $\overline{E_N}$ are again very accurate. Moreover, the obtained values for $\overline{ES_P}$ and $\overline{ES_N}$, i.e., the error bars, vary between 7% and 14%, while in the case of Figure 5.15(b) the respective error varied between 6% and 13%. It is therefore clear that the analysis of Section 5.4.3 regarding variable uncertainty radiuses is verified to be at least as accurate as the respective analysis of Section 5.4.2, which assumes constant uncertainty radius.



(a) (b)

Figure 5.17: (a) Average false negatives / positives and estimation error in each individual query using different model approaches (real data – normal distribution of uncertainty). (b) Average false negatives / positives and their estimations scaling with the query size (real data – bivariate normal distribution of uncertainty).

5.6.2.5. Experiments over Real Data Warehouses

In order to demonstrate the application of the proposed model in a data cube operation, the *DCW* and $DCW_{N\text{-}0,5}$ datasets were used; the added Gaussian noise in the location of each point has σ equal to 0.5% of the space extent along the *x*-axis, since the size of the space is different along the *x*- and *y*-axis. Then, a uniform 60×30 grid is applied along the *x*- and *y*- axis, as illustrated in Figure 5.11(b), forming 1800 buckets overlaying the USA map and counted the number of objects contained inside each cell. Subsequently a *roll-up* operation at the *state* level is performed, as discussed in Section 0. In particular, the estimation of false positives and false negatives were calculated by the MBBs of US states as range queries as discussed in Section 5.4.2. Finally, the original datasets were used in order to determine the actual number of false positives and false negatives.

The error between the estimated and the actual number of false hits obtained as the sum of false negatives and false positives is illustrated in Figure 5.18(a). Clearly, the error in the majority of the US States, is below 25% while the actual weighted average is 16%. Regarding the four outliers, labeled with the name of the state in Figure 5.18(a), the high error presented is due to either the tiny size of the query polygon, i.e., the Delaware case, verifying the result of a previous experiment that the error increases as the query size decreases, or the irregular shape of the query polygon that is not well approximated by its MBB, i.e., the California, Florida and Michigan cases, with their shapes illustrated in Figure Figure 5.18(b).

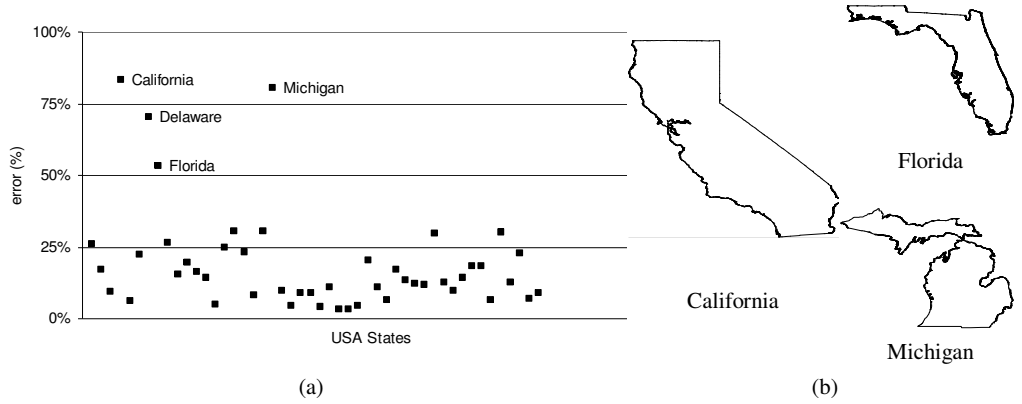


Figure 5.18: (a) Error between the actual number of false hits and their estimation in the roll-up operation from the cell to state level in the USA map, (b) a bad approximation of a state by its MBB

5.6.3. Experiments on the Efficiency

The last experimentation performed on the subject involved the performance of the proposed solutions using an implementation of the proposed model in the PostgreSQL [Post08a] DBMS along with the PostGIS [Post08b] spatial extension. Since the selected DBMS does not natively support MinSkew [APR99] spatial histograms, we have extended it towards this direction; moreover, we have included in our implementation the augmented histogram proposed in Section 3.3. All methods were implemented as functions of the spatial DBMS in the PL/pgSQL language; the developed software is ported in a template database.

Table 5.2: Histogram statistics

	Dataset	*# Objects*	*grid size*	*# grid cells*	*# buckets*	*Construction Execution time (sec)*
Histogram	$NE_{N\text{-}0.01\text{-}1}$	123K	0.001×0.001	920K	1K	21
Augmented Histogram	$NE_{N\text{-}v0.02}$	123K	0.005×0.005× 0.0001	7078K	1K	29

In the first experiment the $NE_{N\text{-}0.01\text{-}1}$ and $NE_{N\text{-}v0.02}$ datasets were utilized and the time required to construct the MinSkew and the augmented MinSkew histograms, respectively, was counted; the results are shown in Table 5.2. Clearly, the processing time is reasonable given the fact that this is an off-line operation, executed only once; then, the constructed histogram buckets are permanently stored in a relational table. Here, it is worth to note that since the MinSkew construction algorithm initially overlays a regular grid on top of the dataset, being subsequently used instead of the original dataset, the time required for constructing a MinSkew histogram does not depend on the dataset size; this is also confirmed in the respective experimental study of [APR99]. Therefore, the execution times illustrated in Table 5.2 can be considered as representatives, given also the other histogram parameters, i.e., the number of buckets and the number of the overlaid grid cells.

In the second experiment the $NE_{N\text{-}0.01\text{-}1}$ dataset and 1000 randomly distributed rectangular queries were employed in order to evaluate the average execution time of the function that implements the proposed model; the query size was also scaled in a way similar to that in Section 5.6.2 from 0.06×0.06

to 0.36×0.36. The respective results showed that regardless of the query size, the execution time required by the DBMS to estimate the false hits introduced in a query was approximately 16 ms, while the time required to process the actual query was 120 ms. Clearly, the proposal of this thesis can be employed as an estimator, since its execution time is restrained to a few milliseconds, given also that the execution of the actual query typically needs one order of magnitude more time. Moreover, it is revealed the expected result that the overhead introduced by the estimator is independent from the query size.

5.7. Conclusions

In this chapter, we presented a theoretical model that estimates the error introduced by each object's location uncertainty in the results of timeslice spatio-temporal queries, as well as, over simple range queries over stationary spatial data. We provided a closed formula of the average number of false hits, classified as false positives and false negatives, under three assumptions: uniform location uncertainty (following the model proposed by [TWHC04] in order to describe the uncertain position of trajectories), uniformly distributed data and constant radius of uncertainty disk. Then, we relaxed these assumptions towards more realistic settings, using the bivariate normal distribution over location uncertainty and *MinSkew* histograms for data and radius distributions.

The accuracy of the proposed model over spatio-temporal trajectory data, as well as over stationary spatial data, was evaluated through extensive experimentation using various synthetic and real spatio-temporal and spatial datasets. Our model shows high accuracy with an average error on $\overline{E_P}$ and $\overline{E_N}$ never exceeding 6% for either random synthetic or real spatio-temporal and stationary spatial data; regarding trajectory data, the model showed values of $\overline{ES_P}$ and $\overline{ES_N}$ near 40%. However, we have to stress again that for typical query and uncertainty sizes the formulas of the proposed model produce values of false negatives / positives between $0.0004 \times N$ and $0.0025 \times N$. It becomes therefore clear that the dataset cardinality should be quite large in order to produce a significant number of false hits, and this significantly affects the quality of the model's output.

Then again, in cases where the cardinality reaches appropriate, i.e, high values, and the random dataset case, the estimation of the number of false hits is accurate regardless of the value of the query size and the radius d of the uncertainty disk, or σ in the case of data with normally distributed uncertainty. Moreover, it has been shown that simple modifications in the single work that is very close to the one presented in this thesis [YM03], could not yield to an accurate estimation of the average number of false hits. The experiments over real spatial data demonstrate accuracy even higher than the one reported for synthetic data, with very low $\overline{ES_P}$ and $\overline{ES_N}$ errors, indicating the advantage introduced by the employment of histograms, even in the case of variable σ. Furthermore, it is verified that in the presence of histograms it is much more appropriate to use the model expressed by Eq.(5.43) and Eq.(5.44) than using the local density estimated by the histogram via traditional operations, i.e., via Eq.(5.41). The results on the application of the proposed model in spatial data cubes and spatial OLAP operations are also very promising. Finally, the implementation of the proposed solutions in real-world

environments has shown the efficiency of this proposal when employed as an estimator, since its execution time is typically only a few milliseconds.

The applications of our proposal include query optimization under the open agoras scenario [Ioa07], interactive database querying, imprecision settings and data warehouse operations, as extensively discussed. The proposed model can be directly employed in spatial database systems in order to provide users with the accuracy of spatial query results based only on known dataset and query features, while off-the-self histograms already employed in spatial databases for query optimization purposes, can serve our model without the need for any additional adjustments.

6. Managing the Effect of Trajectory Compression in Spatio-temporal Querying

The purpose of this chapter is to provide an analysis on the effect of trajectory compression in spatio-temporal querying. The chapter is structured as follows. Section 6.1 introduces basic notions on trajectory compression. Related work is discussed in Section 6.2. Section 6.3 constitutes the core of the chapter presenting our theoretical analysis. Section 6.4 presents the results of our experimental study, while Section 6.5 provides the conclusions of the chapter.

6.1. Introduction

Existing work in Moving Object Databases (MOD), repeatedly addresses that the ever-present positioning devices will eventually start to generate an unprecedented data stream of time-stamped positions. During the last decade the database community continuously contributes on developing novel indexing schemes [AG05], [PJT00], [TP01] and dedicated query processing techniques, in order to handle this excessive amount of data produced by the ubiquitous location-aware devices. However, sooner or later, such enormous volumes of data will lead to storage and computation challenges. Hence the need for trajectory compression techniques arises.

The objectives for trajectory compression are [MB04]: to obtain a lasting reduction in data size, to obtain a data series that still allows various computations at acceptable (low) complexity, and finally, to obtain a data series with known, small margins of error, which are preferably parametrically adjustable. As a consequence, our interest is in lossy compression techniques which eliminate some redundant or unnecessary information under well-defined error bounds. However, existing work in this domain [CWT03], [MB04], [PPS06], [PPS06a], [PPS07] is mainly guided by advances in the field of line simplification, cartographic generalization and time series compression.

Especially on the subject of the error introduced on the produced data by such compression techniques, the single related work [MB04] provides a formula for estimating the mean error of the approximated trajectory in terms of distance from the original data stream. On the other hand, in this chapter, we argue that instead of providing a user of a MOD with the mean error in the position of each (compressed) object at each timestamp (which can be also seen as the data (im)precision), he/she would rather prefer to be informed about the mean error introduced in query results over compressed data. The challenge thus accepted in this chapter is to provide a theoretical model that estimates the

error due to compression in the results of spatio-temporal queries. To the best of our knowledge, this is the first analytical model on the effect of compression in query results over trajectory databases.

Outlining the major issues that will be addressed in this chapter, our main contributions are summarized as follows:

- We describe two types of errors (namely, false negatives and false positives) when executing timeslice queries over compressed trajectories, and we prove a lemma that estimates the average number of the above error types. It is proved that the average number of the false hits of both error types depends on the Synchronous Euclidean Distance [CWT03], [MB04], [PPS06] along the x- and y- axes between the original and the compressed trajectory, and the perimeter (rather than the area) of the query window.
- We show how the cost of evaluating the developed formula can be reduced to a small overhead over the employed compression algorithm, while we discuss how the developed analytical model helps to provide more effective compression algorithms.
- Finally, we conduct a comprehensive set of experiments over synthetic and real trajectory datasets demonstrating the applicability, correctness and accuracy of our analysis.

The model described in this chapter can be employed in MODs so as to estimate the average number of false hits in query results when trajectory data are compressed. For example, it could be utilized right after the compression of a trajectory dataset in order to provide the user with the average error introduced in the results of spatio-temporal queries of several sizes; it could be therefore exploited as an additional criterion for the user in order to decide whether compressed data are suitable for his/her needs, and possibly decide on different compression rates, and so on. Moreover, it could be utilized as to improve the efficiency of the proposed trajectory compression algorithms; given that a model of this kind would expose the actual measures on which the error is depended, it could subsequently provide intuitive directions towards the employment of more sophisticated / efficient solutions.

6.2. Background

In this section we firstly deal with the techniques introduced for compressing trajectories during the last few years, while, we subsequently examine the related work in the field of estimating and handling the error introduced by such compression techniques.

6.2.1. Compressing Trajectories

As already mentioned, existing work in trajectory compression is mainly guided by related work in the field of line simplification and time series compression. Meratnia and By [MB04] exploit existing algorithms used in the line generalization field, presenting one top-down and one opening window algorithm, which can be directly applied to spatio-temporal trajectories. The *top-down* algorithm, named TD-TR, is based on the well known Douglas-Peucker [DP73] algorithm (Figure 6.1) introduced by geographers in cartography. This algorithm calculates the perpendicular distance of each internal point from the line connecting the first and the last point of the polyline (line *AB* in Figure 6.1) and finds the point with the greatest perpendicular distance (point *C*). Then, it creates lines *AC* and *CB* and, recursively, checks these new lines against the remaining points with the same method, and so on.

When the distance of all remaining points from the currently examined line is less than a given threshold (e.g., all the points following *C* against line *BC* in Figure 6.1) the algorithm stops and returns this line segment as part of the new - compressed - polyline. Being aware of the fact that trajectories are polylines evolving in time, the algorithm presented in [MB04] replaces the perpendicular distance used in the DP algorithm with the so-called *Synchronous Euclidean Distance* (*SED*), also discussed in [CWT03], [PPS06], which is the distance between the currently examined point (P_i in Figure 6.2) and the point of the line (P_s, P_e) where the moving object would lie, supposed it was moving on this line, at time instance t_i determined by the point under examination (P_i' in Figure 6.2).

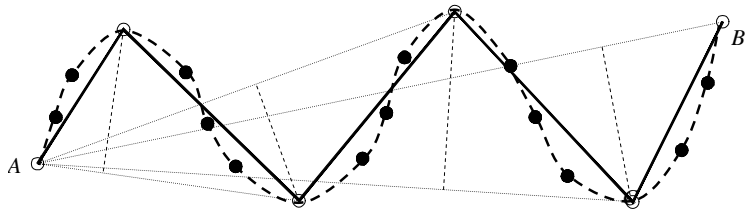


Figure 6.1: Top-down Douglas-Peucker algorithm used for trajectory Compression. Original data points are represented by closed circles [MB04]

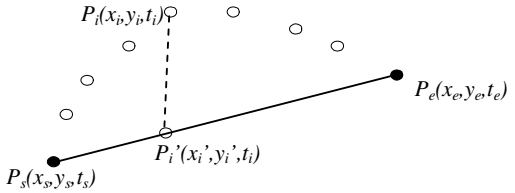


Figure 6.2: The Synchronous Euclidean Distance (SED): The distance is calculated between the point under examination (P_i) and the point P_i' which is determined as the point on the line (P_s, P_e) the time instance t_i [MB04]

The time complexity of the original Douglas-Peucker algorithm (which the TD-TR algorithm is based on) is $O(N^2)$, with N being the number of the original data points, while it can be reduced to $O(N\log N)$ by applying the proposal presented in [HS92]. Although the experimental study presented in [MB04] shows that the TD-TR algorithm is significantly better than the opening window (presented later in this section) in terms of both quality and compression (since it globally optimizes the compression process), the TD-TR algorithm has the disadvantage that it is not an on-line algorithm and, therefore, it is not applicable to newcoming trajectory portions as soon as they feed a MOD. On the contrary, it requires the a priori knowledge of the entire moving object trajectory.

On the other hand, under the previously described conditions of on-line operation, the *opening window* (OW) class of algorithms can be easily applied. These algorithms start by anchoring the first trajectory point, and attempt to approximate the subsequent data points with one gradually longer segment (Figure 6.3). As long as all distances of the subsequent data points from the segment are below the distance threshold, an attempt is made to move the segment's end point one position up in the data series. When the threshold is going to exceed, two strategies can be applied: either the point causing the violation (*Normal Opening Window*, *NOPW*) or the point just before it (*Before Opening Window*,

BOPW) becomes the end point of the current segment, as well as the anchor of the next segment. If the threshold is not exceeded, the float is moved one position up in the data series (i.e., the window opens further) and the algorithm continues until the last point of the trajectory is found; then the whole trajectory is transformed into a linear approximation. While in the original OW class of algorithms each distance is calculated from the point perpendicularly to the segment under examination, in the OPW-TR algorithm presented in [MB04] the *SED* is evaluated. Although OW algorithms are computationally expensive - since their time complexity is $O(N^2)$ - they turned out to be very popular. This is because, they work online, and they can work reasonably well in presence of noise (though only for relatively short data series).

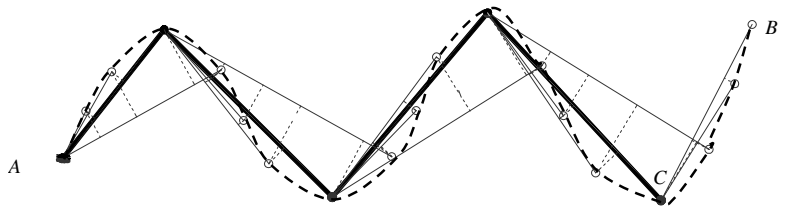


Figure 6.3: Opening Window algorithm used for trajectory Compression. Original data points are represented by closed circles [MB04]

Recently, Potamias et al. [PPS06] proposed several techniques based on uniform and spatio-temporal sampling to compress trajectory streams, under different memory availability settings: fixed memory, logarithmically or linearly increasing memory, or memory not known in advance. Their major contributions are two compression algorithms, namely, the *STTrace* and *Thresholds*. The *STTrace* algorithm, utilizes a constant, for each trajectory, amount of memory *M*. It starts by inserting in the allocated memory the first *M* recorded positions, along with each position's *SED* with respect to its predecessor and successor in the sample. As soon as the allocated memory gets exhausted and a new point is examined for possible insertion, the sample is searched for the item with the lowest *SED*, which represents the least possible loss of information in case it gets discarded. In the sequel, the algorithm checks whether the inserted point has *SED* larger than the minimum one found already in the sample and, if yes, the currently processed point is inserted into the sample at the expense of the point with the lowest *SED*. Finally, the *SED* attributes of the neighboring points of the removed one are recalculated, whereas a search is triggered in the sample for the new minimum *SED*. The proposed algorithm may be easily applied in the multiple trajectory case, by simply calculating a global minimum *SED* of all the trajectories stored inside the allocated memory.

It notably arises from the previous discussion that the vast majority of the proposed trajectory compression algorithms base their decision on whether keeping or discarding a point of the original trajectory on the value of *SED* between the original and the compressed trajectory at this particular timestamp. Consequently, a method for calculating the effect of compression in spatio-temporal querying based on the value of *SED* along the original trajectory data points, would not introduce a considerable overhead in the compression algorithm, since it would require only performing additional operations inside the same algorithm.

6.2.2. Related Work on Error Estimation

To the best of our knowledge, a theoretical study on modeling the error introduced in spatio-temporal query results due to the compression of trajectories is lacking; our work is the first on this topic covering the case of the spatio-temporal timeslice queries. Nevertheless, there are two related subjects: The first is the determination of the error introduced directly in each trajectory by the compression [MB04], being the average value of the *SED* between a trajectory p and its approximation q (also termed as synchronous error $E(q, p)$). [MB04] provide a method for calculating this average value as a function of the distance between p and q along each sampled point. The outcome of this analysis turns to a costly formula, which provides the average error (i.e., mean distance between p and q along their lifetime); however, there is no obvious way on how to use it in order to determine the error introduced in query results.

The second related subject is the work conducted on the context of trajectory uncertainty management, such as [CKP04], [PJ99], [Tra03], [TWHC04]. This is due to the fact that the error introduced by compression can also be seen as uncertainty, and thus related techniques may be applied in the resulted dataset (e.g., probabilistic queries); as such the work presented in the previous chapter could be employed towards our goal. However, such methodology cannot be directly used in the presence of compressed trajectory data, since the task of determining the statistical distribution of the location of the compressed trajectory using information from the original one, is by itself a very complex task.

On the other hand, our approach is based only on the fact that the compression algorithm exploits the *SED* in each original trajectory data point and thus, introduces a very small overhead on the compression algorithm.

6.3. Analysis

The core of our analysis is a lemma that provides the formula used to estimate the average number of false hits per query when executed over a compressed trajectory dataset. In this chapter, we also focus on timeslice queries, which can be used to retrieve the positions of moving objects at a given time point in the past and can be seen as a special case of spatio-temporal range queries, with their temporal extent set to zero. This type of query can also be seen as the combination of a spatial (i.e., query window W) and a temporal (i.e., timestamp t) component. As it will be discussed in Chapter 7, the extension of our model to support range queries with non-zero temporal extent is by no means trivial and is left as future work.

It is important to mention that our model supports arbitrarily distributed trajectory data without concerning about their characteristics (e.g., sampling rate, velocity, heading, agility). Therefore, it can be directly employed in MODs without further modifications. The single assumption we make is that timeslice query windows are uniformly distributed inside the data space. Should this assumption be relaxed, one should mathematically model the query distribution using a probability distribution and modify the following analysis, accordingly. Table 6.1 summarizes the notations used in the rest of the chapter.

Table 6.1: Table of notations

Notation	*Description*
$S, T^{\dagger}, T$	The unit space, a trajectory dataset and its compressed counterpart.
$T_i^{\dagger}$, T_i	an original trajectory and its compressed counterpart.
R, $R_{a\times b}$, W_j	the set of all timeslice queries over S, its subset with sides of length a and b across the x- and y- axes, and a timeslice query window.
n, m_i	the cardinality of dataset T and the number of sampled points inside trajectory $T_i^{\dagger}$.
$SED_i(t)$, $\delta x_i(t)$, $\delta y_i(t)$	the function of the Synchronous Euclidean Distance (SED) between trajectory $T_i^{\dagger}$ and its compressed counterpart T_i, and its projections along the x- and y- axes.
$t_{i,k}$, $SED_{i,k}$, $\delta x_{i,k}$, $\delta y_{i,k}$	the k-th timestamp on which trajectory $T_i^{\dagger}$ sampled its position, its Synchronous Euclidean Distance from its compressed counterpart T_i at the same timestamp, and its projection along the x- and y- axes
$A_{i,j}$	the area inside which the lower-left corner of W_j has to be found at timestamp t_j in order for it to retrieve trajectory T_i as false negative (or false positive).
$AvgP_{i,N}(R_{a\times b})$, $AvgP_{i,P}(R_{a\times b})$	the average probability of all timeslice queries $W_j \in R_{a\times b}$, to retrieve T_i as false negative (or false positives).
$E_N(R_{a\times b})$, $E_P(R_{a\times b})$	the average number of false negatives (or false positives) in the results of a query $W_j \in R_{a\times b}$.

Let us consider the unit 3D (i.e., 2D spatial and 1D temporal) space S containing a set $T^{\dagger}$ of n trajectories $T_i^{\dagger}$ and a set T with their compressed counterparts T_i. Let also R be the set of all timeslice queries posed against datasets $T^{\dagger}$ and T, and $R_{a\times b}$ be the subset of R containing all timeslice queries having sides of length a and b along the x- and y- axis respectively. Two types of errors are introduced when executing a timeslice query $W_j \in R$ over a dataset with the previously described settings:

- *false negatives* are the trajectories which originally qualified the query but their compressed counterparts were not retrieved; formally, the set of false negatives $T_N \subseteq T$ is defined as $T_N = \left\{T_i \in T : T_i \notin W_j \mid T_i^{\dagger} \in W_j\right\}$;
- *false positives* are the compressed trajectories retrieved by the query while their original counterparts are not qualifying it; formally, the set of false positives $T_P \subseteq T$ is defined as $T_P = \left\{T_i \in T : T_i \in W_j \mid T_i^{\dagger} \notin W_j\right\}$.

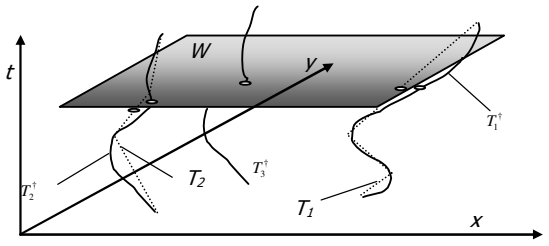


Figure 6.4: Problem setting

Consider for example Figure 6.4 illustrating a set of n uncompressed trajectories $T_i^{\dagger}$, along with their compressed counterparts T_i. Each uncompressed trajectory $T_i^{\dagger}$ is composed by a set of m_i time-

stamped points, applying linear interpolation in-between them. Figure 6.4 also illustrates a timeslice query *W*; though *W* retrieves the compressed trajectory T_1, its original counterpart $T_1^{\dagger}$ does not intersect the query window, encountering a false positive. Conversely, though the original trajectory $T_2^{\dagger}$ intersects *W*, its compressed counterpart T_2 is not present in the query results, forming a false negative. Having described the framework of our work, we state the following lemma

Lemma 6.1*: The average number of false negatives $E_N(R_{a\times b})$ and false positives $E_P(R_{a\times b})$ in the results of a timeslice query $W_j \in R_{a\times b}$ with sides of length a and b along the x- and y- axis, respectively, over a compressed trajectory dataset is given by the following formula:*

$$E_N\left(R_{a\times b}\right) = E_P\left(R_{a\times b}\right) = \sum_{i=1}^{n}\sum_{k=1}^{m_i-1} \frac{\left(t_{i,k+1}-t_{i,k}\right)}{(1+a)\cdot(1+b)} \cdot \left(\frac{b\left(\left|\delta x_{i,k}\right|+\left|\delta x_{i,k+1}\right|\right)}{2} + \frac{a\left(\left|\delta y_{i,k}\right|+\left|\delta y_{i,k+1}\right|\right)}{2} - \frac{e}{6} \right) \quad (6.1)$$

where n is the cardinality of the dataset, m_i the number of sampled points inside trajectory T_i, $\delta x_{i,k}$ and $\delta y_{i,k}$ the projection of the synchronous euclidean distance vector between the original trajectory T_i and its compressed counterpart at timestamp t_k along the x- and y- axes, and $e = 2\left|\delta x_{i,k}\delta y_{i,k}\right| + 2\left|\delta x_{i,k+1}\delta y_{i,k+1}\right| + \left|\delta x_{i,k}\delta y_{i,k+1}\right| + \left|\delta x_{i,k+1}\delta y_{i,k}\right|$.

Eq.(6.1) formulates the fact that the average error in the results of a timeslice query over compressed trajectory data is directly related to the weighted average *SED* along the *x*- and *y*- axis (i.e., $\left(t_{i,k+1}-t_{i,k}\right)$ multiplied by $\left|\delta x_{i,k}\right|+\left|\delta x_{i,k+1}\right|$ or $\left|\delta y_{i,k}\right|+\left|\delta y_{i,k+1}\right|$) multiplied by the respective opposite query dimension (i.e., $b\left(\left|\delta x_{i,k}\right|+\left|\delta x_{i,k+1}\right|\right)$ and $a\left(\left|\delta y_{i,k}\right|+\left|\delta y_{i,k+1}\right|\right)$), while *e* is a sum of minor importance, since it is the sum of the products between pairs of $\left|\delta x_{i,k}\right|, \left|\delta x_{i,k+1}\right|, \left|\delta y_{i,k}\right|, \left|\delta y_{i,k+1}\right|$.

6.3.1. Proof of Lemma 6.1

The average number $E_N(R_{a\times b})$ of trajectories being false negatives in the results of a timeslice query $W_j \in R_{a\times b}$, can be obtained by summing up the probabilities $P\left(T_i \notin W_j \mid T_i^{\dagger} \in W_j\right)$ of all dataset trajectories T_i (i=1,…,n) to be false negative regarding an arbitrary timeslice query window $W_j \in R_{a\times b}$:

$$E_N\left(R_{a\times b}\right) = \sum_{i=1}^{n} AvgP_{i,N}\left(R_{a\times b}\right) \quad (6.2)$$

Similarly, the average number $E_P(R_{a\times b})$ of trajectories being false positives can be calculated by the following formula:

$$E_P\left(R_{a\times b}\right) = \sum_{i=1}^{n} AvgP_{i,P}\left(R_{a\times b}\right) \quad (6.3)$$

Hence, our target is to determine $AvgP_{i,N}(R_{a\times b})$ and $AvgP_{i,P}(R_{a\times b})$. Towards this goal, we formulate the probability of a random trajectory being false negative (or false positive), regarding an arbitrary timeslice query window $W_j \in R_{a\times b}$ invoked at timestamp t_j (i.e., $T_i \notin W_j \mid T_i^{\dagger} \in W_j$, and $T_i \in W_j \mid T_i^{\dagger} \notin W_j$, respectively). As also illustrated in Figure 6.5(b), the intersection of trajectories T_i, $T_i^{\dagger}$ with the plane determined by the temporal component of W_j (i.e., timestamp t_j) will be demonstrated as two points (points $p_{i,j}$ and $p_{i,j}^{\dagger}$, respectively, in Figure 6.5(b)) having in-between them, distance $\delta x_{i,j}$ and $\delta y_{i,j}$ along the *x*- and *y*- axis, respectively.

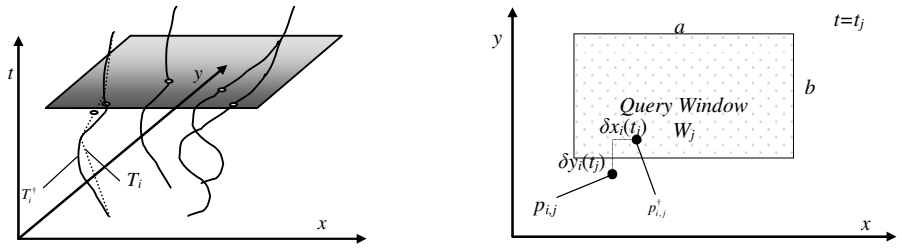
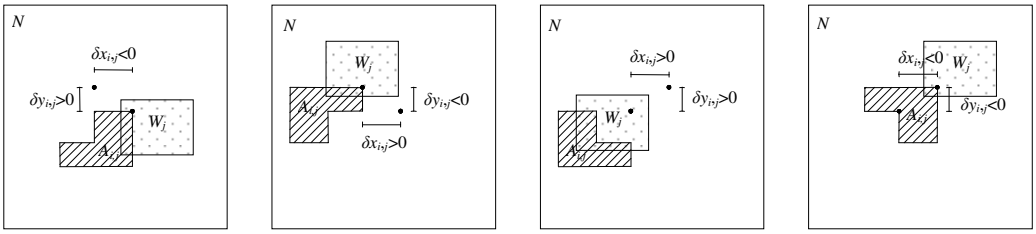


Figure 6.5: The intersection of a trajectory $T_i^\dagger$ and its compressed counterpart T_i, with the plane of a timeslice query at timestamp t_j.

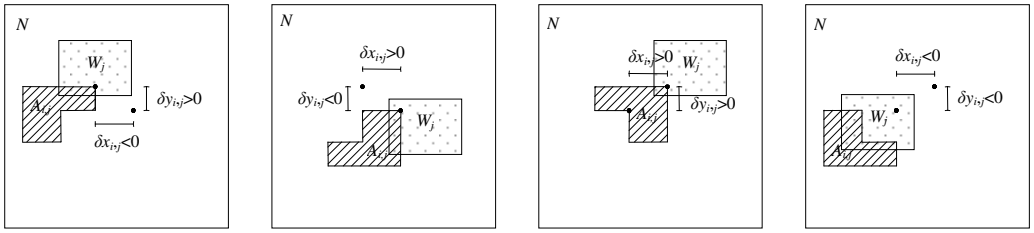
In order to calculate the quantity of timeslice query windows that would retrieve trajectory T_i as a false negative (false positive) at timestamp t_j, we need to distinguish among four cases regarding the signs of $\delta x_{i,j}$ and $\delta y_{i,j}$ as demonstrated in Figure 6.6 (Figure 6.7, respectively). The shaded (with sided stripes) region in all four cases illustrate the area inside which the lower-left query window corner has to be found in order for it to retrieve trajectory T_i as false negative (or false positive, respectively). However, as can be easily derived from these figures, the area of the shaded region in all four cases, is equal for both false negatives and false positives, and can be calculated by the following equation:

$$A_{i,j} = a \cdot b - \left(a - \left|\delta x_{i,j}\right|\right) \cdot \left(b - \left|\delta y_{i,j}\right|\right) \tag{6.4}$$



(a) $\delta x_{i,j} < 0 \wedge \delta y_{i,j} > 0$ (b) $\delta x_{i,j} > 0 \wedge \delta y_{i,j} < 0$ (c) $\delta x_{i,j} > 0 \wedge \delta y_{i,j} > 0$ (d) $\delta x_{i,j} < 0 \wedge \delta y_{i,j} < 0$

Figure 6.6: Regions inside which the lower-left query window corner has to be found in order to retrieve trajectory T_i as false negative



(a) $\delta x_{i,j} < 0 \wedge \delta y_{i,j} > 0$ (b) $\delta x_{i,j} > 0 \wedge \delta y_{i,j} < 0$ (c) $\delta x_{i,j} > 0 \wedge \delta y_{i,j} > 0$ (d) $\delta x_{i,j} < 0 \wedge \delta y_{i,j} < 0$

Figure 6.7: Regions inside which the lower-left query window corner has to be found in order to retrieve trajectory T_i as false positive

Given that W_j is valid when it is (either partially or totally) found inside the unit space, the lower-left query window corner must be found inside a space region of area equal to $(1+a)\cdot(1+b)$. Then, the probability of trajectory T_i to be retrieved as a false negative or false positive at timestamp t_j is:

$$P\left(T_i \notin W_j \mid T_i^{\dagger} \in W_j\right) = P\left(T_i \in W_j \mid T_i^{\dagger} \notin W_j\right) = \frac{A_{i,j}}{(1+a)\cdot(1+b)} = \frac{a\cdot b - \left(a - \left|\delta x_{i,j}\right|\right)\cdot\left(b - \left|\delta y_{i,j}\right|\right)}{(1+a)\cdot(1+b)} \quad (6.5)$$

Given also our assumption regarding the distribution of query windows, the average probability of a trajectory T_i to be false negative regarding an arbitrary query window $W_j \in R_{a\times b}$ at any timestamp can be obtained by integrating Eq.(6.5) over all timestamps inside the unit space. Since $P\left(T_i \notin W_j \mid T_i^{\dagger} \in W_j\right) = P\left(T_i \in W_j \mid T_i^{\dagger} \notin W_j\right)$, it follows that:

$$AvgP_{i,N}\left(R_{a\times b}\right) = AvgP_{i,P}\left(R_{a\times b}\right) = \int_0^1 P\left(T_i \notin W_j \mid T_i^{\dagger} \in W_j\right) dt = \int_0^1 P\left(T_i \in W_j \mid T_i^{\dagger} \notin W_j\right) dt \quad (6.6)$$

However, given that each original trajectory T_i is a set of m_i sampled points applying linear interpolation in between them, Eq.(6.6) is transformed as follows:

$$AvgP_{i,N}\left(R_{a\times b}\right) = AvgP_{i,P}\left(R_{a\times b}\right) =$$
$$\sum_{k=1}^{m_i-1} \frac{1}{t_{i,k+1} - t_{i,k}} \int_{t_k}^{t_{k+1}} P\left(T_i \notin W_j \mid T_i^{\dagger} \in W_j\right) dt = \sum_{k=1}^{m_i-1} \frac{1}{t_{i,k+1} - t_{i,k}} \int_{t_k}^{t_{k+1}} P\left(T_i \in W_j \mid T_i^{\dagger} \notin W_j\right) dt \quad (6.7)$$

and $\delta x_{i,j}$ and $\delta y_{i,j}$ can be trivially formulated as single functions of t when $t_{i,k} \le t \le t_{i,k+1}$, between sampled points:

$$\delta x_i(t) = \delta x_{i,k} + \left(t - t_{i,k}\right)\cdot\frac{\delta x_{i,k+1} - \delta x_{i,k}}{t_{i,k+1} - t_{i,k}}, \quad \text{and} \quad (6.8)$$

$$\delta y_i(t) = \delta y_{i,k} + \left(t - t_{i,k}\right)\cdot\frac{\delta y_{i,k+1} - \delta y_{i,k}}{t_{i,k+1} - t_{i,k}} \quad (6.9)$$

Substituting Eq.(6.8), Eq.(6.9) and Eq.(6.5) into Eq.(6.7) and performing the necessary calculations we result in the following formula:

$$AvgP_{i,N}\left(R_{a\times b}\right) = AvgP_{i,P}\left(R_{a\times b}\right) =$$
$$\sum_{k=1}^{m_i-1} \frac{\left(t_{i,k+1} - t_{i,k}\right)}{(1+a)\cdot(1+b)} \cdot \left(\frac{b\left(\left|\delta x_{i,k}\right| + \left|\delta x_{i,k+1}\right|\right)}{2} + \frac{a\left(\left|\delta y_{i,k}\right| + \left|\delta y_{i,k+1}\right|\right)}{2} - \frac{2\left|\delta x_{i,k}\delta y_{i,k}\right| + 2\left|\delta x_{i,k+1}\delta y_{i,k+1}\right| + \left|\delta x_{i,k}\delta y_{i,k+1}\right| + \left|\delta x_{i,k+1}\delta y_{i,k}\right|}{6} \right) \quad (6.10)$$

Finally, by substituting Eq.(6.10) into Eq.(6.2) and defining

$$e = 2\left|\delta x_{i,k}\delta y_{i,k}\right| + 2\left|\delta x_{i,k+1}\delta y_{i,k+1}\right| + \left|\delta x_{i,k}\delta y_{i,k+1}\right| + \left|\delta x_{i,k+1}\delta y_{i,k}\right| \quad (6.11)$$

we haven proven Lemma 6.1. ■

6.3.2. Discussion on Lemma 6.1

Eq.(6.1), the main result of Lemma 6.1, can be straightforwardly used to estimate the average number of false negatives and false positives for timeslice query windows with known size along the *x*- and *y*-axes (*a* and *b*, respectively). It notably arises from this formula that the average number of false negatives in the results of a timeslice query is equal to the respective average number of false positives, while their values depend mainly on the perimeter of the query window (*a*+*b*), rather than its area ($a\cdot b$). However, it should be explicitly mentioned that Lemma 6.1 holds in the case of uniformly distributed query windows only; as such, the estimated average number of false negatives and false positives serves as a metric estimating data loss due to compression, rather than providing an accurate result regarding individual queries.

Another interesting result is that the error introduced in query results due to trajectory compression depends on the absolute values of $|\delta x_{i,k}|$ and $|\delta y_{i,k}|$ rather than their squares, i.e., $\delta x_{i,k}^2$ and $\delta y_{i,k}^2$. This is not an expected result; as such, it gives rise to the following discussion. In particular, Eq.(6.1) states that the minimization of the error introduced in timeslice query results over compressed trajectories occurs when minimizing $|\delta x_{i,k}|+|\delta y_{i,k}|$, instead of the Synchronous Euclidean Distance (SED), which is considered as the optimization criterion in the majority of the existing trajectory compression algorithms. It is therefore expected that the employment of $|\delta x_{i,k}|+|\delta y_{i,k}|$ instead of *SED* as the minimization criterion in the trajectory compression algorithms, will lead to simplified trajectories that result in smaller values of error introduced in timeslice query results.

Obviously, the evaluation of Eq.(6.1) is a costly operation; given that it involves a double sum, its time complexity is $O(n \cdot m)$ where n is the number of trajectories and m is the (average) number of sampled points per trajectory. In other words, since Eq.(6.1) includes the calculation of $\delta x_{i,k}$, $\delta y_{i,k}$, between each tuple of the initial and compressed trajectories on each timestamp the trajectory was originally sampled, it requires to process the entire original dataset along with its compressed counterpart. On the other hand, as already mentioned in Section 6.2, the vast majority of the proposed trajectory compression algorithms base their decision about the point of the original trajectory data to eliminate, on the value of the *SED*; however, since $SED_i(t)=\sqrt{\delta x_i(t)^2+\delta y_i(t)^2}$, the respective algorithm should first evaluate $\delta x_i(t)$ and $\delta y_i(t)$ at timestamps $t_{i,k}$ producing thus, $\delta x_{i,k}$ and $\delta y_{i,k}$, respectively. Consequently, any trajectory compression algorithm using *SED* as the criterion to decide which trajectory points to eliminate, also calculates $\delta x_{i,k}$ and $\delta y_{i,k}$. As such, Eq.(6.1) can be calculated during the algorithm's execution, adding very small overhead in the original algorithm; the above observation is further confirmed in our experimental study presented in the next section.

Moreover, since Eq.(6.1) involves the query dimensions a and b, it follows that different values of a and b will lead to different calculations for the average error. However, such an approach (i.e., evaluating Eq.(6.1) from the beginning for every different query size), would lead to high computation cost since it would also require $O(n \cdot m)$ time. In order to overcome this drawback, Eq.(6.1) can be rewritten as follows:

$$E_N(R_{a\times b}) = E_P(R_{a\times b}) = \frac{A\cdot a + B\cdot b + C}{(1+a)\cdot(1+b)} \tag{6.12}$$

where

$$A=\sum_{i=1}^{n}\sum_{k=1}^{m_i-1}\left(t_{i,k+1}-t_{i,k}\right)\cdot\frac{|\delta y_{i,k}|+|\delta y_{i,k+1}|}{2},\qquad B=\sum_{i=1}^{n}\sum_{k=1}^{m_i-1}\left(t_{i,k+1}-t_{i,k}\right)\cdot\frac{|\delta x_{i,k}|+|\delta x_{i,k+1}|}{2}$$ and

$$C=-\sum_{i=1}^{n}\sum_{k=1}^{m_i-1}\left(t_{i,k+1}-t_{i,k}\right)\cdot\frac{e}{6}.$$

Therefore, in the case where the average error needs to be determined for a variety of query sizes (i.e., different sizes of a and b), rather than directly calculating Eq.(6.1) for each different query size, the

three factors *A*, *B* and *C* could be calculated first, and be subsequently employed in Eq.(6.12); an approach which dramatically reduces the computation cost to $O(1)$ time.

6.4. Experimental Study

In this section, we present several sets of experiments using synthetic and real trajectory datasets. The goal of our experimental study is two-fold:

- first, to present the overhead introduced in the execution of a compression algorithm when calculating during its execution the values of *A*, *B* and *C* factors introduced in Eq.(6.12), and,
- second, to present the accuracy of the estimation provided by our analytical model regarding the number of false negatives and false positives over synthetic and real trajectory datasets.

6.4.1. Experimental Setup

Once more, we experimented with the real-world datasets used for experimentation purposes in this thesis, the fleet of *trucks* (cf. section 0). We have also used the synthetic dataset NG 2000 (cf. section 1.5.3). All the datasets where normalized in [0,1] space. In order to test the accuracy of our model and produce compressed datasets, we implemented the TD-TR algorithm proposed by [MB04]. Then we executed it over all the above datasets, varying its threshold between 0.001 and 0.02 of the total space, thus producing the respective compressed datasets. Finally, we used the original and compressed datasets and created several 3D R-trees [TVS96] in order to accelerate the querying process used when performing experiments on the quality. Table 6.2 illustrates summary information about the (original and compressed) datasets used. The experiments were performed in a PC running Microsoft Windows XP with AMD Athlon 64 3GHz processor, 1 GB RAM and several GB of disk size.

Table 6.2: Summary Dataset Information

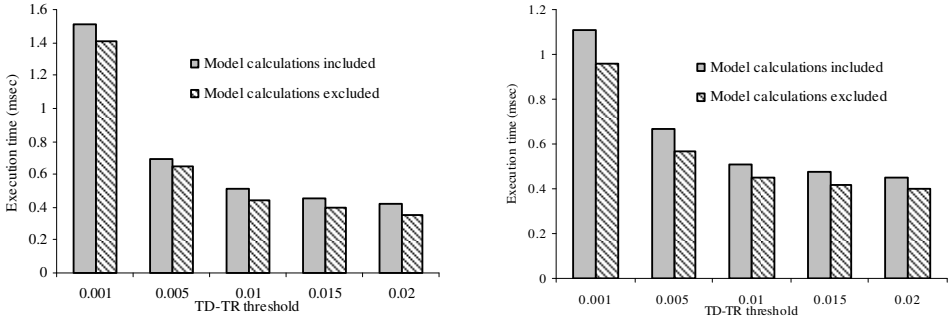
	Original Datasets		*Compressed Datasets (#entries)*				
			TD-TR threshold value				
	#trajectories	*# entries*	*0.001*	*0.005*	*0.010*	*0.015*	*0.020*
Trucks	273	112,203	62,067	20,935	12,636	9,274	7,571
Synthetic	2,000	800,000	229,167	120,437	88,565	74,638	65,410

6.4.2. Experiments on the Performance

In order to demonstrate the applicability of our proposal in trajectory data and estimate the overhead introduced in a trajectory compression algorithm when calculating the values of *A*, *B* and *C* factors introduced in Eq.(6.12), we first ran the TD-TR compression algorithm over the real data and measured the average execution time required for each trajectory, scaling also the threshold of the algorithm. We then modified the algorithm in order to calculate the model parameters (i.e., the values of *A*, *B* and *C* in Eq.(6.12) within its execution and also ran it against the same dataset with the same parameters. The respective results are illustrated in Figure 6.8.

In particular, Figure 6.8(a) and Figure 6.8(b) illustrate the execution time of the TD-TR algorithm per compressed trajectory (in milliseconds), with and without the evaluation of the model parameters, against the trucks, and the synthetic datasets, respectively. A first conclusion is that the execution time of the algorithm is reduced as the value of the TD-TR threshold increases; this is an

expected result, since typically, the number of the algorithm iterations increase, as the value of the threshold degreases. However, the main result gathered from Figure 6.8 is that the overhead introduced is typically small (i.e., the difference between the two bars). In all cases, the overhead introduced in the algorithm is between 7% and 19% of the originally required execution time; furthermore, in absolute times, the overhead introduced never exceeds 0.2 milliseconds per trajectory. As a consequence, the discussion presented in Section 3.2 is further confirmed, and our model can be evaluated as an extension of the compression algorithm's execution, introducing a small, perhaps negligible, overhead.



(a) (b)

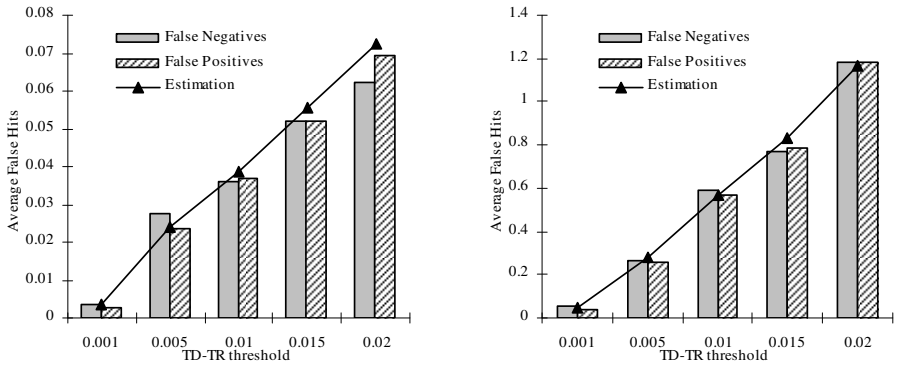
Figure 6.8: Execution time for the TD-TR algorithm with and without the calculation of the model parameters over (a) the trucks, and, (b) the synthetic datasets, scaling the value of the TD-TR threshold.

6.4.3. Experiments on the Quality

The statistical measure employed in order to demonstrate the quality of our estimation, are the reported *average number of false negatives and false positives*, $\overline{E_N}$ and $\overline{E_P}$, respectively. Formally, these measures are defined as:

$$\overline{E_N} = \frac{1}{n}\sum_{i=1..n} E_{N,i}\ ,\ \overline{E_P} = \frac{1}{n}\sum_{i=1..n} E_{P,i}$$

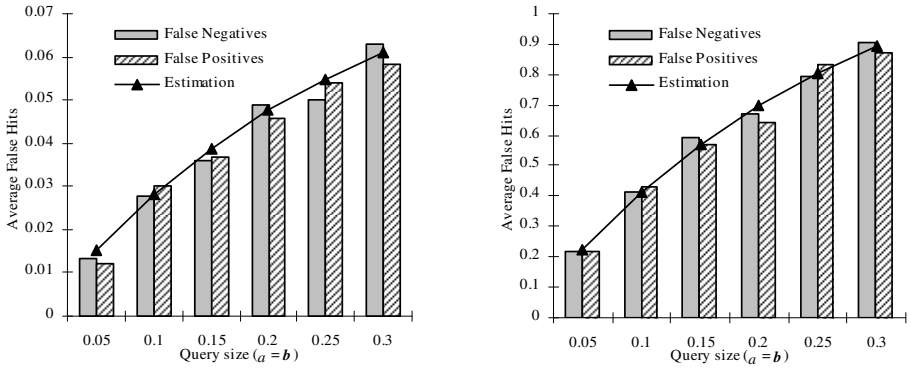
where n is the number of executed queries and $E_{N,i}$ ($E_{P,i}$) the actual number of false negatives (false positives, respectively) in the i-th query. In the next experiments, n is set to 10000 timeslice queries.



(a) (b)

Figure 6.9: Accuracy of the model scaling the value of the TD-TR threshold over (a) the trucks, and, (b) the synthetic datasets

Our first set of experiments was performed over both the real and the synthetic datasets. Specifically, we executed 10,000 square timeslice queries of 0.10×0.10 size (i.e., covering 1% of unit space) randomly distributed inside the unit space, over both the original and the compressed datasets, and then, utilizing the results of each particular query over the two datasets, we counted the actual number of false negatives and false positives, $E_{N,i}$ and $E_{P,i}$, respectively. Figure 6.9 that follows illustrates the results of this experiment scaling the value of the compression threshold over the trucks and the synthetic dataset. A first conclusion is that the average number of false hits (negatives and positives) is linear with the value of the TD-TR compression threshold. Moreover, the estimations, $\overline{E_N}$ and $\overline{E_P}$, of our model are very close to the actual values of average false negatives and false positives reported by the experiments, regardless of the value of the compression threshold. In particular, the average error in the estimation for the synthetic dataset is around 6%, varying between 0.2% and 14%; regarding the trucks dataset, the average error raises up to 10.6%, mainly due to the error introduced in small values of TD-TR threshold.



(a) (b)

Figure 6.10: Accuracy of the model scaling the rectangular query size over (a) the trucks, and, (b) the synthetic datasets

In our second experiment we used the same experimental settings (i.e., datasets, number of queries), but we fixed the TD-TR threshold to 0.01 and scaled the size of the timeslice query window between 0.05×0.05 and 0.30×0.30 (resulting in 0.25% and 9% of unit space, respectively). The corresponding results are illustrated in Figure 6.10(a) and Figure 6.10(b) against the trucks and the synthetic datasets, respectively. Again, it is clear that our model is highly accurate, producing estimates $\overline{E_N}$ and $\overline{E_P}$ with errors for the synthetic dataset between 0.2% and 8.7% and the average being around 2.9% (while the respective average error for the trucks dataset is 7.5%.) Another notable conclusion is that the average number of false positives and false negatives are sub-linear with the query size; an expected result gathered directly from the way that Eq.(6.12) involves the lengths a and b of the query sides.

In the last experiment we verified the effect of using non-square timeslice queries (i.e., $a \neq b$) over the synthetic datasets (while the experiments with the trucks dataset produced similar results). Specifically, we used timeslice query windows with sizes varying from 0.05×0.30 (where $a \ll b$) to 0.30×0.30 (where $a=b$); we also scaled the query size towards the other direction (from 0.30×0.05 to

0.30×0.30). The results of this experiment, illustrated in Figure 6.11(a) and (b) respectively, resulted in similar outcomes as the ones presented in the previous paragraph regarding square (i.e, *a*=*b*) timeslice queries. Specifically, our model is once again very accurate, producing estimates with error between 0.6% and 7.2%, while the average error is 3.5%.

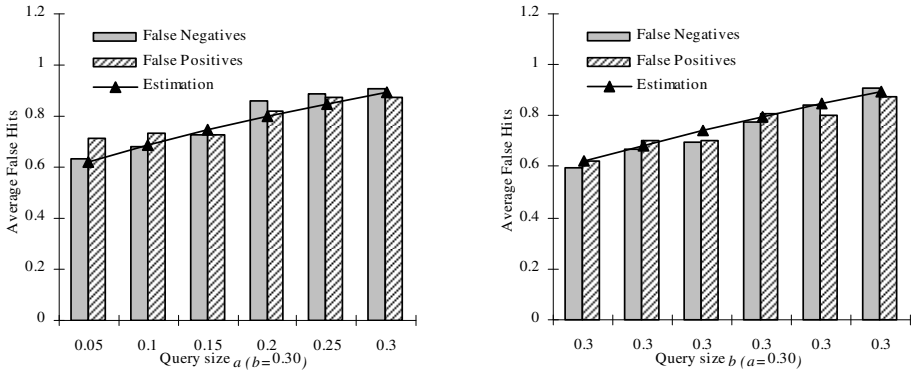


Figure 6.11: Accuracy of the model scaling the non-rectangular query size towards (a) the x- axis, and (b) the y- axis, against the synthetic datasets.

6.5. Conclusions

Related work on the subject of trajectory compression has focused on the development of compression algorithms, also emphasizing on the error introduced in the position of each object from the compression. In this work, acknowledging that users are more likely concerned about the error introduced by the compression in spatio-temporal *query results*, we presented the first theoretical model that estimates this error in the results of timeslice queries. We provided a closed formula of the average number of false hits (false negatives and false positives) covering the case of arbitrarily distributed trajectory data with various speeds, headings etc. Under various synthetic and real trajectory datasets, we first illustrated the applicability of our model under real-life requirements – it turns out that the estimation of the model parameters introduce only a small overhead in the trajectory compression algorithm - and then presented the accuracy of our estimations, with an average error being around 6%.

7. Epilogue

7.1. Conclusions

In this thesis we have presented several techniques to support the efficient management of Trajectory Databases. Specifically, we provided effective mechanisms that allow Moving Object Databases to efficiently store and query historical trajectories, advancing the fields of indexing, query processing, supporting of uncertainty and trajectory compression. Next, we discuss the specific contributions of this thesis.

In Chapter 2, we provided two novel indexing techniques; among them, the first advances an existing solution while the second exploits network-constrained movement so as to outperform general solutions. Specifically, on the case where objects move freely in the space, acknowledging the basic advantages of the TB-tree [PJT00], we proceed one step beyond by proposing a novel index, called TB*-tree. The proposed index overcomes the main disadvantages of its predecessor while at the same time preserving all of its 'desired' properties: it supports trajectory insertions and deletions, trajectory compression, while querying is performed by employing the same algorithms provided in [PJT00]. In the second case of network-constrained objects, we provide the Fixed Network R-tree, which is forest of several 1D (1D) R-trees [Gut84] on top of a single 2D (2D) R-tree [Gut84]. The 2D R-tree is used to index the spatial data of the network graph, while the 1D R-trees are used to index the time interval of each object's movement on a given segment of the network. Additionally, the leaf nodes of all the 1D R-trees are indexed by another 1D R-tree used to answer queries with no spatial extent. We experimentally compared the FNR-tree with the TB*-tree and the traditional 3D R-tree [TVS96] and TB-tree [PJT00]. Under various datasets and range queries, the FNR-tree was shown to outperform all its competitors in the vast majority of settings. The FNR-tree has high space utilization, smaller size per moving object and supports range queries much more efficiently. In general, we argue that the FNR-tree is an access method ideal for fleet management applications. However, the FNR-tree may only be used under the network-constrained scenario; when objects are moving freely in the space, the TB*-tree is shown to outperform the original TB-tree in the vast majority of settings, regarding insertion and querying operations. Moreover, the TB*-tree is more compact than its competitors, behaves well in non-chronological trajectory insertions that appear in real-world environments, and supports trajectory deletions and trajectory compression efficiently.

In Chapter 3, we studied the problem of performing nearest neighbor queries over historical trajectories. Related work on this subject so far, mainly deals with either stationary or moving query

points over static datasets or future (predicted) locations over a set of continuously moving points. In contrast, in this thesis, we presented the first complete treatment of historical NN queries over moving object trajectories stored on R-tree-like structures; as such, the presented solutions may be applied to a variety of indexes, such as the 3D R-tree [TVS96], the TB-tree [PJT00], as well as the novel TB*-tree. We provide a set of novel metrics, and advance existing work in the calculation of the well-known MINDIST metric between line segments and rectangles. The metrics support our ordering and pruning strategies which are subsequently employed in a set of algorithms answering nearest neighbor and historical continuous nearest neighbor queries for stationary or moving query points. The presented algorithms, following both depth-first [RKV95] and best-first [HS99] paradigms, are then generalized to search for the k nearest neighbors. In order to measure the performance of the introduced algorithms we conducted an extensive experimental study based on synthetic and real datasets. Regarding the historical non-continuous algorithms, it has been shown that while the incremental (best-first) approach is always less expensive than the non-incremental (depth-first) in terms of node accesses, its actual execution time heavily depends on the used queue length. In general, the best first approach outperforms its competitor only for point NN queries under small temporal extent (less than 2-4% depending on the index used and under any k), while in all other cases the depth first approach takes less time to be executed. This drawback of the incremental algorithms is mainly due to the queue length which may become huge, especially in the case of the TB-tree and the TB*-tree. Moreover, we demonstrated that our improvement over the *MINDIST* computation can sufficiently increase the performance of the proposed algorithms. Finally, the experimental study shows that the majority of the presented algorithms are linear or sub-linear with the main parameters of our experimental study (in terms of node accesses): the dataset cardinality, the query temporal extent and the number of k.

In Chapter 4, based on our work on NN search, we examined the problem of most similar trajectory (MST) search. More specifically, existing related work on similarity query processing either ignores the temporal dimension of trajectories, or considers trajectories with the same sampling rate. Then again, in this thesis we relaxed these assumptions by defining a novel metric based on the average Euclidean distance between trajectories, called *DISSIM*, while we subsequently presented a complete treatment of historical MST queries over moving object trajectories stored on R-tree like structures avoiding the drawbacks of the existing methods. We proposed a set of metrics, based on simple notions of trajectories, such as the dataset maximum speed, each one followed by a lemma that support our ordering and pruning strategies; then we presented two MST algorithms. Under various synthetic and real trajectory datasets, we illustrated the superiority of the proposed *DISSIM* metric against related work [VKG02], [COO05], in terms of quality, while our algorithms show high pruning ability when processing MST queries, also verified in the case of k-MST queries. Among the algorithms proposed, the `BFMSTSearch` following the best-first paradigm [HS99] seems more promising since it shows better performance over its competitor `DFMSTSearch`; in particular, it demonstrates linear behaviour in terms of execution time and node accesses, while its pruning power remains above 90% in all settings tested during the experimental study (whereas the pruning power of `DFMSTSearch` degrades to very small values as the query length increases).

Here, we have to point out that all the proposed algorithms on nearest neighbor and similarity queries do not require any dedicated index structure and can be directly applied to any member of the R-tree family used to index trajectories, such as the 3D R-tree [TVS96], the TB-tree [PJT00] and the TB*-tree proposed in this thesis. To the best of our knowledge, the application of the proposals of this thesis enables for the first time a spatio-temporal index to support classical range, topological, nearest neighbor and similarity based queries. Moreover, a number of the proposed, in this thesis, techniques have been implemented in the ORACLE Object – Relational DBMS and integrated into the HERMES engine [PFGT08]. In particular, the HERMES engine, so far, has been extended so as to include the TB-tree [PJT00], along with the point and trajectory nearest neighbor algorithms presented in Chapter 3.

Regarding the management of the location uncertainty of spatio-temporal trajectories, in Chapter 5 we argued that there are cases where the user would prefer to know the influence of the objects' uncertainty in the query results, without actually executing the query. Such cases include interactive database querying, imprecision settings, data warehouse operations and querying under the open agoras scenario [Ioa07] as extensively discussed in this chapter. Towards this goal, we provided a theoretical model that estimates the error introduced by each object's location uncertainty in the results of timeslice spatio-temporal queries, as well as, over simple range queries over stationary spatial data. The model proposed consists of a closed formula that calculates the average number of false hits, classified as false positives and false negatives, under three initial assumptions: uniform location uncertainty (following the model proposed by [TWHC04] in order to describe the uncertain position of trajectories), uniformly distributed data, and, constant value of the uncertainty threshold [TWHC04] (radius of uncertainty circle). Then, we relaxed these assumptions towards more realistic settings, using the bivariate normal distribution for describing the location uncertainty and *MinSkew* histograms so as to support arbitrary data and uncertainty radiuses distributions. The accuracy of the proposed model over spatio-temporal trajectory data was evaluated through experimentation using various synthetic spatio-temporal datasets. In particular, our model is shown to provide high accuracy with an average error on $\overline{E_P}$ and $\overline{E_N}$ never exceeding 6% for spatio-temporal and stationary spatial data. Regarding the application of our model over trajectory data, the model showed values of $\overline{ES_P}$ and $\overline{ES_N}$, i.e., average absolute error in each individual query, near 40%. While at a first prospect this error seems to be high, in reality, it is due to the small number of individual trajectories used (according to the formulas, for typical query sizes and uncertainty radiuses we expect $0.0004 \times N$ and $0.0025 \times N$ as false positives / negatives per query). Bearing these values in mind, it becomes clear that for typical query sizes and uncertainty radiuses, the dataset population should be quite large in order to produce a significant number of false hits, suitable to be counted and compared against the results of the proposed model. Therefore, the details of the developed model using a variety of settings were further investigated using synthetic and real spatial datasets of appropriate cardinality. Regarding the applicability of the model in stationary synthetic (random) spatial datasets, the estimation of the number of false hits is accurate regardless of the value of the query size and the radius d of the uncertainty circle, or σ in the case of data with normally distributed uncertainty. The experiments over real spatial data demonstrate accuracy even higher than the one reported for synthetic data, with very

low $\overline{ES_P}$ and $\overline{ES_N}$ errors, indicating the advantage introduced by the employment of histograms, even in the case of variable σ. The results on the application of the proposed model in spatial data cubes and spatial OLAP operations are also very promising. Finally, the implementation of the proposed solutions in real-world environments (PostgreSQL [Post08a] with PostGIS spatial extension [Post08b]) has shown the efficiency of this proposal when employed as an estimator, since its execution time is typically only a few milliseconds. The proposed model, apart from its application in MODs, can be directly employed in existing SDBMS in order to provide users with the accuracy of spatial query results based only on known dataset and query features, while off-the-self histograms already employed in spatial databases for query optimization purposes, can serve our model without the need for any additional adjustments.

The last subject of this thesis is the management of the effect of trajectory compression algorithms in spatio-temporal querying. Related work on this domain so far, has focused on the development of compression algorithms also emphasizing on the error introduced in the position of each object from the compression. On the other hand, in Chapter 6, acknowledging that users are more likely concerned about the error introduced by the compression in spatio-temporal *query results*, we presented the first theoretical model that estimates this error in the results of timeslice queries. We provided a closed formula of the average number of false hits (false negatives and false positives) covering the case of arbitrarily distributed trajectory data with various speeds, headings etc. It turns out that the error is depended on the summation of the absolute values of $\delta x_{i,k}$ and $\delta y_{i,k}$ (i.e., the difference between the compressed and the original trajectory, along the *x*- and *y*- axis, respectively) at every timestamp t_k the original trajectory sampled its position. Moreover, exploiting the developed formula, in that chapter we provide the intuition for a novel approach that may improve the efficiency of existing trajectory compression algorithms. Given that according to the model, the error is depended on the absolute values of $\delta y_{i,k}$ and $\delta x_{i,k}$, its minimization should involve the minimization of $|\delta x_{i,k}|+|\delta y_{i,k}|$, instead of the minimization of $SED_i(t_k)=\sqrt{\delta x_{i,k}^2+\delta y_{i,k}^2}$ which is considered as the optimization criterion in the majority of the existing trajectory compression algorithms. Under various synthetic and real trajectory datasets, we first illustrated the applicability of our model under real-life requirements – it turns out that the estimation of the model parameters introduce only a small overhead in the trajectory compression algorithm - and then presented the accuracy of our estimations, with an average error being around 6%. It has been therefore shown that our model can be utilized right after the compression of a trajectory dataset in order to provide the user with the average error introduced in the results of spatio-temporal queries of several sizes (bringing in only a small overhead). Then the user could use it as an additional criterion so as to decide whether compressed data are suitable for his/her needs, and possibly choose on different compression rates, and so on.

7.2. Open Issues

Several research fields are remained open in the field of trajectory data management. In the next paragraphs we describe the future research work directly fountain by the advances of this thesis.

On the subject of trajectory indexing, database technology has been advanced during the years, proposing indexes that overcome the efficiency of our proposals. Both TB*- and FNR-tree were introduced in the early stages of this thesis; meanwhile, other structures have been proposed in the literature and proved to be more efficient. Currently, the state-of-the-art regarding the indexing of objects moving in unrestricted and network-constrained space is considered the PA-tree [NR07]and the MON-tree [AG05], respectively. On the other hand, according to the results of our respective experimental study, as well as corresponding results published in [AG05], structures exploiting the network-constrained movement are much more efficient than those indexing objects in the unrestricted space; actually, the former usually outperform the lalter by orders of magnitude. However, none of the proposed network-constrained index structures is designed to preserve trajectories: both FNR- and MON-tree by definition lack a mechanism to retrieve trajectories and only care about the processing of coordinate-based queries. Even SETI [CEP03], which is one of the most efficient indexing schemas in unrestricted space regarding coordinate-based queries, suffers from the same drawback. However, as mentioned in Chapter 2, the trajectory preservation is prerequisite to process trajectory-based queries. As such, the first research direction arising on the subject of trajectory indexing is the development of access methods that efficiently support trajectory-based querying under both unrestricted and network-constrained space.

Regarding advanced query processing, our proposal enables R-tree-like structures to efficiently support NN and MST queries; on the other hand, none of the proposed spatio-temporal indexes, apart from R-tree-like structures, consider NN or MST search algorithms. However, for some of them (e.g., FNR-tree), NN querying can be probably supported. A first idea on this subject is that since in the FNR-tree the underlying network is indexed by a conventional R-tree, the best first-algorithm described in [HS99] can be employed in order to find the spatial nearest neighbor; then, given that the network line segments (e.g., the spatial elements of the trajectory segments) are reported in incremental order of their distance from the query object, the algorithm would have to report such nearest segments until retrieving the first overlapping the query in the temporal dimension; a similar approach can be also employed in MON-tree.

Future work on advanced query processing also includes the development of algorithms to support distance join queries ("*find pairs of objects passed nearest to each other (or within distance d from each other) during a certain time interval and/or under a certain space constraint*"), and *Time-Relaxed MST* queries over trajectories using the proposed *DISSIM* metric. This type of query calculates the minimum dissimilarity between trajectories regardless of the time instance in which the query object starts. The algorithms should consider trajectories indexed by R-tree-like structures, which are the most popular trajectory indexes. Yet, the most promising future work is considering the employment of the *DISSIM* metric together with the ordering and pruning techniques developed in this thesis, so as to support efficient similarity range search, a query type having great applications in the data mining domain. In particular, since the application of the generic density-based clustering algorithm OPTICS [ABKS99] according to the *DISSM* metric [NP06] requires finding for each trajectory in the dataset, the number of trajectories being closer (i.e., more similar) than a given value of distance (similarity) the exhaustive scan that is used in the implementation of [NP06] turns to be a

very expensive operation. However, under such circumstances, an R-tree based method for trajectory similarity range queries, would significantly improve performance over alternative indexing and querying strategies.

Finally, future work on advanced query processing should include the development of cost models for NN [TZPM04] and MST queries on historical trajectories. On the same manner, selectivity estimation formulae for query optimization purposes should be developed investing on the work presented in [TSP03] for predictive spatio-temporal queries.

A side advance of this thesis, presented in Section 5.4.2 is the development of a spatio-temporal histogram, based on existing approaches of spatial databases [APR99], for supporting the selectivity estimation of timeslice queries. On the other hand, the estimation of the number of distinct trajectories, for general range queries (i.e., with temporal extent ≠ 0), is not a trivial task, since it involves the well known *distinct-counting problem* [TKC+04]. The distinct-counting problem stands when an object samples its position in several timestamps inside a given query window, resulting to be counted multiple times in the query result. [TKC+04] provide a solution to the aforementioned problem by integrating spatio-temporal indexes with sketches, traditionally used for approximate query processing. However, their proposal reduces the space requirements only a few times (typically about the 40% of the original database size), while the corresponding index structure is maintained on the disk. Clearly, such an approach cannot be utilized instead of histograms (having a typical size of a few kilobytes [APR99]), since it introduces a sizeable overhead in terms of both memory space and processing time requirements.

In the same fashion, a spatio-temporal histogram concerning about the number of distinct trajectories, would have to partition the space into several spatio-temporal buckets, counting the number of distinct trajectories inside each bucket. However, when trying to produce an estimation of the selectivity of a query window which contains more than one bucket, this estimation cannot be computed as the sum of the cardinality of two buckets since trajectories may be counted several times depending on the number of buckets they overlap. Figure 7.1 exemplifies this problem, illustrating four histogram buckets (B_1, B_2, B_3, B_4) along with their respective selectivity $Sel(B_i)$; the total selectivity reported by all four buckets $Sel\left(\bigcup B_i\right) = 3$ is far from being the sum of per bucket selectivities $\sum Sel\left(B_i\right) = 7$ because trajectories T_1, T_2, T_3 will be counted as many times as the buckets each of them overlaps. Moreover, the same problem arises during the histogram construction following the methodology introduced in [APR99] for simple spatial histograms: the construction algorithm initially calculates the number of distinct objects inside each cell produced by a dense spatial grid, and then, in each iteration it aggregates groups of cells to form more wide buckets based on the *MinSkew* heuristic. However, during this aggregation, the number of trajectories inside each resulted bucket has to be calculated, clearly, not as the sum of the trajectories contained inside each fundamental cell.

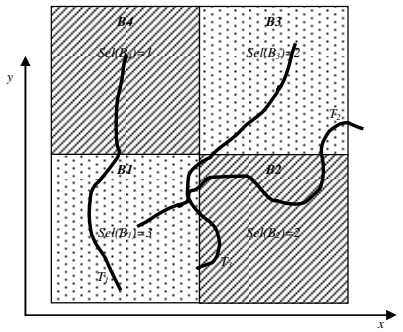


Figure 7.1. The distinct-counting problem in trajectory histograms

Regarding the subject of uncertainty management, there are numerous interesting research directions arising from the work presented in this thesis, including the application of our model in data spaces of higher dimensionality and its extension in order to support general spatio-temporal range queries (i.e., with temporal extent ≠ 0), non-point datasets, non-rectangular query windows as well as nearest neighbor queries. The majority of the aforementioned research directions require significant effort. Among them, the first that must be examined in the context of spatio-temporal databases is its extension in the case of general range queries. This is not a trivial task; nevertheless, we subsequently provide hints towards this direction. Consider, for example, Figure 7.2 illustrating trajectories of three moving objects along with their uncertainty regions (i.e., the dotted areas) in the *x*-*t* space, along with a range query (Due to simplicity reasons all trajectories are illustrated as line segments without loss of generality). Trajectories T_1 and T_2 can not ever encounter a false hit regarding the query window due to the fact that for at least one time instance their uncertainty region was entirely located inside it. On the other hand, trajectory T_3 may encounter a false hit because it is not inside the query window; nevertheless, its uncertainty region crosses it. Generalizing the above observation, we can state that only objects whose uncertainty area crosses the query window without being entirely inside it at any time instance, may contribute to the number of false hits in the results of the query.

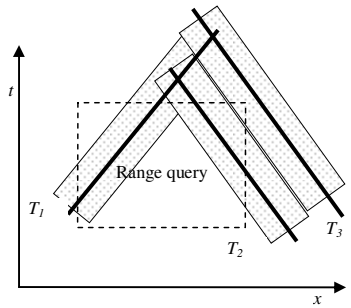


Figure 7.2. The effect of uncertainty in general range queries

The last subject considered in this thesis, i.e., trajectory compression, gives also rise to numerous interesting research directions, including the development of the presented model's counterparts for nearest neighbor queries, or even more, general spatio-temporal range queries. More specifically, the extension of our approach towards the second direction, would require to determine the shape of the spatio-temporal space inside which the lower left range query corner (i.e., the minimum point of the range query) has to be found in order for the compressed trajectory to be retrieved as a false hit

(negative of positive), in accordance with Figure 6.6, Figure 6.7, and subsequently to determine its volume in accordance with Eq.(6.4). Although this volume can be calculated when δx_i and δy_i are expressed as single functions (i.e., between consecutive timestamps), in the general case where δx_i and δy_i are expressed as multi-functions (i.e., different functions in different original trajectory line segments), the respective volume is very hard to be determined. Nevertheless, it remains as a great challenge for future work.

Finally, it would be interesting for one to apply our intuition regarding the appropriate minimization criterion of trajectory compression algorithms, so as to provide a novel approach that improves the efficiency of existing solutions. This efficiently would be measured in terms of compression rates vs. number of false hits introduced in spatio-temporal queries due to compression, contrary to existing approaches which measure it in terms of the average error introduced in the position of each trajectory [MB04].

8. References

[ABKS99] Ankerst, M., Breunig, M., Kriegel, H.,P., and Sander, J.: OPTICS: Ordering Points To Identify the Clustering Structure. *Proceedings of ACM SIGMOD*, 1999

[ACNV99] Arcieri, F., Cammino, C., Nardelli, E., and Venza, A. The Italian Cadastral Information System: a Real-Life Spatio-Temporal DBMS. *Proceedings of STDM* 1999

[AFH02] Agarwal, P. K., Flato, E., Halperin, D., Polygon decomposition for efficient construction of Minkowski sums, *Computational Geometry*, 21(1-2): 39-61 (2002)

[AFS93] Agrawal, R., Faloutsos, C., and Swami, A., Efficient Similarity Search in Sequence Databases. *Proceedings of FODO*, 1993.

[AG05] Guting, R., H., Almeida, V., T., Indexing the Trajectories of Moving Objects in Networks. *GeoInformatica* 9(1):33-60, 2005.

[AGB06] Almeida, V., T., Guting, R., H., and Behr, T. Querying Moving Objects in SECONDO. *Proceedings of MDM*, 2006

[APR99] Acharya, S., Poosala, V., and Ramaswamy, S., Selectivity Estimation in Spatial Databases. *Proceedings of ACM SIGMOD*, 1999.

[BC96] Berndt, J. and Clifford, J., Finding patterns in time series: A dynamic programming approach. Advances in Knowledge Discovery and Data Mining. *AAAI/MIT Press*, 1996

[BJKS02] Benetis, R., Jensen, C., Karciauskas, G., and Saltenis, S., Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. *Proceedings of IDEAS*, 2002.

[BKSS90] Beckmann, N., Kriegel, H.P., Schneider, R., and Seeger, B. The R^*-tree: An Efficient and Robust Access Method for Points and Rectangles. *Proceedings of ACM SIGMOD*, 1990.

[BS03] Beresford, A. R., and Stajano, F. Location Privacy in Pervasive Computing. *IEEE Pervasive Computing*, 2(1):46-55, 2003.

[Bri02] Brinkhoff, T.: A Framework for Generating Network-Based Moving Objects, *Geoinformatica*, 6(2):153-180, 2002

[BW01] Babu, S., and Widom, J., Continuous Queries over Data Streams, *SIGMOD Record*, 30(3):109-120, 2001.

[CC02] Choi, Y.-J., and Chung, C.-W., Selectivity estimation for spatio-temporal queries to moving objects. *Proceedings of ACM SIGMOD*, 2002

[CC07] Chen, J. and Cheng, R., Efficient Evaluation of Imprecise Location-Dependent Queries. *Proceedings of IEEE ICDE*, 2007

[CEP03] Chakka, V.P., Everspaugh, A. and Patel, J., Indexing Large Trajectory Data Sets with SETI. *Proceedings of CIDR*, 2003.

[CF98] Cheung, K.L., and Fu, A.,W., Enhanced Nearest Neighbour Search on the R-tree. *SIGMOD Record*, 27(3):16-21, 1998

[CF99] Chan, K.P., and Fu, A.W-C., Efficient time series matching by Wavelets. *Proceedings of ICDE*, 1999.

[CKP04] Cheng, R., Kalashnikov, D., and Prabhakar, S., Querying Imprecise Data in Moving Object Environments. *IEEE TKDE* 16(9):1112-1127, 2004.

[CN04] Cai, Y., and Ng, R., Indexing spatio-temporal trajectories with Chebyshev polynomials. *Proceedings of ACM SIGMOD*, 2004.

[COO05] Chen, L., Tamer Özsu, M., and Oria, V., Robust and Fast Similarity Search for Moving Object Trajectories. *Proceedings of ACM SIGMOD*, 2005.

[CPZ97] Ciaccia, P., Patella, M., and Zezula, P. M-tree: An efficient access method for similarity search in metric spaces. *Proceedings of VLDB*, 1997

[CR99] Chomicki, J. and Revesz, P., A Geometric Framework for Specifying Spatio-temporal Objects. *Proceedings of TIME*, 1999.

[CWT03] Cao, H., Wolfson, O., and Trajcevski, G., Spatio-temporal Data Reduction with Deterministic Error Bounds. *Proceedings of DIALM–POMC*, 2003.

[CXP+04] Cheng, R., Xia, Y., Prabhakar, S., Shah, R., and Vitter, J.S., Efficient Indexing Methods for Probabilistic Threshold Queries over Uncertain Data. *Proceedings of VLDB*, 2004

[CZBP06] Cheng, R., Zhang, Y., Bertino, E., and Prabhakar, S. Preserving user location privacy in mobile data management infrastructures. *Proceedings of the 6th Workshop on Privacy Enhancing Technologies*, 2006.

[DP73] Douglas, D. H., Peucker, T. K., Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer* 10 (1973) 112–122.

[DYM+05] Dai, X., Yiu, M.L., Mamoulis, N., Tao, Y., and Vaitis, M., Probabilistic Spatial Queries on Existentially Uncertain Data. *Proceedings of SSTD*, 2005.

[EGSV99] Erwig, M. Güting, R. H., Schneider, M., and Varzigiannis, M., Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. *GeoInformatica* 3(3): 265-291, 1999

[FGNS00] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, A Data Model and Data Structures for Moving Objects Databases. *Proceedings of ACM SIGMOD*, 2000.

[FGPT05] Frentzos, E., Gratsias, K., Pelekis, N., and Theodoridis, Y., Nearest Neighbor Search on Moving Object Trajectories. *Proceedings of SSTD,* 2005.

[FGPT07] Frentzos, E., Gratsias, K., Pelekis, N., and Theodoridis, Y., Algorithms for Nearest Neighbor Search on Moving Object Trajectories. *Geoinformatica* 11(2): 159-193 (2007)

[FGT07] Frentzos, E., Gratsias, K., and Thedoridis, Y., Index-based Most Similar Trajectory Search. *Proceedings of ICDE*, 2007

[FGT08] Frentzos, E., Gratsias, K., and Thedoridis, Y., On the Effect of Uncertainty in Spatial Querying, *IEEE TKDE, accepted*, 2008

[Fre02] Frentzos, E., Spatio-temporal Indexing Techniques. *MSc thesis, National Technical Univesity of Athens*, 2003. Available at http://isl.cs.unipi.gr/db/people/efrentzo (in greek).

[Fre03] Frentzos, E., Indexing objects moving on fixed networks. *Proceedings of SSTD*, 2003.

[FT06] Frentzos, E. and Theodoridis, Y., The TB*-tree: Indexing Moving Object Trajectories in Real-World Environments. *UNIPI-ISL-TR-2006-02, Technical Report Series, University of Piraeus*, 2006. Available at http://isl.cs.unipi.gr/db/people/efrentzo.

[FT07] Frentzos, E., and Theodoridis, Y., On the Effect of Trajectory Compression in Spatio-temporal Querying. *Proceedings of ADBIS*, 2007.

[GBE+00] Guting, R., H., Bohlen, M., H., Erwig, M., Jensen, C., S., Lorentzos, N., A., Schneider, M., and Vazirgiannis, M., A Foundation for Representing and Querying Moving Objects. *ACM TODS*, 25(1): 1-42, 2000.

[GL05] Gedik, B., and Liu, L. A customizable k-anonymity model for protecting location privacy. *Proceedings of ICDCS*, 2005.

[GS05] Güting, R.H., and Schneider, M., *Moving Objects Databases*. Morgan Kaufmann Publishers, 2005.

[Gut84] Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Indexing. *Proceedings of ACM SIGMOD*, 1984.

[HKT03] Hadjieleftheriou, M., Kollios, G., and Tsotras, V., Performance Evaluation of Spatio-temporal Selectivity Estimation Techniques. *Proceedings of SSDBM*, 2003

[HKTG02] Hadjieleftheriou, M., Kollios, G., Tsotras, V. J., and Gunopulos, D., Efficient Indexing of Spatio-temporal Objects. *Proceedings of EDBT*, 2002.

[HKTG06] M., Hadjieleftheriou, G., Kollios, V., Tsotras, and D., Gunopulos, Indexing Spatio-temporal Archives. *The VLDB Journal, to appear*

[HS92] Hershberger, J., Snoeyink, J.: Speeding up the Douglas-Peucker line-simplification algorithm. *Proceeedings of SDH,* 1992.

[HS99] Hjaltason, G., and Samet, H., Distance Browsing in Spatial Databases, *ACM TODS*, 24(2): 265-318, 1999.

[HXL05] Hu, H., Xu, J., and Lee, D.L., A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. *Proceedings of ACM SIGMOD*, 2005.

[Ioa93] Ioannidis, Y., Universality of Serial Histograms. *Proceedings of VLDB*, 1993.

[Ioa07] Ioannidis, Y., Emerging Open Agoras of Data and Information. *Proceedings of ICDE,* 2007

[IP95] Ioannidis, Y. and Poosala, V., Balancing histogram optimality and practicality for query result size estimation. *Proceedings of ACM SIGMOD*, 1995.

[ISS03] Iwerks, G.S., Samet, H., and Smith, K., Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. *Proceedings of VLDB*, 2003.

[Keo02] Keogh, E., Exact indexing of dynamic time warping. *Proceedings of VLDB*, 2002.

[KF93] Kamel, I., and Faloutsos, C.: On Packing R-trees. *Proceedings of CIKM*, 1993.

[KGT99] Kollios, G., Gunopulos, D., and Tsotras, V. On Indexing Mobile Objects. *Proceedings of ACM PODS*, 1999.

[KWX+06] Keogh, E., Wei, L., Xi, X., Lee, S.H., and Vlachos, M., LB_Keogh Supports Exact Indexing of Shapes under Rotation Invariance with Arbitrary Representations and Distance Measures. *Proceedings of VLDB*, 2006.

[Lei95] A. Leick, *GPS satellite surveying*, John Wiley and Sons, New York, 1995.

[LS05] Lin, B., and Su, J., Shapes Based Trajectory Queries for Moving Objects. *Proceedings of ACM-GIS*, 2005.

[MB04] Meratnia, N., By, R., Spatio-temporal Compression Techniques for Moving Point Objects. *Proceedings of EDBT*, 2004.

[MFN+08] Marketos, G., Frentzos, E., Ntoutsi, I., Pelekis, N., Raffaeta, A., and Theodoridis, Y., Building Real-World Trajectory Warehouses. *Proceedings of MobiDE*, 2008

[MHP05] Mouratidis K., Hadjieleftheriou M., Papadias, D., Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. *Proceedings of ACM SIGMOD*, 2005.

[MNPT05] Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., and Theodoridis, Y., *R-trees: Theory and Applications.* Springer-Verlag, 2005

[MXA04] Mokbel, M.F., Xiong, X., and Aref, W.G., SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. *Proceedings of ACM SIGMOD*, 2004.

[NP06] Nanni, M., and Pedreschi, D., Time-focused density-based clustering of trajectories of moving objects. *Journal of Intelligent Information Systems*, 27(3):267–289, 2006.

[NR07] Ni, Y., and Ravishankar, C., Indexing Spatio-temporal Trajectories with Efficient Polynomial Approximations, *IEEE TKDE*, 19(5): 663-678, 2007.

[NRB03] Ni, J., Ravishankar, C.V., and Bhanu, B., Probabilistic Spatial Database Operations. *Proceedings of SSTD*, 2003.

[NST99] Nascimento, M., Silva, J.R.O., and Theodoridis, Y. Evaluation of Access Structures for Discretely Moving Points. *Proceedings of STDM*, 1999

[PFGT08] Pelekis, N., Frentzos, E., Giatrakos, N. and Theodoridis, Y., Aggregative LBS via a Trajectory DB Engine. *Proceedings of ACM SIGMOD*, 2008 (to appear)

[Pfo02] Pfoser, D, Indexing the Trajectories of Moving Objects. *IEEE DE Bulletin*, 25(2):2-9, 2002.

[PKM+07] Pelekis, N., Kopanakis, I., Marketos, G., Ntoutsi, I., Andrienko, G., and Theodoridis, Y., Similarity Search in Trajectory Databases. *Proceedings of TIME*, 2007

[PJ99] Pfoser, D. and Jensen, C.S., Capturing the Uncertainty of Moving-Object Representations, *Proceedings of SSD*, 1999

[PJ01] Pfoser, D., and Jensen, C.S., Querying the trajectories of on-line mobile objects. *Proceedings of MobiDE*, 2001

[PJ03] Pfoser, D., and Jensen, C.S., Indexing of network constrained moving objects. *Proceedings of ACM-GIS*, 2003

[PJT00] Pfoser D., Jensen C. S., and Theodoridis, Y., Novel Approaches to the Indexing of Moving Object Trajectories. *Proceedings of VLDB*, 2000.

[Post08a] PostGIS, URL: http://postgis.refractions.net (accessed 15 May 2008)

[Post08b] PostgreSQL, URL: http://www.postgresql.org (accessed 15 May 2008)

[PPS06] Potamias, M., Patroumpas, K. and Sellis, T., Sampling Trajectory Streams with Spatio-temporal Criteria. *Proceedings of SSDBM*, 2006.

[PPS06a] Potamias, M., Patroumpas, K. and Sellis, T., Amnesic Online Synopses for Moving Objects. *Proceedings of CIKM*, 2006.

[PPS07] Potamias, M., Patroumpas, K. and Sellis, T., Online Amnesic Summarization of Streaming Locations. *Proceedings of SSTD*, 2007.

[PT06] Pelekis N., Theodoridis Y. Boosting Location-Based Services with a Moving Object Database Engine. *Proceedings of MobiDE*, 2006.

[PTJ05] Pfoser, D., Tryfona, N., and Jensen, C.S., Indeterminacy and Spatio-temporal Data: Basic Definitions and Case Study, *GeoInformatica* 9(3): 211-236, 2005.

[PTKZ02] Papadias, D., Tao, Y., Kalnis., P., and Zhang, J.: Indexing Spatio-Temporal Data Warehouses. *Proceedings ICDE*, 2002.

[PTVP06] Pelekis N., Theodoridis Y., Vosinakis S., and Panayiotopoulos T.. Hermes - A Framework for Location-Based Data Management. *Proceedings of EDBT*, 2006.

[RKV95] Roussopoulos, N., Kelley, S., and Vincent, F., Nearest Neighbor Queries. *Proceedings of ACM SIGMOD,* 1995.

[SJ02] Saltenis, S. and Jensen, C. S., Indexing of Moving Objects for Location-Based Services. *Proceedings of ICDE*, 2002.

[SJLL00] Saltenis, S., Jensen, C. S., Leutenegger, S. and Lopez, M., Indexing the Positions of Continuously Moving Objects. *Proceedings of ACM SIGMOD*, 2000.

[SKS03] Shahabi, C., Kolahdouzan, M., and Sharifzadeh, M., A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases, *GeoInformatica*, 7(3): 255-273, 2003.

[SR01] Song, Z., and Roussopoulos, N., K-Nearest Neighbor Search for Moving Query Point. *Proceedings of SSTD*, 2001.

[SYF05] Sakurai, Y., Yoshikawa, M., and Faloutsos, C., FTW: Fast Similarity Search under the Time Warping Distance. *Proceedings of PODS*, 2005.

[TCX+05] Tao, Y., Cheng, R., Xiao, X., Ngai, W.K., Kao, B., and Prahbakar, S., Indexing Multi-Dimensional Uncertain Data with Arbitrary Probability Density Functions. *Proceedings of VLDB*, 2005.

[The03] Theodoridis, Y., Ten Benchmark Database Queries for Location-based Services. The *Computer Journal* 46(6): 713-725, 2003.

[TKC+04] Tao, Y., Kollios, G., Considine, J., Li, F., and Papadias, D., Spatio-Temporal Aggregation Using Sketches. *Proceedings of ICDE*, 2004

[TP01] Tao, Y., and Papadias, D., MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. *Proceedings of VLDB*, 2001

[TP02] Tao, Y., and Papadias, D., Time Parameterized Queries in Spatio-Temporal Databases, *Proceedings of ACM SIGMOD*, 2002.

[TPS02] Tao, Y., Papadias, D., and Shen, Q., Continuous Nearest Neighbor Search. *Proceedings of VLDB*, 2002.

[TPS03] Tao, Y., Papadias, D., and Sun, J., An optimized Spatio-temporal Access Method for Predictive Queries. *Proceedings of VLDB*, 2003

[Tra03] Trajcevski, G., Probabilistic Range Queries in Moving Objects Databases with Uncertainty. *Proceedings of MobiDE,* 2003.

[TS96] Theodoridis, Y., and Sellis, T., A Model for the Prediction of R-tree Performance. *Proceedings of ACM PODS*, 1996.

[TSN99] Theodoridis, Y., Silva, J. R. O., and Nascimento, M. A., On the Generation of Spatio-temporal Datasets. *Proceedings of SSD,* 1999.

[TSP03] Tao, Y., Sun, J., and Papadias, D., Analysis of predictive spatio-temporal queries. *ACM TODS*, 28(4):295-336, 2003

[TVS96] Theodoridis, Y., Vazirgiannis, M., and Sellis, T., Spatio-temporal Indexing for Large Multimedia Applications. *Proceedings of ICMCS*, 1996.

[TWZC02] Trajcevski, G., Wolfson, O., Zhang, F., and Chamberlain, S., The geometry of uncertainty in moving objects databases. *Proceedings of EDBT*, 2002.

[TWHC04] Trajcevski, G., Wolfson, O., Hinrichs, K. and Chamberlain, S. Managing uncertainty in moving objects databases, *ACM Trans., Database Systems,* 29(3), 463-507, 2004.

[TZPM04] Tao, Y., Zhang, J., Papadias, D., and Mamoulis, N., An Efficient Cost Model for Optimization of Nearest Neighbor Search in Low and Medium Dimensional Spaces, *IEEE TKDE* 16(10):1169-1184, 2004

[VGD04] Vlachos, M., Gunopulos, D., and Das, G., Rotation Invariant Distance Measures for Trajectories. *Proceedings of SIGKDD*, 2004.

[VKG02] Vlachos, M., Kollios, G., and Gunopulos, D., Discovering Similar Multidimensional Trajectories. *Proceedings of ICDE*, 2002.

[WD04] Worboys, M., and Duckham, K., *GIS: A Computing Perspective*. CRC Press, 2004

[WSCY99] Wolfson, O., Sistla, P.A., Chamberlain, S., and Yesha, Y., Updating and Querying Databases that Track Mobile Units. *Distributed and Parallel Databases*, 7(3):257-387, 1999.

[XMA05] Xiong, X., Mokbel, M., and Aref, W., SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. *Proceedings of ICDE*, 2005.

[XP03] Yuni Xia, Sunil Prabhakar: Q+Rtree: Efficient Indexing for Moving Object Database. *Proceedings of DASFAA*, 2003

[YAS03] Yanagisawa, Y., Akahani, J., and Satoh, T., Shape-Based Similarity Query for Trajectory of mobile Objects. *Proceedings of MDM*, 2003.

[YM03] Yu, X., and Mehrotra, S., Capturing Uncertainty in Spatial Queries over Imprecise Data. *Proceedings of DEXA*, 2003

[YPK05] Yu, X., Pu, K., and Koudas, N., Monitoring k-Nearest Neighbor Queries Over Moving Objects. *Proceedings of ICDE*, 2005.

[ZG02] J. Zhang and M. Goodchild. *Uncertainty in Geographical Information*. Taylor & Francis, 2002.

[ZSI02] Zhu, J, Su, J. and Ibarra, O., Trajectory queries and octagons in moving object databases. *Proceedings of CIKM*, 2002.