Smart Search & Retrieval on Video Databases

by A.H.R. Albers

Master of Science thesis

Project period: September 2004 – September 2005
Report Number: 31-05

Commissioned by:

Supervisors:
Name (TU/e) Prof. dr. ir. P.H.N. de With
Name (BOSCH) Dr. ir. E.G.T. Jaspers
Ir. R.G.J. Wijnhoven

Additional Commission members:
Prof. name (TU/e) Prof. dr. ir. P.H.N. de With
Prof. dr. ir. H. Corporaal

The Department of Electrical Engineering of the Eindhoven University of Technology accepts no responsibility for the contents of M.Sc. theses or practical training reports
Preface

This thesis presents the research done during the graduation project as part of the Master of Science (M.Sc.) studies at the Department of Electrical Engineering and Information Technology of the Eindhoven University of Technology (TU/e). The project was part of the Signal Processing Systems (SPS) chair of the Measurement and Control Systems (MBS) group of the department. The research was carried out at Bosch Security Systems B.V. in Eindhoven as part of the European ITEA project CANDELA.
Abstract

As a result of the established standards for Digital Video Broadcasting and associated storage, the amount of available content for video applications is growing with a rapid pace. This introduces the necessity for content management, which implies interactive video, where a user is able to interact with the video delivery system. A rising problem is how to quickly search for interesting content on large video databases. Straightforward structuring and indexing only partially solves the problem. A more attractive usage paradigm is content retrieval on request, i.e. search capabilities with queries on a user-friendly level. This requires understanding of the video content and implies Video Content Analysis (VCA) to generate video descriptions describing the content.

Our work concentrates on the video surveillance application, where typically weeks of video from different surveillance cameras are stored. Currently, the identification of occurred incidents is not a straightforward task, resulting in playback of videos from many cameras, which is often very time-consuming. To be able to quickly search through the stored video content, generated video descriptions are stored, processed and retrieved to match an abstract search query given by the user.

VCA algorithms performing object tracking generate trajectory metadata on a video frame-by-frame basis, which is generally not suitable for fast retrieval. Therefore, a novel interval-based sampling method is proposed which converts the metadata into a higher semantically form. Moreover, the sampled object features are efficiently stored in a hierarchical database to exploit the available capacity optimally. We have proposed several fast search techniques, using location data of VCA algorithms for efficient storage and retrieval in video databases. The gain of our system is caused by storing sampled features of the objects behaviour, rather than a frame-based extraction of parameters. Additionally, an advantage is the hierarchical storage of trajectory metadata, so that only a part of the stored data has to be examined for object data retrieval.

The results of the research were integrated and evaluated in a demonstrator, that was successfully demonstrated in June at the final CANDELA review meeting in Helsinki, Finland. A graphical user interface enables fast retrieval and shows the benefits of the high-level data processing by applying intuitive semantic search queries. We show that for several scenarios, the search time on large video data sets significantly reduces, compared to traditional methods.

The adopted algorithms can readily be applied to home multimedia or sport video analysis applications, because objects show similar behaviour (e.g. fast retrieval of interesting soccer player activities).
Contents

Preface i

Abstract iii

1 Introduction 1
   1.1 Preliminaries ............................................. 1
   1.2 Problem definition ........................................... 2
   1.3 Related work ................................................. 4
   1.4 Document layout ............................................ 5

2 Metadata Analysis 7
   2.1 Introduction ................................................. 7
   2.2 Interval-based description ..................................... 8
   2.3 Related work ................................................. 9
   2.4 Algorithmic choices ......................................... 10

3 Efficient Storage 13
   3.1 Introduction ................................................. 13
      3.1.1 Point Access Methods .................................... 13
      3.1.2 Spatial Access Methods ................................... 14
   3.2 System approaches ............................................ 16
      3.2.1 Quadtrees ................................................ 16
      3.2.2 R-trees ................................................ 17
   3.3 Algorithmic choices ......................................... 18

4 Search & Retrieval 19
   4.1 Introduction ................................................. 19
   4.2 Architecture ................................................ 19
   4.3 Trajectory Similarity Search ................................ 20
      4.3.1 Introduction ............................................. 20
      4.3.2 Preprocessing ........................................... 20
      4.3.3 Query Execution .......................................... 21
      4.3.4 Postprocessing ........................................... 22
      4.3.5 Rank-Join Results ....................................... 28
   4.4 Tripwire Search .............................................. 29
      4.4.1 Introduction ............................................. 29
      4.4.2 Preprocessing ........................................... 29
      4.4.3 Query Execution .......................................... 30
4.4.4 Postprocessing ........................................... 30
4.4.5 Rank-Join Results ........................................ 32

4.5 Region-of-Interest Search ..................................... 33
4.5.1 Introduction ............................................. 33
4.5.2 Preprocessing ........................................... 33
4.5.3 Query Execution ......................................... 33
4.5.4 Postprocessing ........................................... 34
4.5.5 Rank-Join Results ......................................... 34

5 Performance Optimization ..................................... 35
5.1 Data Representation Optimizations .......................... 35
5.2 Data Indexing Structure Optimizations ..................... 36
5.2.1 Higher-dimensional storage ................................ 36
5.2.2 Date and time storage .................................... 38
5.2.3 Separate data structure .................................... 39

5.3 Search & Retrieval Optimizations .......................... 39
5.3.1 Preprocessing ........................................... 39
5.3.2 Rank-Join ................................................. 42
5.3.3 Trajectory Similarity Search ............................. 42
5.3.4 Tripwire Search .......................................... 43

6 Benchmarking & Results ...................................... 45
6.1 System Implementation ...................................... 45
6.2 Data sets ..................................................... 48
6.3 Results ..................................................... 50
6.3.1 Metadata Analysis ....................................... 50
6.3.2 Efficient Storage ....................................... 52
6.3.3 Search & Retrieval ...................................... 59
6.4 Conclusions ................................................. 62
6.5 Future Outlook .............................................. 63

7 Conclusions and Recommendations ............................ 65
List of Figures .................................................. 69
List of Tables ................................................... 73
References ....................................................... 77
A Postprocessing & Ranking Example ........................... 83
B Data Reduction Methods ....................................... 91
Chapter 1

Introduction

1.1 Preliminaries

The introduction of Digital Video Broadcasting created a broad range of new possibilities for video. Digital compression standards enable high-quality video storage at a low bitrate. Currently, the development in the area of digital video is mainly focused on state-of-the-art video compression (MPEG-4/H.264), describing the video content (MPEG-7), and standardizing a framework to enable interoperability (MPEG-21). Consequently, the amount of available content for video applications is growing with a rapid pace, thereby introducing the necessity for content management. This implies interactive video, where a user is able to interact with the video delivery system. A problem arises how to quickly search for interesting content. Straightforward structuring and indexing only partially solves the problem. An attractive usage paradigm is content retrieval on request, i.e. search capabilities with queries on a human-meaningful semantic level. To do so, developments in the area of digital video need to address the application-specific requirements for content management.

A solution is the annotation of the video stream with a description of the semantic content stored in the form of metadata, thereby allowing the search and retrieval of videos based on the types of actions, objects, behaviour, scenery etc. Because annotation by hand is a very time-consuming process, Video Content Analysis (VCA) algorithms attempt to generate video description automatically. In Fig. 1.1a, a video image is shown, containing two moving objects. State-of-the-art VCA algorithms are able to track objects in the video image over time and generate video descriptions automatically (See Fig. 1.1b). To create a system to search quickly through the video set for interesting video shots, the generated descriptions must be stored, processed and retrieved to match an abstract search query that is given by the user.

This thesis discusses the exploitation of object-tracking information to enable high-semantic search querying on location, trajectories, speed, etc. Typical queries could be the search for vehicles that followed a specific route by sketching the requested trajectory as an overlay on the video window or retrieve certain ball motion combinations in the archive of a snooker games tournament. The contribution of this thesis can be applied in various domains, like sports video analysis applications, home multimedia or video surveillance. However, for a clear understanding, we will focus on video sur-
veillance, in particular the processing of video data from a camera observing traffic behaviour.

![Video image](image1.png) ![Video Content Analysis](image2.png)

*Figure 1.1: A video image (a) and a VCA algorithm performing object tracking (b).*

### 1.2 Problem definition

At present, video surveillance systems act as large-scale digital video recorders. Their primary focus is the application of video compression technology to efficiently store images from a large number of cameras onto disk. They serve two key purposes: providing a human operator with images to react to potential suspicious behaviour and recording evidence for investigative purposes. A trend to ensure high levels of security at public access facilities is to increase the number of surveillance cameras. Simultaneously, the same number of human operators have to beware for incidents (See Fig. 1.2). A recent study concludes that the probability of detecting an incident on the cameras is substantially reduced when the number of cameras, simultaneously monitored by an operator, is increased [1]. Another study demonstrates that human visual attention drops below acceptable levels after only twenty minutes of visual monitoring camera images [2]. The employment of additional operators to monitor all surveillance camera images adequately is expensive and therefore desired to be minimal. Also, the identification of occurred incidents is not a straightforward task, resulting in playback of videos from many cameras, which is often very time-consuming. This implies the introduction of content retrieval on request, e.g. empowering human operators with alarming and search capabilities on a user-friendly level. This to enable real-time threat detection while systems automatically analyse the camera images. Additionally, it enables fast automated searching through large collections of surveillance video content for investigative purposes.

As part of the European ITEA project CANDELA, we try to demonstrate this concept for the surveillance application and prove its feasibility. In Figure 1.3, an overview of the proposed system is shown. The starting point of our research is a state-of-the-art Video Content Analysis (VCA) algorithm that performs automatic object detection and tracking on video frames generated by a Video Acquisition (ACQ) module con-
1.2. PROBLEM DEFINITION

Figure 1.2: Trends in video surveillance: Increasing number of cameras.

Connected to a surveillance camera. The VCA algorithm returns a video frame-by-frame-based description of the object by its bounding boxes. The centre of the bounding box is used as the location of the object. Fig.1.1b shows how the trajectory of an object is defined as the curve connecting the location points over the object's lifetime. For the surveillance application, we have investigated the usage of VCA output in a search and retrieval system to enable search queries that are related to the locations of objects.

The following search queries are identified as interesting: (1) search for objects for which the trajectory is similar as a user-defined trajectory; (2) search for objects for which the trajectory crosses a user-defined tripwire; (3) search for objects for which the trajectory intersects a user-defined region-of-interest.

The data format received from the VCA algorithm is not suitable for efficient storage in a database, nor for fast matching with a query request. If for example the video frame rate is set at 25 frames per second and there are on the average of ten detected objects in the scene, processing of an hour video by the VCA algorithm generates already more than one million location coordinates. To process a similarity query, this requires a computational complexity linear to the amount of frame-by-frame descriptions of all objects. Therefore, to exploit the frame-by-frame-based trajectory information generated by the VCA for fast search and retrieval, a MetaData Analysis (MDA) module is introduced. Effective interval-based trajectory features must be created to store only the relevant information. Note that it is not desirable to sequentially process all stored trajectory information during retrieval, because this still requires a computational complexity that is linear to the amount of objects, and thus approximately linear with the amount of video. Therefore, an Efficient Storage module is necessary to store the object trajectories efficiently. In the Graphical User Interface (GUI), one should be able to draw search queries on location graphically. The sketching on a video image representing the background view of the camera is therefore recommended.

To effectuate a system as visualised in Fig. 1.3, several challenges are distinguished for the three identified modules within the dashed area.

- How to structure the data representation to model trajectory descriptions in a sampled interval-based form to enable effective formatting and efficient storage?
CHAPTER 1. INTRODUCTION

Figure 1.3: Overview of the proposed system.

- Which efficient storage structure is used to provide fast searching, without searching through the whole data set during the query process?

- Finally, we need to define similarity models: Which metrics are going to be used as a distance (quality) measure between trajectories and the search query?

In the following chapters, we will describe the above-mentioned challenges separately, together with the requirements specification.

1.3 Related work

In literature, many algorithms for Video Content Analysis (VCA) have been proposed. In several overview articles [3, 4, 5], state-of-the-art algorithms are discussed that are able to automatically recognize specific object features, distinguish objects from the background and tracks them over time. Typically, the descriptions from the analysis have a low abstraction level. For example, a moving object is commonly described by a bounding box. At a higher abstraction level, additional analysis of the metadata from the VCA is required to enable human understandable search queries. To obtain a higher semantic level of the metadata, a-priori knowledge about the environment is required. Petkovic and Jonker [6] have already proposed a system that separates general VCA processing and additional knowledge processing about the application-specific environment. Several survey papers exist about content-based retrieval systems [7, 8, 9], although they are mainly focused on image-based retrieval systems.

A broad range of articles discusses the topic to raise the semantic level of metadata. Firstly, there is a category of papers that use a combination of spatial and temporal reasoning on top of low-level VCA processing to be able to classify object types, detect certain events and recognize interactions between objects over time. These systems all originate their ideas from the publications of Allen [10] and Egenhofer [11] about temporal and spatial relationships between objects. Examples of such systems can be found in [12, 13].
There are already some publications that identify parts of the concept to graphically construct location-based search queries. Jagadish [14] proposes a novel method to search for objects intersecting each other, but the proposed data representation is not suitable for trajectory similarity search. Articles that describe object trajectories by a polynomial approximation can be found in [15, 16, 17, 18, 19]. The main drawbacks of polynomial approximation techniques for trajectory similarity search are the lack of support for data storage techniques to enable faster-than-linear search time and no support for partial matching of stored trajectories against a search query.

Some other publications describe the object trajectory by a character string [20, 21, 22, 17]. Therefore, the trajectory is split up in several parts, each represented by a code in a predefined alphabet. The generated string for each trajectory is called a chain code or signature. For similarity search, basic string comparison techniques can be used. However, the performance of these methods all depend on sequential searching of the whole data set of trajectories. Some other proposals include a paper by Bashir et al. [23], which uses principal component analysis (PCA) for generation of efficient trajectory descriptions. The drawback of this method is that in a real-time system, the reduced data set containing PCA descriptions has to be recomputed, each time a new trajectory is added. Chen and Chang [24] use a wavelet approach to model efficient trajectory descriptions. However, no efficient storage techniques were used. Yanagisawa et al. [25] use an efficient spatial storage technique to store coordinates of object trajectories. However, the method has not been tested on large data sets and the described similarity measures are doubtful.

Chang et al. [26] and Donderler et al. [27] each describe a video search and retrieval system that combines search properties such as shape, colour and motion of the object in a sketch-based query interface. However, the usage is limited to small data sets and only applicable to home multimedia applications.

The related work showed us that there are up to now no publications, articles or books available that address the architecture as described in Fig. 1.3. Although solutions exist for parts of the problem, literature was not found that addresses the total path from video acquisition to VCA modules, successively towards an effective formatting of the metadata in a MDA module, to enable storage for fast retrieval on large video databases.

### 1.4 Document layout

Based on the challenges distinguished in the introduction, the proposed modules for the system of Fig. 1.3 will be discussed in the following chapters. Chapter 2 discusses the conversion from frame-by-frame based trajectory descriptions into an effective interval-based trajectory feature representation. Subsequently, in Chapter 3, efficient storage techniques are discussed to achieve a faster than linear search time. In Chapter 4, the search and retrieval architecture is defined. Chapter 5 proposes several performance optimizations and explains the adjustable parameters settings in the system. In Chapter 6, performance results for typical real-world scenarios are given for the proposed applications. Chapter 7 presents conclusions. In the appendices, an example is elaborated to show the performance of the proposed algorithms in more detail.
Chapter 2

Metadata Analysis

2.1 Introduction

In this chapter, the conversion from a frame-by-frame-based trajectory parameter representation towards effective interval-based trajectory features will be described. The proposed architecture is already explained in Fig. 1.3. In a Video Acquisition (ACQ) module, video image frames are captured from a surveillance camera and the image frames are send towards a Video Content Analysis (VCA) module. This module is able to distinguish the stationary background of the video image from the active foreground and tracks moving objects over time on a video frame-by-frame basis. A trajectory of an object is defined by the set of location points over the object's lifetime. See Fig. 1.1 for an example. For calculating similarity between object trajectories, the system has to find the sub-trajectories that have similar contour as the contour of the query trajectory. The data format received from the VCA algorithm is not suitable for fast matching with a query request. To process a similarity query, a computational complexity is introduced linear to the amount of frame-by-frame descriptions of all objects. This results in a very long search time before results can be presented. Sampled interval-based trajectory descriptions, which significantly lower this computational expense, are a prerequisite. A MetaData Analysis (MDA) module is introduced in the system to fulfill this need.

Object movement is continuous in nature. However, the object-tracking information gathered from the VCA module samples an object trajectory at discrete time points. Hence, a trajectory is a sequence of time-stamped positions. To determine the objects spatial location at any position of the continuous trajectory, an interpolation technique must be used. Therefore, in the MDA module, a Piecewise Linear Approximation (PLA) is used [28], due to its algorithmic simplicity and low computational complexity.

The next paragraph describes the MDA module in more detail, addressing several data-reduction methods to create effective interval-based features. The storage and indexing of object trajectories in the Efficient Storage module will be explained in Chapter 3.
2.2 Interval-based description

As a requirement, the system should support the storage of trajectory data in an efficient storage system, to calculate similarity without examining the entire stored data set. Because trajectory data can be seen as spatial data sequences, by omitting the time dimension, we can take advantage of existing multi-dimensional indexing structures to store and search the data efficiently. These methods are also known as Spatial Access Methods (SAMs).

A SAM that does not examine the entire data set could potentially suffer from two problems, false positives and false negatives. False positives are results that do not match with the query. False negatives are trajectories that match the query, but are not returned with the search. Because false positives can be removed in a post-processing stage, they can be tolerated as long as they occur relatively infrequently. In contrast, false negatives are usually not acceptable.

Noted by Faloutsos et al. [29], there are several highly desirable properties for data-reduction methods for usage in a SAM:

1) it should enable much faster searching than sequential processing;
2) it should require little storage overhead;
3) it should be able to handle queries of various lengths;
4) it should allow insertions and deletions without requiring the SAM to be rebuilt;
5) it should be correct, i.e. there should be no false negatives;
6) it should be possible to build the SAM in "reasonable time";
7) the method should be able to handle different distance measures.

Before presenting SAMs for efficient storage of trajectory data in Chapter 3, we first mention a problem common to all spatial access methods with time-series sequences. In order to cope with fast partial matching of time-series data against a search query, a window of size \( n \) is placed over a time-series data sequence, therefore converting a sequence of \( n \) points \((x, t)\) into one \((n \times n)\)-dimensional subsequence vector containing the whole time-series \((x_1, t_1; x_2, t_2; ...; x_n, t_n)\). Experiments have shown that the search performance of SAMs degrades dramatically as the dimensionality of stored sequence vectors increases, and eventually reduces to sequential searching or even worse. Because object trajectories may contain hundreds of coordinates in its frame-by-frame-based form, it is not acceptable to code such sequences directly as \((n \times n)\)-dimensional sequence vectors in a SAM. This phenomenon is known in literature as the curse of dimensionality [30]. Hence, effective interval-based trajectory descriptions are needed to take full advantage of the power of SAMs.

In literature, there exist many methods to reduce a data sequence to remain only a relevant feature description. A framework called GEMINI (GEneric Multimedia INdexing) [29] is introduced to accommodate any data-reduction method to allow efficient indexing of time-series data. Because a sequence of object trajectory coordinates can be seen as a two-dimensional time-series, the techniques for reducing time-series sequence data are evaluated as a basis for creating interval-based features of trajectory data. In the next subsection, we describe the methods that are able to efficiently represent time-series data sequences by only a few relevant features.
2.3 Related work

The methods discussed in this section are based on techniques for reducing the dimensionality of vector data and all support the GEMINI framework. Not all methods are applicable for an interval-based trajectory representation because of the stated requirements. In Subsection 2.4, the architectural choices will be explained.

- **Discrete Fourier Transform** [29]: The basic idea of the DFT is that any realistic signal can be characterized by the superposition of a finite number of sine/cosine waves, each of which is represented by a single complex number known as a Fourier coefficient. A signal of length $n$ can be decomposed into $n$ sine/cosine waves that can be recombined into the original signal. However, many of the Fourier coefficients have a very low amplitude and thus contribute little to the reconstructed signal. These low amplitude coefficients can be discarded without much loss of information and thereby provide data reduction.

- **Discrete Wavelet Transform**: The wavelet transform that is proposed for data reduction is the Haar Wavelet Transform [31]. It is a sequence of averaging and differentiating operations in which we compute the average and difference between every two adjacent values of a data sequence. One important difference with the DFT is that the wavelet coefficients can represent small, local subsections of the data whereas Fourier coefficients always represent global contributions to the data. This former property is very useful for multi-resolution analysis of the data. The first few coefficients contain an overall, coarse approximation of the data; addition coefficients can be added to areas of high detail.

- **Chebyshev Polynomials**: Thanks to their orthogonality and special properties, Chebyshev Polynomials have played a significant role in nearly every area of numerical analysis, including polynomial approximation, numerical integration, integral equations, and spectral methods for partial differential equations. A key property is that the system of Chebyshev polynomials is orthogonal. Two polynomials are orthogonal if their inner product is equal to zero. Given the orthogonality of the Chebyshev polynomials, they can be used as a basis for approximating any function, and thereby enable data reduction [32].

- **Singular Value Decomposition**: Singular Value Decomposition (SVD) [33] is a linear transformation that uses a new coordinate system for the data set such that the greatest variance by a projection of the data set is located on the first axis, the second greatest variance on the second axis, and so on. It is a global transformation technique, and requires the computation of eigenvalues and eigenvectors of large matrices. The entire data set is examined and is then rotated such that the first axis has the maximum possible variance, the second axes have the maximum possible variance orthogonal to the first, etc. Therefore, SVD provides the best least squares fit to any matrix of data points.

- **Piecewise Aggregate Approximation**: PAA reduces the size of time-series of length (dimensionality) $N$ to $n$ ($1 \leq n \leq N$) by dividing each time-series of length $N$ into $n$ segments of equal length, and uses the average value of each segment as a new coordinate. The idea is introduced independently by Yi and Faloutsos [34] and Keogh et al. [35]. The seemingly simple data-reduction scheme has many advantages: it is easy to understand and implement; it is much faster than most other transforms; it can be as efficient as other approaches; it can handle more distance measures than only the general Euclidean norms and has support for weighted distance functions.
• **Adaptive Piecewise Constant Approximation:** APCA [36] is a generalization of the previously discussed PAA by relaxing the requirement that each segment must be of the same length. Intuitively, regions with great fluctuations are represented with several short segments, while relatively flat regions are represented with fewer long segments. As a result, APCA requires the storage of more information per coordinate, namely besides the mean value of all the points in the segment also the segment length to the previous coordinate. Experiments have shown that the APCA representation has a very high fidelity to the original signal and therefore a relatively low reconstruction error.

### 2.4 Algorithmic choices

Looking at the desirable properties for a spatial access method (SAM), as stated at the beginning of this chapter, an overview of the pros and cons of the discussed data-reduction methods can be made. See Table 2.1, for a summary. For the interested reader, a detailed overview can be found in Appendix B.

<table>
<thead>
<tr>
<th>Property</th>
<th>DFT</th>
<th>DWT</th>
<th>Chebyshev</th>
<th>SVD</th>
<th>PAA</th>
<th>APCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Much faster search than seq. processing</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>2. Requires little storage overhead</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3. Handle queries of various lengths</td>
<td>--</td>
<td>-</td>
<td>+</td>
<td>--</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>4. Insert / delete without rebuilding the SAM</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>--</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>5. Guarantee no false negatives</td>
<td>-</td>
<td>+</td>
<td>--</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>6. Build the SAM in 'reasonable time'</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>7. Handle different distance measures</td>
<td>--</td>
<td>-</td>
<td>+</td>
<td>--</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

**Table 2.1:** Summary of the discussed data-reduction methods.

A query trajectory drawn by a user contains the trajectory contour of interest in the spatial \((x,y)\) dimension (Fig 2.1c). Object trajectories processed from the VCA module can be stored as three-dimensional spatio-temporal moving-point patterns \((x,y,t)\) (Fig 2.1a). But then it will be hard or even impossible to match the stored spatio-temporal trajectory coordinates with a spatial query. To overcome this problem, the temporal dimension will be omitted in the MDA module (Fig 2.1b). To do so, the desired data-reduction method should be able to operate solely on the spatial \((x,y)\) dimension. Spatial \((x,y)\) series trajectory data without temporal dimension can have the property to contain more than one \(y\)-coordinate for a single \(x\)-coordinate. Because querying for trajectories should not be limited to certain curves, this behaviour must be supported by the proposed data-reduction method. The candidate methods DFT, DWT and Chebyshev do not function on this kind of spatial data series. Therefore, these three methods are disqualified. The SVD method is a global transformation. Since the whole data set has to be examined before a transformation matrix can be computed, SVD is not incremental. A single insertion to the data set results in a recomputation of the entire transformation matrix. For that reason, SVD is abandoned as a candidate method.
2.4. ALGORITHMIC CHOICES

The candidate methods that remain are the relatively simple data transformation PAA and its adaptive variant, APCA. These two methods are competitive with more sophisticated methods such as DWT and SVD for the description of time-series data and are extensible to deal with spatial trajectory sequences. The PAA and APCA are used as a basis for the conversion from a frame-by-frame-based trajectory representation towards a sampled interval-based trajectory description. Our method stores an averaged value of the coordinates of each processed trajectory segment of equal length $MaxDist$. To better represent parts of the frame-by-frame-based trajectory representation with great fluctuations, the method is adaptive to the value $MaxDirDist$, defined by the distance between a line in the averaged direction of the first $n$ coordinates and the currently processed coordinate. For an example, see Fig. 2.2, where the two limiting values $MaxDist$ (a) and $MaxDirDist$ (b) are visualized. As an example, in Fig. 2.3, the frame-by-frame-based trajectory description versus the resulting sampled interval-based trajectory description are shown for one object on a traffic crossing. A large value for $MaxDist$ and $MaxDirDist$ will result in a very rough trajectory approximation and a large minimum query length to guarantee no false negatives (trajectories that are rejected by the search process, but actually belong to the results). On the other hand, small values for $MaxDist$ and $MaxDirDist$ approximate the object-tracking information very well, but enlarge the storage requirements substantially and require more coordinates to be matched during the search.

In Chapter 6, we benchmark several values for $MaxDist$ and $MaxDirDist$, to be able to select the optimal value. For correct functioning of the algorithms that will be
described in Chapter 4 on search and retrieval, several additional properties are stored with each coordinate in the interval-based trajectory description:

1) a unique object trajectory Id;
2) an object trajectory coordinate Id;
3) the actual interval-based trajectory coordinate \((x,y)\);
4) the distance from the current coordinate to the previous coordinate;
5) the distance from the current coordinate to the next coordinate;
6) the direction angle from the current coordinate to the previous coordinate;
7) the direction angle from the current coordinate to the next coordinate;
8) the start-time and date of the object’s first appearance in the scene.
Chapter 3

Efficient Storage

3.1 Introduction

For calculating similarity between object trajectories, the system has to find the sub-trajectories that have similar contour as the contour of the query trajectory. A requirement of our system is that the interval-based trajectory description, as explained in Chapter 2, supports the storage in an efficient storage system, to calculate similarity without examining the entire data set of trajectories. Because trajectory data can be seen as spatial data sequences, by omitting the time dimension, we can take advantage of existing multi-dimensional indexing structures to store and search the data efficiently.

Gaede et al. [37] classify multi-dimensional indexing structures into Point Access Methods (PAM) and Spatial Access Methods (SAM). PAMs are methods that are primarily designed to perform spatial searches on point databases; databases that store only multi-dimensional points that do not have spatial extension, such as shape, direction etc. On the other hand, SAMs manage objects that, apart from their position in space, have spatial characteristics. Such objects are lines, polygons or higher-dimensional polygons. Object trajectories can be stored in either a PAM or SAM multi-dimensional indexing structure. In the following subsections, the benefits and drawbacks of the most important PAM and SAM methods are explained while looking at the requirements for our system, together with experiences adopted from recent research in the field of multi-dimensional indexing structures and applications.

3.1.1 Point Access Methods

Point access methods generally organize the point data in buckets, each corresponding to a disk page or memory block. The buckets are indexed by either flat or hierarchical data structures. There exist various classifications of PAM, but the author of [37] reports the following categorization for point access methods:

- **Multi-dimensional hashing access methods.** These methods use hashing to index higher-dimensional points. Heuristic techniques are used to ensure that two objects that are close to each other in the multi-dimensional space, will be indexed the same, or close buckets. An example of a hashing method is the grid file [38], as visualised in Fig 3.1.
CHAPTER 3. EFFICIENT STORAGE

Object Tracking Grid file indexing

Figure 3.1: Example of a grid file (b) with two-dimensional point-data (a).

**Hierarchical access methods.** These methods use hierarchical data structures to manage point data. PAMs that fall in this category are the k-d-tree [39] (Fig. 3.2), the k-d-B-tree [40] and the quadtree [41] (Fig. 3.3).

Figure 3.2: Example of a KD-tree (b) with two-dimensional point-data (a).

In recent publications about data storage systems for geographic information systems (GIS), data storage systems for moving points in global positioning systems (GPS) and overview articles, the best performing point access methods are all quadtree variants [42], which we discuss in more detail in the next section.

### 3.1.2 Spatial Access Methods

Point access methods cannot directly be used to manage spatial objects. Spatial access methods are often extensions of PAM, used to cover this need. The authors of [37] classify these methods according to the techniques they use to extend PAM, as follows:
3.1. INTRODUCTION

- **Object mapping methods.** These methods map geometric objects into points in a higher-dimensional space. For instance, a rectangle in $R^2$ can be viewed as one coordinate in $R^4$. Then they use existing PAM such as the quadtree to manage the points.

- **Object bounding methods.** Being the most popular SAM, these methods decompose the space in a hierarchical manner. Objects are grouped and the minimum bounding rectangles (MBRs) are stored at the leaves of the hierarchical structures. Nodes at the same level may overlap each other, so the number of paths that have to be followed in the search for an object can vary. The most promising object bounding methods are the R-tree [43] (Fig. 3.4) and R*-tree [44] (Fig. 3.5), which we discuss in more detail in the next section.

- **Clipping methods.** These methods use hierarchical data structures, as the object bounding methods do, but they use clipping of objects to prevent over-
Object Tracking

Hierarchical R*-tree indexing

Figure 3.5: Example of an R*-tree (b) with two-dimensional point-data (a).

lapping of intermediate nodes at the same level. To achieve this, inserted objects have to be divided in two or more MBRs, which means that specific object entries may be duplicated and redundantly stored in various nodes. Therefore, this redundancy works in the opposite direction of decreasing the retrieval performance in case of window queries. An example of a clipping method is the R+-tree [45].

• Multiple-layer methods. Multiple layer methods partition the space more than one time and each partition is referred to as a layer. Layers are organized in a hierarchical manner. An example of a multiple layer access method is the multi-layer grid file [46].

3.2 System approaches

In this section, we will focus on two structures that have proven their practicability in data storage systems for geographic information systems (GIS) and data storage systems for global positioning systems (GPS). These structures are the quadtree family and the R-tree family.

3.2.1 Quadtrees

Quadtrees are one of the first data structures for higher-dimensional data. They were developed by Finkel and Bentley in 1974 [41]. Since then, there have been been many publications dealing with quadtrees [47, 48]. A quadtree is a tree in which the root has four edges. Every vertex in the quadtree corresponds to a square. If a vertex v has edges, then their corresponding squares are the four quadrants of the square of v. This implies that the squares of the leaves together form a subdivision of the square of the root. Figure 3.3 gives an example of a quadtree. The four edges from the root are labeled NE, NW, SW, and SE to indicate to which quadrant they correspond. The recursive definition of a quadtree immediately translates into a recursive algorithm: split the current square into four quadrants, partition the input data set accordingly and recursively construct quadtrees for each quadrant with its associated input data set. Currently, quadtrees are used mainly for point data, areas, curves, surfaces, and
volumes. The resolution of the decomposition (the number of times that the decom­
sition process is applied) may be fixed at forehand, or it may be governed by properties
of the input data or meeting a certain quality criterion.

Searching in a quadtree is similar to searching in a binary search tree. At each level,
one has to decide which of the four subtrees need to be included in the successive
search step. In the case of a point query, typically only one subtree qualifies, whereas
for range and window queries there are often several. In Figure 3.3b, a point-recursive
quadtree example is given, applied to the output of the object tracking algorithm of
Fig. 3.3a. Quadtree structures are designed for storage in a main background memory.
That is, they do not consider memory paging. As a result, the access to the data in
the memory is not optimal in terms of speed and therefore not very suitable for large
data sets.

3.2.2 R-trees

The R-tree, introduced by Guttman in 1984 [43], is a hierarchical data structure, meant
for efficient indexing of multi-dimensional objects with spatial extensions. R-trees are
used to store the minimum boundary rectangles (MBRs) of objects, instead of the origi­
nal objects. The MBR of a multi-dimensional object is defined to be the minimum
multi-dimensional rectangle that contains the original object. Similar to the classical
one-dimensional B-trees [49], the R-trees are equally distributed over its vertices
(balanced) and they ensure efficient storage utilization on disk. Because the R-trees
manage MBRs and not the actual objects, they can not answer the query completely,
unless the objects in the database are equal to their MBRs. In general, R-trees are
used to efficiently apply filtering to the result set because they immediately indicate
the intersecting MBRs of the queried objects. Thereafter, a postprocessing step is used
for calculating exact results. Figure 3.4 gives an example of an R-tree. Note that the
MBRs in the edges at the same vertex may intersect one another. While querying, the
paths where the MBRs do not intersect with the query-MBR are pruned. The R-tree
does not guarantee that traversing one path of the tree is enough when searching for
an object, as the MBRs of entries in the same nodes may overlap one another. In the
worst case, the search algorithm may have to visit all vertices in order to return the
results of a query.

Inserting a spatial object into the R-tree means inserting the MBR of the object
in the correct order in the R-tree along with a reference to the original object. For
point objects the MBR equals the actual object, so they are directly inserted into the
tree. Only one path of the tree is traversed and the new entry is inserted as a branch
in one of the lowest edges. If the MBR of the object intersects with many entries of an
intermediate node, the edge is chosen whose MBR is less enlarged after the insertion.
The object is inserted only at one vertex and if it causes the vertex page to overflow,
the vertex is split up in two, after applying several splitting criteria. If an insertion
causes enlargement of the edge pages MBR, it is adjusted properly and the changes
are propagated upwards.

Active research on the search performance concentrates on the minimization of over­
lap between sibling nodes. Guttman [43] suggests various policies to minimize overlap
during insertion. The variant of R-tree that is considered to handle dynamic object
insertion very efficiently is the R*-tree [44]. It introduces a new insertion policy, that
considerably improves the search performance of the tree. The R*-tree comes up with
a solution to effectively minimize the overlap region between sibling nodes in the tree.
In Fig. 3.4b, an R*-tree example is given, applied to the output of the object tracking
algorithm as shown in Fig. 3.4a. The main advantage of the R*-tree is the minimiza-
tion of the tree paths that are traversed for an object search. The advantages of the
new insertion algorithm over the original R-tree can be summarized as follows:

- while traversing the insertion path, the insertion algorithm follows the vertices
  whose MBR has the minimum increase of overlap. Thus, the search performance
  is improved;
- whenever a new entry has to be stored into a full node, the node is not necessarily
  split, but some entries are deleted and re-inserted to sibling nodes. This feature
  of the algorithm increases storage utilization and improves the quality of the
  hierarchical partitioning, making it almost independent of the sequential order
  in which new entries are inserted;
- the algorithm for splitting a node is totally different from its R-tree equivalent.
  Firstly, the entries are sorted by the lower value, then by the upper value of
  the MBRs. For each sort, \((M - 2m + 2)\) distributions of the \((M + 1)\) entries
  are determined into two groups. The optimal split-axis is determined by the
  distribution that has the minimum value by of the sum of all margin-values. For
  the chosen split axis, the algorithm chooses the distribution that results in a
  minimum overlap between the MBRs. The algorithm has proven a better MBR
  partitioning over the tree.

3.3 Algorithmic choices

There have been published several benchmark articles, comparing R-tree and quadtree
implementations. In some comparisons, the quadtree is the winner, in others, the R-
tree is preferred [50, 51]. But the overall conclusions from these comparisons are
the same. The choice for an indexing structure heavily depends on the data set to be
stored and how much data is concerned. While for small low-dimensional random data
sets, quadtrees may be favourable due to its optimal in-memory structure, for non-
random ordered and large higher-dimensional data sets, a disk-page optimized indexing
structure is advantageous. Real-world surveillance data is not randomly ordered and
most likely contains only a few routes in the video scene were 90 per cent of the objects
are located. Therefore, quadtree implementations are doubtful. The quadtree will be
very tall and unbalanced and does not fit in main background memory anymore if
large amounts of data has to be stored. Another problem with quadtrees is the minor
support for high-dimensional data sequences. Trajectories are mapped onto one or
more higher-dimensional points, instead of directly storing the MBRs, as can be done
with R-trees. This makes the quadtree not very flexible for future needs. Apart from
efficient indexing, the structure should be optimized for disk-based access. Hence it
has become evident that R-trees satisfy our needs best. As a result, an R-tree variant
is adopted for our work in favour of the quadtree implementation.
Chapter 4

Search & Retrieval

4.1 Introduction

After the definition of an effective interval-based trajectory description in Chapter 2, which enables efficient spatial data indexing methods, as discussed in Chapter 3, we explain the search for interesting objects with the identified search queries on location data. In Section 4.2, the overall architecture is explained, identifying the modules in the system. Subsequently, for each of the identified search queries, the architectural modules are discussed in more detail.

4.2 Architecture

The proposed architecture for search and retrieval is presented in Fig. 4.1. It is divided in the following five modules:

- Graphical User Interface
- Preprocessing
- Query Execution
- Postprocessing
- Rank-Join

In the Graphical User Interface (GUI) module, one is able to draw a search query in the spatial plane of the video image and search for objects that match with the drawn search query. In the Preprocessing module, the defined search query is preprocessed to meet the specific requirements of the data indexing structure, to enable fast searching. If the search query is composed by multiple sub-query parts, an Query Execution module is executed for each part, returning a candidate result set containing objects from the video archive that match with the defined query. After additional computations for each candidate result set in the Postprocessing module, a result set is defined. It is ranked in a best-match order to the query. Finally, a Rank-Join module joins the result sets of each individual query part into one final result set and re-ranks it in a best-match order to the complete query. Subsequently, this result set is send back to the GUI.

In the following sections, for each of the three identified search queries in Section 1.2 (trajectory similarity search, tripwire search and region-of-interest search), the above-mentioned modules will be explained in more detail.
4.3 Trajectory Similarity Search

4.3.1 Introduction

With Trajectory Similarity Search, we mean the search for object behaviour similar to a user-specific drawn trajectory contour on the video image displaying the actual camera view. In the graphical user interface, one is able to sketch a query trajectory by drawing lines on the video image and selecting the time range for which results wants to be retrieved. In Fig. 4.3a, query trajectory is sketched on the video image, containing three line-segments. In the next subsections, the architectural modules are discussed in more detail, specifically for the Trajectory Similarity Search.

4.3.2 Preprocessing

A Query Preprocessing module is adopted to convert the drawn query trajectory into a form to efficiently search through the R*-tree hierarchical data indexing structure, where all object trajectories from the video archive are stored. Each object trajectory is stored in the R*-tree as two or more trajectory coordinates. The distance between successive trajectory coordinates is less than, or equals MaxDist. See Section 2.4 and Section 3.3, for more information about the trajectory descriptions and the R*-tree data indexing structure.

Distance calculations between trajectory coordinates and the query trajectory are relatively expensive. By exploiting the search capabilities of the R*-tree, as will be explained in the next subsection, only a small subset of all trajectory coordinates has to be examined for proper object data retrieval. Therefore, the R*-tree search algorithm is used as a filtering step, to avoid a computational complexity linear to the number of trajectory coordinates stored in the data indexing structure.

To retrieve all trajectories in the filtering step located near the query trajectory without false negatives (trajectories that are wrongly rejected from the candidate result set), the system considers the area with distance \( r \) from the query trajectory. This area is indicated in Fig. 4.3 as a convex hull. By choosing distance \( r \) at half MaxDist, we retrieve all trajectories near the query trajectory, without having any false negatives. See Fig. 4.2, where an example shows that false negatives can occur if \( r \) is chosen smaller than half MaxDist.
4.3. TRAJECTORY SIMILARITY SEARCH

To use the R*-tree search algorithm effectively, a rectangular search area is given as input. Therefore, a minimum bounding rectangle (query-MBR) is projected on each convex hull of the query trajectory. An example is visualized in Fig. 4.3. A query trajectory is drawn, containing three line-segments. For each line-segment, the convex hull is determined using radius $r$ and finally a query-MBR is projected on top of it. In the next subsection, the Query Execution module will be explained, where for each query-MBR as defined in this subsection, the hierarchical data indexing structure will be searched.

![Figure 4.2: Distance $r$ at $\frac{1}{2} \cdot \text{MaxDist}$ (a) and distance $r$ at $\frac{1}{4} \cdot \text{MaxDist}$ (b).](image)

**Figure 4.2:** Distance $r$ at $\frac{1}{2} \cdot \text{MaxDist}$ (a) and distance $r$ at $\frac{1}{4} \cdot \text{MaxDist}$ (b).

4.3.3 Query Execution

As shown in Fig. 4.3, for each line-segment of the query trajectory, the search range is defined by the query-MBR of the generated convex hull. The query execution is divided into two stages. First, a range-search returns all edges from the R*-tree where the stored bounding rectangles overlap with the query-MBR. A second stage takes the output of the range-search as input and returns all trajectory coordinates that are located inside the query-MBR. In Fig. 4.4a, an example is shown for which the R*-tree data indexing structure contains only two object trajectories. In Fig. 4.4c, the corresponding tree structure is shown.

![Figure 4.3: Trajectory Similarity Search - Preprocessing.](image)

**Figure 4.3:** Trajectory Similarity Search - Preprocessing.

Each individual range-search will start at the root of the hierarchical R*-tree data indexing structure. For all edges beginning at the root of the R*-tree, the range-search checks if the query-MBR overlaps with the stored bounding rectangles at the leaves of the R*-tree. The vertices for which there is no overlap with the query-MBR are ignored from the search process and the range-search continues with the next edge.
on the same level in the R*-tree. If for one or more paths in the R-tree the leaves are reached, the trajectory coordinates are passed to the second stage. For all trajectory coordinates that are located inside the query-MBR, the coordinates are returned in a candidate result set. This behaviour repeats itself until all the leaves in the R*-tree are reached for which the bounding rectangles overlap with the query-MBR. In Fig. 4.4b, the two stages are visualized. The bounding rectangles from the R*-tree that overlap with the query-MBR are the rectangles B and C. The black-coloured trajectory coordinates are the coordinates located inside the query-MBR.

4.3.4 Postprocessing

In the previous module, trajectory coordinates from different object trajectories are retrieved from the R*-tree data indexing structure and stored in a candidate result set. To be able to rank the object trajectories in a best-match order with the query trajectory, cost values need to be defined for each trajectory coordinate in the result set. The cost values will be computed with corresponding cost functions depending on the distance to the query trajectory and the direction difference with the query trajectory, as we will explain in the next subsections. Subsequently, the total similarity will be computed using a global error measure determined by the cost values of all retrieved coordinates of each object trajectory.

Two-dimensional Euclidean point-to-line distance

To derive a cost value of each retrieved trajectory coordinate with respect to the query trajectory, two cost functions are defined. The first cost function represents the actual distance between the retrieved trajectory coordinates and the query trajectory. The used algorithm is known in literature as the two dimensional Euclidean point-to-line distance [52].

If a line-segment of a drawn query trajectory is specified by two points \( a = (x_a, y_a) \) and \( b = (x_b, y_b) \), see Fig. 4.5, then a vector \( \vec{v} \) perpendicular to the line is given by:

\[
\vec{v} = \begin{bmatrix} y_b - y_a \\ - (x_b - x_a) \end{bmatrix}. \tag{4.1}
\]

Let \( \vec{s} \) be a vector from the coordinate point \( p = (x_p, y_p) \) to the first point on the line

\[
\vec{s} = \begin{bmatrix} x_a - x_p \\ y_a - y_p \end{bmatrix}. \tag{4.2}
\]
then the distance from \((x_p, y_p)\) to the line is given by projecting \(\mathbf{s}\) onto \(\mathbf{v}\), giving

\[
d = \frac{|\mathbf{v} \cdot \mathbf{s}|}{|\mathbf{v}|} = \frac{|(x_b - x_a)(y_a - y_p) - (x_a - x_p)(y_b - y_a)|}{\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}}.
\] (4.3)

![Figure 4.5: Two-dimensional Euclidean point-to-line distance.](image)

Figure 4.5: Two-dimensional Euclidean point-to-line distance.

The formula above corresponds to the more general distance computation in the three-dimensional domain:

\[
d = \frac{|\text{det}((\mathbf{b} - \mathbf{a}), (\mathbf{a} - \mathbf{p}))|}{|\mathbf{b} - \mathbf{a}|}
\] (4.4)

with all vectors having zero z-components.

The first cost value \(C_{\text{dist}}\) is defined as the division of two-dimensional Euclidean point-to-line distance through the maximum distance \(r\), as defined in Subsection 4.3.2, and is limited by 1.

\[
C_{\text{dist}} = \begin{cases} 
\frac{d}{r} & 0 \leq d < r \\
1 & d \geq r
\end{cases}
\] (4.5)

![Figure 4.6: Cost value \(C_{\text{dist}}\).](image)

In Fig. 4.7a, an example is shown for which the R*-tree data indexing structure contains only two object trajectories. In Fig. 4.7b, a query trajectory is shown together with the five retrieved trajectory coordinates from the R*-tree data indexing structure. In Fig. 4.7c, \(C_{\text{dist}}\) values are shown for the retrieved trajectory coordinates located inside the query-MBR. Although the R*-tree algorithm requires a query-MBR, the actual desired coordinates are located within the convex hull of the query. Hence,
trajectory coordinates with a $C_{\text{dist}}$ value of 1 are ignored for further processing.

**Directional Difference**

The second cost function represents the directional difference between two adjacent coordinates of a trajectory and the query trajectory. Therefore, for the coordinates in the candidate result set, additional information must be retrieved. First, the directional angle $\alpha$ of the line between the current and the previous trajectory coordinate in the result set is derived. Moreover, also the length of the line is computed. Next, the directional angle $\beta$ of the line between the current and the next trajectory coordinate is derived. Again, also the length of this line is computed. The directional differences $Dir_{\text{prev}}$ and $Dir_{\text{next}}$ are computed as the angular differences by the substitution of respectively $\alpha$ and $\beta$ with the directional angle of the query trajectory sketch $\varphi$, see Fig. 4.8.

![Diagram of directional difference](image)

**Figure 4.8: Directional difference.**

A maximum direction difference threshold $MaxDir$ is proposed, to derive the second cost value $C_{\text{dir}}$, depending on the directional difference. The cost value is computed by dividing the minimum of $Dir_{\text{prev}}$ and $Dir_{\text{next}}$ through the maximum direction difference $MaxDir$, limited by 1.

$$C_{\text{dir}} = \begin{cases} \frac{\min(\text{Dir}_{\text{prev}}, \text{Dir}_{\text{next}})}{\text{MaxDir}} & 0 \leq \min(\text{Dir}_{\text{prev}}, \text{Dir}_{\text{next}}) < \text{MaxDir} \\ 1 & \min(\text{Dir}_{\text{prev}}, \text{Dir}_{\text{next}}) \geq \text{MaxDir} \end{cases}$$ (4.6)

Trajectory coordinates representing straight parts in the object trajectory have more influence on the subjective similarity ranking than trajectory coordinates representing directional changes. Trajectory coordinates that represent directional changes in the
4.3. TRAJECTORY SIMILARITY SEARCH

The cost value $C_{\text{dir}}$ is weighted by a distance factor $w_{\text{dist}}$, limited by 1:

$$w_{\text{dist}} = \frac{\min(Dist_{\text{prev}}, Dist_{\text{next}})}{\text{MaxDist}}, \quad 0 \leq w_{\text{dist}} \leq 1. \quad (4.7)$$

The weighted cost value $(C_{\text{dir}})_{\text{weighted}}$ is defined as

$$(C_{\text{dir}})_{\text{weighted}} \equiv \begin{cases} \frac{C_{\text{dir}}}{w_{\text{dist}}} & 0 \leq \frac{C_{\text{dir}}}{w_{\text{dist}}} < 1 \\ 1 & \frac{C_{\text{dir}}}{w_{\text{dist}}} \geq 1 \end{cases} \quad (4.8)$$

In order to rank the trajectories in a best match order with respect to the drawn query trajectory, two error measures are adopted. A local error represents the matching error of the individual retrieved trajectory coordinates to the query trajectory. A global error represents the total matching error of all retrieved trajectory coordinates together to the query trajectory. In the remainder of this section, both error measures are explained.

**Local Error**

For each retrieved trajectory coordinate, a local error measure $E_{\text{local}}$ is computed. This measure is dependent on the cost values $C_{\text{dist}}$ and $C_{\text{dir}}$. For the local error measure, the two cost values $C_{\text{dist}}$ and $C_{\text{dir}}$ are not weighted equally. This has been done because the directional error is seen as a more important factor in the subjective similarity ranking.

As an example, see Fig. 4.10. A coordinate from object trajectory (1) is visualized with a relatively large value $C_{\text{dist}}$ and a small value $C_{\text{dir}}$. On the other hand, a coordinate from object trajectory (2) has a large cost value $C_{\text{dir}}$ and a small value $C_{\text{dist}}$. As can be seen from the figure, object trajectory (1) is subjective more similar to the query than object trajectory (2). Note that the coordinates outside the dashed hull are ignored.
The local error, graphically visualized in Fig. 4.11, is formally defined by:

$$E_{local} = \begin{cases} 
\frac{C_{dist} + C_{dir}}{2} & 0 \leq (C_{dist} + C_{dir}) < \frac{1}{2} \\
\frac{1}{2} & \frac{1}{2} \leq (C_{dist} + C_{dir}) < 1 \\
\frac{1}{2} + \frac{3}{2} \cdot (C_{dir} - \frac{1}{2}) & 0 \leq C_{dist} < 1, \ 0 \leq C_{dir} < \frac{1}{2} \\
\frac{1}{2} & 0 \leq C_{dir} < 1, \ \frac{1}{2} \leq C_{dist} < 1 
\end{cases} \quad (4.9)$$
4.3. TRAJECTORY SIMILARITY SEARCH

endpoint of the query that is the most far away from the previously selected coordinate. $E_{\text{trajectory-query}}$ is defined by the Euclidean distance between (1), the point on the query that has the smallest distance to one of the retrieved coordinates of a trajectory, and (2), the retrieved coordinate that has the largest distance from the previously selected point on the query trajectory. See Fig. 4.12 for an example. Performance measurements indicated that the ranking of trajectories by using the Hausdorff distance is very sensitive to outliers. To overcome this drawback and effectuate a proper ranking even in the case of noisy trajectory descriptions, we adopt a global error measure defined by the weighted average of individually calculated local error measures. It is defined by:

$$(E_{\text{global}})_n = \frac{1}{N} \sum_{i=0}^{N-1} (E_{\text{local}})_{n,i},$$

where $n$ represents the trajectory and $i$ represents a trajectory coordinate.

Object trajectories for which the retrieved coordinates are in match with the complete contour of the query are found more important than object trajectories for which only a certain part of the retrieved coordinates matches with the contour of the query. Therefore, the global error $E_{\text{global}}$ is scaled with match factors $w_{\text{matched}}$ and $w_{\text{non-matched}}$. Consequently, a higher ranking of object trajectories that match the complete contour of the query is effectuated. Factor $w_{\text{matched}}$ represents the matched part of the query and factor $w_{\text{non-matched}}$ represents the non-matched part of the query.

$$w_{\text{matched}} = \frac{\text{MatchLength}}{\text{TotalLength}}, \quad 0 \leq w_{\text{matched}} \leq 1,$$

$$w_{\text{non-matched}} = (1 - w_{\text{matched}}), \quad 0 \leq w_{\text{non-matched}} \leq 1,$$

$$(E_{\text{global}})_{\text{weighted}} = (E_{\text{global}} \cdot w_{\text{matched}}) + (1 \cdot w_{\text{non-matched}}).$$

See Fig. 4.13 for an example, where for two retrieved object trajectories the MatchLength to a query trajectory is overlaid on the query trajectory.
4.3.5 Rank-Join Results

In the previous section, the local and global error computations have been introduced. However, they consider only a single-line trajectory query. Common user queries consist of multiple line-segments (multiple parts), as shown in Fig. 4.3. For each part, a result set has been generated in the previous modules. However, for each query, only one result set will be presented to the user. Therefore, the individual result sets have to be combined. For better ranking, shorter query line-segments should have less influence on the total similarity than longer query line-segments.

Hence, a scaling factor $w_{proportion}$ is proposed. It is defined by:

$$ (w_{proportion})_i = \frac{Length_i}{TotalLength_query}, \quad 0 \leq (w_{proportion})_i \leq 1, \quad (4.15) $$

where $i$ represents the actual query line-segment.

The global error $E_{global}$ for the multi-line query trajectory is defined as:

$$ (E_{global})_{multi-line,n} = \frac{1}{N} \sum_{i=0}^{N-1} (E_{global})_{n,i} \cdot (w_{proportion})_i, \quad (4.16) $$

where $i$ represents the query line-segment and $n$ a retrieved trajectory and $0 \leq (E_{global})_{multi-line,n} \leq 1$. 

---

Figure 4.13: MatchLength against a query, visualized for two retrieved trajectories.
4.4 Tripwire Search

4.4.1 Introduction

With Tripwire Search, we mean the search for object trajectories crossing user-drawable line(s) (tripwire(s)) in the video image. See Fig. 4.15a for an example tripwire query. To start a new Tripwire Search, a user draws a tripwire on the video image in the GUI and specifies the direction in which the objects should pass the wire, together with the time range in which the tripwire crossing has occurred. Also, search queries containing multiple tripwires can be drawn. For each drawn tripwire, a logical relation (AND/OR) must be selected. In the search process, each tripwire is treated independently. In the last step of the process, the results of multiple tripwires are joined, taking the logical relation between the tripwires into account. In the next subsections, the architectural modules are discussed in more detail specifically for the Tripwire Search.

4.4.2 Preprocessing

Similar to Section 4.3, the Tripwire Search adopts R*-trees and applies minimum bounding rectangles (query-MBRs) to search efficiently. Therefore, a Query Pre-processing module is needed to convert the drawn tripwire query into a form to efficiently search through the R*-tree hierarchical data indexing structure, where all object trajectories from the video archive are stored. To retrieve all trajectories in the filtering step, located near the query trajectory, without false negative (trajectories that are wrongly rejected from the candidate result set), the system considers the area with distances from the tripwire query. This area is indicated in Fig. 4.15 as a convex hull. By choosing distance \( s \) at MaxDist, we retrieve all trajectories near the query trajectory that possibly intersect with the tripwire, without having any false negatives. If distance \( s \) is chosen smaller than MaxDist, the two trajectory coordinates of an intersecting trajectory may lie outside the hull. Consequently, the intersection coordinate could not be found. See Fig. 4.14, where an example shows that false negatives can occur if \( s \) is chosen smaller than MaxDist.

![Figure 4.14: Distance \( s \) at MaxDist (a) and distance \( s \) at \( \frac{1}{2} \times \text{MaxDist} \) (b).](image)

A minimum bounding rectangle (query-MBR) is projected on each convex hull of the tripwire query to search efficiently. An example is visualized in Fig. 4.15. A query trajectory is drawn, containing two tripwires. For each tripwire, the convex hull is determined using distance \( s \) and a query-MBR is projected on top of it. In the next subsection, the Query Execution module will be explained, where for each query-MBR as defined in this subsection, the hierarchical data indexing structure will be searched.
4.4.3 Query Execution

As shown in Fig. 4.15, for each tripwire, the search range is defined by the query-MBR of the generated convex hull. Because the Query Execution module used for tripwire search is almost equal to the Query Execution module used for trajectory similarity search, we refer for the details to Section 4.3.3. In Fig. 4.16a, an example is shown for which the R*-tree contains two object trajectories. In Fig. 4.16b, the range-search stages are visualized. The bounding rectangles from the R*-tree that overlap with the query-MBR are the rectangles A and B and the black-coloured trajectory coordinates are the coordinates located inside the query-MBR. In Fig. 4.16c, the corresponding tree structure is shown.

Trajectory coordinates retrieved by the Query Execution module could potentially lie outside the convex hull of the tripwire query. Note that the query-MBR is by definition larger than the hull. A Postprocessing module is needed to reject the trajectory coordinates that are located outside the indicated convex hull. Subsequently, all coordinate-pairs of each retrieved trajectory are checked for intersection with the tripwire query. Finally, if there are multiple tripwires specified in the query, in a Rank-Join module, each single result set is joined into one total result set. The above-described modules will be explained in the next section.

4.4.4 Postprocessing

The previously discussed Query Execution module serves as a filtering step for a Postprocessing module where the actual crossing between retrieved trajectories and the
4.4. TRIPWIRE SEARCH

query is examined. Prior to examining possible tripwire crossings, the coordinates that are located outside the indicated convex hull must be rejected from the result set. Therefore, we adopt the two-dimensional Euclidean point-to-line distance algorithm from Subsection 4.3.4.

For the coordinates in the reduced candidate result set, a line-line-intersection algorithm [54] checks for possible tripwire crossings of the retrieved trajectories. In Fig. 4.17, the algorithms are visualized for one of the tripwires from the query of Fig. 4.15, where the tripwire is labelled as $AB$ and the two consecutive trajectory coordinates as $CD$. The algorithms has as input the start- and endpoint of the drawn tripwire query, together with two consecutive trajectory coordinates from the candidate result set. If the two line-segments intersect, the algorithm outputs the intersection point $P$. The linear descriptions of a tripwire $AB$ and the two consecutive trajectory coordinates $CD$ have the form of equations

$$y = c_1 \cdot x + c_2,$$

$$y = c_3 \cdot x + c_4.$$  

If tripwire $AB$ and trajectory $CD$ intersect each other, as in Fig. 4.17, than it holds that

$$c_1 \cdot x + c_2 = c_3 \cdot x + c_4,$$  

where for the two-dimensional case this results in

$$\begin{cases}
    r(X_B - X_A) + X_A = s(X_D - X_C) + X_C, \\
    r(Y_B - Y_A) + Y_A = s(Y_D - Y_C) + Y_C,
\end{cases}$$

for $0 \leq r \leq 1$ and $0 \leq s \leq 1$.

Solving the above equation for $r$ and $s$ yields

$$r = \frac{(Y_A - Y_C)(X_D - X_C) - (X_A - X_C)(Y_D - Y_C)}{(X_B - X_A)(Y_D - Y_C) - (Y_B - Y_A)(X_D - X_C)},$$

$$s = \frac{(Y_A - Y_C)(X_B - X_A) - (X_A - X_C)(Y_B - Y_A)}{(X_B - X_A)(Y_D - Y_C) - (Y_B - Y_A)(X_D - X_C)}.$$  

By examining parts of the above two equations, we can add limiting conditions to the usage of the relatively more expensive division operation:

![Figure 4.17: Object trajectory crossing the tripwire query.](image)
- if the denominator in Equation (4.21) is zero, $AB$ and $CD$ are parallel;
- if the numerator in Equation (4.21) is also zero, $AB$ and $CD$ are equal.

If the above conditions do not hold, we have to solve the equations and decide if an intersection coordinate exists with the following conditions:

- if $0 \leq r \leq 1$ and $0 \leq s \leq 1$, intersection exists;
- if $r < 0$ or $r > 1$ or $s < 0$ or $s > 1$, line-segments $AB$ and $CD$ do not intersect.

Finally, if an intersection exists, the intersection coordinate point $P$ can be computed as follows:

$$
\begin{align*}
X_P &= X_A + r(X_B - X_A) \\
Y_P &= Y_A + r(Y_B - Y_A)
\end{align*}
$$

If a crossing direction constraint is specified, the angle between the trajectory coordinates will be derived. If the angle is not in match with the crossing direction range, the trajectory is discarded from the result set. As an example see Fig. 4.18, where a retrieved trajectory crosses the tripwire query with angle $\alpha$. The angle of the tripwire query is $\varphi$ and the preferred crossing direction is indicated in the figure with an arrow. Because $\alpha$ is between a $\pm 90^\circ$ direction range of $\varphi$, the constraint is satisfied.

![Figure 4.18: Crossing direction constraint from top to bottom.](image)

### 4.4.5 Rank-Join Results

In the previous sections, for each drawn tripwire query a range-search has been executed on the hierarchical R*-tree. Next, the retrieved candidate result set is post-processed to eliminate the trajectories that do not cross the drawn tripwire query. However, the previous modules consider only a single tripwire query. Since multiple tripwires can be drawn simultaneously in the Tripwire Search (see Fig. 4.15), additional processing is required. For each drawn tripwire, a result set has been generated in the previous modules. For each query, only one result set will be presented to the user. Therefore, the individual result sets have to be joined. The join process is dependent on the logical relation that has been selected for the drawn query. For the OR-relation, a simply join operation on the individual result sets suffices. If the AND-relation is specified, only the trajectories that cross all drawn tripwires will be added to the result set. Therefore, for each trajectory in a result set, the other result sets have to be examined to see if the trajectory can also be found there. Finally, proper ranking of trajectories in the result set is accomplished by sorting on the date/time properties of the object.
4.5 Region-of-Interest Search

4.5.1 Introduction

With Region-of-Interest Search, we mean the search for object trajectories crossing user drawable region(s)-of-interest in the video image. See Fig. 4.19a for an example Region-of-Interest Search query. To start a new Region-of-Interest Search, a user draws a rectangular region onto the video image in the GUI and selects the time range in which the objects will be found. Also multiple regions can be drawn, where for each drawn region, a logical relation (AND/OR) is selected. In the search process, each drawn region-of-interest is treated independently. As already explained, in the last step of the search process, results from the multiple regions-of-interest are combined taking the indicated logical relation into account.

4.5.2 Preprocessing

Similar to Section 4.3, the Region-of-Interest Search adopts R*-trees and applies minimum bounding rectangles (query-MBRs) to search efficiently. Therefore, a Query Preprocessing module is needed to convert the drawn region-of-interest query into a form to efficiently search through the R*-tree hierarchical data indexing structure. To retrieve all trajectories in the filtering step located near the query trajectory without false negatives (trajectories that are wrongly rejected from the candidate result set), the system considers an area of minimum MaxDist length and height. Therefore, we retrieve all trajectories that possibly intersect the indicated region, without having any false negatives. If the vertical or horizontal side of the drawn region is smaller than MaxDist, the side is enlarged. This is done because otherwise the trajectory coordinates of an intersecting trajectory may lie outside the drawn region. Consequently, the intersecting trajectory would not be found. An example is visualized in Fig. 4.19a, where two regions of interest are drawn. For each region, the size is adjusted to meet the requirements. The query-MBRs are defined by the adjusted rectangles as shown in Fig. 4.19c. In the next subsection, the Query Execution module will be explained, where for each query-MBR as defined in this subsection, the hierarchical data indexing structure will be searched.

4.5.3 Query Execution

As shown in Fig. 4.19, for each region-of-interest, the search range is defined by the query-MBR of the drawn region rectangle. Because the Query Execution module
used for region-of-interest searching is almost equal to the Query Execution module used for Trajectory Similarity Search, we refer for the details to Section 4.3.3. In Fig. 4.20a, an example is shown for which the R*-tree contains two object trajectories. In Fig. 4.20b, the range-search stages are visualized. The bounding rectangles from the R*-tree that overlap with the query-MBR are the rectangles A, C and D. The black-coloured trajectory coordinates are the coordinates located inside a query-MBR. In Fig. 4.20c, the corresponding tree structure is shown.

Figure 4.20: Region-of-Interest Search - Query Execution.

4.5.4 Postprocessing

In the previous module, trajectories are retrieved in a candidate result set for which one or more coordinates cross the drawn region-of-interest. Because ranking of the retrieved trajectories in the candidate result set is only useful according to the data / time properties of the object, further postprocessing is not needed.

4.5.5 Rank-Join Results

In the previous sections, for each drawn region-of-interest a range-search has been executed on the hierarchical R*-tree. Subsequently, the retrieved trajectories are stored in candidate result sets. However, the previous modules consider each region query independently. Since multiple regions can be drawn simultaneously (see Fig. 4.19), additional processing is needed. Similar as for the Tripwire Search, for each drawn region, a result set has been generated in the previous modules. For each query, only one result set will be presented to the user. Therefore, the individual result sets have to be combined. The join process is dependant on the logical relation indicated with the query. For the OR-relation, a simply join operation on the individual result sets suffices. If the AND-relation is specified, only trajectories that cross all drawn regions will be added to the result set. Therefore, for each trajectory of a result set, the other result sets have to be examined. The ranking of objects in the result set is done by sorting on the date/time properties.
Chapter 5

Performance Optimization

In the previous chapter, the search and retrieval architecture has been discussed. Several architectural modules contain adjustable parameters. Maximum performance of the system will be obtained with an optimal parameter setting. The adjustable parameters of the modules will be discussed in the next sections and in addition several performance optimizations are proposed. Starting with the optimizations for the trajectory data representation, we continue with optimizations for searching in the R*-tree data indexing structure. Subsequently, the proposed improvements for each module of the search and retrieval architecture will be discussed. In Chapter 6, the results obtained by the proposed optimizations are quantified for different scenarios.

5.1 Data Representation Optimizations

The trajectory data representation, as defined in Chapter 2, has several adjustable parameters that influence the retrieval performance. These parameters include:
- the maximum distance between succeeding trajectory coordinates $MaxDist$;
- the maximum directional angle between succeeding trajectory coordinates $MaxDirDist$;
- the trajectory start direction angle computation;
- the usage of a relative or absolute coordinate system.

For clear understanding, we now consider an example which creates the proposed sampled interval-based trajectory descriptions from the frame-by-frame-based object trajectory. In the example, VCA real-time object-tracking for one object on a traffic crossing is visualized, together with the processed frame-by-frame-based trajectory description. When the method decides that the location of the object at the current frame is relevant, it is added to the new representation. The relevance is determined by two criteria. Firstly, the distance between the currently processed location coordinate and the start coordinate has to be smaller than $MaxDist$. Secondly, the distance between an averaged trajectory line of the first $n$ coordinates and the currently processed coordinate may not exceed $MaxDirDist$. When the method decides that any of these two criteria are violated, a new trajectory coordinate is generated and added to the new representation. In Fig. 5.1a through Fig. 5.1g, the intermediate processing of the proposed method is shown for the object tracking data of Fig. 5.1a. In Fig. 5.1h, the resulting sampled interval-based description is shown.
In the example of Fig. 5.1, the MaxDist values are chosen quite large, for clear understanding of the algorithm in only a small number of steps. A large value for MaxDist and MaxDirDist will result in a very rough trajectory approximation and a large minimum query length to guarantee no false negatives (trajectories that are rejected by the search process, but actually belong to the results). On the other hand, small values for MaxDist and MaxDirDist approximate the object tracking information very well, but enlarge the storage requirements substantially and require more coordinates to be matched during the search. In Chapter 6, we benchmark several values for MaxDir and MaxDist, to be able to select the optimal value.

The trajectory start direction is determined by averaging the first $n$ directional angles of the frame-by-frame-based trajectory parameter description, as explained in Section 2.4. The number of coordinates used in the averaging process is variable and the optimal value is determined in Chapter 6.

To be able to use the system in a broader multi-camera setup, a coordinate system relative to global positioning coordinates (e.g. GPS) is a requisite, to be able to connect the trajectories from one camera image to another. A single matrix multiplication in the Data Representation module will do, but because benchmarking against a multi-camera system is beyond the scope of this thesis, we consider this as future work.

### 5.2 Data Indexing Structure Optimizations

After the trajectory has been sampled into interval-based features, it will be stored in an R*-tree hierarchical data indexing structure (See Chapter 3). This indexing can be done on several data dimensions and can be divided into separate data structures, for faster retrieval. The optimizations include:

- **higher-dimensional storage**: store $n$ coordinates together as one $(n \times n)$-dimensional point in the R*-tree;
- **date and time storage**: store the date and time as additional dimensions in the R*-tree;
- **separate data structures**: create small separate R*-trees for the last day or week of recently processed video.

In the next subsections, we explain the above-mentioned optimizations in more detail.

#### 5.2.1 Higher-dimensional storage

Currently, the trajectory storage system uses an R*-tree spatial data indexing method which is efficient for searching multi-dimensional data with rectangular range-search queries. The hierarchical indexing structure is built using individual coordinates of object trajectories as spatial $(x, y)$ points. A problem that remains with the trajectory similarity search is the retrieval of many false positives. For example, trajectories are retrieved as candidate results that are moving in perpendicular direction to the query trajectory. To lower the false positive ratio directly in the range-search, a solution is to store $n$ coordinates together as one $(n \times n)$-dimensional point $(x_1, y_1, x_2, y_2, \ldots, x_n, y_n)$ in the R*-tree.
5.2. DATA INDEXING STRUCTURE OPTIMIZATIONS

Frame-based trajectory description

Figure 5.1: Example of the proposed interval-based trajectory sampling algorithm.
As an example, in Fig. 5.2, a four-dimensional R*-tree is shown. Two trajectory coordinates \((x,y)\) are stored as one four-dimensional point in the hierarchy. Some storage overhead is created, because to store the complete contour of the trajectory, the start- and endpoint are stored twice in the R*-tree data structure. For clear understanding, we show the creation of four-dimensional points in two separate figures (Fig. 5.2a and b), for two example trajectories (A) and (B), to better visualize this overhead. While searching, the query-MBR will be projected on each \((x,y)\) range in the R*-tree. Therefore, candidate results retrieved from the R*-tree are trajectories that have a minimum of two succeeding coordinates in match with the query-MBR. As an example see Fig. 5.2c, where only the coordinates from trajectory (A) are retrieved as candidate matches.

![Figure 5.2: Higher-dimensional storage model (a) and (b), trajectory similarity search (c).](image)

Note that the convex hull has on the height-side an overhead of two times \(MaxDist\) to be certain to retrieve all similar trajectory coordinates. In Chapter 6, we benchmark the \((n \times n)\)-dimensional storage system against the current implementation for different scenarios.

### 5.2.2 Date and time storage

The date and time are currently not indexed in the R*-tree, but separately stored. Therefore, it is not possible to exploit date/time constraints directly in the range-search module to reject object trajectories internally that have a wrong timestamp. To be able to do so, first, a conversion must be made from the current date and time format into an integer format to be compatible with the R*-tree data structure. Subsequently, the integer representation of the date and time of the object's first appearance in the video image are added as two additional dimensions \((x,y, date, time)\) in the R*-tree.

With the storage of the object's start time, querying is only applicable on objects that have entered the scene within the selected time range. If for example an object enters the scene at a certain timestamp (1), a query between timestamp (2) and (10) will not be able to retrieve the object. Therefore, as a requirement, time ranges should be taken large enough to retrieve all candidate results. If no date or time constraints are selected with the search query, searching will be started over the whole date and time ranges. In Chapter 6, we benchmark the optimization compared to the current implementation for different scenarios.
5.2.3 Separate data structure

Besides the storage of the date and time inside the R*-tree indexing structure, another optimization is the creation of a separate R*-tree data structure for the last day or week of recently processed video. Normally, one large data structure is created which is too large to store in main-memory. To speed up intermediate result set processing for the most recently recorded video data, an additional routine can be enabled in the MDA module. This module stores the recently processed video into a separate R*-tree structure, which is small enough to be kept in main-memory. In Chapter 6, we elaborate on the performance increase against the current implementation.

5.3 Search & Retrieval Optimizations

The search and retrieval architecture, as visualized in Fig 4.1, is divided into several modules. For the Preprocessing module, the adjustable parameters together with the proposed optimizations to speed up the Query Execution module will be discussed in the next subsection. Subsequently the optimization proposed for the Rank-Join module is discussed. For the Trajectory Similarity Search and the Tripwire Search, specific optimizations are considered in two separate subsections. The Query Execution module will not be discussed since its performance depends solely upon the choices made for the R*-tree data indexing structure, which are clarified in Section 5.2.

5.3.1 Preprocessing

Minimizing the dead-zone

While processing search queries from drawn trajectories or tripwires on the R*-tree, a minimum bounding rectangle (query-MBR) for the drawn line-segment is determined. In Fig. 5.3, this is visualized for the Trajectory Similarity Search. In Fig. 5.3b, convex hulls are placed automatically over the query trajectory line-segments of Fig. 5.3a to indicate the maximum matching range and to prevent false negatives, as explained in Subsection 4.3.2. In Fig. 5.3c, minimum bounding rectangles (query-MBRs) are projected on the convex hulls as the R*-tree search algorithm expects a rectangular search area. If a drawn trajectory or tripwire is relatively long and has a directional angle near ±45°, the search space outside the convex hull (dead-zone) is significantly large, see Fig. 5.4.

![Figure 5.3: Trajectory Similarity Search - Preprocessing.](image)

A large dead-zone in the query-MBR has several drawbacks on the search performance:
- longer retrieval time, because more paths in the R*-tree have to be examined and more results have to be processed;
- more false negatives in the candidate results, i.e. trajectory coordinates that are returned, but do not have any similarity with the query.

Figure 5.4: Large dead-zone.

To resolve the drawbacks, we propose a novel query-MBR split-up policy, that mini-
mizes the dead-zone. The number of query-MBR split-ups is indicated by a split-up factor \( k \), which is determined by the length cost value \( C_{\text{length}} \) and the angle cost value \( C_{\text{angle}} \) of the drawn query line-segment. The split-up factor \( k \) is adaptive to a maximum value \( \text{MaxSplitUps} \). After several measurements on a common set of query trajectories, the preferred number of split-ups is determined for each query. From these results, we empirically obtained the following formula for \( k \):

\[
k = \text{MaxSplitUps} - \left( \frac{2 \cdot C_{\text{angle}} + C_{\text{length}}}{3} \right) \cdot \text{MaxSplitUps},
\]

where \( C_{\text{angle}} \) and \( C_{\text{length}} \) are determined by:

\[
C_{\text{angle}} = \begin{cases} 
0 & 40^\circ \leq \text{angle} < 50^\circ \\
\frac{1}{4} & 30^\circ \leq \text{QueryAngle} < 40^\circ \text{ or } 50^\circ \leq \text{QueryAngle} < 60^\circ \\
\frac{1}{3} & 20^\circ \leq \text{QueryAngle} < 30^\circ \text{ or } 60^\circ \leq \text{QueryAngle} < 70^\circ \\
\frac{1}{2} & 10^\circ \leq \text{QueryAngle} < 20^\circ \text{ or } 70^\circ \leq \text{QueryAngle} < 80^\circ \\
1 & \text{elsewhere}
\end{cases}
\]

\[
C_{\text{length}} = \begin{cases} 
0 & \text{QueryLength} > 5 \cdot \text{MaxDist} \\
\frac{1}{2} & 4 \cdot \text{MaxDist} \leq \text{QueryLength} < 5 \cdot \text{MaxDist} \\
\frac{3}{4} & 3 \cdot \text{MaxDist} \leq \text{QueryLength} < 4 \cdot \text{MaxDist} \\
\frac{7}{8} & 2 \cdot \text{MaxDist} \leq \text{QueryLength} < 3 \cdot \text{MaxDist} \\
1 & 0 \leq \text{QueryLength} < 2 \cdot \text{MaxDist}
\end{cases}
\]

If the split-up factor \( k \) is determined, the original query-MBR is divided into \( k \) smaller parts. Each part has to be enlarged to again completely contain the convex hull of the query. This enlarging step can be resolved in two ways, horizontally or vertically (See Fig. 5.5). Subsequently, the axis is selected that requires the minimum enlargement of the newly created query-MBRs. In Fig. 5.6, the split up is visualized for different values of \( k \). Note that the dead-zone is significantly reduced for higher values of \( k \).
5.3. SEARCH & RETRIEVAL OPTIMIZATIONS

Sub-query execution ordering

Object trajectories captured from a fixed video camera, monitoring traffic behaviour, are not randomly ordered over the video image plane. Therefore, it seems logical that a search for object trajectories in a high-activity region takes more time than a search in a low-activity region. Because a requirement of the system is to show intermediate results during the search, the overall result after processing each sub-query should be presented. Therefore, it seems useful to determine an execution order of sub-queries if there are multiple queries drawn on the video image, depending on the motion activity near each sub-query. Query-MBRs that cover the least amount of trajectory coordinates are given higher priority, to be able to retrieve intermediate results as fast as possible. As an example, see Fig. 5.7b, where a query containing two line-segments is overlaid on the motion activity map of a traffic crossing. Sub-query (2) will be given higher priority than sub-query (1), because there is less motion activity near sub-query (2). When searching sub-query (1) at first, intermediate results will be retrieved as fast as possible. Moreover, if querying for suspicious behaviour, trajectories of interest will be expected more often in the least activity areas in the scene, thereby indicating that the most attractive results will be presented to the user with the first intermediate result set.

To create such an activity-dependent ordering, a matrix is created in the Data Representation module of the system, where the number of stored trajectory coordinates for each matrix record are maintained. In the Preprocessing module, described in Subsection 4.3.2, a lookup table is added to determine the amount of object trajectories in each query-MBR. Subsequently, a proper search ordering is defined. The lookup table
is graphically visualized as a motion activity map in Fig. 5.7a for the video scenario already presented in Fig. 1.1. The legend symbolizes the average number of processed trajectory coordinates. For an expert user, it is also possible to adjust the search ordering to their own preference. In Chapter 6, we show the performance increase against the current implementation on a common query set for two video scenarios.

5.3.2 Rank-Join

The performance optimization proposed for the Rank-Join module is to enable intermediate result set processing. As proposed in Subsection 4.3.5, for all result sets received from a Postprocessing module, the Rank-Join module generates one final result set. In the optimized version, the Rank-Join module generates result sets each time a new result set is received. For faster user-interaction, intermediate result sets are returned to the graphical user interface (GUI). In Chapter 6, we show the decrease in search time, after enabling intermediate result set processing.

5.3.3 Trajectory Similarity Search

In areas with much motion activity, a high number of trajectory coordinates are stored in the R*-tree structure (See Fig. 5.7). When searching for trajectories similar to a user's sketch in high-activity areas, many candidate results will be retrieved, which takes a relatively high load to the range-search algorithm of the R*-tree. To resolve this, one can use the output of path learning algorithms, as proposed in [55, 56], to reduce the load on the data indexing structure.

State-of-the-art path-learning algorithms are able to define a graph of commonly visited routes in the scene according to full or partly self-learned branches. When a query trajectory is defined, a path search in a graph of commonly visited routes will check for matches with the query trajectory. If the query trajectory matches with a common visited route, only one table lookup is sufficient to retrieve all similar object trajectories.

Let us explain this with an example, visualized in Fig. 5.8. Label "ACFE" indicates a trajectory that traversed from the bottom (A) to the left (C) and finally to the top via (F) and (E). See [56] for a similar proposal. The labelling is performed in the VCA module of the system during the tracking of the objects. Subsequently, the label is stored in a database. Similarly, when a search query is applied by drawing a trajectory on the video image, a label is constructed in a similar way. For the query trajectory,
5.3. SEARCH & RETRIEVAL OPTIMIZATIONS

the created label can be easily searched for with common graph-search algorithms. If the trajectory matches with a commonly visited path in the graph as shown in Fig. 5.8b, only one table lookup is sufficient to retrieve all similar trajectories, without any postprocessing. If the query trajectory does not match with a path in the graph, traditional R*-tree search processing will be executed, as explained in Section 4.3.

If the adopted path-learning algorithm is performing very well, only for the uncommon sketched query trajectories in the video image, matches have to be retrieved via the R*-tree data indexing structure. Typically, the suspicious trajectories in the R*-tree are located in the least busy areas of the tree, so all power of the R*-tree can be exploited to search for similar trajectories. The implementation of a path-learning algorithm in our system is beyond the scope of this thesis, so the optimization is considered as future work.

5.3.4 Tripwire Search

While searching for object trajectories that cross a drawn tripwire, the exact crossing coordinate is not relevant. Therefore, an alternative line-line-intersection algorithm could potentially decrease the retrieval time of trajectories with a tripwire search.
In Fig. 5.9, the optimization is visualized in an example. For two successive trajectory coordinates, a relatively low computational expensive algorithm indicates if the retrieved object trajectory crosses the tripwire query. The algorithm checks in which region the retrieved trajectory coordinates are located by using the pre-computed \((x, y)\) equation of the tripwire. If the \(x\)-value of the current trajectory coordinate is located within the \(x\)-axis range of the drawn tripwire, enlarged to the borders of the convex hull, it is filled in the tripwire equation. This results in the \(y\)-value located on the video tripwire. If the \(y\)-value is smaller than the value of the \(y\)-coordinate from the trajectory coordinate, the current trajectory coordinate is located in region (1). Otherwise, it is located in region (2). Subsequently, the check is repeated for the succeeding coordinate.

Compared to the originally proposed line-line-intersection algorithm, the optimized algorithm has only four subtraction and two multiplications. See Table 5.1 for more details concerning the computational complexity of the two algorithms.

<table>
<thead>
<tr>
<th></th>
<th>Original algorithm</th>
<th>Optimized algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additions</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Subtractions</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>Multiplications</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Divisions</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 5.1: Computational complexity for the two proposed algorithms.*
Chapter 6

Benchmarking & Results

6.1 System Implementation

To show the feasibility and enable benchmarking, the performance of the proposed search and retrieval architecture as explained in Chapter 2 through 4 is measured. A software system has been built, implementing the modules of Section 1.2 (See Fig. 6.1). Several experiments have been conducted to determine the optimal settings for the adjustable parameters in the system. In this section, we enumerate the important details of the built software system. Subsequently, scenarios for benchmarking and the generated test results are discussed. Finally, we discuss the optimal parameter settings and the usability of the proposed optimizations from Chapter 5.

![Diagram](image)

*Figure 6.1: Identified modules (Chapter 1) - search and retrieval system.*
Middleware
A standard middleware library has been adopted, based on the Simple Object Access Protocol (SOAP) [57]. The middleware provides message passing between the different modules in the system over a standard IP network. Each module in the system is autonomous, i.e. it reads the input data, processes this data, and outputs the results. For further reading, we refer to [58], where a more detailed description of the used middleware can be found.

Metadata format
For the communication of all metadata generated in the total system, the multimedia description standard MPEG-7 has been used [59]. Because the description of metadata is quite different for the various modules in the system, three basic MPEG-7 structures have been specified. The first structure describes the video properties sent into the system by the Video Acquisition (ACQ) module. The second structure specifies objects in the scene on a video frame-by-frame basis and is used for transferring trajectory parameter descriptions. The third structure describes objects for a certain time-interval. This last structure allows communication of detected events and image data and is used for transferring the interval-based trajectory descriptions.

Video Acquisition (ACQ)
The video data is transferred through the system in the MPEG-4 format [60] at a bitrate of 1 Mbit per second, using the Advanced Simple profile at CIF resolution (352 x 288 pixels) at a rate of 12.5 video frames per second. The compressed video data is decoded in the middleware layer of each module, so the VCA algorithms receive uncompressed video. Since the communication of the video and metadata is communicated over an IP network, the communication bandwidth should be limited to prevent network congestion.

Video Content Analysis (VCA)
A state-of-the-art object segmentation and tracking algorithm is encapsulated in the VCA module. After processing several image frames, the algorithm outputs the generated object tracking information. The generated tracking information contains a unique object ID for each detected object, together with location information of the object on a frame-by-frame basis in absolute bounding box coordinates $(x_{min}, y_{min}; x_{max}, y_{max})$. The bounding box is the minimum bounding rectangle around the detected object, specified for each video frame.

Metadata Analysis (MDA)
In this module, the frame-by-frame-based MPEG-7 descriptions are sampled to create interval-based features, which enables efficient storage and fast retrieval for location information. For each sampled object trajectory, an MPEG-7 structure is transferred to the Efficient Storage module.

Efficient Storage
To exploit the metadata generated by the modules in the system, the metadata is extracted from the three defined MPEG-7 structures and each of them is stored in a database structure. Video frames are stored in a relational MySQL [61] database by grouping the frames as binary large objects (BLOBs). Subsequently, a PostgreSQL [62]
database is used for general metadata storage (e.g. object types, events, descriptions), where the Generalized Search Tree (GIST) library [63] is used for hierarchical storage of object trajectory descriptions in an R*-tree.

**Graphical User Interface**

The Graphical User Interface (GUI) is built with Trolltech's QT [64] cross-platform C++ library for usability on Windows, Linux and Mac computers. For access to the databases containing video- and metadata, the standard Microsoft ODBC communication layer [65] has been adopted.

The GUI is divided into a query- and a retrieval part. A user is able to define graphical search queries on spatial locations by sketching on the video image displaying the actual camera view. Queries can be applied for similar object trajectories (search for similar motion paths of objects in the scene), queries on tripwires (search for objects in the scene that cross drawn lines) and queries on areas of interest (search for objects in the scene that intersect drawn rectangles). In combination with general search queries such as the recorded date, time and the determined object type (person, bike, car etc.), the search can be made even more specific.

The system presents the results, ranked in order of similarity to the search query. Results are visualized as video thumbnails. When clicking on a thumbnail, the video clip with the selected object is played back and the resulting object trajectory is overlaid on the video image. See Fig. 6.2 for a screenshot of the GUI, showing its functionality.

![Figure 6.2: Screenshot of the Search & Retrieval GUI.](image-url)
6.2 Data sets

For generating test results and benchmark purposes, we created two video-data sets recorded at different camera positions. The first video-data set is recorded with a surveillance camera observing traffic behaviour at a crossing (See Fig. 6.3a). The second video-data set is recorded with a surveillance camera observing traffic behaviour at a parking lot (See Fig. 6.3b). Thirty-one days of video (744 hours) for each set has been processed by the VCA module in the system, resulting in 561,337 objects for the set directed at a traffic crossing and 114,328 objects for the set directed at a parking lot. For each data set, queries are defined for benchmarking. For the data set with the traffic crossing, trajectory similarity queries are found very suitable for benchmarking. For the data set with a parking lot, tripwires and region-of-interest queries are very attractive. The set of trajectory similarity queries, used for benchmarking the system, is visualized in Fig. 6.4a through c, for typical long-line queries, short-line queries and multi-line queries. The tripwire search and region-of-interest search queries are visualized in Fig. 6.4d and Fig. 6.4e.

![Figure 6.3: Video scenes used for benchmarking. A traffic crossing (a) and a parking lot (b).](image)
Figure 6.4: Trajectory similarity queries (a,b,c), tripwires (d) and region-of-interest queries (e) used for benchmarking.
6.3 Results

Several experiments have been conducted to determine the optimal settings for the adjustable parameters in the system. In the next subsections, we discuss the obtained results for each module in the proposed system architecture. The performance measurements were run on an IBM desktop computer with a Microsoft Windows 2000 operating system, equipped with a Pentium 4 processor at 2.66 GHz and 512 MB DDR-RAM memory at 266 MHz. The tests were performed using the system implementation as discussed in Section 6.1.

6.3.1 Metadata Analysis

The frame-by-frame-based trajectory descriptions from the VCA module are processed and sampled to create effective interval-based trajectory features. The following adjustable parameters are benchmarked on the common query set as defined in Section 6.2:

- the number of frame-by-frame-based trajectory descriptions used for determining the start direction angle;
- the maximum distance between succeeding trajectory feature coordinates \( \text{MaxDist} \);
- the maximum directional angle between succeeding trajectory feature coordinates \( \text{MaxDirDist} \);

To determine the best possible start-direction angle, several subjective measurements are adopted on the frame-by-frame-based trajectory descriptions. In Fig. 6.5a, one of these measurements is visualized, showing a frame-by-frame-based trajectory sequence, together with the directional angle from the start point. In Fig. 6.5b, the averaged direction angle is computed for different values of \( n \), where \( n \) means the number of used coordinates in the averaging step. Empirically evaluation of different measurements clarified that usage of three coordinates in the averaging step, the most accurate results are obtained.

![Figure 6.5: Start-direction angle computation. Individual direction angle (a) and direction angle for various values of \( n \) (b).](image)

To determine the best possible values for \( \text{MaxDist} \) and \( \text{MaxDirDist} \), measurements have been adopted using a selection of ten trajectory parameter descriptions, obtained from the VCA module (See Fig. 6.6). For different values of \( \text{MaxDist} \) and \( \text{MaxDirDist} \) (in pixels), sampled interval-based trajectory descriptions have been generated. If the interval-based sampling method introduces a large visual error, it is rejected as a candidate method.
The sampled trajectory features that approximate the frame-by-frame-based descriptions with the lowest visual error are the trajectories sampled with:

- \((\text{Max Dist}, \text{Max Dir Dist}) = (40, 40)\);
- \((\text{Max Dist}, \text{Max Dir Dist}) = (50, 30)\).

These two combinations have been compared to each other. Visual inspections showed that setting \((\text{Max Dist}, \text{Max Dir Dist}) = (40, 40)\) best approximates the frame-by-frame-based trajectory. In Fig. 6.7, the resulting interval-based trajectory descriptions are shown for the ten benchmarked frame-by-frame-based trajectory descriptions.
6.3.2 Efficient Storage

After the trajectory has been sampled and interval-based features have been generated in the MDA module, the resulting data will be stored in an R*-tree hierarchical data indexing structure, as explained in Section 3.3. The hierarchical indexing can be done on several data dimensions and can be divided into separate data structures. In this section, the following sets of storage methods will be benchmarked on the common query set as defined in Section 6.2:

1. This set measures the search performance of traditional storage methods compared to the proposed R*-tree based storage for the following cases:
   a) non-indexed storage of frame-by-frame-based trajectory descriptions (sequential searching for matches);
   b) non-indexed storage of sampled interval-based trajectory descriptions (sequential searching for matches);
   c) R*-tree indexed storage of frame-by-frame-based trajectory descriptions (hierarchical searching for matches);
   d) R*-tree indexed storage of sampled interval-based trajectory descriptions (hierarchical searching for matches).

2. This set measures the search performance of the proposed optimizations on R*-tree based storage:
   a) storage of each coordinate in a separate data entry \( (x, y) \) (the same as in set 1);
   b) storage of each coordinate in a separate data entry, with added date and time info \( (x, y, date, time) \);
   c) storage of two coordinates in one data entry \( (x_1, y_1; x_2, y_2) \);

The measurements have been performed on the queries as visualized in Fig. 6.4a through c. For each query, the time to retrieve a candidate result set and the number of retrieved results is measured. For a clear understanding, the results are visualized in tables showing the relative time percentages and absolute time values for each storage method. For the best performing method, absolute time values for each individual query are visualized, together with the number of retrieved candidate results. To eliminate the influence of a physical disk on the performance measurements, all benchmark tests were performed using a RAMdisk [66]. A RAMdisk is a kernel level driver that presents a standard disk drive to the operating system, however, it stores and retrieves data from the system memory instead of an actual, physical disk.

Storage method Set 1 - Query 1–20 - Crossing scene

In Fig. 6.8a through c, the benchmark results for query trajectories 1–20 are shown in relative and absolute time values for the three storage methods. The conversion from frame-by-frame-based into interval-based trajectory descriptions reduces the search time with almost 87 per cent, because the data set is much smaller. After R*-tree indexing of the interval-based trajectory descriptions, the search time is again decreased with a factor 8 to 12940, compared to traditional searching in the interval-based descriptions. The differences in search time can be explained by the length differences between the queries and the high differences in the number of retrieved results. The
6.3. RESULTS

Search time for R*-tree indexing of the frame-by-frame-based trajectory descriptions is not measured here, due to the large storage requirements for the data set. However, measurements on a smaller data set show that the search time decreases with only 35 per cent compared to traditional searching in the frame-by-frame-based descriptions.

In Fig. 6.8d through f, individual time values are shown for query trajectory 1–20 for interval-based R*-tree indexed search. The long-line query trajectories 1-10 are slower than the short-line queries 11–15, because for short queries, the R*-tree search is more efficient. Also, within each set of query trajectories, differences exist. For example, query trajectory 1 and 10 are up to four times slower than query trajectory 5 and 8. These differences in search time are caused by the very different number of retrieved candidate results. In Subsection 6.3.3, the proposed optimizations from Subsection 5.3.1 (smaller dead-zone by split-up of the query and the usage of an adaptive query ordering) are examined which try to lower the differences in search time.

**Storage method Set 2 - Query 1–20 - Crossing scene**

For the measurements, an R*-tree is created with added date / time information. Note that the traditional R*-tree contains as much trajectory data as the R*-tree with added date / time information. Search time for different date/time constraints has been measured, to benchmark the performance compared to traditional R*-tree storage. In Fig. 6.9a through c, the results are shown. Querying for large date/time ranges is slower than traditional searching without date/time ranges. However, for ranges up to ten days, query trajectory 1–10 and 16–20 are faster than traditional R*-tree storage. For the five short-line queries 11–15, traditional R*-tree searching is always faster. However, to be able to return only the candidate trajectories in a selected time range, an additional postprocessing stage needs to be added to filter out the results that do not match with the date/time constraints. For the R*-tree with added date and time information, such a postprocessing stage is not needed, because the filtering is done internally. (See Fig. 6.9d through f).

In Fig. 6.9g through i, benchmark results are shown for the higher-dimensional R*-tree which stores two trajectory coordinates in the R*-tree as one four-dimensional point. The average search time is decreased with more than 60 per cent, compared to traditional R*-tree indexing. Because of the introduced storage overhead as explained in Subsection 5.2.1, the number of retrieved candidate results for the long-line and multi-line query trajectories are higher than expected, compared to traditional R*-tree indexing (See Fig. 6.9j and l). For the small-line query trajectories (Fig. 6.9k), a 50 per cent decrease of candidate results is achieved, compared to traditional R*-tree indexing. Here, the storage overhead is not noticeable, because no more than two coordinates of a stored trajectory will match with the query. We measured only the performance increase of storing two trajectory coordinates in one data entry. Although storage of more coordinates is possible, while searching, the query-MBRs have to be enlarged to guarantee no false negatives (See subsection 5.2.1). If the query-MBRs are enlarged too much, the search performance will be drastically decreased. Therefore, the storage of more than two coordinates in one data entry is not justified.
Figure 6.6: Benchmark results for storage set 1 for query trajectories 1-20.

(a) Average search time for 3 storage types over all queries
(b) Average search time for 3 storage types over all queries
(c) Average search time for 3 storage types over all queries
(d) Individual search time (Interval-based Indexed Search)
(e) Individual search time (Interval-based Indexed Search)
(f) Individual search time (Interval-based Indexed Search)
(g) Average number of results over all queries
(h) Average number of results over all queries
(i) Average number of results over all queries
Figure 6.4: Benchmark results for storage set 2 for query trajectory 1-10.

(a) Average search time (seconds) for 8 date/time intervals

(b) Average search time (seconds) for 8 date/time intervals

(c) Average search time (seconds) for 8 date/time intervals

(d) Average number of results for 8 date/time intervals

(e) Average number of results for 8 date/time intervals

(f) Average number of results for 8 date/time intervals

(g) Average search time for two data types

(h) Average time values for two data types

(i) Average time values for two data types

(j) Average number of results for two data types

(k) Average number of results for two data types

(l) Average number of results for two data types
Storage method Set 1 - Query 21–28 - Parking scene
In Fig. 6.10a and b, the benchmark results for tripwire queries 21–25 and region-of-interest queries 25–28 are shown in relative and absolute time values for the three storage methods. The conversion from frame-by-frame-based into interval-based trajectory descriptions reduces the search time with more than 92 per cent, because the data set is much smaller. Compared to the crossing data set as shown in Fig. 6.8, differences in relative search time exists. These differences can be explained by the fact that for the parking scene, objects have generally a lower speed in the video, compared to the crossing scene, and thus many frame-by-frame-based trajectory coordinates are located on top of each other. Subsequently, by sampling the frame-by-frame-based descriptions into interval-based descriptions independent of the frame-rate, the trajectory is compressed in only a few relevant feature coordinates (See Section 2.4). The differences between the number of coordinates of a frame-by-frame-based description and an interval-based description are therefore higher for the parking scene as for the crossing scene. This explains the relatively lower sequential search time on interval-based descriptions for the parking scene, compared to the crossing scene (7.7% vs. 12.9%).

After R*-tree indexing of the interval-based trajectory descriptions, the average search time is again decreased with a factor 25 for the tripwire queries and a factor 90 for the region-of-interest queries, compared to traditional searching in the interval-based descriptions. The differences in search time can be explained by the high differences in the number of retrieved results. The search time for R*-tree indexing of the frame-by-frame-based trajectory descriptions is not measured here, due to the large storage requirements for the data set.

Storage requirements
Storage requirements for different storage methods are visualized in Fig. 6.11 and Fig. 6.12, for the crossing scene and the parking scene. Fig. 6.11 shows the storage requirements for the crossing scene in percentages for the non-indexed frame-by-frame-based descriptions (a), the R*-tree indexed frame-by-frame-based descriptions (b), the non-indexed interval-based descriptions (c) and several variants of R*-tree indexed storage of interval-based descriptions (d,e,f). Note that the conversion from frame-by-frame-based to interval-based descriptions lowers the storage requirements with more than 91 per cent. The R*-tree indexed trajectory data has an overhead of a factor 2.1 compared to the storage requirements for non-indexed storage. If also the date and time are stored in the hierarchy, the storage requirements will be increased with a factor 2.8 compared to non-indexed storage. If the dimensionality of the R*-tree is increased by storing two trajectory coordinates together as one four-dimensional point in the hierarchy, the storage requirements will be increased with a factor 3.2 compared to non-indexed storage. However, when comparing the storage requirements to non-indexed frame-by-frame-based descriptions of trajectories, the R*-tree storage of interval-based trajectory descriptions still lowers the storage requirements with at least 65 per cent.

Fig. 6.12 shows the storage requirements for the parking scene in percentages for the non-indexed frame-by-frame-based descriptions (a), the R*-tree indexed frame-by-frame-based descriptions (b), the non-indexed interval-based descriptions (c) and traditional R*-tree indexed storage of interval-based descriptions (d). Note that the conversion from frame-by-frame-based to interval-based descriptions lowers the stor-
6.3. RESULTS

(a) Average search time for 3 storage types over all queries

(b) Average search time for 3 storage types over all queries

(c) Individual search time (Interval-based Indexed search)

(d) Individual search time (Interval-based Indexed Search)

(e) Average number of results over all queries

(f) Average number of results over all queries
age requirements with more than 93 per cent. The R*-tree indexed trajectory data has an overhead of a factor 2.2 compared to the storage requirements for non-indexed storage.

Figure 6.11: Storage requirements for different trajectory data storage methods (crossing scene).

Figure 6.12: Storage requirements for different trajectory data storage methods (parking scene).
6.3.3 Search & Retrieval

After the search query is defined in the Graphical User Interface (GUI), the query follows a predefined path through the system. For clear understanding, Fig. 6.13 shows the identified search and retrieval architecture from Chapter 4. In the following subsections, the optimizations and measurement results for the Preprocessing, Postprocessing and Rank-Join module will be explained.

**Preprocessing**

The following optimizations as addressed in Subsection 5.3.1, will be benchmarked on the common query set defined in Section 6.2:

- query-MBR split-up factor to minimize the dead-zone;
- sub-query execution ordering, dependent on the object trajectory data density.

In Subsection 5.3.1, a novel query-MBR split-up policy is proposed that minimizes the dead-zone of the query-MBRs. The number of query-MBR split-ups is indicated by a split-up factor $k$, which is determined by the length cost value $C_{\text{length}}$ and the angle cost value $C_{\text{angle}}$ of the drawn query line-segment. The split-up factor $k$ is limited by a maximum value $\text{MaxSplitUps}$. We define $k_{\text{total}}$ as the number of split-ups $k$, summed over all line-segments of the query trajectory. For different values of $\text{MaxSplitUps}$, the performance has been measured for query trajectory 16-20 as defined in Fig. 6.4. In Fig. 6.14a, the decrease in search time is shown for each query, together with the $k_{\text{total}}$ value. In Fig. 6.14b, the number of candidate results is shown. Note that the difference in search time is larger for low values of split-up factor $k$ compared to higher values of $k$. Increasing the $\text{MaxSplitUps}$ factor also increases the overhead on query processing. This explains why the differences between higher values of split-up factor $k$ are smaller compared to lower values of $k$. The optimal value of $\text{MaxSplitUps}$ depends solely on the expected number of retrieved results, which can be determined automatically by the sub-query execution ordering optimization.

In Subsection 5.3.1, an optimization is proposed that enables fast intermediate result set processing. Therefore, an activity dependent ordering of the drawn queries is effectuated. A lookup table, containing information about the density of object trajectories is generated in the MDA module. This table is graphically visualized...
as a motion activity map in Fig. 6.16 for the parking scene and in Fig. 6.17 for the crossing scene. The legend indicates the density of frame-by-frame-based trajectory coordinates. Experiments showed that usage of the proposed adaptive query ordering will result in a very efficient query execution. In Fig. 6.15, individual time value are shown, measured for the sub-queries of query trajectory 16–20. As an example, we consider query trajectory 20. With the usage of adaptive query ordering, sub-query two is executed first, therefore reducing the search time for the first intermediate set of returned results up to 70 per cent, compared to random ordering. If adaptive ordering is combined with the earlier proposed query-MBR split-up policy, the benefits can be even higher.

---

**Figure 6.14:** Time values and $k_{total}$ (a) and candidate results (b) for query trajectory 16–20.

**Figure 6.15:** Time values for each line-segment of query trajectory 16–20.
If querying for suspicious behaviour, trajectories of interest will be expected more often in the least activity areas in the scene, thereby indicating that the most attractive results will be presented with the first intermediate result set, while using the adaptive query ordering optimization.

![Motion activity map](image)

**Figure 6.16:** Motion activity map (a) displaying the density of trajectory coordinates on the parking lot scene (b).

![Motion activity map](image)

**Figure 6.17:** Motion activity map (a) displaying the density of trajectory coordinates on the traffic crossing scene (b).

**Postprocessing & Ranking**

When a result set, containing matching candidate trajectories, is retrieved from the R*-tree, a postprocessing module enables a ranking in order of similarity to the defined query. When a search query is selected that contains multiple parts, a Rank-Join module joins each two received result sets from the Postprocessing module, ranks the trajectories in order of similarity to the defined query and returns an intermediate result set. For the trajectory queries of Fig. 6.4c, we measured the quality of the Postprocessing module. Results, showing the ten best-ranked results for query trajectory 16–20 are visualized in Fig. 6.18. In Appendix A, the functionality of the Postprocessing and Rank-Join module are explained, visualizing the algorithms for one of the query trajectories in more detail.
6.4 Conclusions

Several conclusions can be drawn from the obtained results for each module of the system.

**Metadata analysis**
The best performing MDA setting is \((\text{MaxDist}, \text{MaxDirDist}, n) = (40, 40, 3)\). With this parameter setting, proper sampled interval-based trajectory features are obtained from a frame-by-frame-based extraction of trajectory parameters.

**Efficient storage**
The R*-tree storage method is used for efficient trajectory storage. It lowers the search time drastically compared to traditional searching. Performance increase by storing more than one coordinate together in the tree has shown very effective to further lower the number of candidate results and to decrease the search time even more. Storage of date and time features of the object's first appearance in the video will only be advantageous while querying for less than ten days of video. To be able to query for larger time ranges, other proposals are needed to guarantee a low search time (See Section 6.5).

**Search & Retrieval**
Query preprocessing, divided into a novel query-MBR split-up policy, to minimize the dead-zone, subsequently followed by an efficient sub-query execution ordering, significantly lowers the search time for the first intermediate result set that will be returned to the user. The optimal value for the \(\text{MaxSplitUps} \) should be adaptive to the motion activity near the query. By using the proposed optimizations for the Preprocessing module, graphical search queries are executed very efficient, returning intermediate result sets as fast as possible. Postprocessing and ranking algorithms are adopted to allow (partial) matching of stored trajectories to a query in a best match order.
6.5 Future Outlook

Because searching in an optimized R*-tree with added date and time is only advantageous for time ranges shorter than ten days, as concluded in Section 6.3.2, we considered the idea of building separate R*-trees for each five days of processed video. Then, searching with an optimized R*-tree with added date and time will always be faster than traditional R*-tree searching. Minimum search time will be obtained if the R*-tree is processed in main background-memory. Currently, the cost for one megabyte memory storage is approximately 0.10 euros, which is much higher than storage on a physical disk. The storage requirements for five days of processed video for the frame-by-frame-based trajectory descriptions will be approximately 280 MB for the traffic crossing scene. If stored in-memory, a storage requirement cost of 28 euros is introduced. To argue about the usage of the proposed optimizations, we explicate the search time versus the relative storage requirement for the proposed storage methods in Fig. 6.19.

![Figure 6.19: Storage requirements for different trajectory data storage methods.](image)

As shown in Fig. 6.19, R*-tree (c) is a factor 20.5 faster than non-indexed interval-based descriptions. The storage requirement cost increases with a factor 2.1 to 5.02 euros. Note that R*-tree (e), with added date and time information, decreases the search time with a factor 3.7 compared to R*-tree (c). However, the storage requirement cost increases with a factor 1.3 to 6.60 euros. R*-tree (d) with higher-dimensional storage of two coordinates in one data entry cannot beat R*-tree (e) with added date and time information on search performance. Because also the storage requirement costs for R*-tree (d) are higher than for R*-tree (e), the method should not be considered in a future system. R*-tree (f), with added date and time information, combined with the storage of two trajectory coordinates in one data entry, decreases the search time with a factor 1.5, compared to R*-tree (e). However, the storage requirement cost increases with a factor 1.6 to 10.24 euros. The results indicate that the additional cost increase while using R*-tree (f) is not justified anymore by the decrease in search time. The most attractive method is therefore R*-tree (e).
Chapter 7

Conclusions and Recommendations

A system for smart search and retrieval on video databases has been proposed with the emphasis on a surveillance application. For this purpose we have designed algorithms for efficient storage of object-tracking data. The system allows search queries on location data, exploiting the results of video content analysis algorithms. This implies searching for interesting object behaviour using trajectory similarity queries, tripwire queries and region-of-interest queries for preselected date and time ranges.

The gain of our system is caused by effectively storing sampled interval-based features of the object behaviour, rather than a video frame-by-frame based extraction of parameters. The efficient data representation methods Piecewise Aggregate Approximation and Adaptive Piecewise Constant Approximation are used as a basis for the conversion. Additionally, an advantage is the hierarchical storage of trajectory data in an R*-tree, so that only an interesting part of the stored trajectory features has to be examined for object data retrieval.

Graphical search queries are preprocessed into bounding rectangles to efficiently search the hierarchical R*-tree. After searching, a result set is returned containing candidate matching trajectory coordinates. Subsequently, a postprocessing module determines the similarity and ranking with the query request. For matching the individual coordinates with the query, cost values are defined using a Euclidean line-distance metric, together with a metric to calculate the difference in directions between line parts. To retrieve a correct ranking of trajectories in a best matching order to the query, an adaptive weighted sum is used, which enables proper (partial) matching of stored trajectories against a drawn query.

Besides trajectory similarity search, tripwires and region-of-interest queries are supported by the system. For these two types of search queries, the same search strategy is used, except for the postprocessing and ranking. Tripwire queries use a line-line-intersection metric together with the Euclidean line-distance metric to determine the correct match results. The region-of-interest query will return all candidate trajectories from the R*-tree as correct matches. Additionally, to enable even more specific searching, the system supports the combination of multiple tripwires or region-of-interest queries with logical relations.
Thirty-one days (744 hours) of video data has been processed for two different video scenes, resulting in 561,337 objects for a video directed at a traffic crossing and 114,328 objects for a video directed at a parking lot. Multiple heuristics for each part of the system have been proposed and their performance has been compared. Results show that with the proposed algorithms, a strong reduction in retrieval time is obtained, compared to traditional methods. Trajectory similarity search is approximately a factor 1440, 1950 or 815 faster compared to traditional methods, while searching with longer-line, shorter-line or multiple-line query trajectories in the traffic crossing data set. Tripwire and region-of-interest search are approximately a factor 200 to 600 faster compared to traditional methods, while searching in the parking data set, depending on the sizes of the query.

The adopted algorithms can readily be applied to home multimedia or sport video, because objects show similar behaviour (e.g. fast retrieval of interesting soccer player activities). All concepts have been evaluated on a practical demonstrator platform which was successfully demonstrated in June at the final CANDELA review meeting in Helsinki, Finland. A graphical user interface enables fast retrieval and shows the benefits of the high-level data processing by applying intuitive semantic search queries. We showed for several scenarios, that the search time on large video data sets significantly reduces, compared to traditional methods.

With respect to possible improvements for the future, the following algorithmic details are proposed:

- To enable fast intermediate result set processing, the usage of the proposed adaptive query ordering is advisable. The system will order the multiple queries effectively on estimated execution time, therefore giving preference on queries that cover the least activity areas. Also, it is advantageous to adaptively determine the MaxSplitUps factor inside this module. While querying for suspicious behaviour, trajectories of interest will be expected more often in the least activity areas in the scene, thereby indicating that the most attractive results will be presented while first querying the least activity areas.

- To further increase the intermediate retrieval speed, an idea is proposed to store each five days of processed video into a separate R*-tree with added date and time information. After a user selects date and time constraints, the R*-trees of the selected ranges will be copied in-memory, for fast retrieval. The optimized R*-tree with added date and time information has proven to be the most efficient method at an acceptable storage requirement cost.

For complete system performance improvement, the usage of a learning-based techniques can offer interesting benefits. Path-learning algorithms, for example, offer the possibility to match queries at beforehand to common visited routes in the video scene, thereby increasing the search performance.

To enable the proposed search and retrieval system in a multi-camera setup, the object's coordinate reference system has to be transformed from local image coordinates into real-world (e.g. GPS) coordinates. Video content analysis algorithms that are able to track objects across multiple cameras have already been presented in literature [67, 68]. By concerning multiple cameras, querying for objects by drawing routes on a local city map becomes reality. This will extensively expand the search capabilities for future surveillance systems.
The system can simply be enhanced with real-time alarming functionality. Therefore, an additional module must be added to the system. This module checks in real-time if the most recent object trajectories are in match with a predefined alarm-rule set. By defining alarm rules with tripwires, region-of-interest queries and trajectory similarity queries, suspicious behaviour and possible threats can be automatically noticed without further human operator interference.

To allow a combination of location-based search queries with other extracted object features such as object colour, object shape or predefined object type, our work can be integrated in a multi-feature querying system. A selection of the most promising techniques from recent literature can be found in [69, 70, 71, 72, 73, 74]. As a result, even more specific search querying is possible.
# List of Figures

1.1 A video image (a) and a VCA algorithm performing object tracking (b). 2
1.2 Trends in video surveillance: Increasing number of cameras. 3
1.3 Overview of the proposed system. 4

2.1 Spatio-temporal trajectory (a) vs. Spatial trajectory (b) vs. Query-by-sketch (c). 11
2.2 Graphical definition of MaxDist (a) and MaxDirDist (b). 11
2.3 Frame-by-frame-based description (a) and Sampled Interval-based description (b). 12

3.1 Example of a grid file (b) with two-dimensional point-data (a). 14
3.2 Example of a KD-tree (b) with two-dimensional point-data (a). 14
3.3 Example of a two-dimensional Point-Recursive Quadtree (b) with two-dimensional point-data (a). 15
3.4 Example of an R-tree (b) with two-dimensional point-data (a). 15
3.5 Example of an R*-tree (b) with two-dimensional point-data (a). 16

4.1 Proposed architecture for search and retrieval. 20
4.2 Distance \( r \) at \( \frac{1}{2} \times \text{MaxDist} \) (a) and distance \( r \) at \( \frac{1}{2} \times \text{MaxDist} \) (b). 21
4.3 Trajectory Similarity Search - Preprocessing. 21
4.4 Trajectory Similarity Search - Query Execution. 22
4.5 Two-dimensional Euclidean point-to-line distance. 23
4.6 Cost value \( C_{\text{dist}} \). 23
4.7 Postprocessing - 2D Euclidean point-to-line distance. 24
4.8 Directional difference. 24
4.9 Cost value \( C_{\text{dir}} \). 25
4.10 Differences between \( C_{\text{dist}} \) and \( C_{\text{dir}} \) in the local error measure \( E_{\text{local}} \). 26
4.11 Graphically visualisation of the Local Error defined by \( C_{\text{dist}} \) and \( C_{\text{dir}} \). 26
4.12 Hausdorff distance. 27
4.13 MatchLength against a query, visualized for two retrieved trajectories. 28
4.14 Distance \( s \) at MaxDist (a) and distance \( s \) at \( \frac{1}{2} \times \text{MaxDist} \) (b). 29
4.15 Tripwire Search - Preprocessing. 30
4.16 Tripwire Search - Query Execution. 30
4.17 Object trajectory crossing the tripwire query. 31
4.18 Crossing direction constraint from top to bottom. 32
4.19 Region-of-Interest Search - Preprocessing. 33
4.20 Region-of-Interest Search - Query Execution. 34
5.1 Example of the proposed interval-based trajectory sampling algorithm. 37
5.2 Higher-dimensional storage model (a) and (b), trajectory similarity search (c). 38
5.3 Trajectory Similarity Search - Preprocessing. 39
5.4 Large dead-zone. 40
5.5 Vertical enlargement (a) vs. horizontal enlargement (b) of split-up rectangles. 41
5.6 Visualization of the query-MBR split-up for different values of \( k \). 42
5.7 Motion activity map (a) displaying the density of trajectory coordinates (b). 43
5.8 Spatial representation (a) and graph representation (b) of paths. 43
5.9 Alternative algorithm for usage in Tripwire Search. 43

6.1 Identified modules (Chapter 1) - search and retrieval system. 45
6.2 Screenshot of the Search & Retrieval GUI. 47
6.3 Video scenes used for benchmarking. A traffic crossing (a) and a parking lot (b). 48
6.4 Trajectory similarity queries (a,b,c), tripwires (d) and region-of-interest queries (e) used for benchmarking. 49
6.5 Start-direction angle computation. Individual direction angle (a) and direction angle for various values of \( n \) (b). 50
6.6 The ten adopted frame-by-frame-based trajectory descriptions. 51
6.7 The ten sampled interval-based trajectory features, \((MaxDist, MaxDirDist) = (40,40)\). 51
6.8 Benchmark results for storage set 1 for query trajectory 1-20. 54
6.9 Benchmark results for storage set 2 for query trajectory 1-20. 55
6.10 Benchmark results for storage set 1 for tripwire query 21-25 and region-of-interest query 26-28. 57
6.11 Storage requirements for different trajectory data storage methods (crossing scene). 58
6.12 Storage requirements for different trajectory data storage methods (parking scene). 58
6.13 Identified search and retrieval architecture (Chapter 4). 59
6.14 Time values and \( k_{total} \) (a) and candidate results (b) for query trajectory 16-20. 60
6.15 Time values for each line-segment of query trajectory 16-20. 60
6.16 Motion activity map (a) displaying the density of trajectory coordinates on the parking lot scene (b). 61
6.17 Motion activity map (a) displaying the density of trajectory coordinates on the traffic crossing scene (b). 61
6.18 Top-10 ranking of retrieved results for query trajectory 16-20. 62
6.19 Storage requirements for different trajectory data storage methods. 63

A.1 Search & Retrieval - Pseudo code. 83
A.2 Query trajectory 16 (a) and the ten best ranked results (b). 84
A.3 Cost value \( C_{dist} \). 84
A.4 Cost value \( C_{dir} \). 85
A.5 \( C_{dist} \) and \( C_{dir} \) for sub-query 1 and sub-query 2. 85
A.6 Graphically representation of the Local Error defined by \( C_{dist} \) and \( C_{dir} \). 86
A.7 \( E_{local} \) for each received coordinate vs. \( E_{global} \) for sub-query 1 and 2. 87
A.8 $E_{global}, E_{local}$ with $MatchLength$ and $MatchLength$ for each trajectory. 88
A.9 Visualization of the five best-ranked results (a–e) against the query trajectory. 89
List of Tables

2.1 Summary of the discussed data-reduction methods. 10

5.1 Computational complexity for the two proposed algorithms. 44

B.1 Overview of the discussed data reduction methods, part 1. 91
B.2 Overview of the discussed data reduction methods, part 2. 92
This thesis presents the results of a year of research, done as the graduation project for a Master of Science (M.Sc.) title at the Department of Electrical Engineering of the Eindhoven University of Technology (TU/e). The graduation project was carried out at BOSCH Security Systems B.V. in Eindhoven as part of the European ITEA project CANDELA [75].

The work presented in this thesis has been successfully integrated in the surveillance demonstrator presented at the final review for the CANDELA project, July 2005, Helsinki. Its feasibility has also been shown at the ICME 2005 international conference [76], July 2005, Amsterdam. Additionally, I've been invited to write a publication that has been accepted for the ICCE 2006 conference in Las Vegas [77]. Finally, my graduation work has resulted in five patent proposals for BOSCH.

There are quite some people that I would like to thank, who helped me during the project. First of all, my graduation professor, Prof. dr. ir. P.H.N. de With (Peter), for giving me the opportunity to carry out my internship in the industry within a European research project. Additionally, I would like to thank my supervisors at BOSCH Security Systems, Dr. ir. E.G.T. Jaspers (Egbert), Ir. R.G.J. Wijnhoven (Rob) and Ir. P. Merkus (Paul), for their great cooperation and continuously support when needed. I won't forget the many interesting discussions we had, often resulted in new ideas and solutions.

To realize a working surveillance demonstrator as part of the CANDELA project, close cooperation with people from the companies Multitel, Vrije Universiteit Brussel (VUB), Eindhoven University of Technology (TU/e) and BOSCH Research, has given much added value to my graduation work.

Furthermore, I would like to thank my family, friends and colleagues who have supported me during my time at the TU/e and during my graduation project. Finally, I owe much gratitude to Marloes, for her patience and continuously support in me.
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


Appendix A

Postprocessing & Ranking

Example

The internal functionality of the proposed trajectory similarity search algorithms will be explained with an example. Obtained results for query trajectory 16 from Fig. 6.4 are visualized, using the adopted postprocessing and ranking algorithms. In Fig. A.1, the pseudo-code for the search and retrieval module is shown. We will focus on the pseudo-code section that covers the Postprocessing and Ranking sub-modules. For more information about the other sub-modules, the reader is referred to Sections 4.3 and 5.3.

```
sendQuery(Trajectory)
{
  preprocessing()
  // determine query-MBR windows
  // determine search ordering
  }
  // for each query-MBR window
queryExecution() // R*-tree hierarchical search
postprocessing()
  // for each candidate trajectory
  // for each candidate coordinate
  { determineDist(); determineDir(); determineLocal();
    determineTrajectoryMatchLength();
    determineGlobal();
  }
  // for each two sets of results
rankJoin()
  combineResultSets()
  updateTrajectoryMatchLength();
  updateGlobals(); // trajectory match length
  // query length proportions
  sortResultSet();
  returnResultSet();

Figure A.1: Search & Retrieval - Pseudo code.
```


Firstly, we show the query trajectory used through this chapter, together with the finally ten best-ranked results after querying the data set of trajectories (Fig. A.2). After preprocessing the query (Section 4.3.2), two query-MBR windows are created. When executing the query-MBR windows on the hierarchical R*-tree (Subsection 4.3.3), candidate trajectory matches are retrieved from disk.

\[ C_{\text{dist}} = \begin{cases} \frac{d}{r} & 0 \leq d < r \\ 1 & d \geq r \end{cases} \]  

Figure A.3: Cost value $C_{\text{dist}}$.

A maximum direction difference threshold $MaxDir$ is proposed, to derive the second cost value $C_{\text{dir}}$, depending on the directional difference. The cost value is computed by dividing the minimum of $\text{Dir}_{\text{prev}}$ and $\text{Dir}_{\text{next}}$ through the maximum direction difference $MaxDir$, limited by 1.
\[
C_{\text{dir}} = \begin{cases} 
\frac{\min(\text{Dir}_{\text{prev}}, \text{Dir}_{\text{next}})}{\MaxDir} & 0 \leq \min(\text{Dir}_{\text{prev}}, \text{Dir}_{\text{next}}) < \MaxDir \\
1 & \min(\text{Dir}_{\text{prev}}, \text{Dir}_{\text{next}}) \geq \MaxDir
\end{cases}
\] (A.2)

The obtained results for the finally ten best-ranked trajectories are shown in Fig. A.5 for each of the two sub-queries. The horizontal axis represents the received coordinates.

After \(C_{\text{dist}}\) and \(C_{\text{dir}}\) are computed, a local error measure is determined for each coordinate. For the local error measure, the two cost values \(C_{\text{dist}}\) and \(C_{\text{dir}}\) are not weighted equally. The reason for this is that the directional error is seen as a more important factor in the subjective similarity ranking.
The local error, graphically visualized in Fig. A.6, is defined for the following four situations:

A) When for $C_{dist}$ and $C_{dir}$ holds that $(C_{dist} + C_{dir} < \frac{1}{2})$, equal weighting is applied.

B) When for $C_{dist}$ and $C_{dir}$ holds that $\frac{1}{2} \leq (C_{dist} + C_{dir}) < 1$, local error $E_{local} = \frac{1}{4}$

C) When $C_{dist}$ is equal or larger than $\frac{1}{2}$ and $C_{dir}$ is smaller than $\frac{1}{2}$, local error $E_{local} = \frac{1}{4}$

D) When $C_{dir}$ is equal or larger than $\frac{1}{2}$, local error $E_{local} = \frac{1}{4} + \frac{3}{4} \times (C_{dir} - \frac{1}{2})$

Subsequently, the MatchLength of the current trajectory against the query is determined. Finally, a global error for each trajectory of each sub-query is computed by averaging the computed local errors for each coordinate (See Fig. A.7). It is defined in Subsection 4.3.4 by:

$$ (E_{global})_n = \frac{1}{N} \sum_{i=0}^{N-1} (E_{local})_{n,i} \quad (A.3) $$

After the global error values for all received trajectories have been determined for each sub-query, a Rank-Join module combines the two result sets. First of all, the MatchLength for each trajectory found in the two sub-queries are combined. Because shorter query line-segments should have less influence on the total similarity ranking than longer query line-segments, the global error values are combined by weighted averaging proportional to the length of the sub-queries (See Subsection 4.3.4).
A scaling factor \( w_{\text{proportion}} \) is defined by:

\[
(w_{\text{proportion}})_i = \frac{\text{Length}_i}{\text{TotalLength}_{\text{query}}}
\]  

(A.4)

where \( i \) represents a sub-query.

The global error \( E_{\text{global}} \) for the multi-line query trajectory is defined in Subsection 4.3.4 as:

\[
(E_{\text{global}})_{\text{multi-line, } n} = \frac{1}{N} \sum_{i=0}^{N-1} (E_{\text{global}})_{n,i} \cdot (w_{\text{proportion}})_i,
\]  

(A.5)

where \( i \) represents a sub-query and \( n \) a retrieved trajectory. See Fig. A.7b and d, for \( E_{\text{global}} \) of the finally ten best-ranked results against the two sub-queries.

Finally, the total global error for each received trajectory is weighted with a factor to better represent the \( \text{MatchLength} \). Factor \( w_{\text{matched}} \) represents the matched part of the query and factor \( w_{\text{non-matched}} \) represents the non-matched part of the query.

\[
w_{\text{matched}} = \frac{\text{MatchLength}}{\text{TotalLength}}, \quad 0 \leq w_{\text{matched}} \leq 1,
\]  

(A.6)

\[
w_{\text{non-matched}} = (1 - w_{\text{matched}}), \quad 0 \leq w_{\text{non-matched}} \leq 1,
\]  

(A.7)

\[
(E_{\text{global}})_{\text{weighted}} = (E_{\text{global}} \cdot w_{\text{matched}}) + (1 \cdot w_{\text{non-matched}}),
\]  

(A.8)
Subsequently, the results are sorted and returned to the user. The calculations are shown in Fig. A.8. For clear understanding, in Fig. A.9, the five best-ranked results are shown individually, as an overlay on the query trajectory and query-MBR windows.

Because the global error rankings before weighting with MatchLength are very close to each other, the retrieved trajectory with the highest \( w_{\text{matched}} \) factor will be ranked best. The currently adopted algorithm for the determination of \( w_{\text{matched}} \) has a small deviation of maximum \( \text{MaxDist} \), which is the maximum distance between succeeding stored trajectory coordinates.

However, measurements on the overall retrieval performance showed that with usage of the MatchLength factors in the ranking module, all resulting trajectories that are in full match with the query trajectory are ranked above resulting trajectories only partial in match with the query trajectory.

![Graphs showing global error with and without MatchLength](image)

Figure A.8: \( E_{\text{global}} \), \( E_{\text{local}} \) with MatchLength and MatchLength for each trajectory.
Figure A.9: Visualization of the five best-ranked results (a–e) against the query trajectory.
Appendix B

Data Reduction Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Good ability to compress most natural signals</th>
<th>Fast DFT algorithms exist $O(\log(n))$</th>
<th>Difficult to deal with sequences of different lengths</th>
<th>Cannot support weighted distance measures</th>
<th>Cannot operate solely on the spatial $(x,y)$ dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFT</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DWT</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chebyshev Polynomials</td>
<td>+ Most efficient approach</td>
<td>+ Time series can be of arbitrary length</td>
<td>+ Only $O(n)$ time complexity</td>
<td>+ Is able to support multi-dimensional time series</td>
<td>+ Cannot operate solely on the spatial $(x,y)$ dimension</td>
</tr>
</tbody>
</table>

Table B.1: Overview of the discussed data reduction methods, part 1.
APPENDIX B. DATA REDUCTION METHODS

Singular Value Decomposition (SVD)
- Optimal linear data reduction technique
- The eigenvalues tell us something about the underlying structure of the data
- Computationally very expensive
  Time: $O(Mn^2)$
  Space: $O(Mn)$
- A single insertion into the database requires recomputing of the entire SVD
- Don't support weighted distance measures or non-Euclidean measures

Piecewise Aggregate Approximation (PAA)
- Extremely fast to calculate
- As efficient as other approaches (empirically)
- Support queries of arbitrary lengths
- Supports non-Euclidean measures
- Supports weighted Euclidean distance
- Is able to support multi-dimensional time series
- Can operate solely on the spatial ($x,y$) dimension

Adaptive Piecewise Constant Approximation (APCA)
- Fast to calculate
- More efficient as other approaches (where the level of detail varies)
- Support queries of arbitrary lengths
- Supports non-Euclidean measures
- Supports weighted Euclidean distance
- Is able to support multi-dimensional time series
- Can operate solely on the spatial ($x,y$) dimension
- Somewhat complex implementation

Table B.2: Overview of the discussed data reduction methods, part 2.