Erlend Tøssebro

Representing uncertainty in spatial and spatiotemporal databases

Doctoral Thesis

Submitted in Partial Fulfilment of the
Requirements for the Degree

Doktor Ingeniør

Norwegian University of Science and Technology
Department of Computer and Information Science

May 2002

NTNU Trondheim
Norges teknisk-naturvitenskapelige universitet
Doktor ingeniør avhandling 2002:88

Institutt for datateknikk og Informasjonsvitenskap (IDI)
IDI-rapport 2002:07

ISBN 82-471-5488-9
ISSN 0802-6394

Preface

This is a doctoral thesis submitted to the Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), in partial fulfilment of the degree "Doktor Ingeniør" (Ph.D.). The work has been carried out at the Database Systems Group[1] in the years 1998-2002. Parts of the work were conducted during a 6 month visit to the group at Praktische Informatik IV[2], FernUniversität Hagen, Germany.

Acknowledgements

This work could not be carried out without the direct or indirect support and help from others. I would therefore like to express my gratitude and appreciation to all of those who gave me comments, assistance and support during my work on this thesis.

My first and foremost thanks go to my supervisor Mads Nygård at NTNU for the enormous time that he spent with me while doing this work. This work could not have been done without his direction, guidance and encouragement.

Secondly, I would like to thank Ralf Hartmut Güting at FernUniversität Hagen, both for allowing me to stay for 6 months in Hagen, Germany, pointing out a lot of literature for me to read, and suggesting possible directions for my research.

Third, I would like to thank Kjetil Nørvåg at NTNU for providing various types of advice on being a Ph.D. student and getting the thesis finished.

1. See http://www.idi.ntnu.no/grupper/db/
2. See http://www.informatik.fernuni-hagen.de/import/pi4/index.html

Fourth, I would like to thank Heri Ramampiaro at NTNU for help in writing this thesis and using FrameMaker.

Fifth, I would like to thank Jan Terje Bjørke for providing reading material and courses in the basics of geographic information systems. Without his help in gaining the knowledge required to complete this thesis, it would have taken me much longer.

I would also like to thank all the other colleagues who have provided assistance, both in the database group in Trondheim and at the FernUniversität Hagen. In Trondheim these were: Kjell Bratbergsengen, Jon Olav Hauglid, Jon Heggland, Tore Mallaug, Roger Midtstraum, Maitrayi Sabaratman and Olav Sandstå. In Hagen these were: Dirk Ansorge, Stefan Dieker, Anne Jahn, Jose Antonio Cotelo Lema, Miguel Rodriguez Luaces and Markus Schneider.

Last, I would like to thank Stewart Clark at NTNU for checking the language of this thesis as well as providing pointers to writing correct scientific English.

Abstract

The theme of this thesis is uncertainty in spatial and spatiotemporal databases. Due to lack of accurate measurements, or rapid changes in time, spatial and spatiotemporal data are often uncertain. This thesis presents new abstract and discrete models for uncertain spatial and spatiotemporal information. The models are based on the principle that one knows that the uncertain object, regardless of type, must be within a certain area.

The first part of this thesis concerns an abstract model. To this author's knowledge, this is the first attempt to create a general type system for uncertainty with spatial data. Individual uncertain types have been modelled before, but no work has studied points, lines and regions and used the same principles to model all three. It also seems to be the first model to handle temporal as well as spatial uncertainty. This thesis contains mathematical definitions of uncertain points, lines, regions and temporal versions of these. The thesis also contains definitions of relevant operations on these types. These operations are also evaluated for their usefulness with regard to uncertain data.

The second part of this thesis concerns three discrete models which are all based on the abstract model mentioned earlier. One of these is an advanced model that manages to model almost all of the aspects of the abstract model, but at the cost of increased need for storage space. It is also difficult to compute probabilities in a consistent manner for this model.

The second model is of medium complexity, and balances storage use and modelling power. It also has the advantage that computing probabilities in a consistent manner is much easier than for the advanced model. The third model is an attempt to bring the storage space needed as low as possible. It therefore has somewhat limited modelling power. Unlike the two other discrete models, it cannot be extended to handle spatiotemporal data.

The handling of uncertain spatiotemporal data is based on how crisp spatiotemporal data are handled in [GBE+00] and [FGNS00]. This thesis makes two important additions to these models so that they can handle uncertain data. First, it presents ways of generating a sliced representation when the times the snapshots were taken are uncertain. Second, it details how operations change as a result of uncertainty. The *Initial* and *Final* operations exemplify this as in the crisp case they return the initial and final shapes of an object, but they cannot be defined in the uncertain case. This thesis discusses how these operations can be replaced in the uncertain case.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

The theme of this thesis is spatial and spatiotemporal databases. It will look more specifically at how to model uncertainty in spatial and spatiotemporal databases. This chapter will outline the motivations, research questions, and important contributions of this thesis. There is also a quick overview of the contents of the other chapters to serve as a road map for the reader.

1.1. Motivation

Spatial databases are becoming more and more common, especially in the form of geographic information systems. These systems have traditionally relied on data being clearly defined and having crisp boundaries. This is unfortunately not the case for many types of geographical data. Some phenomena might be difficult or expensive to measure accurately, and others might change faster than it is feasible to measure them.

For these reasons there is a need for systems that can store the uncertain nature of the information and can contain information about the uncertainty itself. One might want to ask how uncertain a given piece of information is. This may be important in estimating the reliability of the results.

The following subsections will give some real-world examples of when one might want to represent uncertainty in spatial and spatiotemporal databases.

1.1.1. Points

Surveillance of vehicles. One might have limited coverage of the road network such that in some cases one knows precisely where a given vehicle is,

but at other times one can only say that it is within a certain area.

Surveillance of animals. If one has tagged an animal with a radio transmitter, one may get its approximate position, at least as long as it is within range of the receivers. However, if the animal moves outside this range, one can again only say that it must be within a certain area.

Location of submarine. Submarines are designed to be silent, and sonar arrays designed to detect them might get disturbance from the ocean floor or other sources. This may mean that one only has a very general idea of where the submarine is. It may also be difficult for the crew of the submarine itself to determine precisely where they are.

Translating textual descriptions of movement. If one has a textual description of the movements of a person (a moving point), that description might just say “the person was in Trondheim (a region) from 11 am to 7 pm.” Then all one knows is that this person was somewhere in that region during that time.

1.1.2. Lines

Partially dry rivers. Some rivers in dry areas contain water only at some times in the year or only after a period of rain. Such a river can be said to be uncertain as it is only a river at certain times

Coastline. Because of the tides, the actual location of the coastline changes continuously. If one does not want to track this change, the coastline can be stored as an uncertain line. The coastline should also be represented as an uncertain line in a map of the coast because the map cannot be updated to reflect the tide at any particular moment.

Front line. In a war, one never quite knows where the front line is, but one might have some idea.

The border between two uncertain regions. If there are two uncertain regions and one knows that they share a certain stretch of border, that stretch of border may be an uncertain line

Disputed borders. If two countries disagree on where the border between them lies, this part of the border may be stored as an uncertain line. Examples of this are the border in Kashmir between India and Pakistan, and the border between Israel and Lebanon.

1.1.3. Regions

Soil Types. According to [LAB96], soil types can be both vague and uncertain. Vagueness in this case is when one soil type gradually turns into another. Uncertainty in this case is when an underlying layer changes without any visible features on the surface. In this case, it is very difficult to measure where exactly the change occurs, even if the change is fairly abrupt.

Mineral or oil deposits. Like soil types, these can be both vague and uncertain, and for the same reasons. Additionally, mineral and oil deposits may change over time as the minerals or oil is extracted.

Digitization of paper maps. When a paper map is turned into a digital map, there may be small rounding errors in the vertexes. This means that vertexes that should be the same are not. This means that there might be a small sliver between two regions in the digitized version that were neighbours in the original. This is one of the most well studied sources of uncertainty.

Lake reservoir. A lake that is used as a reservoir for a power plant may have a water level that varies considerably based on the power consumption and the amount of precipitation in the area. The varying water level means that the extent of the lake changes.

1.1.4. Time

Measurement period. If a phenomenon is measured at slightly different times at different places, the time of the phenomenon as a whole is vague because it may have changed slightly in the time between the earliest and latest measurements. This problem may also arise when one generalizes the data to display it on a small scale. When aggregating several features into one, and these features were recorded at slightly different times, the time that the snapshot of the aggregated feature was taken is vague.

Habitat of species. If a new species of animal or plant is discovered in an area, one now knows that it is there, and one might know that it was not there ten years ago, but one probably does not know when in the intervening time the species first arrived.

Combining databases with different granularities. If one combines several databases in which time is stored with different granularities into a single database, the data stored in the databases with course granularity will be uncertain if the granularity chosen for the resulting database is that of a fine-grained database.

1.2. Research questions

Based on the motivation underlying this work in Section 1.1, the main research question becomes:

How can one efficiently model and store uncertainty in spatial and spatiotemporal data?

This question leads to the follow-up questions, determining the development of this work:

Q1 ***Current Situation:*** *Are there efforts that already have answered the main question or addressed parts of it?*

Q2 ***Requirements:*** *What are the requirements for a system that addresses the main research question?*

Q3 ***Abstract Solution:*** *How should uncertainty in spatial and spatiotemporal data be modelled in general?*

Q4 ***Discrete Solution:*** *How should the abstract solution be implemented in a computer?*

Q5 ***Evaluation:*** *How well does this research address the challenges, and how do the solutions presented compare to previous work?*

1.3. Research approach

According to [BCW95], research typically starts with a research topic. In this thesis, the topic is uncertainty in spatial databases. Then, one should formulate a more specific research question which one wants to answer. The research questions for this thesis are given in the previous section. Now we need to find a rationale for why this work should be done. In this thesis, the rationale is:

Rationale: *To be able to store and retrieve uncertain spatial and spatiotemporal data, and to be able to answer queries about such data.*

In pure research, the rationale is to know something. In applied research, the rationale is to be able to do something. Given this definition, we can see that this is applied research.

This work has mainly been theoretical. After formulating the research questions, it was possible to set up a hypothesis:

Hypothesis: *One can model uncertainty in spatial and spatiotemporal data by storing a region that the object is certain to be inside, and a probability function describing the likelihood of the object being in various places inside that region.*

From this starting point, I went on to design a mathematical abstract model. This abstract model is described in Chapter 3. This is the first step towards being able to model and store something according to [EGSV98]. The abstract model was inspired by the model already proposed for crisp spatiotemporal data in [GBE+00]. The abstract model shows that it is at least conceptually possible to model uncertainty in spatial and spatiotemporal data in the manner that I thought.

Then I began constructing an implementable discrete model from the abstract model. This is the second step according to [EGSV98]. This work eventually became the three discrete models presented in Chapter 4 of this thesis. I constructed three models rather than one because there is not a single best model. All the three models presented in Chapter 4 have their own advantages and drawbacks.

So far, all the work described has been theoretical. I had shown how something could potentially be implemented. To test the implementability of my models, I then implemented parts of my medium complexity model. The results and experiences from that implementation are described in Chapter 5.

1.4. Research environment

The research that went into this thesis was mainly done at the Department of Computer and Information Science at the Norwegian University of Science and Technology. However, some of the main ideas behind the thesis were developed when I was in Hagen, Germany in the spring of 2000.

1.5. Requirements

To be able to evaluate this work, I have compiled a set of requirements that the model should fulfil. I arrived at these requirements by analysing which queries a user might want to ask about uncertain spatial and spatiotemporal data. Another goal was to enable the user to ask all the same queries about uncertain data that existing models for crisp data allow:

R1 *One must be able to tell whether an object is uncertain or not.*

R2 *One should be able to say something about how uncertain an object is.*

R3 *One must be able to say where the object certainly is not*

R4 *One must be able to say where the region certainly is*

R5 *One should be able to compute the probability that the object overlaps or is inside a given area*

R6 *One should be able to compute the probability that a given crisp point is inside an uncertain region*

R7 *The model should be able to handle all numerical and spatial data types*

R8 *The model should be able to store objects with temporal as well as spatial uncertainty*

R9 *One should be able to get crisp versions of uncertain objects with varying degrees of confidence*

R10 *One should be able to compute all operations that can be run in standard spatiotemporal models like* [GBE+00]

1.6. Publications

This thesis is partly covered by the papers that I have co-authored during the years I worked on this thesis.

- Erlend Tøssebro and Ralf Hartmut Güting: *Creating Representations for Continuously Moving Regions from Observations.* In the proceedings of the **7th Int. Symposium on Spatial and Temporal Databases (SSTD01)**, pages 321-344, July 2001. ([TG01])

  This paper deals with the problem of creating a representation for a region that is capable of showing that the region is continuously changing when the original data is in the form of snapshots of the region.

- Erlend Tøssebro and Mads Nygård: *Abstract and Discrete models for Uncertain Spatiotemporal Data.* Poster presentation at **14th International Conference on Scientific and Statistical Database Management (SSDBM 2002).** An abstract is in the proceedings ([TN02f]).

  This is an overview of the work that is presented in this thesis.

- Erlend Tøssebro and Mads Nygård: *Representing Uncertainty in Spatiotemporal Databases.* To be submitted for journal publication ([TN02g]).

  This paper is basically Chapter 3 of this thesis.

- Erlend Tøssebro and Mads Nygård: *Three Discrete Models for Uncertainty in Spatiotemporal Databases.* To be submitted for journal publication ([TN02h]).

  This paper is basically Chapter 4 of this thesis.

- Erlend Tøssebro and Mads Nygård: *Representing Uncertainty in Spatial Databases.* Submitted for conference publication ([TN02a]).

  This paper contains an abstract model for uncertainty in spatial databases. It is based on the idea that one knows that any spatial object is within a region.

- Erlend Tøssebro and Mads Nygård: *Uncertainty in Spatiotemporal Databases*. To be published in the proceedings of the **Second Biennial International Conference on Advances in Information Systems (ADVIS)** ([TN02b]).

  This paper extends the model described in the previous paper so that it can also model temporal information. It also contains some more operations. This paper and the preceding one together cover Chapter 3.

- Erlend Tøssebro and Mads Nygård: *Advanced Discrete Model for Uncertain Spatial Data.* To be published in the proceedings of the **Third International Conference of Web-Age Information Management (WAIM).** ([TN02c])

  This paper describes the first of the three discrete models based on the abstract model from the two previous papers. This is called the advanced model because it is capable of modelling more than the others.

- Erlend Tøssebro and Mads Nygård: *Medium Complexity Discrete Model for Uncertain Spatial Data.* Submitted for conference publication ([TN02d]).

  This paper describes the second of the three discrete models. The model presented here requires less storage space than the one from the previous paper.

- Erlend Tøssebro and Mads Nygård: *Extending Discrete Models for Uncertain Spatial Data to Spatiotemporal Data.* Submitted for conference publication ([TN02e]).

This paper describes how the models presented in the two previous papers can be extended to model time as well as space. This paper and the two preceding ones together cover Chapter 4.

1.7. Contributions

The main contributions of this thesis are as follows:

- *It shows how uncertain spatial data can be modelled by a support region and a core of the appropriate type.*

  In Chapter 3, spatial data types for points, lines and regions based on this principle are shown.

- *It discusses useful operations for spatial and spatiotemporal data, and shows how some of the operations change in the uncertain case.*

  As a part of this, a list of operations from earlier work is evaluated for use in the uncertain case in Chapter 3.

- *It discusses operations that measure uncertainty.*

  The thesis introduces some new operations as well as some operations that exist for other types of data and shows how they can be used for uncertain spatial and spatiotemporal data in Chapter 3.

- *It shows how the model for uncertain spatial and spatiotemporal information can be implemented.*

  Chapter 4 presents three possible ways of implementing this model.

- *It examines how to deal with temporal uncertainty in a spatial database.*

  Chapter 3 contains a method for extending uncertain non-temporal types into corresponding temporal types that can represent temporal uncertainty. In Chapter 4, several ways of dealing with temporal uncertainty are described.

- *It examines how to use probability functions to estimate probabilities of operations.*

  The model presented in Chapter 3 stores probability functions in the

uncertain data types. Chapter 4 discusses several ways of using standard one-dimensional functions to compute spatial probabilities.

1.8. Organization of the thesis

This thesis is divided into seven chapters as well as two appendices.

- *Chapter 1* contains this introduction.

- *Chapter 2* contains an overview of past achievements in spatial databases in general. It defines many of the terms that are used later in the thesis. More specific overviews of past achievements for specific parts of the thesis are given at the beginning of each chapter.

- *Chapter 3* contains a high-level abstract model for uncertain spatial and spatiotemporal information. It uses infinite point sets and takes ideas from both fuzzy set theory and probability theory. It is based on the idea that one knows an area within which the uncertain object is certain to be.

- *Chapter 4* contains three different discrete models that are all lower-level versions of the abstract model from Chapter 3. The different advantages and disadvantages of the three models are discussed in detail.

- *Chapter 5* contains a description of a test implementation of parts of my medium complexity model from Chapter 4. It also discusses some new aspects of the model that became evident during the implementation.

- *Chapter 6* contains a discussion of the pros and cons of the various models presented in Chapters 3, 4 and 5.

- *Chapter 7* concludes and describes possible extensions of this work.

- *Appendix A* contains a paper co-authored by me while I was in Germany. This paper is somewhat outside the scope of the thesis itself, but it is referenced several times in Chapter 4.

- *Appendix B* contains a list of the classes that the implementation described in Chapter 5 consists of. It also contains lists of the functions that these classes contain.

Chapter 2
Introduction to Spatial Databases

2.1. Introduction

Spatial databases have become much more common in recent years. Several commercial products have already been developed for one of the most important application areas of spatial databases, Geographic Information Systems (GIS). Although spatial databases are mainly used in geographic information systems today, they can also have other uses. It could for instance be useful to geologists to have a three-dimensional database which contains the type of rock at different points and depths underground.

This chapter is divided into six sections, which will discuss various aspects of spatial databases. Section 2.2 discusses different ways to model spatial data. To be able to store spatial data in a computer, one must find a proper representation, or model, for it. Unlike, for instance, ones bank balance, there is no single clear, intuitive way to store spatial data. There are also different types of spatial data, which should be stored in different ways. The location of a house can be stored as a set of coordinate values, whereas the plot of land in which the house lies must be stored differently, since one wants to store its shape as well as its position. Section 2.2 will therefore consider the various types of data which one might want to store in a spatial database and how they might be modelled and stored. The section will also introduce a number of terms which will be used in the later sections.

Section 2.2.1, Query methods for spatial databases, will discuss common types of queries which users might want to ask of a spatial database. This issue is important in choosing models and designing indices for spatial data, because one wants a data structure which is efficient in answering common queries.

Section 2.3 will discuss how to index data where the key has more than one dimension in an efficient manner, and discuss the benefits and drawbacks of the different indexes. Indexes are an important ingredient of all databases, especially when they store large amounts of data. In recent years, huge volumes of spatial data have been generated by satellites, and this stream of data will only increase in the years to come as more sophisticated satellites are launched. Although an index is not strictly necessary to find the data, they provide a significant boost in speed. Without an index, finding a given piece of data will take O(n) time, whereas it will usually take O(log(n)) time with a tree-based index. If the database contains 1 million disk blocks storing one type of data, it will take on average 1/2 million time units to find a particular data item without an index, and maybe 6 time units with a tree-based index, which is a speed-up of over 50000 times.

Section 2.4 is called “automated generalization”. Generalization is the process of removing unnecessary detail when producing a map on a small scale from source data on a larger scale. If one for instance had a collection of maps at the scale of 1:50000 and wanted to produce a map at the scale of 1:250000, one could not just reduce the size of everything and plot it on the map. The symbols would either become so small that they could not be seen or be placed so near each other that they overlapped and would therefore become unreadable. Therefore, map makers remove less important features when they make maps on smaller scales. A geographical information system, which is by far the most common form of spatial database, may be used to create maps from the data that it contains. It would be a great advantage if such a system could perform this generalization process automatically, because this would allow the user to “browse” through the map data. The user could be presented a very generalized map over a large area and then zoom to the area that the user was really interested in. More and more details would be added as the scale became larger until the user had what he/she needed. This process could also be used to assist cartographers in creating generalized paper maps, but so far the results of manual generalization look much better than those from automated generalization.

Section 2.5 is about spatiotemporal data, which is a relatively new field combining research in spatial databases and temporal databases. The time dimension has received more attention in the last years. The capacity of the stor-

age media has become large enough that old data do not have to be deleted. The advantage of this is that the user can ask the database about historical information, for example where the borders of one particular country were sixty years ago.

Section 2.6 contains an overview on existing research in representing uncertainty and vagueness in spatial and temporal databases. This section contains the answer of research question *Q1* from Section 1.2.

2.2. Data models for spatial data

This section will describe the various types of spatial data and how these data types are usually modelled in a computer. One cannot store an exact representation of most types of spatial data. A line, for example, such as the coast of Norway, consists of infinitely many points, and all these points cannot be stored in a computer. This means that an inexact representation using a finite number of points must be used to represent the line.

Section 2.2.1 describes some common queries that users might ask a spatial database. Section 2.2.2 describes the various data types which can be stored in a spatial database, and the two next sections describe various ways to model these kinds of data. Section 2.2.5 compares the different models for some of the data types. Section 2.2.6 discusses a commonly used way of storing different kinds of spatial data in the same database called layering, and how this model has been altered and improved into an object-oriented model. Section 2.2.7 discusses the problem of getting different spatial databases to work together. This is included in this section, because much of the problem lies in the fact that different databases use different models for the same kinds of data. (If they had used the same type of model, it would be easy to write a conversion program).

2.2.1. Common query types

Before discussing the data types and data models which are used to store spatial data, a discussion of some of the most frequently asked query types might be useful, because some of the models which are described later were made to answer one or more of these types. The spatial join is especially important because it is both frequently used and computationally expensive.

- Region Query (select): The user wants to see all objects of a certain kind that lie within or overlap a certain area. Example Query (1): Show all roads, rivers and terrain elevation within a given bounding box.

- Neighbour Query (select): The user wants to know which object is nearest in space to a given object or the user wants to know which objects lie within a certain distance from the object. Example Query (2): Which elementary school lies closest to a given house? (Used to determine which school the children living in that house should attend)
- Similarity Query (select): Finds all objects that are similar in shape to a given object. Example Query (5): In a geological database: Find all cases where a certain type of rock layer has a certain shape. (Used to find likely oil deposits).
- Spatial Join (join): Checks which objects overlap a given object or area or checks overlaps between the objects in two sets of objects. Examples: Query (3): Which plots of land contain a river? Query (4): which houses lie less than 200 metres from a river?

2.2.2. Data types

This subsection describes four different data types that are common in two-dimensional spatial databases.

2.2.2.1. Point data

Point data is data for which the position, but not the shape, size or other spatial properties are of interest. Examples of such features might be buildings in rural areas, mountain tops, and view points. Many point features arise because of generalization. Such point features should be modelled as the feature types they have on high scales and then generalized into points.

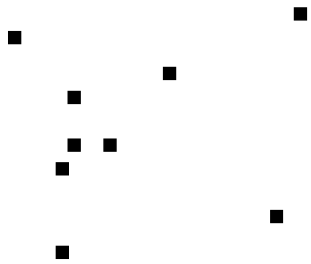


Figure 2.1 *Point data*

Points are often used in the representation of other kinds of features. A line, for instance, is often stored as a set of points with straight lines between them. Therefore, storing points is important in a spatial database even if there are not a lot of point features.

2.2.2.2. Line data

Line data consists of objects which have a length and a shape, but no discernible area. One good example of a line feature would be a network of power lines. Other features which are often represented by lines are roads and rivers, but these are really long, narrow areas rather than lines. They may be stored as

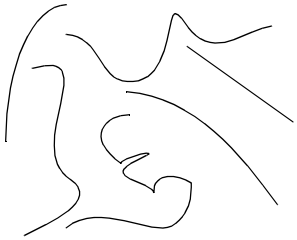


Figure 2.2 *Line data*

lines if the database is only going to be used to generate small-scale maps, or they can be stored as areas and generalized into lines for small scales. See Section 2.4 for more on generalization.

2.2.2.3. Field data

Field data is data which varies continually across the area of interest rather than being divided into discrete areas. Terrain models, soil types, pollution levels, and some forms of geological data are all good examples of such data. On traditional paper maps such data are either presented using contour lines, or by colour-coding the points in the map. Fields can have any number of dimensions. (Terrain models are two-dimensional, while geological data are three-dimensional. Meteorological data might be considered to be a four-dimensional field with time as the fourth dimension)

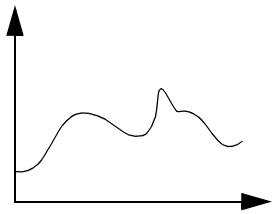


Figure 2.3 *One-dimensional field*

2.2.2.4. Region features

A region is a geographical object where the shape and size are of interest, such as a plot of land or a country. Field features are sometimes represented as region features, but are a class of their own and should be represented differently in most cases. The main difference is that a region represents strictly bounded, discrete areas such as countries, while field data represent phenomena which vary continuously in the area of interest.

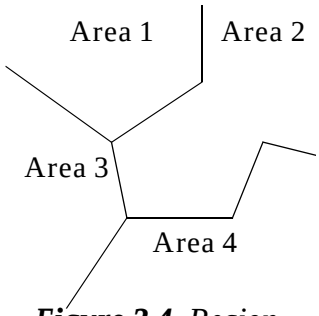


Figure 2.4 *Region features*

2.2.3. Vector models

A vector model consists of one or more networks of lines. The model does not have to represent line features, though.

A line in continuous space consists of infinitely many points. It is obviously not possible to represent this in a computer, so instead a line is represented by a set of points and an interpolation rule which is used between the points. The most common way of storing a line is to store a sequence of points and draw straight lines between points which are next to each other in the sequence. Other options include storing

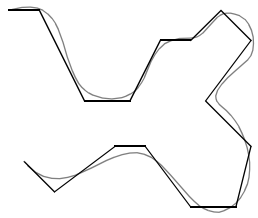


Figure 2.5 *A line and its vector representation*

the control points for a set of Bezier or spline curves which lie next to one another.

2.2.3.1. Modelling curved lines with straight line segments

There are basically two ways of storing a curved line using straight line segments. The first method, which is often called a spaghetti model, consists of storing an unordered set of straight line segments. When all of them are put together, they form a continuous line. The second method consists of storing an ordered sequence of points, in which neighbours are connected by straight lines.

2.2.3.2. Modelling region data with vectors

A set of regions can be modelled as a linear network where the lines represent the borders between the various regions. There are several versions of this model. In the earliest version, the border of each area is represented by a closed line[1]. In this early model, finding the location of an area is easy, but finding adjacent areas is difficult, since each border line is effectively stored twice, once for each area.There are two versions of this model which are better at storing neighbour information which are commonly used. Both are based on storing the border of an area as multiple lines which form a closed line together. In these versions each line represents a border with one particular neighbour. The first version is to store a link to the twin of each line in the neighbouring area. The second is to store only one line and store links to the areas on each side of that line.

2.2.3.3. Modelling fields: Iso-lines

Iso-lines is the method which is most commonly used to display terrain elevation in ordinary paper maps. The method consists of drawing lines in which the feature of interest remains constant. If these lines and the value they represent are stored in a database, a reasonably accurate model of the terrain elevation can be made. The problem with this type of data model is that producing iso-lines requires quite a lot of preprocessing, possibly involving another model. Measurements of fields are usually either height measurements in individual points done by cartographers on the site, or they are height-rasters generated by a satellite passing over the area. It would then be better to store this original model and either create the iso-lines when needed or store them as helpful information to the display software.

1. A closed line is a line where the start point and the end point are the same.

One argument for using iso-lines to store elevation data is that one of the most used sources of elevation data for spatial databases is the iso-lines of paper maps. These would be easier to store with iso-lines in the computer than with any other model.

The reason why iso-lines are used in paper maps is that they are relatively easy for people to understand as well as easy to draw. The other models, such as TINs, are better suited to storing the data in a database, but cannot be used directly to present the data to the user.

2.2.3.4. Modelling fields: Triangular Irregular Networks

A triangular irregular network, or TIN for short, is a set of non-overlapping triangles covering the entire area of interest. This can be used to represent a field by using points which have been sampled from the field as the corners of the triangles. Because all the triangles are planar[1], interpolating a value for the field for any point in the area of interest is easy. All one has to do is to enter the x- and y-coordinates into the formula for the 3D plane which is defined by the triangle surrounding the point of interest. Interpolating the field value in a randomly chosen point is much more difficult in an iso-line representation. Additionally, the interpolated value is not unique in such a representation.

Another reason for using TINs is that one can easily generate a TIN from a collection of sample points (Some algorithms for doing this are described in [LS80] and [Tsai93]).

2.2.4. Raster models

In a raster model, the entire area of interest is divided into small cells with the same shape. The most commonly used shapes are squares and cubes, but others are also possible. Each cell is then assigned a value which depends on the feature that particular raster is supposed to model. Rasters are poorly suited for storing point and line data, and are therefore rarely used for these purposes. However, both field data and area features can be stored as rasters.

1. A triangle in 3D space defines a plane passing through the triangle. (3D space because there are two spatial dimensions and the value of the field is the third dimension.)

2.2.4.1. Storing fields as rasters

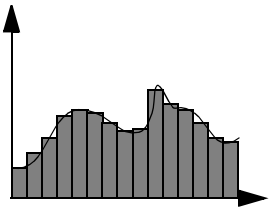


Figure 2.6 *"Raster" rep. of 1D field.*

The raster is the oldest method for storing fields. It divides the area covered by the database with a fine, rectangular grid, and stores the value of the feature in each grid cell. This is the same method that is used to store images in computers. The problem with rasters is that they require a lot of space, and they require either that the feature of interest is sampled at regular intervals (one sample per cell), or that the values in the cells are computed from the samples. The problem with space is usually a lot worse for GIS rasters than for images, because rasters typically contain a lot more cells than images, and the user is usually only interested in parts of the raster. The benefit of rasters is that they are easy to manipulate. Several rasters can be placed "on top of" each other easily, which produces a composite raster, and the resolution can be reduced by turning four points[1] into a single points.

2.2.4.2. Storing regions with rasters

Area features can also be represented by rasters. In this form of representation, each area object receives a particular number, and that number is stored in all raster cells which are completely within the area. Those that are on a border will either get a special "border" number or they will get the number of the area which covers the largest part of them.

2.2.5. Comparing models

This section compares the various models for storing lines and areas. Points and lines are usually stored in a single format, so there is no basis for comparison. (Lines can be stored in a more complex manner than chains of straight lines, but these methods have never, to my knowledge, been implemented in any system.) Areas and fields, however, can be stored in several different ways.

2.2.5.1. Raster vs. linear network for modelling regions

According to Section 2.2.3.2 and Section 2.2.4.2, regions can be modelled by linear networks or rasters. Of these models, the linear network consumes far less space than the raster. This is because in the linear network model, only the edge of the region and its non-spatial properties must be represented, whereas in a raster, the properties of all the raster cells must be represented, and to get a

1. Or any other number of points with the same number of points in each direction, such as 9 (3*3) or 16 (4*4).

fairly accurate representation of the shape of the region, a large number of raster cells is needed to represent it. The benefit of rasters is that combining different rasters with the same scale is fairly easy, while comparing two linear networks requires a lot more computation. (Line intersection tests for the lines in both linear networks.)

2.2.5.2. Raster vs. TIN for modelling fields

The popularity of rasters and TINs has fluctuated over the last twenty years. The raster was created first, and was the only model available for a while. In the 1980s TINs became increasingly popular because of their compactness and adaption to an uneven distribution of sampled points. Whereas a raster has an equal density of points throughout the area, a TIN may have a higher point density in areas with rapid change than in areas where the value remains stable.

In the 1990s the raster has gained popularity once more. This is mainly due to satellite imaging. Field features are often sampled by satellites taking high-resolution orthogonal pictures of the feature. This way of sampling produces “small” rasters as output, so that storing a collection of such images as rasters is straightforward. It is usually best to store the data in a format that is as close to the original format as possible, because users may want to perform new analyses on the original data, rather than a pre-analysed version which may have removed data which are important to this new user.

Another factor which has contributed to the increased popularity of rasters is that the capacity of storage media has increased.

2.2.5.3. Summary

In Table 2.1, the various models are summarized and compared with respect to space usage, the cost of performing a spatial join (See “Common query types” on page 13), and how accurately the models represent the feature that they are supposed to model.

2.2.6. Layering

Layering is commonly used as a model in GIS to represent various kinds of information in the same database. In this method, each type of information is assigned to its own layer. There is usually one index for each layer, so that information from each layer can be indexed independently. This means that roads would be one layer, rivers another, plots of land a third and terrain model a fourth. the user could then combine the layers when performing queries. This gives a simple interface for specifying what one wants in a map. Re-

Table 2.1 *Models for spatial data*

Model type	Model name	Model domain	Space required	join cost	accuracy
Raster	raster	areas, fields	high	small	ok
Vector	TIN	fields	low	high	ok
	Iso-lines	fields	medium	high	ok (poor)
	line network	lines, areas	low	high	good
Points	point collection	points, fields	low	medium	good, poor[1]

1. Good accuracy for points, poor for fields.

cent research has concentrated more on object-oriented databases which do not use layering directly. Instead, each type of data is its own class. There might be one class for roads, one for rivers, etc., and although all of these classes will store a line (or a pointer to a line object), they will store different kinds of non-spatial data. Thus, one “layer” often becomes a class in object-oriented systems.

However, the type hierarchies in object-oriented systems offer a lot more flexibility than a layer-based system. For instance, both roads and rivers are usually represented as line features. These could then inherit from a “generic line” object and yet be distinct types. The superuser of the database might then choose whether all lines should be stored in the same index or each subtype should be indexed independently.

2.2.7. The interoperability problem

This is a problem which has received a lot of attention in the last few years. The problem is that various users operate with different data models for the same kind of data in their databases. One company might store the terrain model as a raster, another as a TIN, and a third as a collection of sample points. If the companies using these three databases want to cooperate, they might want to create a single application using data from all three databases. This would be very difficult even if they all used the same model, because they might store the data differently. With a standard, the last problem would be all but eliminated, and the problem of different models would be reduced from one database not comprehending what is stored in the other to the problem of converting one model into another.

Another aspect of interoperability is the ability for servers and clients created by different developers to work together. This last problem is not spe-

cific to GIS, and systems such as CORBA have been developed to deal with it, but a GIS needs its own set of common interfaces to use these services.

Another problem is to recognize what type of information a given piece of data is supposed to model. If one has a collection of points, these could be point features, the points in a linear network, or even sample points for a field. One would need some form of meta-data to differentiate between these. OGC[1] is the largest and most well known organization which is working to create a standard for this kind of meta-data.

OGIS, the standard which OGC is creating, uses object-oriented architectures such as CORBA and OLE, which permit small programs called “objects” to cooperate through commonly known interfaces. The OGIS specification contains such interfaces for objects dealing with geographical data. Thus OGIS allows one to build distributed GIS applications using a variety of components. OGIS also contains the Open Geodata Model, which is a general set of geographic data types, which is supposed to cover all kinds of geographical data. This can be found on their web-site: www.opengis.org.

2.2.8. Relational vs. object-oriented

Many scientists see object-oriented databases as the future for spatiotemporal databases, because the normal relational model is too restrictive for this type of data. If one has a table of points, one for lines which refer to their endpoints, one for regions and one that connects lines and regions, the computer must do three joins to draw a map for a given area. (Four tables are necessary to make the scema 3NF). The fact that there are so many tables means that many joins must be performed even for simple queries. Another problem is that there are no data types for spatial data and therefore no means to create a spatial index. The object-oriented method also allows each class to implement specialized access methods for its instances.

The problems with object oriented databases in general is that there are no standard query languages, and that object-oriented databases typically are bound to one particular programming language. (Although many C++ databases now also support Java). This means that one is forced to use this programming language when interacting with the database. If this programming language later becomes obsolete (like Cobol today), it would still have to be used for those parts of the system which interacted with this legacy database. Relational databases, however, are independent of programming language (A system / extension for interacting with SQL is found in most programming lan-

1. Open GIS Consortium

guages.) However, despite these problems, object-oriented databases seem to be the better choice for spatial data.

Object-relational databases are also an alternative. In this hybrid strategy, one keeps the table structure of a relational database, but allows user-defined types of arbitrary complexity as column values. This allows some of the flexibility of the object-oriented approach, while maintaining the structure of a relational database. For instance, a river could be a table including such attributes as length, how much water flows in it and name as well as a line representing it. This line would be treated as an atomic attribute even though it is actually stored as a series of points with straight lines between them. Such composite attributes might include functions to extract or alter component values. the user should be able to call such functions from the SQL-equivalent that is used in the database. (The SQL3 standard is supposed to cover this kind of thing)

Among the commercial products, ARC/INFO is a geo-relational database, which means that the geographic part is stored in a proprietary database, while the other information (non-spatial information on the objects) is stored in a relational database (any SQL database might be used here). Smallworld is an object-oriented database using its own language (Magik). GRASS uses the file system for storing data.

Some years ago, the commercial systems were usually either vector-based or raster-based. GRASS was a raster database, while ARC/INFO was a vector database. Now, however, most of the commercial systems support both storage formats.

In recent years, major vendors of relational databases like Oracle and Informix have created spatial extensions to their databases. These support spatial data types through extensions to SQL, and often uses special access methods for indexing spatial data. The Informix Spatial Datablade uses R-trees, for instance.

2.3. Spatial access methods

Most ordinary databases use some form of index or access method to access data, as both exact matches and similarity searches are much faster with an index. Hashing is the fastest. However, the most popular index for normal databases is the B-tree. The problem with hashing is that it requires relatively stable data to work, and that it can only find exact matches, while B-trees can also be used to find approximate matches. Some hashing methods, such as extendible hashing and linear hashing, have been developed to cope with unstable

data, that is, a lot of insertions and deletions. No known hashing method can find approximate matches.

The B-tree is organized as a tree where each node is a disk block which contains pointers to its children. B-trees typically have a large fan-out, because the nodes store as many references to child nodes as can be fitted into a disk block, and disk blocks are usually quite large compared to the space required for such a reference. The leaves either contain pointers to disk blocks containing data items or to the data items themselves. This means that only logarithmic time is needed to find either an exact or an approximate match. For comparison, linear time is required by a sequential search. Hashing provides exact matches in constant time, but it cannot do approximate matching, so a sequential search must be performed.

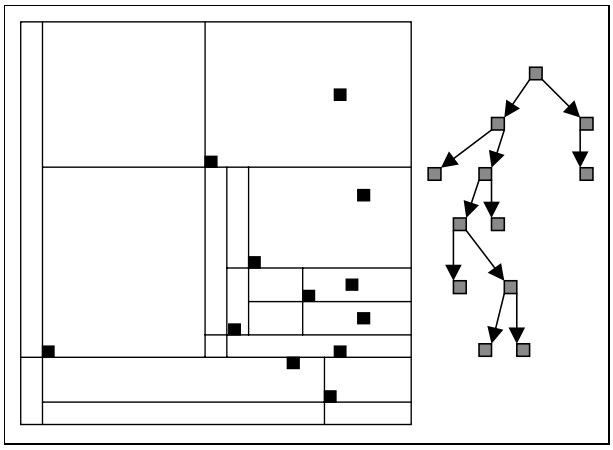
Both B-trees and hashing assume that the data items have one totally ordered key. This is unfortunately not true for spatial data, at least if one wants to search its spatial component. Although the coordinate values are totally ordered individually, their combination is not. Is for example the coordinates (5, 3) greater than or less than the coordinates (4, 9)? This means that new methods are needed to obtain the benefits of an index when accessing spatial data. These access methods can also be used when accessing some other forms of multidimensional data.

A wide variety of spatial access methods have been developed in the last 25 years. These can roughly be divided into two groups: access methods for points and access methods for extended regions. Most of the data structures have been developed with two-dimensional data in mind, but can be generalized to data with a higher number of dimensions. However, the performance of many of the data structures degenerate when the number of dimensions becomes too large.

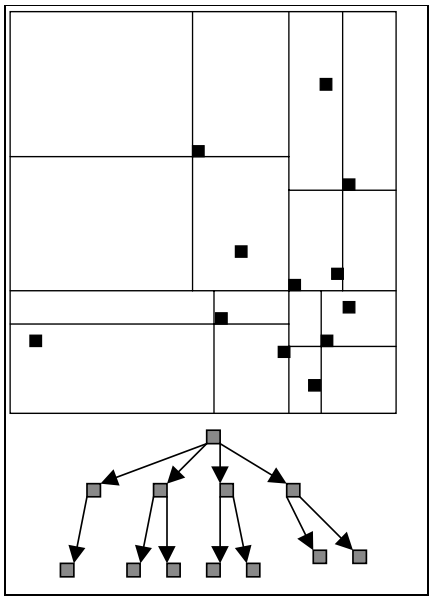
Many of the early data structures, and some of the later ones as well, are main-memory based, that is they do not take into consideration that data is transferred from disks one block at a time, not one byte at a time. Disk-based structures, on the other hand, are made to exploit this property.

2.3.1. Access methods for point data

This subsection contains descriptions of many data structures for point data. The Quad- and Kd-trees are early, main-memory-based access methods for point data. Both are based on simple binary trees.The Grid file is one of the early disk-based structures, and is not based on a tree. The BANG-file was developed to solve some of the problems with the grid file. The KDB-tree and hB-



a) Point quadtree with unfavorable order



b) Point quadtree with favorable order

Figure 2.7 *Point quadtree examples*

tree are attempts to modify the B-tree for use in spatial databases. The KDB-tree is the earliest attempt, while the hB-tree is much more recent. There are many point data structures that are not mentioned in this review. For a more thorough survey, see [GG98]. The structures presented here are either fundamental structures like the Quad-tree, kd-tree and grid-file, or more successful recent structures.

2.3.1.1. Quad tree

The quad tree [Sam90] is one of the earliest data structures that was developed to access two-dimensional data. It is similar to a binary tree with the exception that each node has four children, rather than two. In a quad tree, each node represents a rectangular part of the total space. (The root node represents the entire area). The children of a node represent a subdivision of that node. The nodes are split by two split lines, one horizontal and the other vertical. Versions of this type of tree with more than two dimensions will have 2^n children per parent, where n is the number of dimensions.

There are several variants of quad trees. The **Point quadtree** stores one point in each node. The subdivision lines of that node go through the point that is stored in the node. This means that the order in which the points are inserted is very important for the structure of the quadtree. With a very unfavourable order, the tree can become quite deep and unbalanced, while a good distribu-

tion will yield an almost perfectly balanced tree, that is, that all leaf nodes are on the same level in the tree. This can be clearly seen by comparing Figure 2.7a) and Figure 2.7b). These two quad-trees contain exactly the same points.

The **Region quadtree** is a regular subdivision of the area which is used to index objects or fields. Unlike the point quadtree, the nodes in the region quadtree are always split in four parts with the same size and shape. Each node represents a small area, and the nodes are subdivided until the entire area covered by the node has the same value for the property the quadtree is used to index or the node has reached its minimum size. This type of quad tree can be used to compress rasters with a small number of different colours. With a large number of possible values, all the cells must probably be subdivided to their smallest size, which would yield a plain raster with a tree index structure on top of it.

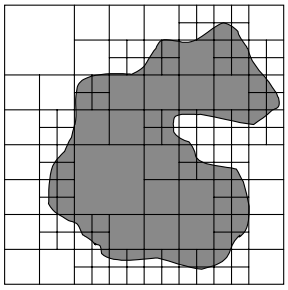


Figure 2.8 *Region QT*

The **Point-Region quadtree** is similar to the region quadtree, except it is used to store points rather than field data. Each cell in this quad tree represents a disk block or something else with a limited storage capacity. Each leaf node can therefore contain at most a certain number of points, and must be split if more are inserted. The problem with this data structure is that there is no guaranteed space utilization, even if one permits several cells to point to the same disk block. If the data are very unevenly distributed, one may split a node only to find that all the points went into just one subnode. Then this subnode would have to be split as well, and one would have three empty nodes corresponding to three empty disk blocks. (One if sharing disk blocks is permitted) Figure 2.9 shows a PR quadtree where only one point may be stored in each leaf.

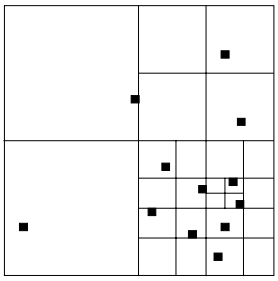


Figure 2.9 *Point-Region QT*

The **MX (matrix) quadtree** also stores points, but it does so in a different way from the PR quadtree. In the MX quadtree, nodes are always split until the lowest possible level around a point. The point coordinates themselves are not stored, only the presence of a point in one cell of minimum size. This leads to a very large index relative to the number of points, but it is sometimes used. The increased size of the index

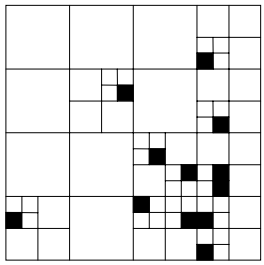


Figure 2.10 *Matrix QT*

Table 2.2 *Properties of quad trees*

Quad tree type	Data type stored	Subdivision type	Subdivision criterion
Point quadtree	points	irregular	Point inserted
Region quadtree	areas, fields	regular	Node contains border
Point-Region qt.	points	regular	More than max. number of points in node.
MX quadtree	points	regular	Nodes split to smallest size around points.
PM quadtree	lines	regular	Varies

can be seen by comparing the number of cells (leaves) in Figure 2.10 and Figure 2.9. Another way to use an MX quadtree, is to store only the quad-tree nodes which contains a data points. This can be done by using a space-filling curve (See Section 2.3.1.7 on page 34) and storing the positions of the points along this curve.

The **PM quadtree** is a class of quad trees used to store lines. All PM quadtrees split nodes into equal-sized subnodes. They differ in the splitting policy. The two most commonly used PM quadtrees are the PM3 and the PMR quadtree. The PM3 quadtree splits nodes when they contain a certain number of points, while the PMR quadtree splits nodes when a certain number of lines intersect with them. To avoid an infinite number of splits when one point has many incoming lines, the insertion algorithm tests whether one of the subnodes still contains too many lines after the split. If this is the case, that node is assigned extra storage space so that it can store a number of lines that is large enough that it does not need to be split further. Figure 2.11 shows a PMR quadtree where at most two lines may intersect the node. There is one node, however, which contains four lines. This node has been assigned extra storage space to accommodate four lines, because it is not possible to split it in such a way as to get only two lines in it.

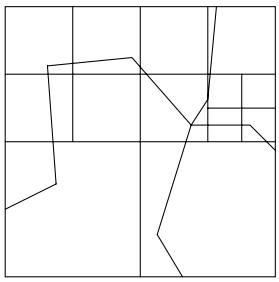


Figure 2.11 *PMR quadtree.*

An overview of the different quad tree types is given in Table 2.2. The subdivision type in this table is how the nodes are divided when they must be split. An irregular subdivision splits the node along the coordinate values of one of the points it contains, whereas a regular subdivision divides the node into four nodes with equal area.

2.3.1.2. Kd-tree

The **Kd-tree** [Ben75], [Ben79] is a binary tree that resembles the point quadtree. Instead of splitting space in all the dimensions when splitting a node, the kd-tree splits space in only one dimension. The tree usually alternates between the dimensions, so that in the two-dimensional case it splits the root node along the x-axis, the children of the root node along the y-axis, the children of those nodes along the x-axis and so forth. An example of a kd-tree is given in Figure 2.12.

The **adaptive kd-tree** [BF79] does not split nodes along one of the coordinates of a point, but rather splits it such that the two new nodes get an equal number of points. Unlike the regular kd-tree, all points are stored in the leaf nodes of this tree.

2.3.1.3. Grid file

The grid file presented in [NHS84] is a flat (non-hierarchical), disk-based data structure for storing point data. It divides space into a number of cells by means of horizontal and vertical grid lines. Each grid cell contains a pointer to a data bucket. Several grid cells can point to the same data bucket, but all these grid cells must form a rectangular area.

A data bucket is a place which can store up to a certain amount of data, usually a disk block. The fact that data items are stored in buckets rather than individually in tree leaves is one of the most important differences between memory-based and disk-based indexes. Another important difference is that a disk-based index itself can be easily stored on disk. Many quad-tree nodes can be stored in one disk block, which creates the problem of which nodes to store in which blocks. In contrast, B-tree based indexes store one node in one disk block. Grid files store a certain number of grid cells in one disk block in such a

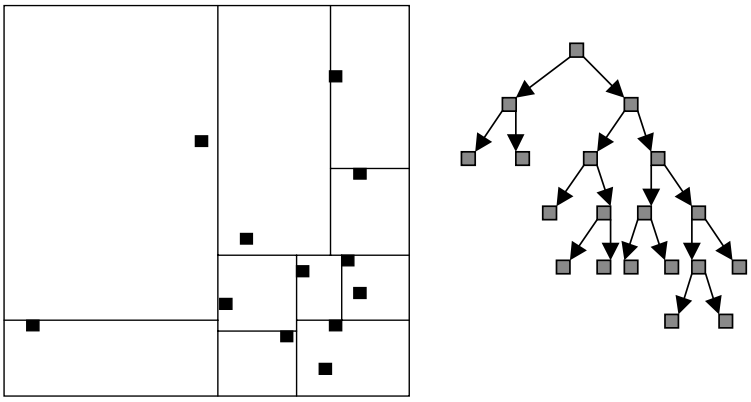


Figure 2.12 *Kd-tree*

way that it is easy to calculate which disk block to read based on which grid lines the query point is between. The grid lines are usually stored in main memory.

An empty grid file consists of one cell with a pointer to one data bucket. Whenever a data bucket is full, it has to be split. When a data bucket is split, the program first checks how many grid cells point to that bucket. If more than one grid cell points to the same bucket, the bucket is split by one of the lines dividing the grid cells. All the grid cells on one side of this line point to the same new bucket. If only one grid cell points to the bucket, that grid cell must be split. The cell is split along a line which is parallel to one of the axes. Which dimension is used depends on which dimension was used in the last split. If the last split was horizontal, this split is vertical and vice versa. The coordinate of the split line is determined in such a way that the points in the cell are equally divided among the two new cells. The new split line is then drawn across the entire space, and all cells which are intersected by the line are split, even if they are not full. If a cell that is not full has to be split, the two new resulting cells will point to the same data bucket. In this way storage space can be preserved.

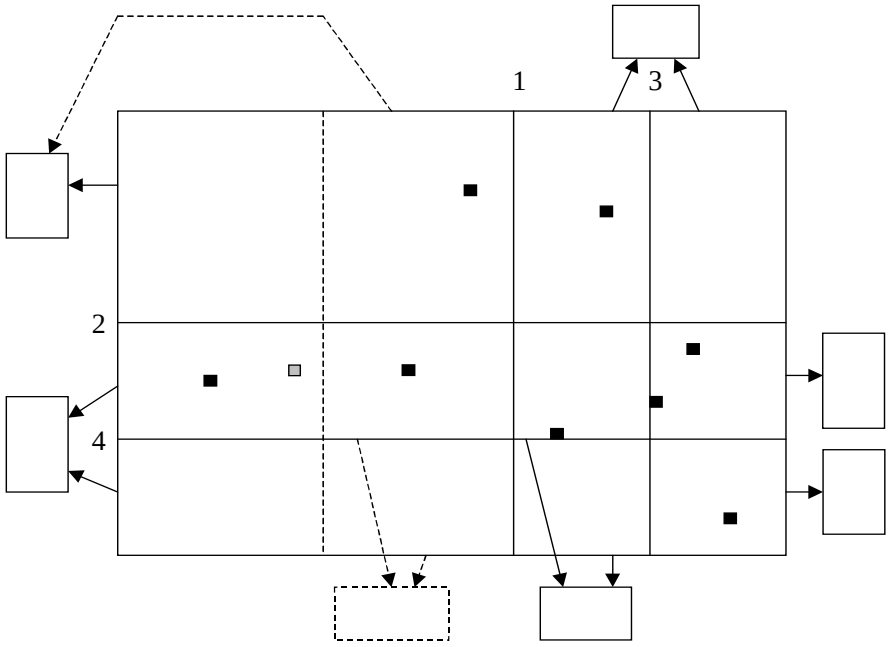


Figure 2.13 *Grid file*

The main problem with this data structure is that the index grows faster than the number of points. If the data are clustered around a few central locations, a lot of index nodes must be split unnecessarily. This is shown in Figure 2.13. When the light grey point is inserted into the grid file, its grid cell has to be split. As the last split of that cell was horizontal, this split must be vertical, as

shown by the dashed line. As the split line intersects the cells above and below the cell in which the point was inserted, these two cells also have to be split, despite the fact that they are not full. The end effect of the insertion is that there is one new data bucket and three new index entries. The number of data buckets increases from six to seven, which is an increase of 1/6. The number of index entries increases from nine to twelve, which is an increase of 1/3.

This splitting policy ensures that the cell structure remains simple, so that it can be represented by a normal matrix where each column has the same number of rows and vice versa. The grid file is relatively fast to search. (Searching occurs by comparing the coordinates of the split lines with the coordinate of the point or region in question.) If the number of cells is proportional to the number of points, the number of splitting lines is proportional to the square root of the number of points.

There are two variants of the grid file, one of which seeks to reduce the growth of the index structure and the second seeks to improve storage utilization. Table 2.3 compares the basic grid file with these two alternatives. The **two-level grid file**, which is described in [Hinr85], has a two-level directory instead of the one-level directory of the regular grid file. The grid cells in the top layer point to buckets containing new grid directories, not data buckets. The grid directories at the second level point to the data buckets. This technique reduces the growth of the index structure, but according to [GG98] it does not eliminate the problem.

The **twin grid file**, which is described in [HSW88], was designed to increase storage utilization. It uses two grid files and assigns a new point to one of them so that storage utilization is maximized. In a normal grid file, when a bucket is split the two resulting buckets are only 50 % full. This leads to an average storage utilization of ln(2), which is 69 %. (This is normal for all data structures which split the nodes in half when they are full, which is true for most multidimensional data structures and many single-dimensional data structures). The twin grid file reduces this problem. According to the article, they manage a storage utilization close to 90 % with the twin grid file. It is, however, somewhat slower than the regular grid file. (More accesses to the index to search for or insert points.)

2.3.1.4. BANG-file

The main problem with the grid file is that the index grows faster than the data. One way to solve this problem is if the directory could contain the bucket regions[1] instead of the grid cells. The first attempt at doing this is the **Interpolation-based Grid-file**. In this file, each bucket region is represented by a pair

Table 2.3 *Characteristics of grid file types*

type	storage utilization	index growth	access time[1]
ordinary	medium(69 %)	poor	2
twin	good(90 %)	poor	4
two-level	medium(69 %)	somewhat better	2-3[2]

1. Disk accesses required for exact match if grid lines stored in memory.
2. Two disk accesses required if top-level grid stored in memory, three otherwise.

of numbers,(r, l), where r is the region-number and l is the granularity level. Each granularity level is created by splitting each bucket region in the previous level in half along one dimension. The region numbers are generated by adding a 0 or 1 to the back[1] of the number of the parent region.

The directory in the interpolation-based grid file contains the number pairs of all the bucket regions. This reduces the problem of index growth in some cases, but not all. The problem is that when a grid cell is split in half, there is a risk that all the points end up in one of the parts, which will have to be split again. This leads to a poor storage utilization and does not really solve the index growth problem.

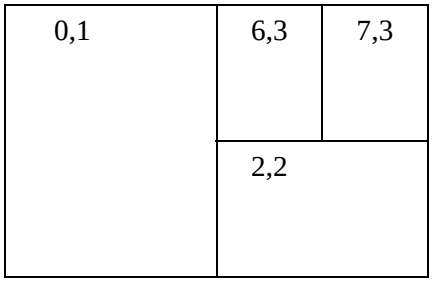


Figure 2.14 *Bucket regions with numbering*

The **BANG-file**[2] [Free87] solves this problem by allowing bucket regions to overlap. Each time a point is inserted into the BANG-file, it is inserted into the smallest region which contains it. In this manner, one can split off smaller regions of a large bucket region. For example, in Figure 2.15, when the grey point is inserted, a small part (one eighth) of the original bucket region is split off. In this manner, the regions of space which are actually stored in a given bucket are rectangular areas with smaller rectangles extracted from them. The regions need not even be continuous.

1. A bucket region is the region which is stored in one disk bucket.
1. According to [Free87] the bit is added to the front. According to [GG98] it is added to the back (as in Z-ordering). The numbering in [GG98] will be used here. Which of these ways the bucket regions are numbered by has no effect on the effectiveness of the Interpolation-based grid file or the BANG file.
2. Balanced And Nested Grid-file

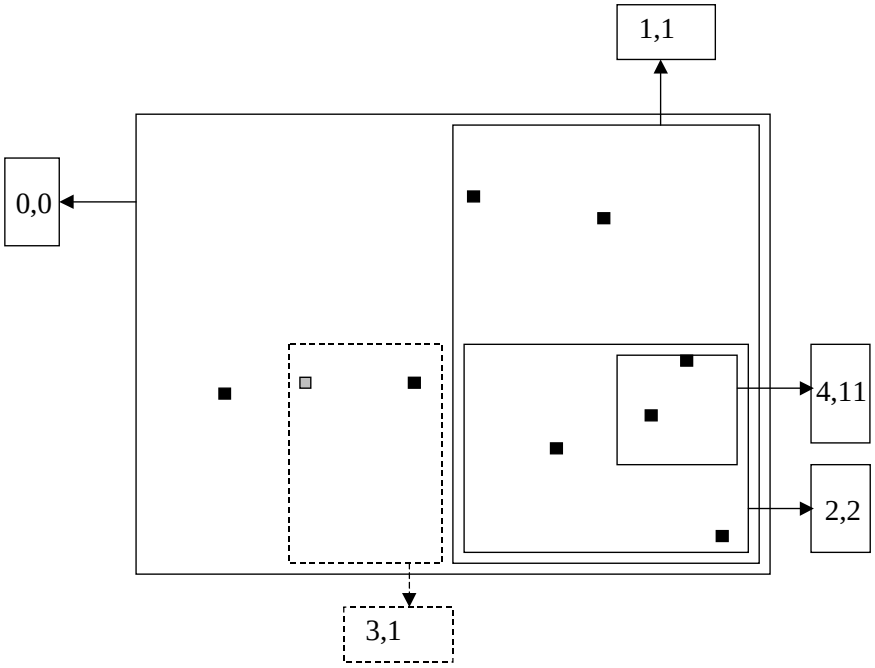


Figure 2.15 *BANG-file*

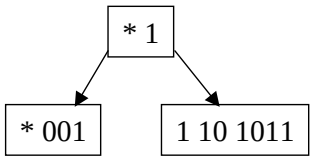


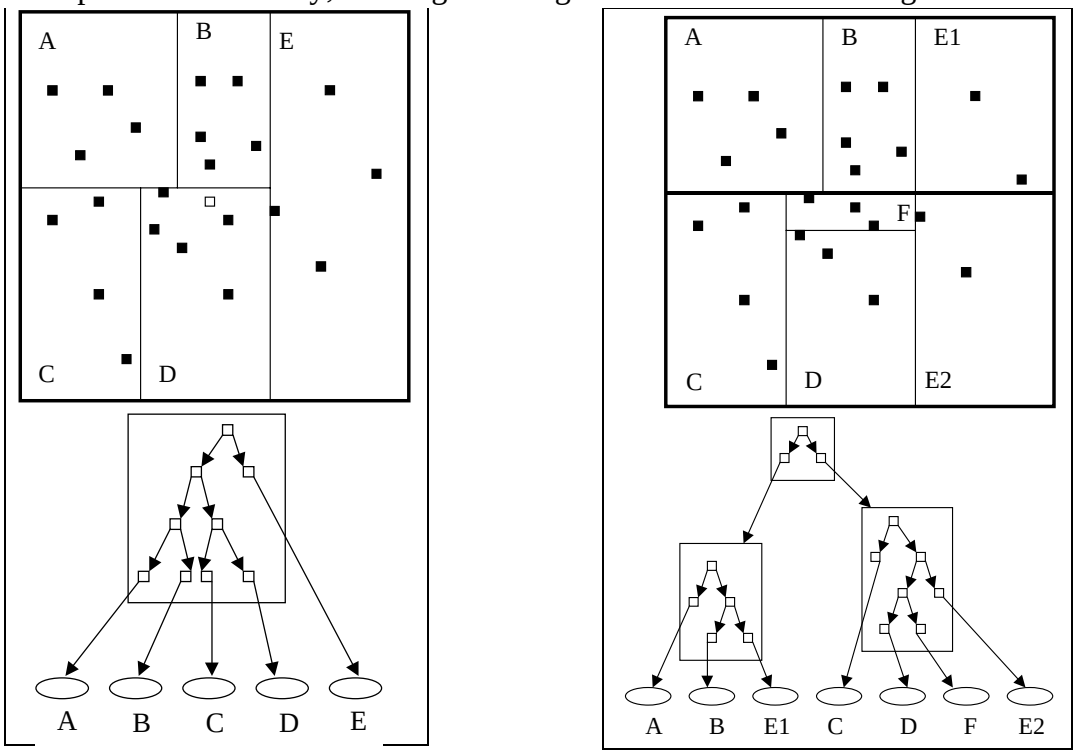
Figure 2.16 *Directory of the BANG-file*

The directory for the BANG-file resembles the B-tree in that it is a perfectly balanced tree with all the leaves at the bottom level, and that each node occupies one disk block. Each leaf corresponds to a bucket region and contains the points themselves or references to them. Each internal node in the "B-tree" contains the longest possible prefix of the binary versions of the region numbers of every leaf which is underneath it. The region numbers are generated in the same way as in the interpolation-based grid file. The directory for the BANG-file in Figure 2.15 is shown in Figure 2.16. The numbers in the nodes in Figure 2.16 are binary versions of the numbers of the bucket regions in Figure 2.15. The '*' sign means the empty bit string.

2.3.1.5. KDB-tree

The KDB-tree [Rob81] was the first attempt to create a multidimensional version of the B-tree. Like the B-tree, each node is split when it is full and the parent node is updated with the location of the two new children. However, in the multidimensional case one cannot separate between the nodes using a simple value. The solution used in the KDB-tree is to store an adaptive kd-tree in each internal node, and each leaf in this kd-tree points to a child of this node. When a node is split, its corresponding node in the kd-tree of the parent is also split along the same line.

If an internal node must be split, all its children must be split along the same line even if they are not full. This may cause nodes further down the tree to be split unnecessarily, leading to a large index and a low storage utilization.



a) KDB-tree before split **b)** KDB-tree after split

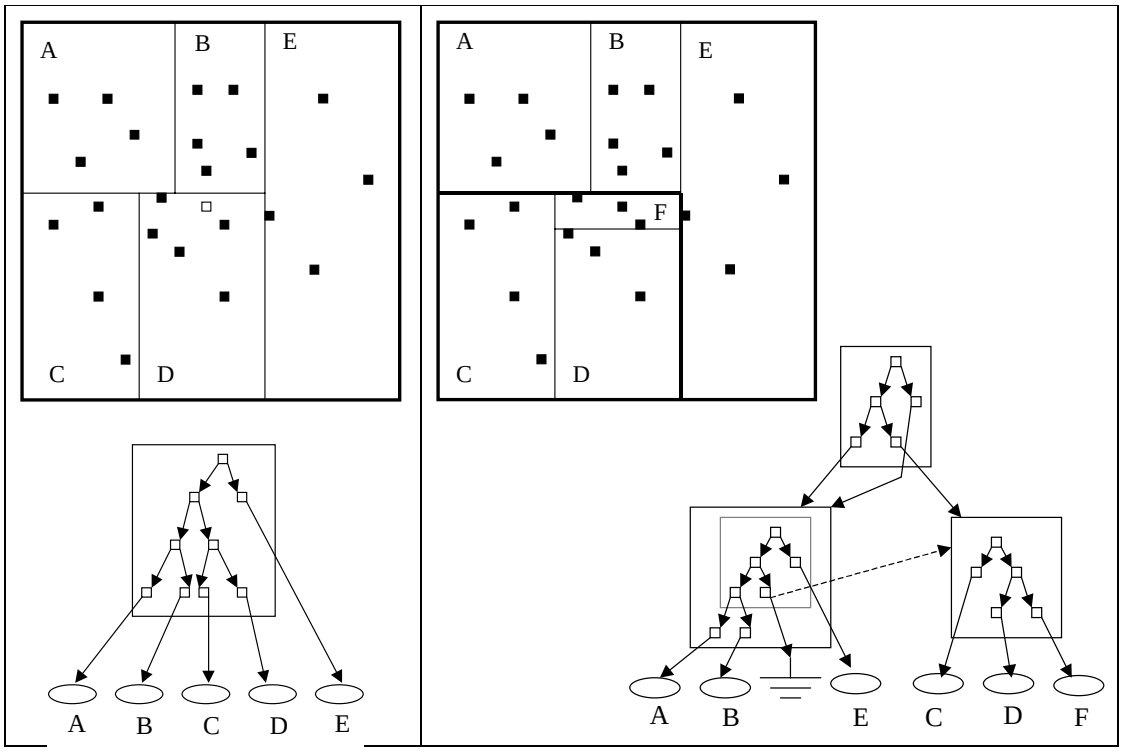
Figure 2.17 *Splitting a KDB-tree*

This can be clearly seen from Figure 2.17. In the tree structures, the large boxes are the B-tree nodes and the small boxes are the kd-tree nodes. The ovals are disk blocks. In this case, the insertion of a point causes a split in the B-tree node which in the first figure contains the entire tree. To split this node, a new root has to be added, and its kd-tree must contain two child nodes, which point to the two new B-tree nodes. One of the split lines in the kd-tree of the original node is then chosen in such a manner that each subnode of the new root contains at least two subnodes in the B-tree (in this case disk blocks). The new root then contains a kd-tree which is split along that line (Which is indicated in the second figure by a thicker line). However, notice that node E is split by this line, and therefore has to be split into nodes E1 and E2, even though it was not full. As this can happen, the KDB-tree cannot guarantee any minimum storage utilization.

Queries to this file can be answered simply by a traversal of the tree. This is like traversing a B-tree, except that the internal kd-trees of the nodes also have to be traversed.

2.3.1.6. HB-tree

This data structure from [LS89] is based on the KDB-tree. In the KDB-tree, each node is a rectangular area. In the hB-tree, the nodes are rectangular areas with smaller rectangular areas extracted from them. Each internal node contains an adaptive kd-tree which contains the references to the smaller regions which the node consists of. One difference between the KDB-tree and the hB-tree, is that more than one of the leaf nodes of this kd-tree can point to the same child node in the B-tree. Because of this, the area covered by the child node is not necessarily square. Because kd-trees contain representations of square areas, the kd-tree of the child node must contain a child covering the “hole” in the area. This kd-tree node must contain a special value signifying that this B-tree node does not cover this area.



a) hB-tree before split. **b)** hB-tree after split.

Figure 2.18 *Splitting an hB-tree*

Figure 2.18 shows an hB-tree in the same situation as the KDB-tree in Figure 2.17. Again, the borders in the new root are indicated by thicker lines. It is clear that block E no longer needs to be split, so the hB-tree has a better storage utilization than the KDB-tree in this case. The problem, however, is that in this case the kd-tree in the B-tree root node is split by some of the same lines as the kd-tree in the left B-tree child, which means that the same split lines are stored more than once in the index[1]. This problem might be less pronounced for hB-

trees which can have a larger number of kd-tree nodes inside their B-tree nodes than the example tree. It also occurs only if a B-tree child is referred to by more then one kd-tree child in its parent. According to [GG98], an experiment conducted by the creators of the hB-tree showed that more than 95 % of the index nodes of the hB-tree have only one reference to them.

One of the children in the kd-tree of the left child in the B-tree is a pointer which simply says that this region is not covered by this node. In a more recent variant of the hB-tree, the hB^{Π}-tree, this node would instead contain a reference to the node which contains the region. This is shown by a dashed line in the figure.

2.3.1.7. Space-filling curves

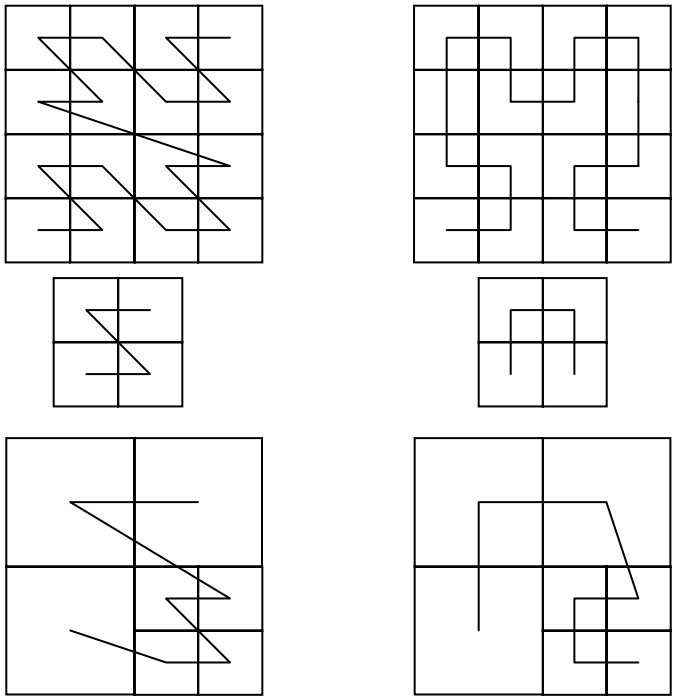
A space-filling curve is a curve which is shaped in such a manner that it passes through all the points in space. Each point in space can then be given a number based on its position along this curve. In this manner one can impose a total order on spatial objects. This allows the objects to be indexed using a traditional index method such as a B-tree. To be able to find approximate matches, it would be best if the space-filling curve is such that points which are close in space are also close along the curve.

The simplest space-filling curve is the scan-line curve. This curve moves like the electron beam of a TV or monitor, that is, it passes from 0 to maximum in the first row, and then in the second row, and so forth. The problem with this curve is that many point that are close to each other in space are not close to one another along this curve.

The two most popular curves are the **Z-curve** [OM84] and the **Hilbert curve**, both of which are shown in the figure below. The simplest form of the curves is represented in the small, middle figures. When one of the square “points” is divided into four, the single corner in it is replaced by a copy of the entire figure, as shown in the figures furthest down. The figures on the top show the structure when all the four cells have been divided. The main difference between the Z-curve and the Hilbert curve is that the latter is often rotated when a point is divided in four, whereas the Z-curve is always the same way. This makes the value of a cell along the Z-curve very easy to calculate, it can be calculated by interleaving the bits in the values of the X and Y-coordinates. The Hilbert curve, on the other hand, requires a much more complicated formula. The benefit of the Hilbert curve is that the likelihood that a point which is close

1. The kd-tree in the root is the same as the kd-tree part in the grey box in the left B-tree child.

in space is close on the curve is slightly higher for the Hilbert curve according to [GG98].



a) Z-curve **b)** Hilbert curve

Figure 2.19 *Space-filling curves*

2.3.2. Access methods for region data

Another problem with spatial data is that many objects have a spatial extent, that is, they are areas instead of points. In ordinary databases, even ones with multiple keys, the keys have definite values. For an area object, the coordinate key is a range of values rather than a single value. For this reason, extended areas must be handled differently from points when they are indexed.

Indexing areas by their centroids is too simple. One common query is the spatial join, in which the user wants to check whether the members of two sets of objects overlap one another. Because objects can be arbitrarily large or small, an index based on the centroid alone will not give any benefit in this case, because the centroids of two overlapping objects may be arbitrarily far apart.

There are two classes of data structures which have been developed to index region data. The first is the R-tree, an adaptation of the B-tree with the purpose of indexing extended areas. Numerous variants of the R-tree have been developed, and the most important of them are mentioned. The second method is to somehow transform an index for accessing point data into an index for ac-

cessing region data. Both these techniques have been fairly successful, and there is no clear winner.

2.3.2.1. R-trees

The **R-tree** [Gutt84] is a tree structure for representing region data. It is based on the B-tree [BM72]. An empty R-tree contains only one node, the root, which points to a disk block. If more and more objects are inserted into the tree, the block will eventually become full. When that happens, the block has to be split. When a block is split, half the objects are stored in one and the other half in the other block. There are several different algorithms for selecting which objects go into which block, but there are basically two criteria.

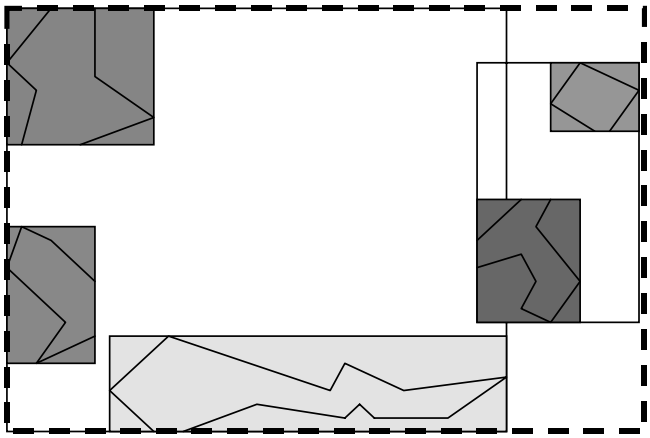


Figure 2.20 *R-tree*

Each node in an R-tree is associated with a minimum bounding region. An MBR is the smallest quadratic region which contains all the objects which are stored in that node or in children of the node. All objects are stored in the leaf nodes of the tree, the internal nodes only store pointers to their children. Also, all the leaves are on the same level of the tree, the tree is perfectly balanced. (The R-tree inherits these properties from the B-tree.) When splitting a node, the objects may be allocated to the two new nodes according to two heuristics. One can either minimize the total area covered by the MBRs of the two new nodes or minimize the area in which the two new MBRs overlap. When a node is split, the parent node is updated with entries for the two new nodes. If this causes the parent node to become full, it is split in the same manner. If the root node becomes full, it is split, and a new root is added on top of the two new nodes.

The advantage of R-trees is that they can store regions with a spatial extent without resorting to transformations. It is also straightforward to search in an R-tree. The problem with R-trees is that internal nodes may overlap, so that the search procedure may have to traverse several branches of the tree in order to find an object.

There are several variants of R-trees. The **R+-tree** prevents overlapping internal nodes by splitting the objects along the split lines of the internal nodes.

This may cause the same object to be found in several leaf nodes of the tree. The **R*-tree** [BKSS90] works much like the standard R-tree, but uses slightly different algorithms for searching and node splitting, which improves the performance. The algorithm for assigning the objects to one of the two new nodes takes several criteria into consideration rather than only the area criterion. These are: minimize area, minimize overlap, minimize ratio of boundary to area. The last criterion is there to try to prevent long, thin MBRs. The R*-tree also uses the technique of forced reinsertion to create a better distribution of data in the nodes. Each time a node becomes full, some of the entries in the node are removed and reinserted into the tree. If the node is still full, it is split. The entries to be reinserted are chosen based on the distance between the centroids of the entries and the centroid of the node. According to [BKSS90] this technique increases the average number of disk accesses for inserts by only 4 % and improves search performance by 20 %. The R*-tree is said to be the fastest of the R-trees in [GG98].

2.3.2.2. Transformation technique

The transformation technique is a method for storing and retrieving region data in point data structures. The main idea is to transform the MBR of a region into a point with twice the number of dimensions of the original region. This can be done by using the coordinates of two diagonally opposed points. If the MBR goes from (x1, y1) to (x2, y2), the point representation could be (x1, y1, x2, y2). this allows any data structure for storing points to store regions if it can handle a high number of dimensions.

This works fine for exact queries, but there are some problems with other kinds of searches, an intersection test for example. Because objects can be arbitrarily large, and the coordinate values of the endpoints therefore can be arbitrarily far apart. This may cause the search space to become infinite for query regions which were finite in ordinary space. Another problem is that the data distribution becomes uneven, as half of the data space is never used (Corresponding to that x2 is smaller than x1 or y2 smaller than y1). An additional problem is that queries are often much more difficult to express in transformed space than in original space.

Another technique is to use the centroid and the distance to the edges along the coordinate axes as coordinate values. Although this method is often preferred to using diagonally opposite points, it solves none of the problems of transformation.

Another way to transform a point data structure into a region data structure is to allow overlapping regions. The centroid of the MBR determines

which region a new object is to be inserted in, and this region is then extended to cover the new object entirely.

However, despite these problems, some of the point data structures perform almost as well as the R*-tree when the transformation technique is used according to [GG98].

2.3.3. Access method summary

Table 2.4 *Access methods*

Access method	Disk or memory based	hierarchical or flat	point or area	branch factor	storage utilization
Quad tree	memory	hierarchical	varies	4	varies
Kd-tree	memory	hierarchical	point	2	ok
Grid-file	disk	flat	point	N/A	ok[1]
BANG-file	disk	hierarchical	point[2]	high	ok
KDB-tree	disk	hierarchical	point	high	poor
hB-tree	disk	hierarchical	point[b]	high	ok
Z-ordering	varies	varies	varies	varies	varies
R-tree	disk	hierarchical	area	high	ok

1. Twin grid file has good storage utilization
2. Can be transformed into an area access method

2.4. Automated generalization

Generalization is the problem of reducing the amount of information to display on a map when producing maps on small scales from original data on large scales. If for instance you wanted to make a map of Norway and had the data collected for the 1:50000 series[1] available, you could not plot all of that on the map of Norway. The map would become completely unreadable because there were so many things on it. Instead, you would have to choose what data to use and how to represent them.

One way of querying a GIS would be to mark an area on the world map. The GIS would then display that area at greater detail. The user might mark areas for enlargement several times before he/she got the area he/she wanted.

1. The 1:50000 series is the most detailed series of maps which covers all of Norway.

This would require that the GIS was able to perform generalization by itself, that it could select the data to represent at each scale without the aid of humans.

There are several things that have been called “automated generalization”. Here they will be divided into two categories, data reduction and information reduction. It is the information reduction which really should be called generalization. (Data reduction might be called compression.)

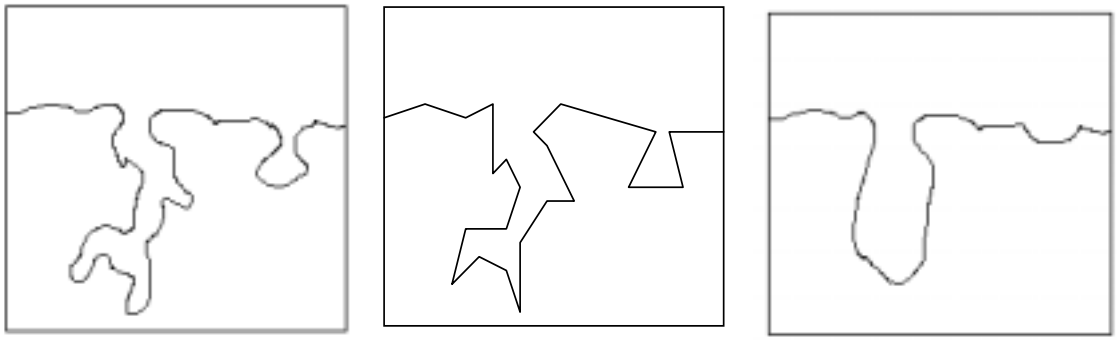
2.4.1. Data reduction

Data reduction is to reduce the amount of data that has to be displayed. If for instance the terrain elevation is stored as a raster where each point is a 10x10 metre square, and the user wants to display the map on a scale in which the points on the screen have a size of 20x20 metres, the computer has to calculate the values of the points on the screen based on the points in the database as quickly as possible. (A regular user does not want to wait for several minutes before the map is displayed.)

Another use of data reduction is to minimize the amount of data that has to be transmitted across a network. Lossless data compression is a good example of pure data reduction, because all the information can be reconstructed from the compressed data. The user will probably be sitting at a PC which requests the geographic data from a server through the Internet, a slow, unreliable medium, especially if the PC is connected via a modem.

2.4.2. Information reduction

If a database includes information on all the roads of Norway, and the user requests a map of the entire country, the database should not display all the roads. If it did, the map would be only black lines with roads and names that overlapped each other so much that they would be impossible to read. Therefore, only the major roads should be displayed at that scale. The problem is on which scales should which objects be represented and what form should they be represented in? If the map should display a cluster of houses, should these be represented by a symbol for each house or a single symbol for the entire cluster? Another example would be the Norwegian coastline. If data reduction was the goal, reducing the number of control points with an algorithm such as Douglas-Peucker [DP73] would be a solution. This algorithm tries to represent the line as accurately as possible with fewer control points. Information reduction, on the other hand, would be to eliminate less important bays and fjords and only display the more important features.



a) Original coastline

b) Coastline with data reduction

c) Coastline with information reduction

Figure 2.21 *Data and information reduction*

An example of this is shown in Figure 2.21b). The data reduction has reduced the number of control points while trying to keep the line as close to the original as possible, which leads to a line with many sharp angles, which would appear almost as cluttered as the original line at smaller scales. The information reduction, on the other hand, has removed the information on the smaller bays and rather kept only the main feature, the fjord itself. This map, Figure 2.21c), would appear much less cluttered on smaller scales than the two others. The fjord in Figure 2.21c) is also wider than it is in Figure 2.21a). This is to make it more apparent at small scales.

Scientists have tried to automate information reduction for a number of years, but with relatively little success. Purely algorithmic solutions created very poor quality maps. Rule-based systems, which have been successfully applied in medical diagnostics, also proved to be too limited for this problem. This caused scientists to concentrate more on so-called amplified intelligence rather than expert systems for some years according to [Wei95] and [Kel95]. Amplified intelligence is a method where the computer presents a solution to a problem and a human operative then tells it whether it was good or bad, and how it should have been solved. The computer then learns from its mistakes. Case-based reasoning is the amplified intelligence approach which is most promising according to [Wei95] and [Kel95]. The problem with this approach is that it requires a user who is familiar with cartography. However, many GIS products should be usable to an ordinary person with no such experience. For this reason, scientists are now trying again to create a completely automated generalization.

There are two types of data, object data and field data. Generalization of these two types of data must use different methods. Most of the research so far

has been on the generalization of object data such as lines or areas. Of these, line generalization is the area which is most well developed.

2.4.3. Generalization of object data

Object data in this context are points, lines and areas (not fields). These features can represent any kind of geographical phenomenon.

Generalization operators for object data (From [MS92])

- Simplification: Reducing the number of points in a line. This operator reduces the number of points that a line goes through. (A line is made up of a number of points with straight line segments between them.) A good example of a simplification algorithm is Douglas-Peucker from [DP73].
- Smoothing: Makes the line smoother by moving the points it goes through.
- Aggregation: Replaces a group of points or lines by a single point or line representing the group.
- Amalgamation: Unites several area features into a single larger area.
- Merge: If a rail station has 10 tracks, and the scale has become so small that these begin to overlap in the representation, pairs of tracks can be merged so only five tracks need to be shown. This reduces the amount of information, but shows that there are several parallel tracks.
- Collapse: Turns an area feature into a line feature or point feature. A river is an area on a large scale, but on a smaller scale the width of the river becomes so small that it cannot be seen. Then it is better to draw the river as a visible line feature if the river is important to the intended user of the map. A house is an area on a very large scale, but on a smaller scale the area becomes so small it is better to represent the house with a symbol, which is usually a point feature.
- Refinement: If you have a map depicting a river basin with all the tributaries, it can become cluttered on a small scale because there are so many small rivers and brooks on the map. Refinement is to remove the smaller brooks and only keep the larger or more significant rivers.
- Enhancement: Since a bridge is the same width as the road, it must be represented by a special symbol on smaller scales to be visible. Another example would be roads of differing importance. Different roads are often shown as lines with different widths to mark importance. These lines are often a lot wider than the road itself would be if it was represented on the map's scale.

- Displacement: If a road is next to a river, these two features could come so close that they are drawn on top of each other on a small-scale map. The solution to this is to place them further apart so that both are visible.
- Classification: Representing similar features with a single symbol type to reduce the number of different symbols on the map.
- Symbolization: Under collapse, changing a house from an area to a point is mentioned. That change also involves symbolization. The house, which was an area with a given shape, is replaced with a symbol which indicates that there is a house here.

[SW99] describes a framework for generalizing graphs using the operations of amalgamation and selection. In this paper, the definition of the amalgamation operation is more general than that in [MS92]. Amalgamation is defined in [SW99] as joining several features (regardless of type) that are indistinguishable at the smaller scale into a single feature. For regions, this is the same as the amalgamation definition from [MS92], but this new definition also applies to line and point features.

2.4.4. Generalization of field data

2.4.4.1. Generalization of raster data

[MS92] uses the following operators for raster generalization, which are largely drawn from image processing operations:

- Structural generalization: There are several variants of this. The simplest is resolution reduction. this can either be done by calculating the averages of the points or by resampling. Another variant is vector to raster conversion. You can for example transform a lake with numerous islands to a raster, use a generalization operator on the raster and transform it back to vector form. This can be used for amalgamation.
- Numerical generalization: This consists of using various filters[1] on the raster to reduce the amount of information in it or to enhance specific features. A low-pass filter can make the raster more smooth, while a high-pass filter will enhance areas of rapid change.
- Numerical categorization (Image classification): Reduces the amount of information by assigning categories to points (instead of individual val-

1. A filter in this context is an operator which is applied to all the pixels in the image. A typical low-pass, or smoothing filter calculates a weighted average of the point and all its neighbours and assigns this value as the new value of the point. In a high-pass filter the value in the point itself will have a high positive weight, while the neighbours will have small negative weights.

ues). Example: Many small-scale maps do not show the terrain elevation in individual points, but rather divide the range of elevations into broad categories and show which category each point belongs to by giving it a particular colour.

- Categorical generalization: Reducing the number of categories.

2.4.4.2. Generalization of TINs

Triangular irregular networks are more difficult to generalize than rasters. The “dynamic” algorithms for building TINs build them from the top down, that is, from a single triangle covering the entire area, which is refined until it reaches the desired resolution. This refinement works by calculating the difference between the interpolated value of a point which has not yet been inserted and the interpolated value in the existing TIN for that point. It then inserts the point with maximum difference and modifies the existing TIN so that that point is included (See Section 2.2.3.4, “Modelling fields: Triangular Irregular Networks,” on page 17). This process continues until the difference between the interpolated value and the actual value is smaller than a given tolerance. This means that the entire process for creating a TIN must be run each time one wants to generalize it to a new resolution unless one already have stored one for a coarser resolution, in which case the TIN with coarser resolution can work as a start. One solution would be to store several TINs with different tolerances and switch between them. The problem with this is that it causes abrupt changes in resolution instead of gradual changes.

[FP95] describes a way to build a hierarchical TIN where each triangle is subdivided into smaller triangles. The problem is that to create a set of sub-triangles that is reasonably good (Where the triangles are not too enlongated), the lines of the original triangle have to be split, and the likelihood that some of the points used to create the smaller triangles lie exactly on the line is rather small. How they plan to solve this problem was not described in the article, which concentrated on the concept.

2.4.5. When to generalize

One of the main problems in generalization is to determine which of these operators to use and which parameters should be used with them. [MS92] describes some measures which can be used to determine when to generalize. However, several of these measures are subjective, and much of that book seems to assume that a human operator is involved in the generalization process. However, some of the spatial measurements can be calculated by a computer.

[MS92] uses the following measures:

Geometric conditions

- Congestion: Too many geographical features are represented in a limited area.
- Coalescence: Two or more features are so close together that they appear as a single feature on the output device.
- Conflict: Logical error. Example: If a river has been removed by generalization, there will be a bridge symbol on the road that crosses the river, but no corresponding river on the map.
- Complication:
- Inconsistency: Generalization operator is applied in an inconsistent manner across the map. This may or may not be desirable. (Buildings can be represented as individual building symbols in rural areas, and as "urban areas" where there are many buildings close together.)
- Imperceptibility: An area feature has grown so small that it cannot be seen or can be seen only with great difficulty.

Spatial measures:

- Density measures: How many features are there per unit of area?
- Distribution measures: Are the features uniformly distributed, randomly distributed or clustered around a few points?
- Length and sinuosity measures: How long is the line feature, and how much does it curve. Is it almost straight, or does it go through a lot of twists and turns. Are these twists and turns so small that they should be generalized away?
- Shape measures: There are some measures of shape. The simplest of these is the ratio between the area of the object and the length of its circumference. This can be used as an indication of how enlongated the object is. (The object with the largest ratio is the circle)
- Distance measures: How far is a given object from its nearest neighbour?
- Gestalt measures: Perceptual characteristics of a feature or group of features. No good measures have been developed.
- Abstract measures: Examples of abstract measures: Complexity, homogeneity, symmetry, repetition and recurrence.

2.4.6. Generalization in commercial products

There do not seem to be much capability for generalization in commercial spatial databases, at least their web-sides do not mention it. They can be made to show different maps on different scales, but these maps must be manually generalized. Commercial databases also support simple zooming, but they do not dynamically generalize the data. (If you zoom in, the map will be displayed on a larger scale, but it will not show more features than at the smaller scales. It is like looking at a paper map through a magnifying glass). If they store different maps for different scales, they can show the map which is the closest in scale to the scale the user wants.

One example of such a product is the on-line map service at www.mapquest.com. This service seems to have some set levels of generalization. At in-between scales it zooms one of the preset maps. This effect is seen by the fact that at certain scales a lot of new features appear on the map compared to a slightly smaller scale, while at others there is no change in the number of features compared to the next smaller scale.

2.5. Spatiotemporal data

Spatiotemporal data is a research field which has emerged in the last few years. It is a union between earlier research in spatial and temporal data, which used to be completely separate fields. To understand spatiotemporal data, one first needs an introduction to purely temporal data.

2.5.1. Temporal data

Temporal databases usually store several versions of the same data item which were valid at different times. These times are usually stored as intervals, so that a data item was valid from time 1 to time 2. If it is still valid, time 2 is “now”.

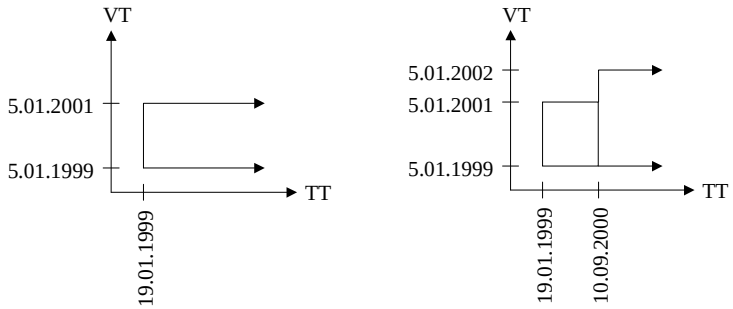
There are two different times that can be stored in databases: Transaction time and valid time. Databases which support transaction time store the time when all transactions were performed and when their results ceased to be valid in the database. Databases which support valid time store the time when the results of a transaction became/becomes true in the real world and when it stops being true. For example if Lisa the accountant was hired on 5 January 1999 on a two-year contract, but the paperwork was delayed so that it was registered in the database on 19 January, then valid time would be from 5 January 1999 to 5 January 2001, and transaction time would be from 19 January until now. This situation is shown in Figure 2.22a. In this manner future events can

be stored in the database if they are known in advance. This data can also be changed. If on 10 September 2000 the firm decided to extend Lisa's contract for a year, they could create a new transaction with transaction time 10 September 2000 which extends Lisa's employment to 5 January 2002. The old transaction (giving the final date as 5 January 2001) would not be erased, but would get a new final transaction time as 10 September 2000. This new situation is shown in Figure 2.22b. A bitemporal database will give not only a history of the domain the database models, but also a history of the database itself.

There are several types of temporal databases:

- Snapshot database: This is the traditional database, which stores the data as they are now, and discards earlier states. Another type of snapshot database is one which stores "snapshots" of the domain at various distinct times.
- Historical database: This database supports valid time.
- Bitemporal database: This database support both transaction time and valid time.

There has been some research on indexing bitemporal data. One proposed structure is a variant of the two-dimensional R-tree with transaction time and valid time as the two dimensions. This variant, which is described in [BJSS98], allows two kinds of new nodes. The first kind of node is the node in which one of the dimensions has the value "now", that is, it is continually expanded as time passes, since the objects within expand. (For each new day, it is known that they were valid for one more day.) The second class is the "staircase" shaped regions. These arise when both transaction time and valid time is "until now". Thus, each day one knows that both the transaction and its effects were valid for another day.



a) Before change **b)** After change

Figure 2.22 *Bitemporal graphs*

2.5.2. Combining space and time

In the last few years, scientists have been writing about databases which can store both spatial and temporal information on the same objects. This is important in applications such as meteorology, where the changes are of great interest. A survey of this work can be found in [AR99]

There are two main types of queries a user might want to ask in a spatiotemporal database. One is a snapshot view of the database or parts of it at a given point in time. The other is how an object or group of objects has changed over time. This would have to be displayed as a kind of "movie" in which the events are displayed in sequence. If this "movie" is supposed to run very fast relative to the number of events in the database, temporal generalization might be required.

According to [Peu99], there are four types of events:

- Continuous: the object is slowly, but steadily changing in a certain way. (Example: deserts are increasing in size)
- Majorative: This event happens most of the time. (Example: The sun shines in Sahara)
- Sporadic: This event happens some times. (Example: El Niño)
- Unique: This event only happens once (Example: The opening of the channel tunnel)

The same article lists a number of data models which have been proposed by researchers in spatiotemporal databases. None of these have been implemented in a commercial system yet. No spatiotemporal commercial systems exist although some temporal extensions exist. Oracle, for instance, has a temporal extension to their database.

- Snapshot model: In this model "snapshots" of the world at different times are stored in the database as layers. This method is rather primitive, but is good when the data is updated relatively infrequently. The maps in the 1:50000 series from Statens Kartverk in Norway are updated once every 20 years, and would therefore be suited to this form of storage. The problem with this approach is that a lot of data is stored redundantly. If a feature has not changed in the last 20 years, it will still be stored in both layers.
- Temporal grid: This is a variant of the raster in which a linked list of versions are stored in each cell instead of a single value. This list contains the time when the change occurred, so that the correct value can be chosen.

- Amendment vectors: When a line changes, for instance when a river changes its course, a special amendment vector is stored along with that line indicating its new course. This line also stores the time in which the change became valid.
- Object oriented: Object may have links to older versions of themselves, or the object itself might store various versions of itself.
- Temporal vector: This type of database stores an “original state” and a time vector. This vector contains all the events which have occurred in the database and what they changed. This representation is good for answering change-based queries, but is not good at answering queries about the current state.

2.6. Uncertainty in spatiotemporal data

This section describes several different methods for modelling uncertainty and vagueness in spatial and spatiotemporal data.

In [DMSW01], an ontology of different kinds of uncertainty is defined. The hierarchy of forms of uncertainty, or imperfection, is shown in Figure 2.23. Imperfection is considered to be the general form of uncertainty. Error is when measurements do not reflect reality. Imprecision is when measurements are lacking in specificity or are incomplete. [DMSW01] considers vagueness[1], to be a subcategory of imprecision. The basic goal of this thesis is representing uncertainty in the position or extent of an object, regardless of the source of that uncertainty. In the rest of this thesis, uncertainty therefore means either measurement error or imprecision due to incomplete knowledge, but does not cover vagueness. This definition of uncertainty is shown by the dashed box in Figure 2.23.

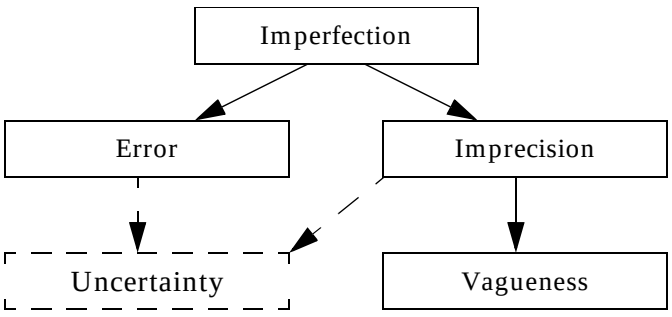


Figure 2.23 Hierarchy of the types of imperfection

1. An example of a vague statement would be that Bergen is in the south of Norway, because the south of Norway is not clearly defined.

2.6.1. Raster models

One approach to modelling uncertainty is to use a raster. In such a model, any spatial object is represented by a raster where each cell contains the probability that the cell is a member of the object. Examples of such models are presented in [Alt94] and [Low94]. These models represent uncertain regions. Figure 2.24 shows an uncertain region represented in the model from [Alt94]. The advantage of a raster model is that it can store probabilities explicitly. You do not have to compute them when you need them. Raster models can also store the evaluations of an arbitrarily complex probability function. The disadvantage is that raster models require much more space than vector models.

0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0
0.0	0.0	0.0	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.3	0.7	0.9	0.8	0.4	0.1
0.0	0.2	0.7	1.0	1.0	1.0	0.7	0.2
0.1	0.4	0.8	1.0	1.0	0.8	0.3	0.1
0.1	0.7	1.0	1.0	0.9	0.3	0.0	0.0
0.0	0.5	0.8	0.6	0.5	0.0	0.0	0.0
0.0	0.1	0.2	0.1	0.0	0.0	0.0	0.0

Figure 2.24 *Raster model of an uncertain region*

[HG95] discusses how to handle errors in digital elevation maps. They use the example that they need to know which parts of the land will be flooded by a new dam. The dam will be 350m above sea level, so all the land behind it that is lower than that will be flooded. However, the digital elevation data contains errors. According to [HG95], the best way to handle this situation is to start with a height raster and for each raster cell generate the probability that it is below 350m using a normal probability distribution. This can then be visualized with a gray-scale or colour-scale map of the region in which only the region in which there is uncertainty of 2.5 % or greater is shown. [HG95] also describes epsilon bands, which in this example would be to generate the 340m and 360m contour and say that the land between them will maybe be flooded.

[CMB97] and [CM99a] describes a way of extracting fuzzy objects from observations. In [CMB97], methods for assigning raster cells to various objects are described. These methods use a combination of fuzzy sets and probability theory. The model that they describe is a raster model because fuzzy membership values are stored in each cell. However, they also group the cells into ob-

Figure 2.25 CF and FC objects

jects according to different criteria. A Crisp-Fuzzy (CF) object is an object with a crisp border, but where the interior of the object may be fuzzy. When generating objects from field data, each raster cell is assigned to the object in which it has the largest fuzzy membership. A Fuzzy-Crisp (FC) object is an object with a fuzzy boundary but a crisp interior. FC objects may overlap while CF objects may not. Figure 2.25 illustrates these two types of objects. CF objects cannot overlap but may have no membership values of 1.0. Therefore each raster cell is assigned to one and only one object when making CF objects. FC objects must have some value which is 1.0 (the crisp interior), but may overlap each other. An object that overlaps other objects and has no 1.0 fuzzy membership values is a FF (Fuzzy-Fuzzy) object, while a normal crisp object is a CC (Crisp-Crisp) object.

[CM99b] describes a temporal extension to this model. This temporal extension is essentially a snapshot model from Section 2.5.2 in which each snapshot uses the model from [CM99a]. In addition, corresponding objects in different snapshots are identified and labelled as different versions of the same object. Fuzzy overlap is used as the criterion for deciding which objects in one snapshot corresponds to which objects in the next snapshot.

[Wor98] uses rough sets to define the outer and inner boundaries of possibly imprecise spatial objects. [Wor98] defines resolution objects that are partitions on the underlying space, and shows how to convert objects from one resolution to another. This process may introduce imprecision even if the original representation was precise, because the object may only partially overlap one of the new partition parts.

2.6.2. Abstract models using broad boundaries

Several abstract models for uncertain regions employing broad boundaries have been proposed. Abstract in this context means that they base themselves on point set theory rather than a model that can be represented in a computer. A broad boundary is a boundary that has an area and is not just a line. A region with a broad boundary is often represented as two regions. An outer region indicates where the region might possibly be, while an inner re-

gion marks where the region certainly is. An uncertain region with a broad boundary is shown in Figure 2.26. The grey area in this figure is the broad boundary.

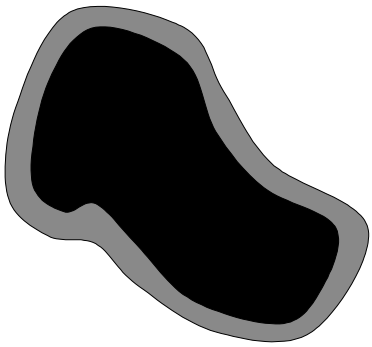


Figure 2.26 *Region with broad boundary*

Two examples of such models are the broad boundary model from [CF96] and the egg-yolk model from [CG96]. These models are used to deduce which topological relationships that might possibly exist between regions with broad boundaries. [CF96] describes 44 different relationships, and [CG96] describes 46 different relationships. For comparison, there are only nine such relationships for regions without uncertainty.

[MC89] describes a way to model uncertainty in the location of the boundary of a region that uses probabilistic error bands. This means that on each side of the estimated border there is an area with a certain width in which the border can be. Additionally, the probability that a point p is inside the area is a function of the distance from the estimated border to p.

[ES97] describes another model of this type for uncertain regions. This model defines a vague region as two regions, a kernel region and a boundary region. The boundary region represents the broad boundary. [ES97] defines the meaning of normal set operations like *Union* and *Intersection* for regions with broad boundaries. The paper uses a three-valued logic[1] for predicates.

2.6.3. Abstract models using fuzzy sets

Some abstract models for uncertainty in spatial data employing fuzzy sets have also been proposed. In Chapter 8 of [Ren99], a model for vague spatiotemporal regions using fuzzy sets is proposed. This model basically states that a fuzzy spatiotemporal region is a function from space and time to [0,1]. It also introduces an operation which can check how much the region has grown or

1. The values used are 1 (True), ? (Maybe) and 0 (False).

shrunk between two time instants. An example of such a fuzzy region is given in Figure 2.27. In this figure, darker colours indicate higher fuzzy membership values.

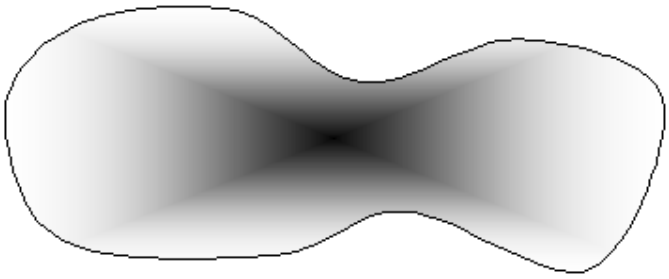


Figure 2.27 *Fuzzy region*

[WH96] describes a model for fuzzy boundaries between regions in which the fuzzy membership function indicates how sharp the boundary is. A membership of 1.0 indicates a crisp boundary. [WH96] also describes several ways of computing this membership function including taking the derivative of the characteristic under consideration and looking at how different the adjacent regions are in that characteristic.

[Sch99] contains a model for vague points, lines and regions using fuzzy sets. This model defines a region as a function over the plane. A vague line is defined as a function over a crisp line that returns a number between 0 and 1. This number represents the fuzzy membership of that point in the line. These vague lines have a crisp[1] position in space. [Sch99] proposes two models for vague points: one of which has a vague point as a point with a fuzzy membership value, and in the second, the vague point has a vague position. A vague position is modelled as a function over the plane that has the value of 1.0 where the point is expected to be and lower values for other points.

2.6.4. Discrete vector models

Vector models using broad boundaries also exist. [Sch96] proposes such a model for uncertain regions. This model is based on the ROSE algebra presented in [GS95]. Like the abstract models with broad boundaries, an uncertain region in this model consists of two crisp regions, one inside the other. An example of such an uncertain region is given in Figure 2.28.

An example of a vector model for uncertainty in moving point data is presented in [PJ99]. That paper studies the uncertainty that arises from the sam-

1. As opposed to uncertain.

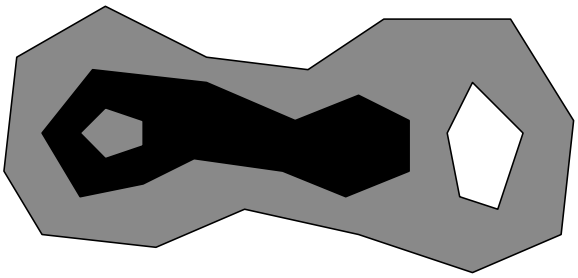


Figure 2.28 *Discrete vector model of an uncertain region*

pling rate itself. If an object could possibly move as far as 150m between two sample times, and two actual samples show a movement of 100m, the object could have been anywhere in an elliptical area between the two samples. [PJ99] describes a model in which the trajectory of the moving point is stored as a set of line segments along with the information needed to reconstruct the elliptical area in which the point could have been in between.

2.6.5. Temporal uncertainty

Uncertainty has also been studied in the temporal database community. One of these efforts is described in [DS98]. This paper contains a description of how uncertainty may be modelled in a temporal database, as well as the extensions to SQL that would be necessary. The model employs discrete time, that is the time line is divided into units called chronons. These chronons are the smallest units of time that the database will store. Probability distributions are stored as probability mass functions which give the probability that a particular event happened at particular chronons. Figure 2.29 shows an uncertain time instant in this model. The grey bars represent one chronon, and their height give the probability that the time instant is “located” in that particular chronon.

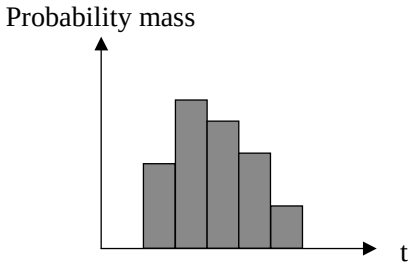


Figure 2.29 *Uncertain time instant*

[DS98] also presents a way to store the results of probability mass functions, which can be used for uncertain time instants spanning different numbers of chronons.

2.6.6. Uncertainty in future projections

Some researchers have also studied the uncertainty that results from attempting to project current knowledge to future situations or to project past knowledge to the current situation. One example is an airline control tower which gets to know the position, speed and direction of travel of an aeroplane at certain times. Then the computers in the control tower must project the currently known positions and speeds of the aeroplanes into the future to detect possible collisions.

[WXCJ98] describes a way to handle the uncertainty that occurs in such cases. In their model, an object is expected to send an update on its position, speed and direction of travel when the difference between the actual position and the interpolated position in the database reaches a certain limit. Figure 2.30 shows a moving point with registered locations at various time instants. There has been no update since *t3*. Therefore, the current position of the point is uncertain. It can be anywhere in the grey area. The dashed line represents the expected path of the point.

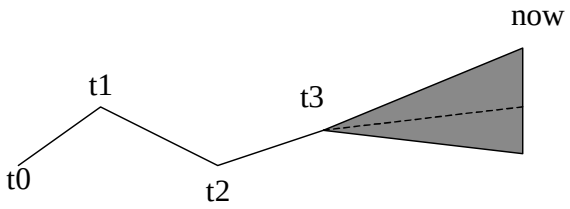


Figure 2.30 *Uncertainty in future projections*

Chapter 3
An Abstract Model for Uncertainty in Spatiotemporal Data

Due to lack of accurate measurements, or rapid changes in time, spatial and spatiotemporal data are often uncertain. This chapter presents a new abstract model for uncertain spatial and spatiotemporal information. The model is based on the principle that one knows that the uncertain object, regardless of type, must be within a certain area. This is to the authors' knowledge the first attempt to create a general type system for uncertainty with spatial data. Individual uncertain types have been modelled before, but no previous work has studied points, lines and regions and used the same principles to model all three. It also seems to be the first model to handle temporal as well as spatial uncertainty. This chapter contains mathematical definitions of uncertain points, lines, regions and temporal versions of these. The chapter also contains definitions of relevant operations on these types. These operations are also evaluated for their usefulness with regard to uncertain data.

3.1. Introduction

Databases which store information about geographic objects are becoming increasingly common in modern society. However, many forms of spatial data cannot be measured exactly, or they may vary with time in such a manner that one cannot know exactly where the spatial object is at any given time. Examples of the former are soil types or geological formations, which may change

abruptly without any sign of this on the surface. This makes it difficult to measure them exactly. An example of the latter is given below:

> **Example 1:** A lake is used as a reservoir for a hydroelectric power plant. Because of differences in energy demand and precipitation in the area, the water level, and thus the extent of the lake may vary considerably. There may also be small islands that are submerged when the water level is high.

Although one could store the exact size of the lake at any time by taking measurements frequently enough, this would be costly both in terms of manpower (taking the measurements) and space. A better solution might be to store the lake in a manner that indicates that it is uncertain. This uncertainty includes both position and the exact shape of the object.

Models for static uncertain regions exist already, and are well documented. See Section 3.2 for examples. However, other types of spatial data may also be uncertain, and the uncertainty may vary in time. There may also be uncertainty in the timing of events. Let us consider two further examples to illustrate this:

> **Example 2:** If one is monitoring an animal with a radio transmitter, and the animal wanders outside the coverage of the radio receivers, one probably only has a vague idea where the animal is. One might only know that it is within a certain area.

> **Example 3:** A new species of animal is observed in an area. One then knows that this species was there at that time, and one might know that it was not there five years ago, but one does not know when in this interval the species first appeared.

To the authors' knowledge, no general type system for spatial or spatiotemporal uncertainty exists. Although there are some definitions of individual types, the authors know of no general framework for all the types.

It is common to distinguish between two types of spatial indeterminacy: vagueness and uncertainty. Vagueness is uncertainty in classification. Asked if a particular person is tall or not, different people will answer both yes and no. One example of this from geographical data is the fact that geographic features may change gradually rather than abruptly. An example of this is an animal habitat, which may contain a core where the animals are frequently found and a large region outside this where individual animals occasionally may be found. Uncertainty, on the other hand, is the uncertainty in position or shape mentioned earlier. This thesis will just consider uncertainty.

In this chapter, we will attempt to create a set of data types and operations for uncertain data, building on earlier work in spatiotemporal databases and models for vague and uncertain data. Three types of uncertainty will be studied: positional uncertainty, shape uncertainty and existence uncertainty. Positional uncertainty is when one does not quite know where an object is, either in space or in time. Shape uncertainty is when one does not know the exact shape of the object. Existence uncertainty is when one does not know whether an object exists or not.

3.2. Related work

There are several different types of models for spatial data. For spatiotemporal data, [EGSV98] describes two modelling levels, abstract and discrete. Discrete models for spatiotemporal data can be directly implemented and are based on discrete representations such as vector or raster models. Abstract models are higher-level and usually model spatiotemporal data with point sets. In many abstract models, such as the one described in [GBE+00], lines and regions are modelled as infinite point sets in the Euclidean plane. This makes the model simpler, and may provide ideas for query operation designs that might be missed if one immediately went to the discrete level.

Abstract models also usually contain some rules to ensure that it is possible to store the data, although discrete models contain a lot more such rules. Three discrete versions of this abstract model are presented in chapter 4.

One early model for uncertain points and lines is presented in [Dut92]. In this model, a point is represented as a central point with a circular deviation and a Gaussian distribution function over this area. A line is represented as a series of such points. The line segments between the points are represented by the union of the straight line segments going between all possible positions of the two points. The paper then shows that such a line will look like a knotted rope, as the greatest variance is in the points themselves, and the least variance is in the centre of the lines between the points.

The egg-yolk model described in [CG96] models an uncertain region with only one face as two regions, one inside the other. The inner region is referred to as the ‘yolk’ and the outer region as the ‘white’ in the egg. This representation is then used to find a lot of different topological relations between uncertain regions, each consisting of only one component. A very similar model is described in [CF96], which uses the term “broad boundary” to describe the ‘white’ in the egg. Most of the operations in this article are the same as in [CG96].

One problem with these types of models is that they cannot model certain kinds of uncertainty. For example, if there is a lake with a dam, the water level in this lake might vary significantly, thereby producing uncertainty as to the size of the lake. Such a lake may also have low islands which are submerged if the water level is high enough. Such uncertain holes cannot be modelled by these models.

Models based on fuzzy sets have been frequently used to model vague regions. Fuzzy sets [Zad65] are sets in which the membership of any individual point in the set is not either yes or no, but rather a number between 0 and 1. Many of these models, such as the discrete models presented in [LAB96] and [Low94][1], use rasters to represent the fuzzy sets. The models described in [Sch99] and [ES97] represent another type of fuzzy set model, because these, like the abstract models for crisp[2] objects, use infinite point sets. The most comprehensive model for vague data using fuzzy sets is the one presented in [Sch99], which models all the standard spatial types (points, lines and regions) using fuzzy sets. Although these fuzzy models cannot be used as they are to model positional uncertainty, some of the ideas from them may be adopted.

Another possible model for uncertain regions, regardless of the type of uncertainty, is the vector-based discrete model presented in [Sch96]. This model bases itself on two boundaries, like the egg-yolk models. However, unlike the egg-yolk approach, it can also model holes and multicomponent regions.

Uncertainty has also been studied in the temporal database community. [DS98] describes a model and an extension to SQL for handling temporal uncertainty. They use a probability function to indicate the likelihood that the event occurred at a given time unit.

There has been an effort to create a comprehensive type system for different kinds of spatial databases. [GBE+00] describes such a type system for spatiotemporal databases. [Sch99] describes a similar kind of model for vague spatial data.

3.3. Basis for the new model

The new model presented in this chapter takes ideas from several of the models described earlier. The model in [Dut92] is adequate for modelling digitization error, but not adequate for some other applications. One example of

1. [Low94] models indeterminacy, regardless of type, with fuzzy sets.
2. Crisp: Usual meaning is the opposite of vague.

this is Example 2 from the introduction. This example cannot be modelled by the one in [Dut92] because the region may have an arbitrary shape. However, the concept that the point is known to be located within a region and has a certain probability distribution can be used in the new model. Another example is that the approach suggested in [Dut92] cannot model uncertainty about the length of a line. As with points, the concept of a line with a probability distribution function is useful for our work.

[Sch99] describes an abstract model for vague spatial data. The region model in that paper may be used as a basis for a model for uncertain regions. [Sch99] models a vague region as a fuzzy set where places which are certainly members of the region have values of 1 and regions which are only partially members have values between 0 and 1. In the same way, an uncertain region may be modelled as a probability function where points which are certainly members have a value of 1 and points for which membership is uncertain have values between 0 and 1.

Schneider's model as given in [Sch99] for a vague line or vague point, however, is not so useful for uncertain data. A vague line is a line with a crisp position but uncertain membership. In Schneider's model, this uncertain membership is indicated by a function which gives values between 0 and 1 for each member of the crisp line. An uncertain line has uncertainty about position as well, which means that a different type of model must be used. However, an uncertain line may also have uncertainty about whether it exists or not. This existence uncertainty may be modelled in the same way as vagueness. The difference between vague and uncertain points is the same as for lines.

An important difference between our new model and Schneider's is that his model uses somewhat different mathematics. While Schneider uses fuzzy sets, our new model uses probability theory. This is both because uncertainty is best modelled by probabilities, and because the probabilities for uncertain points and lines must be modelled by probability density functions. The authors do not know of a similar concept in fuzzy set theory.

[GBE+00] describes a complete model for crisp spatiotemporal data. That model is also a basis for some of this chapter. In particular, many of the operations described come from [GBE+00]. Also, some of the ideas for transforming a spatial model into a spatiotemporal model are reused. For operations, this chapter describes how useful that model's operations will be in the uncertain case, and whether it can be used in new ways or on new types of data. However, the chapter also describes new operations which are not from [GBE+00].

Some of the types from [GBE+00] are also used as building blocks for the types presented here. Therefore, a brief description of these types is given now. The basic type for points is A_{points}, which is a finite set of points. The type for a single point is A_{point}. A line in [GBE+00] (of type A_{line}) is defined as a set of curves forming a graph. A curve is defined by a function from a variable *t*, which is between 0 and 1, to the *X*-*Y* plane. Curves cannot intersect with themselves. The carrier set of this type is called A_{curve}.A region [GBE+00] is an infinite set of points in the plane with the condition that there may be no singleton points or lines. That is, the region must be a valid result of a regularized set operation. A region may consist of a finite set of disjoint components, or faces. These again can have a finite number of holes. The carrier set of faces is called A_{face}, while the carrier set of regions is A_{region}.

Table 3.1 *Carrier sets for the data types from [GBE+00]*

Individual	Set
A_{point}	A_{points}
A_{curve}	A_{line}
A_{face}	A_{region}

3.4. Data types for uncertain spatial information

This section describes a set of data types for modelling uncertain spatial information. The first subsection will describe how to model the basic datatypes such as numbers. The other subsections will describe uncertain points, lines and regions. All the types will be defined by their carrier sets.

In the following descriptions, the word “crisp” will be used as the opposite of uncertain.

To define the data types that follow, the operation *support* is needed. This operation comes from fuzzy set theory, but has a slightly wider application here. In this thesis, *support* is defined as follows for any function $f: z \rightarrow \Re$:

$$Support(f) \equiv \{z | f(z) > 0\}$$

The *z* in this formula is a member of whatever type or set of types the function *f* accepts as input values. This means that support is defined for all uncertain types, whether they are spatial or not. A more complete definition and discussion of this operation can be found in Section 3.6.1.3.

All the uncertain data types defined in this thesis rely on probabilities or probability density functions. The properties of these are defined by the following functions:

- Probability Density:
  $ProbDens(P) \equiv (\forall x{:}P(x) \geq 0) \wedge \int_x P(x) \leq 1$
- Spatial Probability Density:
  $SProbDens(P) \equiv (\forall x \forall y{:}P(x, y) \geq 0) \wedge \int_x \int_y P(x, y) \leq 1$
- Probability Function:
  $ProbFunc(P) \equiv \forall x{:}(P(x) \geq 0 \wedge P(x) \leq 1)$
- Spatial Probability Function:
  $SProbFunc(P) \equiv \forall x \forall y{:}(P(x, y) \geq 0 \wedge P(x, y) \leq 1)$

3.4.1. Base types

An uncertain number can easily be modelled by a probability distribution function. For a real number, this function would have to be defined as a probability density function, whereas for integers, it might be just a collection of probabilities for the number having particular values.

Definition 1: An uncertain number is defined as follows.

$$A_{UNumber} \equiv \{NP(x) | ProbDens(NP) \wedge ((PieceCont(NP) \wedge Support(NP) \in A_{Range(number)}) \vee (DiracDelta(NP) \wedge Support(NP) \in A_{number}))\}$$

PieceCont(*F*) is true if the function *F* is piecewise continuous. *DiracDelta*(*F*) is true if *F* is a dirac delta function.

In this chapter, both uncertain real numbers and uncertain time instants are modelled with the $A_{UNumber}$ type. This simplifies the definitions of operations which are applicable to both these types.

Many queries in spatial databases return Boolean values for data without uncertainty. Because a single Boolean value cannot indicate uncertainty, different ways of answering these queries must be found. The simplest alternative is to introduce a third “Boolean” value, *Maybe*, which indicates that the answer is not known. This is described in [ES97]. However, many times a better approach is to return the probability of the answer being true.

Both of these types of uncertain Boolean values will be used in different cases. If it is possible and meaningful to compute the probability of the answer

being true, then that probability will be returned. If this is not possible or not meaningful, the *Maybe* value will be used to indicate uncertainty.

These two forms of Boolean values are treated as two different types in this thesis. The uncertain Boolean is the version with three values, and the other is called a probability. A third type which is useful for uncertain data is a type which indicates to which degree a statement is true. Some operations may return a degree of truthfulness which cannot be interpreted as a probability.

$$A_{UBool} \equiv \{False, Maybe, True\}$$

$$A_{Prob} \equiv [0, 1]$$

$$A_{Degree} \equiv [0, 1]$$

3.4.2. Uncertain points

An uncertain point is a point for which an exact position is not known. However, one usually knows that the point is within a certain area. One may also know in which parts of this area the point is most likely to be. An uncertain point is therefore defined as a probability density function P(x, y) on the plane. The support of this function is the area in which the point may be. To be able to store the function P(x, y) in a computer, it must be piecewise continuous. The probability that the uncertain point exists at all is the double integral of P(x, y) over the plane. To be able to model crisp points, P(x, y) must be allowed to be a dirac delta function.

Definition 2: An uncertain point is defined as follows.

$$\begin{aligned} A_{UPoint} \equiv \{ & PP(x, y) | SProbDens(PP) \\ & \wedge ((Support(PP) \in A_{region} \wedge PieceCont(PP)) \\ & \vee (Support(PP) \in A_{point} \wedge DiracDelta(PP)))\} \end{aligned}$$

A possible uncertain point is shown in Figure 3.1a. Figures 3.1b and 3.1c show views of the X and Y directions. The central spikes indicate the expected value of the points, and that a single point is being modelled, one just does not know where it is. The numbers indicate that the integral must be between 0 and 1. The thick bar underneath indicates the area of uncertainty.

The model described here can model Example 2 because it allows the point to be inside an arbitrarily shaped region, and not just a circle like [Dut92]. It also enables a point to be modelled where its existence is not certain.

One problem with this model is how to determine the probability density function so that the double integral of it over the universe becomes 1 if the point is certain to exist.

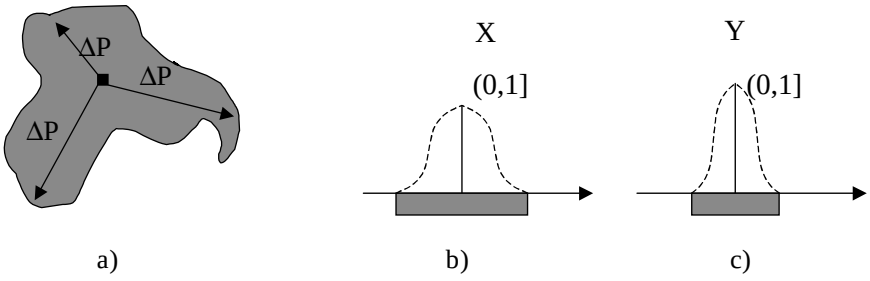


Figure 3.1 *Uncertain point*

Definition 3: The *uncertain points* set type is defined as follows.

$$A_{UPoints} \equiv \{UP \subseteq A_{UPoint} | Finite(UP)\}$$

The *Finite* function returns *True* if the set contains a finite number of elements and *False* otherwise.

3.4.3. Uncertain lines

The line type as defined in [GBE+00] is a set of curves where each member is a simple curve. The first step in developing a model for an uncertain line is therefore to create a model for a curve. An uncertain curve is a curve for which the exact shape, position or length is not known, but it is known in which area the curve must be. An example of an uncertain curve is shown in Figure 3.2a. It may also be known where in this area the curve is most likely to be. The dotted line in Figure 3.2a exemplifies this.

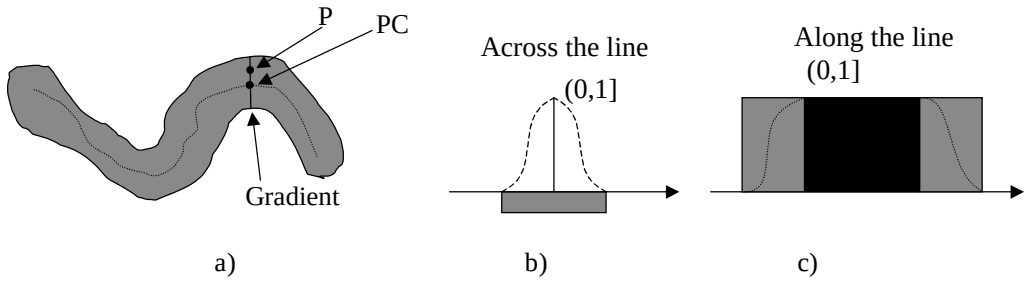


Figure 3.2 *Uncertain curve*

When seen along a line crossing it, a crisp curve would look like a point, or a set of points in the case of multiple crossings. When seen along the same line, an uncertain curve should be a probability density function indicating where the curve is most likely to cross. Such a function is shown in Figure 3.2b.

This function may apply along the line marked "Gradient" in Figure 3.2a. Formally this line and its probability density function may be defined as follows:

$$A_{gradient} \equiv \{(gc, fg) | gc \in A_{curve} \wedge (\forall (p \in gc):(fg:p \rightarrow \Re)) \wedge ProbDens(fg)\}$$

When seen along its length, the uncertain curve has a probability of existing at each point. In Figure 3.2c, one common example of such a probability function is shown. In this example, there is uncertainty about the length of the line. This means that the line is certain to exist in the middle, and the probability of the line existing becomes lower the closer one comes to the ends.

One way of modelling this probability is that the uncertain line has a central line with a probability function associated with it. This probability function should not have areas in the middle where it is 0, because a curve with such a function is really two curves and not one, and should therefore be modelled as two curves. Such an illegal function is shown in Figure 3.3.

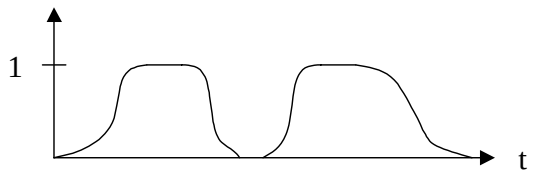


Figure 3.3 *Illegal probability function for uncertain curve*

If there is uncertainty as to the number of curves, this may be modelled by a function which is less than 1 in a period between two places in which it is 1. This is shown in Figure 3.4.

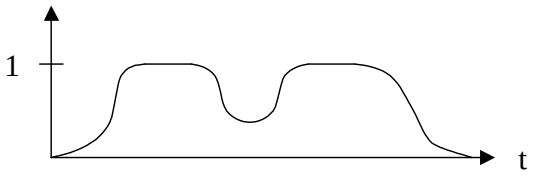


Figure 3.4 *Probability function indicating uncertainty about the number of curves*

This property of a function may be expressed mathematically as follows:

$$NoDip(f) \equiv (\forall x \forall y \forall z:((f(x) > 0) \wedge (f(z) > 0) \wedge x < y < z) \rightarrow f(y) > 0)$$

An uncertain line may be defined as a central line and a set of gradient lines. This set of gradient lines models the positional and shape uncertainty of the line. For each point of the central line there should be one and only one gradient line crossing it. The expected values of the probability functions of all the gradients should be somewhere on the central line. This is ensured by the following three conditions:

- The gradients do not share points or parts:
  $NoCross(G \subseteq A_{gradient}) \equiv (\forall(g_1, g_2 \in G):(g_1 \cap g_2 \neq \emptyset) \rightarrow (g_1 = g_2))$
- For each point *p* on the curve, there is a gradient. The expected value of this gradient is *p*:
  $ExpectedCurve(ec \in A_{Curve}, G \subseteq A_{gradient}) \equiv (\forall(p \in ec)\exists(g \in G):E(g.fg) = p)$
- The expected value of all the gradient lines are on the central line:
  $CurveExpected(ec \in A_{Curve}, G \subseteq A_{gradient}) \equiv (\forall(g \in G)\exists(p \in ec):E(g.fg) = p)$

For all of these functions, E(x) is the expected value for a probability density function.

To ensure that the type is implementable, the probability density values of points that are close to one another should have similar values. To ensure this, we use the condition that all iso-lines of probability must be continuous. This means that for all possible probability density values, the set of points formed from the points along all the gradient lines that have this probability density should form either a continuous line along the central curve or a set of continuous cycles. The set of points from all the gradient lines that have a given probability density value is returned by the *ISet* function, which is defined as follows:

$$ISet(i > 0, G \subseteq A_{gradient}) \equiv$$
$$\{x | \exists(g \in G):(x \in g.gc \wedge g.fg(x) = i)\}$$

To ensure that the iso-lines are continuous cycles, the following condition is used:

$$ContIso(ec \in A_{Curve}, G \subseteq A_{gradient}) \equiv$$
$$(\forall i:((Iset(i, G) \subseteq Points(ec)) \vee$$
$$\exists(C \subseteq A_{cycle}):Finite(C) \wedge points(C) = Iset(i, G)))$$

The function *points(C)* returns a set containing all the points which are parts of at least one cycle in the set of cycles.

To compute the area in which the uncertain line may be, one can take the union of all the gradient lines. The following condition ensures that the union of all the gradient lines forms a crisp face:

$$FormFace(G \subseteq A_{gradient}) \equiv$$
$$(\{x | \exists(g \in G): x \in g.gc\} \in A_{Face})$$

Definition 4: An uncertain curve is defined as follows.

$$A_{UCurve} \equiv \{(ec, fe, G) | ec \in A_{curve} \wedge G \subseteq A_{gradient} \wedge$$
$$\forall(p \in ec): (fe: p \rightarrow A_{Prob}) \wedge$$
$$NoDip(fe) \wedge NoCross(G) \wedge$$
$$ExpectedCurve(ec, G) \wedge CurveExpected(ec, G) \wedge$$
$$ContIso(ec, G) \wedge FormFace(G)\}$$

This type definition is quite complex, and is the most complex type of the three main ones. The reason for this is that a point is a probability density function, a region is a probability function where each point has a probability of being in the region. A line, however, is a little of both, as shown in Figure 3.2.

Note that this definition of a curve does not allow a curve that is partially crisp and partially uncertain. This is because there would be a point where the uncertain area ends and the crisp area begins where the probability density function of the gradients rises until it becomes infinite. In this place, some of the iso-lines would not be cycles as they will end right next to the point where the line becomes crisp.

Both points and regions are defined as functions over the plane. To make it simpler to define operations which are common to all uncertain spatial types, a view of the uncertain curve as a function over the plane is therefore also given:

Computational definition. Uncertain curve (*C*) as function over the plane:

$$C.f(x, y) = gl.fg(x, y) \cdot C.fc(cp)$$

In this function, *gl* is the member of *C.G* on which the point (x,y) lies and *cp* is the point at which *gl* crosses *C.ec*.

The line type is a set of curves for the same reasons as given for points.

Definition 5: The uncertain line is defined as a set of uncertain curves.

$$A_{ULine} \equiv \{UC \subseteq A_{UCurve} | Finite(UC) \wedge$$
$$\forall(ac \in UC)\forall(bc \in UC): (ac \neq bc \rightarrow \neg Cross(ac, bc))\}$$

The requirement that two curves should not cross is there to ensure the uniqueness of the representation. If two curves that cross are added to the same set, they must be divided so that all four get the crossing as their end points. The *Cross* operator is defined in Section 3.6.1.2.

The problem that a curve cannot be partially uncertain can be solved by defining such a curve as a line with several curves, some uncertain and some crisp. The crisp curve is defined by having all its gradient lines have length 0 and their probability functions being dirac delta functions.

One problem with this model for lines is that it involves fairly complex mathematics, such as finding gradients of a function. Also, some operations, such as testing whether two lines cross each other, are much more complex in this model than in models for crisp or vague lines. This complexity exists because the uncertain curve is neither a simple probability density like for the uncertain point nor a simple probability function for each point like in an uncertain face. The line type is therefore the most complex of the three basic spatial types.

3.4.4. Uncertain regions

An uncertain region is a set of uncertain faces. An uncertain face is one where the location of the boundary or even the existence of the face itself is uncertain. This may be modelled as a probability function P(x,y) which gives the probability that the point (x,y) belongs to the face. *Support*(P) must be a valid crisp face. Additionally, an alpha-cut operation must yield a valid crisp region for all input values between 0 and 1. The alpha-cut function is defined as follows:

$$\alpha cut(f, i) = \{z | f(z) > i\}$$

A more complete definition may be found in Section 3.6.1.3. Note that the *Support* operation is the same as an alpha-cut with i=0.

Definition 6: An uncertain face is defined as follows.

$$\begin{aligned} &A_{UFace} = \{FP(x, y) | \\ &SProbFunc(FP) \wedge Support(FP) \in A_{face} \wedge \\ &\forall (i \in [0, 1]): \alpha cut(FP, i) \in A_{region} \wedge \\ &PieceCont(FP)\} \end{aligned}$$

This definition is used because it is very general, and gives the capability of modelling uncertain regions in which the exact number of faces is unknown. This is possible because the uncertain face can have a core which contains multiple crisp faces like the uncertain face shown in Figure 3.6. It also allows holes

which are not certain to exist (such as the submerged islands in Example 1) because there may be an area with a function value less than one inside an area with function value one.

Figure 3.5a shows an example of an uncertain face where the black area is the area in which the face is certain to exist and the grey area is the area of uncertainty. Figures 3.5b and 3.5c show views of the probability distribution along the X and Y axis.

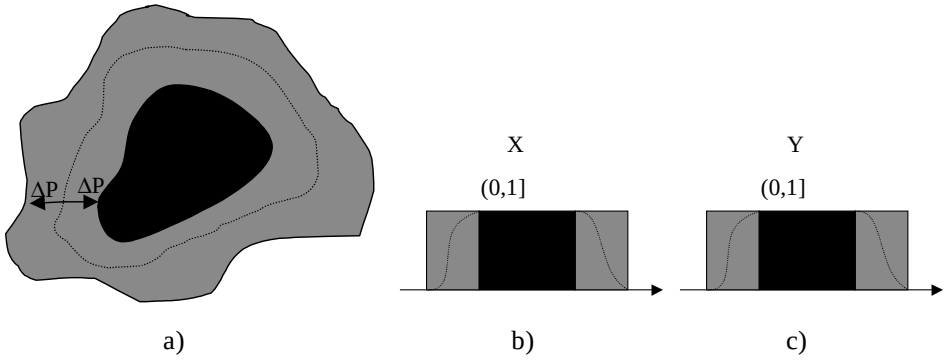


Figure 3.5 *Uncertain face*

An uncertain face is known to exist if at least one point has probability one of being a member of the face. Notice also that, unlike the crisp case, the boundary for an uncertain face is not necessarily an uncertain line. See Figure 3.6 for an example of this. In this example, there are two possible configurations, one in which there are actually two faces, and one in which there is only a single face. There may therefore be either one or two boundary lines in the actual object.

Definition 7: The uncertain region is defined as a set of uncertain faces.

$$A_{URegion} \equiv \{ UF \subseteq A_{UFace} \mid Finite(UF) \wedge \forall(af \in UF)\forall(bf \in UF):(af \neq bf \rightarrow Disjoint(af, bf))\}$$

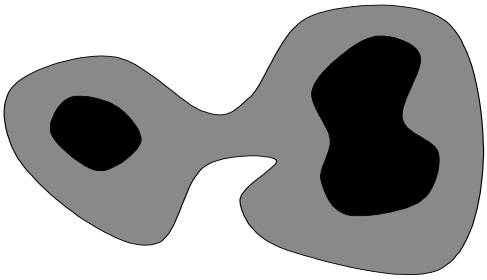


Figure 3.6 *Face where the border is not a valid uncertain line*

Disjoint for uncertain types is defined as follows:

$$Disjoint(A, B) \equiv Union(Support(A), Support(B)) = \varnothing$$

This type of model for faces and regions has the advantage that such faces and regions are well documented for the vague case using fuzzy sets in [Sch99] and [ES97]. It is also very general, capable of modelling any kind of uncertainty, even uncertainty about how many components the region really has. Error-band based models can only model uncertainty about the position and shape, not the number of components or holes such as in Example 1 above.

3.5. The time type and temporal uncertainty

Temporal uncertainty has been studied in [DS98]. However, their model is based on discrete time. To be consistent with the way the spatial dimension has been modelled, the temporal dimension should also be continuous. Therefore, the model defined here uses continuous time.

A crisp time instant can be modelled as a real number. An uncertain time instant may therefore be modelled by an uncertain real. The modelling of time intervals requires a type for a set of uncertain intervals. This type takes the A_{range} type constructor from [GBE+00] and adds uncertainty to it.

An uncertain interval ($A_{UInterval}$) is a probability function over the real number line which indicates the likelihood of the interval existing for that number. Figure 3.7 shows one example of such a function. In this example, there is uncertainty about the length of the time interval, represented by the two time intervals marked *dT*. Because an interval should be continuous, the probability function should not have areas in the middle where it is 0. Such an illegal function is shown in Figure 3.3. The interval must be defined over a particular type of number (α), such as integer or real.

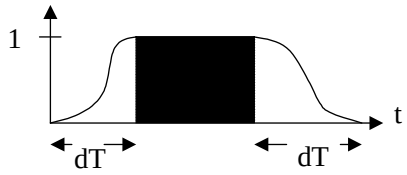


Figure 3.7 *Example of uncertain time interval*

Definition 8: An uncertain interval is defined as follows.

$$A_{UInterval(\alpha)} \equiv \{IF_{\alpha_I} \mid ProbFunc(IF_{\alpha_I}) \wedge PieceCont(IF_{\alpha_I}) \wedge Support(IF_{\alpha_I}) \in A_{Interval(\alpha)}) \wedge NoDip(IF_{\alpha_I})\}$$

$A_{Interval(\alpha)}$ is the set of all continuous intervals over the type α.

This defines a single interval, but many operations may return a set of such intervals. This requires the A_{URange} type.

Definition 9: The uncertain range is defined as a set of uncertain intervals.

$$A_{URange(\alpha)} \equiv \{UR \subseteq A_{UInterval(\alpha)} \mid Finite(UR) \wedge \forall(ai \in UR)\forall(bi \in UR):(ai \neq bi \rightarrow Disjoint(ai, bi))\}$$

To find the time instant at which the time interval starts, take the derivative of the probability function of the time interval in the *dT* intervals. Each disjoint region on the real number line in which this derivative is not zero is a time instant which bounds the time interval.

Our types for time instants and intervals are chosen because they can model both uncertainty as to the length of a time interval and uncertainty as to exactly how many time intervals there are in a set. The last can be done in the same way as modelling uncertainty as to the number of curves as shown in Figure 3.4. In this case, the set of bounding time instants described in the previous paragraph will also contain some uncertain time instants corresponding to the area of uncertainty in the middle of Figure 3.4.

3.5.1. Turning spatial types into spatiotemporal types

Types for uncertain temporal data may be derived from the non-temporal data types. If *A* is a data type, let *TA* be its temporal version. The value of *TA* in each time instant in which it exists must be a valid member of *A*. It is therefore natural to define *TA* as a function from time to *A*. Uncertainty about when an object first appeared or ceased to exist may be indicated by making the existence of the *A*'s close to the start or end uncertain.

The uncertain spatial data types in Section 3.4 may be transformed into spatiotemporal types using the "*UMoving*(α)" type constructor.

Definition 10: The *Uncertain Moving* type constructor.

$$A_{UMoving(\alpha)} \equiv \left\{ f_{\alpha_M} \middle| f_{\alpha_M} : \overline{A}_{instant} \rightarrow \overline{A}_{\alpha} \wedge PartFunc(f_{\alpha_M}) \right.$$
$$\left. \wedge Finite(Components(f_{\alpha_M})) \wedge PieceCont(f_{\alpha_M}) \right\}$$

This type constructor creates a temporal type from a non-temporal type. For instance, $A_{UMoving(A_{UPoint})}$ is the carrier set for a temporal, or moving, uncertain point. This carrier set contains all functions from time to A_{UPoint} which satisfies the above criteria.

In this definition, f_{α_M} is the function from *TA* to *A*. The first two conditions in the definition are the same as those of the *moving(*α*)* type constructor from [GBE+00], and are there to ensure implementability. *PartFunc(f)* is true if *f* is a partial function. The difference between A and $\overline{A}$ is that $\overline{A}$ does not include the empty set.

The final condition ensures that the probability values or probability densities are piecewise continuous in time as well as in space. This is necessary for implementability as well as to ensure that the return values of some operations are valid.

The function f_{α_M} is piecewise continuous if the probability or probability density values are piecewise continuous in time for all points in space.

An example of this is the moving uncertain point, $A_{Umoving(A_{UPoint})}$. This type is a function *f* from $A_{instant}$ to A_{UPoint}, that is, for any time instant in which *f* is valid, a value of type A_{UPoint} is returned. Notice that *f* takes a crisp time instant, not an uncertain one. Because this function must be piecewise continuous, there may not be any time instant where the point is at a completely different location from the time instants immediately before and after it. Piecewise continuous means that the probability density in a single crisp point for the A_{UPoint} values returned by *f* at different time instants must be piecewise continuous.

3.6. Operations on uncertain data

An important part of a set of data types is a general definition of the operations that can be applied to them. This section begins with an overview of operations on non-temporal uncertain spatial data, and then moves on to temporal uncertain data. For both sets of operations the operations described in [GBE+00] have been evaluated. The operations from [GBE+00] that the evalua-

tion showed to be most important as well as some new operations for uncertain data are described. The operations are divided into three categories, those that are applied to data with no uncertainty, but which are meaningless for uncertain data, those that can be applied to both kinds of data, and new operations for uncertain data.

The set of operations described in this section is somewhat redundant, as some of the operations can be defined in terms of other operations. However, it is often easier to use a comprehensive set of operations rather than a minimal one, because a user might not know how to put together the operations in a minimal set to get the result that he/she wants. It may also be faster to implement some of the redundant operations directly rather than as a sequence of other operations.

In this section, the letter name of the variable describes its type as given in Table 3.2. A signature of the type $S \times S \rightarrow S$ means that both the inputs must be of the same type, and the output is of the same type as the input. In a signature of the type $S \times R \rightarrow B$, the S input is neither limited to the type of the other input nor of the output.

In the semantics for the operations, the letter *R* is used for the result, *A* for the first input and *B* for the second input.

3.6.1. Operations on non-temporal uncertain data

The *Core* and *Support* operations from fuzzy set theory will be used for operations on uncertain data. These have slightly different semantics than in the vague case because of the differences between uncertainty and vagueness. *Core* is defined as follows.

- *Core*$(T \rightarrow CT)$*:* For a region or interval, this operation returns the crisp set containing all the points or values having membership 1 in *A*. For other types it returns other values. For a complete definition, see Section 3.6.1.3.

- For *Support*, see Section 3.4.

3.6.1.1. Operations on crisp data which are meaningless for uncertain data

The operations described in this subsection are listed in Table 3.3 and Table 3.4. They are useless for uncertain data because they either cannot be determined or become identical to other operations. For some of these operations one may determine whether the operation is certainly false or not. The formula for determining this is given in the table when it exists.

Table 3.2 *Type* indicators[1]

Letter	Type
Po	Point
Ps	Points
C	Curve
L	Line
F	Face
Re	Region
S	Spatial (Point, Curve or Face)
Ss	Spatial Set (Points, Line or Region)
N	Number (Real or Time Instant)
I	Interval (of Number)
Ra	Range (of Number)
NI	Non-Spatial (Number or Range)
T	Any non-spatial or spatial type
B	Boolean
Pr	Probability
D	Degree
MOV(X)	Moving(X)
CX	Crisp X

1. All these stand for uncertain data types except for CX

Equal: One cannot determine equality between two uncertain objects. Even if the two objects have exactly the same type and probability function, they are not necessarily equal, because the real objects may be different even if the uncertain representation is “equal”. For instance, if two regions both have the representation given in Figure 3.8, one of them can be bordered by line A and the other by line B. The regions produced by A and B are clearly not equal, but they can both correspond to the same uncertain region. The only way one

Table 3.3 *Operations for which a positive answer is impossible for uncertain data*

Operation	Signature	Semantics
Equal	$S \times S \rightarrow B$	Maybe: $(core(A) \cap support(B) = core(A)) \wedge (core(B) \cap support(A) = core(B))$ No otherwise
Meet	$C \times C \rightarrow B$ $I \times I \rightarrow B$	See text
Touch	$F \times F \rightarrow B$	Maybe: $(core(A) \cap core(B) = \emptyset) \wedge (support(A) \cap support(B) \neq \emptyset)$ No otherwise

can know that two uncertain objects are equal is if they are in fact the same object, with the same object identity or primary key value.

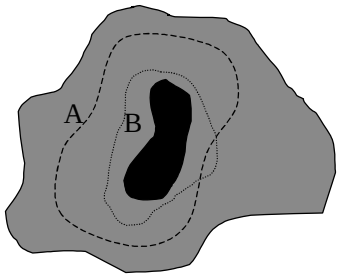


Figure 3.8 *An uncertain region and two possible "real regions"*

Meet: Unless the fact that two curves meet is explicitly stored, one cannot know for sure whether they meet or not. The two curves definitively do not meet if their supports do not overlap. Otherwise, the answer can be no better than "Maybe". Even if the end points are known and identical, the two curves may still end in different parts of the end point. This is also true of uncertain intervals.

Touch: In the uncertain case this operation determines the possibility that two faces have a common border. Even if the supports overlap and the cores do not, one cannot be sure whether they actually have common borders or the borders just cross each other. Therefore, the operation cannot return "Yes" when

there is uncertainty, unless the fact that the two faces have a common border is explicitly stored.

Table 3.4 *Operations which become identical to other operations for uncertain data*

Operation
Attached
Overlap
On_Border
In_Interior

Attached[1], *Overlap*[2], *On_border*[3], *In_interior*[4]: These operations from [GBE+00] depend on point set topology. Specifically they test for intersection with the boundary as opposed to the interior of an object. Point set topology for indeterminate data is outside the scope of this thesis. This has been studied in several other papers, such as [CG96], [CF96] and [Win00]. In the model presented in this chapter, it is impossible to say anything certain about the relationships between the boundary of two uncertain objects unless this is explicitly stored. In our model, *Attached, Overlap* and *In_Interior* become identical to *Intersect* from Section 3.6.1.2. *On_Border* has probability 0 of being true because the likelihood of the two border lines being in exactly the same place is 0.

3.6.1.2. Operations which may be used on both crisp data and uncertain data

The operations in this section are divided into five categories, depending on the types of their input and output: set operations, operations applicable to all uncertain spatial data types, operations for uncertain regions, operations for uncertain lines and projections. There are no operations that are only applicable to uncertain points and cannot also be applied to other types.

Set operations

The set operations for the *points* and *line* data types are the same as in the crisp case. Using a set operation on the individual *point*s and *curve*s does not make sense. The *point* data type is not a set. Performing a set operation on a *curve* will most likely produce an illegal value. An uncertain *region* is in essence an infinite point set where the individual points have a certain probability of

1. Does the boundary of A overlap the interior of B?
2. Do the interiors of A and B overlap?
3. Is point A on the border of region B?
4. Is point A in the interior of B?

Table 3.5 *Normal set operations applicable to all uncertain spatial data*

Operation	Signature	Semantics
Union	$Ss \times Ss \rightarrow Ss$	Points, Line: $A \cup B$ Region: $R(x, y) = A(x, y) + (1 - A(x, y)) \cdot B(x, y)$
Intersection	$Ss \times Ss \rightarrow Ss$	Points, Line: $A \cap B$ Region: $R(x, y) = A(x, y) \cdot B(x, y)$
Minus	$Ss \times Ss \rightarrow Ss$	Points, Line: $A - B$ Region: $R(x, y) = A(x, y) \cdot (1 - B(x, y))$
No_Components	$Ss \rightarrow N$	Number of elements in set
Decompose	$Ss \rightarrow \{S\}$	A set of the corresponding single-element type, each containing one component

being members. Each set operation should therefore return a set that for each point gives the probability of the operation being true. This is easiest to do by combining the probability functions of the two input sets. Probability theory has been used to arrive at the formulas given in Table 3.5. The events that a point belongs to regions A and B are considered to be independent.

The *union* operator is not defined for arguments of different types because this would produce meaningless results such as a region with a line sticking out of it. Such results are also not legal values of any of the types defined here. The *intersection* operator returns a result of the lowest dimension of the two inputs. An *intersection* between a point and a line does not make sense in the uncertain case because the probability that the two are at exactly the same place is 0. An *intersection* between either a point or a line and a region uses the same semantics as the intersection of two regions. The output type is point or line.

Because it is impossible to know exactly where the border of a region or end point of a line is, the *intersection* operation cannot produce results of a lower dimension than the lowest dimension input. Unlike the crisp case, there is no need for specialized versions of *intersect* that return these.

The *Minus* operator is not defined for arguments of different types because the results of this operation are not useful even for crisp data. For instance, a region minus a point is the region. This also applies in the uncertain case.

The *No_Components* and *Decompose* operations may be defined exactly as in the crisp case because the uncertain model contains the same types of curves

and faces as the model in [GBE+00]. For uncertain lines and regions, this means that the *number of components* operation returns the minimal number of components, and *decompose* decomposes the object into the minimum number of components. For instance, the region in Figure 3.9 has two components according to this definition, while in reality it might have three.

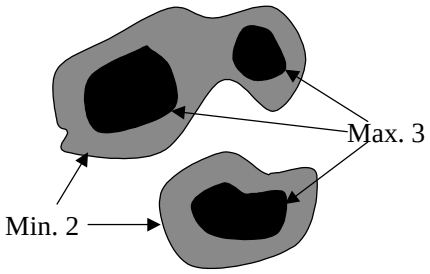


Figure 3.9 *Minimum and maximum number of components*

Other operations applicable to all uncertain spatial data types

This is a collection of operations which are not set operations but which nevertheless can be used on all uncertain spatial data types. Their semantics are defined in Table 3.6.

Area: For uncertain lines and points, it may be useful to determine the size of the area of uncertainty. Therefore the *area* operation is defined for them as well as for regions. For uncertain regions there are two alternatives to answering this question. One is to return minimum and maximum possible sizes. The problem with this is that the area operator now returns two numbers, which makes it different from the crisp case. The other alternative, which we go for, is to compute the average size using the area operator from [Sch00b]. This operator can be applied to uncertain regions by taking the integral of the function of the region over the universe.

Resemble: This operator determines how much two uncertain spatial objects resemble one another. It may be used to replace *equal* for uncertain objects. For crisp regions it is used to determine similarity in shape. The function *min*

Table 3.6 *Other operations applicable to all uncertain spatial data*

Operation	Signature	Semantics
Area	$S \rightarrow N$	Point, Curve: *area*(*Support*(A)) Face: $\int_x \int_y A(x, y) dy dx$
Resemble	$S \times S \rightarrow A_{Degree}$	$(area(min(A, B)))/(area(max(A, B)))$

Table 3.7 *Operations for uncertain regions*

Operation	Signature	Semantics
Inside	$S \times F \rightarrow B$	See text
Intersect	$S \times F \rightarrow A_{Prob}$	$Existence(A \cap B)$
Negation	$Re \rightarrow Re$	$R(x, y) = 1 - A(x, y)$

returns the minimum probability or probability density value of A and B. *Max* returns the maximum.

Operations for uncertain regions

This is a set of operations for which at least one input must be an uncertain face or region. The semantics for these operations are defined in Table 3.7.

Inside: This operation has somewhat more complex semantics in the uncertain case than in the crisp case because the result may be uncertain. This is indicated by the "Maybe" value.

- For faces: If $Support(A) \cap Core(B) = Support(A)$, return "Yes". If $Core(A) \cap Support(B) = Core(A)$, return "Maybe". Otherwise return "No".
- For curves and points: If $A \cap B = A$ return "Yes". If $A \cap Support(B) = A$ and A is a curve, return "Maybe". If *Existence*[1]$(A \cap B) > 0$ and A is a point, return "Maybe". Otherwise return "No".

Intersect: This operator determines the probability that *A* and *B* intersect. This is the "overlap" criterion used in many spatial searches.

Negation: This is a set operation which is meaningless for uncertain points and lines. The negation of an uncertain point or line would contain an infinite number of points or curves, some of which would be almost identical to those removed. This result is meaningless, so the negations for these datatypes are not defined

1. The *Existence* operator returns the probability that an object exists. It is defined in Section 3.6.1.3.

Table 3.8 *Operations for uncertain lines*

Operation	Signature	Semantics
Cross	$C \times C \rightarrow B$	See text
Length	$C \rightarrow N$ $I \rightarrow N$	Line: See text Range: $R = \int_t f(t)dt$

Operations on uncertain lines

The semantics of these operations are given in Table 3.8 and are defined on single curves. *Crosses* is only useful for single curves. The length of an uncertain line is the sum of the lengths of its curves.

Cross: Determining whether or not two uncertain curves cross each other is far more complex than for crisp curves because one does not know quite where the curves are. If $support(A) \cap support(B) = \emptyset$, the two curves cannot cross. Otherwise they may cross. To know for sure, the following conditions must be checked:

- Both curves must exist in the entire area in which they cross. The following formula test whether curve *A* exists in the entire area. The test is analogous for *B*.

  $$\forall(g \in A.G):(g \cap ca \neq \emptyset) \rightarrow (A.fe(g \cap A.ec) = 1)$$

  In this formula, $ca = support(A) \cap support(B)$. This prevents lines which are not guaranteed to exist in the area in which the lines may cross from getting "Yes" to the question *Crosses*(*A*, *B*)

- Let $ba = boundary(ca) \cap boundary(support(A))$ and $bb = boundary(ca) \cap boundary(support(B))$. Both *ba* and *bb* must consist of at least two crisp curves.

- Each component of *ba* and *bb* must cross line *ec* of the other line an odd number of times. This applies to both lines. This prevents lines such as line *A* in Figure 3.10 from getting "Yes" to the question *Crosses*(*A*, *L*).

Length: The true length of an uncertain line cannot be determined. For instance, Figure 3.11 depicts an uncertain line and three possible lines which all may be the correct line. All these three lines have different lengths. In fact, the "possible line" has much more than twice the length of the minimal line. Because the actual line may be arbitrarily long, only the minimal length can be

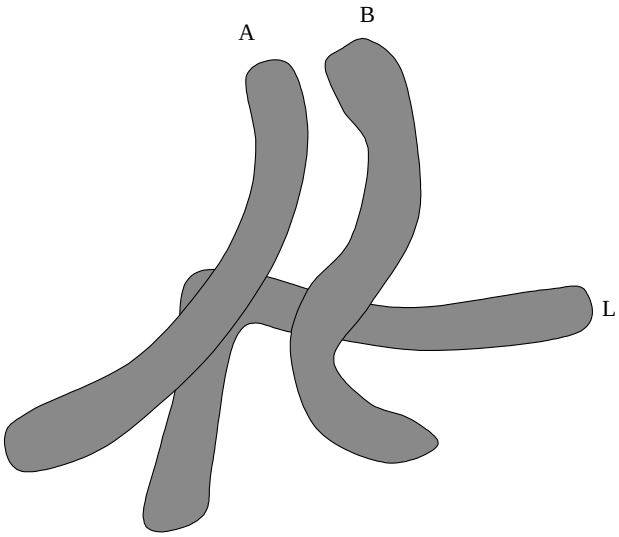


Figure 3.10 *A curve which may cross, and a curve which certainly crosses, the curve L*

computed. In the example shown in Figure 3.11, the minimal line is just a straight line.

To compute the minimal length, one may use the following procedure: Let $c=Core(A)$. For each crisp curve $s \in c$ find the two gradient lines that are at each end of *s*. Then find the shortest line *l* between these gradients that lies entirely within the support of *A*. The minimal length of the line is the sum of all of these *l*'s.

The *l*'s must also go in the same direction as the central line. Otherwise one might get inappropriate results for uncertain cycles. The support of an uncertain cycle is a torus, and one might get a shorter distance by going the wrong way.

Projections

Many operations in both spatial and spatiotemporal databases involve projecting a data type down into a universe with fewer dimensions than the universe in which the argument was defined. The exact probabilities of the projection cannot be determined without knowledge of the underlying probability

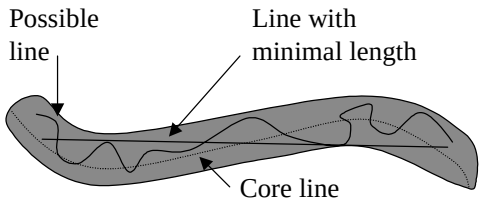


Figure 3.11 *An uncertain line, its core, its minimal line, and another possible line*

model[1]. Therefore, the best one can do is to create approximations. One could for instance project the support of the data type to get an outline of the projected result, and this might be enough for some applications. Alternatively, for each projected value one might use the maximal value of all the values projected into it. One example of this would be if one wanted to know where a moving region has been at any point in time, one might use the maximal probability for each point.

3.6.1.3. New operations for uncertain data

These operations are new for uncertain data because they determine different aspects of that uncertainty. A list of these operations and their semantics is given in Table 3.9.

Alpha_Cut: This operation was first described for fuzzy sets in [Zhan98]. It returns a crisp set which contains all the points which have a membership value or probability density value above the given number in *A*. The support operation can be seen as a special case of the alpha cut operation with *B*=0.

Core: This operation returns the set of values which the object must contain. For lines it returns the central line. It is defined for lines because some other operations need this definition. For points and numbers, the *Core* does not exist unless the point or number is crisp. In this case, *Core* returns a crisp version of the point or number.

Expected_Value: This operator returns the crisp object of the same type which represents the shape and position the object has the greatest likelihood of having. For lines, this is different from both *Core* and *Alpha-Cut*(*A*, 0.5) as shown in Figure 3.12.

Support: This operation returns the set of values which the object might possibly contain or be at.

Accuracy: This operator indicates how accurately an uncertain object is given. For lines and points, one can only determine whether the object is crisp (Accuracy 1) or uncertain (Accuracy 0).

Existence: This operation returns the probability that object *S* exists.

More_Accurate_Than: This operator checks whether *A* might be a more accurate version of *B*.

1. The underlying probability model means what exactly it is that varies. In Example 1, there is really only one random variable, the water level. For a geological feature with uncertain measurements, however, the location and shape of different parts of the boundary are independent random variables.

Breadth: This operation returns how wide the area of uncertainty for an uncertain line is at a particular point.

3.6.2. Operations on temporal uncertain data

In the same way as the previous section, this one is also divided into three subsections, one for operations which are no longer useful for uncertain data,

Table 3.9 *New operations for selected uncertain data types*

Operation	Signature	Semantics
Alpha_Cut	$S \times [0, 1] \rightarrow \{CP\}$ $NI \times [0, 1] \rightarrow \{CN\}$	Spatial types: $R = \{P \in CP \mid A(P) > B\}$ Number, Range: $R = \{N \in CN \mid A(N) > B\}$
Core	$S \rightarrow \{CP\}$ $NI \rightarrow CRa$	Point, Number: See text Line: $R = \{P \in CP \mid P \in A.ec \wedge A.fc(P) = 1\}$ Region: $R = \{P \in CP \mid A(P) = 1\}$ Range: $R = \{N \in CN \mid A(N) = 1\}$
Expected_Value	$T \rightarrow CT$	Point: E(*PP*) Number: E(*NP*) Line: $R = \{P \in CP \mid P \in A.ec \wedge A.fc(P) > 0.5\}$ Region, Range: *Alpha_Cut*(*A*, 0.5)
Support	$S \rightarrow \{CP\}$ $NI \rightarrow CRa$	Spatial types: $R = \{P \in CP \mid A(P) > 0\}$ Number, Range: $R = \{N \in CN \mid A(N) > 0\}$
Accuracy	$T \rightarrow [0, 1]$	Point, Line, Number: If *Crisp*(*A*) return 1 otherwise 0 Region: $\frac{(area(Core(A)))}{(area(Support(A)))}$ Range: $\frac{(length(Core(A)))}{(length(Support(A)))}$
Existence	$T \rightarrow [0, 1]$	Point: $\int_0^{max}\int_0^{max} A(x, y)dydx$ Line, Region: See text on projections Number: $\int_0^{max} A(v)dv$ Range: See text on projections
More_Accurate_Than	$T \times T \rightarrow CB$	All types: $Support(A) \subset Support(B) \wedge Core(A) \supseteq Core(B)$
Breadth	$C \times CP \rightarrow CN$	$length(l \in A.G \mid B \in Points(l))$

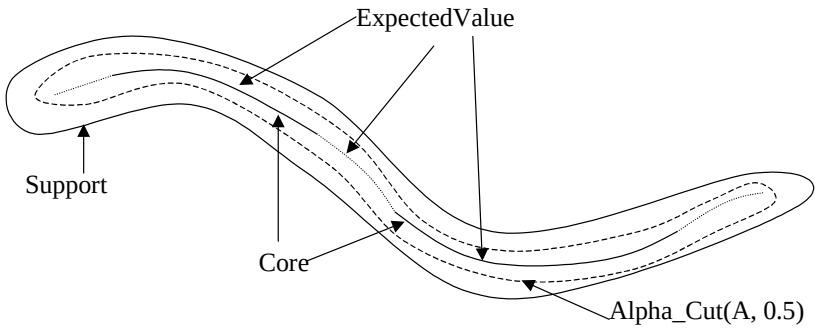


Figure 3.12 *Expected value*

one for operations that work for both certain and uncertain data, and one for new operations on uncertain data.

Like operations for crisp data, operations for uncertain data can be *lifted* into operations for uncertain temporal data as described in [GBE+00]. The basic idea is that the lifted operation takes temporal versions of the normal input parameters as input and returns a temporal version of the normal output, which is such that for any time instant this output is the same as would be returned by the non-lifted version of the function on the inputs as they are at that time instant. For a function that returns numbers in the [0,1] range, this corresponds to an $A_{URange(A_{Instant})}$ because both are functions from a time instant to a value in the [0,1] range.

The prefix *MOV* is used to indicate a moving, or temporal, type as opposed to a non-moving type.

Let *LA* be the lifted version of operation *A*, and *I* and *J* be temporal versions of the input parameters of *A*. Also let *O=LA*(*I*, *J*). Then for all time instants *t*:

$$O(t) = A(I(t), J(t))$$

As an example of lifting, the lifted version of *Union* has the signature $MOV(Ss) \times MOV(Ss) \rightarrow MOV(Ss)$. For regions, its semantics is:

$$R(t)(x, y) = A(t)(x, y) + (1 - A(t)(x, y)) \cdot B(t)(x, y)$$

Another example is the lifted version of the *Area* operator. It has the signature $MOV(S) \rightarrow MOV(N)$, and has the following semantics for faces:

$$R(t) = \int_x \int_y A(t)(x, y) dy dx$$

The result here is dependent on time, but not on the coordinate because both the *x* and *y* axes are eliminated by the integrals, while the time dimension

is not. Thus the lifted version of *Area* returns a moving number which can change over time as the object changes.

3.6.2.1. Operations on crisp data which are meaningless for uncertain data

There are two operations for temporal data which cannot be determined when there is uncertainty about the time. They are listed in Table 3.10.

Table 3.10 *Operations which are meaningless for temporal uncertain data*

Operation
Initial
Final

Initial ($MOV(T) \rightarrow T$): If there is uncertainty about when an object first appeared, its initial value cannot be determined. As an example, a region *B* is known to exist at time 3, and to overlap region *A*. It is also known that region *B* did not overlap region *A* before time 2, and that region *B* did not exist prior to time 1. In this case determining an initial shape is impossible. If *B* began to exist before time 2, it could not have overlapped region *A* at the start, but if it came into being after time 2, it could have overlapped region *A* right from the start.

Final ($MOV(T) \rightarrow T$): If there is uncertainty about when an object ceased to exist, its final value cannot be determined. The argument is the same as for *Initial*.

3.6.2.2. Operations which may be used on both crisp data and uncertain data

Temporal restriction operations

These are operations which restrict a spatiotemporal object to certain times. The semantics of these operations are given in Table 3.11.

At: This operation limits *A* to the times and places in which it intersects *B*. Checking *At* with a second argument other than a face or interval does not make sense in the uncertain case because the probability that the point or curve *A* intersects the point or curve *B* is 0.

At_Instant: This operation returns the shape of the object at a time instant. For the uncertain case an integral is used because one must consider the shape of the object in the entire period of uncertainty.

Table 3.11 *Temporal restriction operations*

Operation	Signature	Semantics
At	$MOV(NI) \times I \rightarrow MOV(NI)$ $MOV(S) \times F \rightarrow MOV(S)$	$R(t)(x) = A(t)(x) \cdot B(x)$ $R(t)(x, y) = A(t)(x, y) \cdot B(x, y)$
At_Instant	$MOV(T) \times CN \rightarrow T$ $MOV(T) \times N \rightarrow T$	$R(v) = A(B)(v)$ $R(v) = \int_t A(t)(v) \cdot B(t)dt$
At_Periods	$MOV(T) \times Ra \rightarrow MOV(T)$	$R(t)(v) = A(t)(v) \cdot B(t)$
When	$MOV(T) \times (T \rightarrow B) \rightarrow MOV(T)$ $MOV(T) \times (T \rightarrow A_{Prob}) \rightarrow MOV(T)$	See text $R(t)(x, y) = A(t)(x, y) \cdot B(A(t))$

At_Periods: This operation limits the first input so that it exists only in the time range given by the second input. In our model, this can be implemented by multiplying the functions for the two inputs.

When: This operation limits A to the times when the function B is true with respect to A. This operation may be used both when the function returns a probability and when it returns an uncertain Boolean value. For uncertain Boolean values, one may assume that *Maybe* has a "probability value" of 0.5 and use the semantics for a probability.

Projections from space-time

These operations create projections of a spatiotemporal object into either space or time. The semantics of these operations are defined in Table 3.12.

Def_Time: This operation returns the uncertain time interval in which the object A exists. Because this is the same as would be returned by a lifted version of *Existence*, no new definition is needed for this operation.

Locations: This operation returns the portions of the projection of an uncertain moving point which are points themselves. The projection of an uncertain moving point will only be a point when it stands still. This operation is therefore defined to return all the uncertain locations in which the point has stood still.

Present: This function must return a probability rather than a Boolean value, because it may be uncertain whether the object is present or not.

Trajectory: This operation returns the portions of the projection of an uncertain moving point which are lines. These are the projections of the uncertain

Table 3.12 *Projections from space-time into space or time*

Operation	Signature	Semantics
Def_Time	$MOV(T) \rightarrow Ra$	This is a lifted version of *Existence*
Locations	$MOV(P) \rightarrow \{P\}$	$R = \{p \mid \exists t:(p = A(t)) \wedge \exists(\varepsilon > 0)\forall a \forall b: (t - \varepsilon < a < t + \varepsilon \wedge t - \varepsilon < b < t + \varepsilon) \rightarrow (A(a) = A(b))\}$
Present	$MOV(T) \times N \rightarrow A_{Prob}$	*Existence*(*At_Instant*(*A, B*))
Trajectory	$MOV(P) \rightarrow L$	Projection (See Section 3.6.1.2)
Routes	$MOV(C) \rightarrow \{C\}$	$R = \{c \mid \exists t:(c = A(t)) \wedge \exists(\varepsilon > 0)\forall a \forall b: (t - \varepsilon < a < t + \varepsilon \wedge t - \varepsilon < b < t + \varepsilon) \rightarrow (A(a) = A(b))\}$
Traversed	$MOV(C \vee F) \rightarrow Re$	Projection (See Section 3.6.1.2)
Speed	$MOV(P) \rightarrow MOV(N)$	*Speed*(P) = *Speed*(*Expected_Value*(*P*))

point when it is moving. It contains the entire projection except for those periods in which the point has been standing still.

Routes: This operation returns the portions of the projection of an uncertain moving curve that are curves themselves. For an uncertain line, this operation must be run individually on all the curves of the line. The semantics of this operation uses the same principle as *Locations*. This definition does not cover the cases where the uncertain curve moves in such a fashion that the result would still be an uncertain curve.

Traversed: This operation returns the portion of the projection of an uncertain curve that are regions. This operation must use uncertain projection, but is otherwise identical to the crisp case.

Speed: The exact speed of a moving, uncertain point cannot be determined, because the exact location of the point is not known. However, a fairly good approximation is to compute the speed of the expected position of the point. If an uncertain point has expected value *a* at time 1 and expected value *b* at time 2, the speed of the point is *distance*(*b-a*) / (*time2 - time1*).

3.6.2.3. New operations for uncertain data

Although there are no truly unique new operations for uncertain spatiotemporal data, one may run any of the new operations from Table 3.9 on the time aspect of the object as well as the spatial aspect. These operations may also

be lifted to become new operations on uncertain spatiotemporal data. Thus, one gains two new operations from each one in Table 3.9. As an example, the lifted version of *Support* returns the spatial support of the object as a function of time, whereas *Support* run on the time aspect gives the time intervals in which the object may have existed. Examples of these two versions of *Support* for a moving uncertain interval are shown in Figure 3.13. The lifted support of the interval in the example is the total of the grey and black areas of the moving interval.

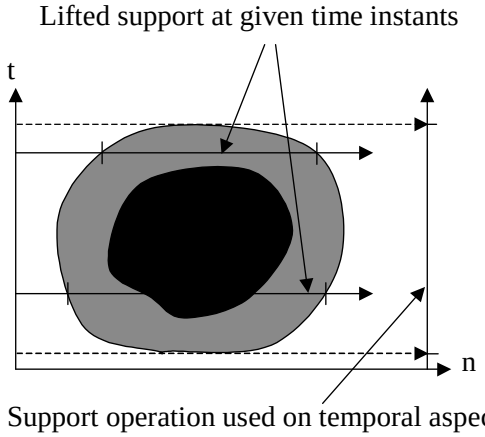


Figure 3.13 *Different variants of support for a moving interval*

3.7. Discussion

This work is, to the authors' knowledge, the first attempt to produce a general type system for uncertain spatial and spatiotemporal data. It also uses a more general definition for uncertain points and lines than earlier work, and these definitions facilitate the modelling of more types of uncertainty.

The type system described is also uniform in that all the types are defined in roughly the same way. For all the types there is a "region" indicating where the object might be. In this sense all the uncertain types are based on the crisp region. For all the types there is also a probability function indicating where the object is most likely to be.

This is similar to Schneider's model for vague regions, but in our work that method is applied for all the data types. The type system in this chapter is also capable of modelling vagueness. If one removes the positional and shape uncertainty from the point and line types, there is still the uncertainty about existence, and this can be used to model vagueness as well. The new model becomes almost identical to Schneider's model in that case.

One advantage of our abstract model is that existing discrete models are possible implementations of parts of it. The models described in [Dut92] and [Sch96] are possible discrete implementations of parts of this model because both ultimately describe probability functions over the plane which are legal according to our model.

This is an advantage because it shows that the model is fairly general (which is one of the aims of our abstract model). If existing discrete models did not implement parts of this model, this might indicate that it could be too specialized or had ignored significant types of uncertainty.

Our work also introduces types for continuous uncertain time instants and time intervals and uses these to create types for spatiotemporal data with uncertainty both in the spatial and temporal dimensions.

Additionally, many of the operations from [GBE+00] are evaluated for use in the uncertain case. Some of the operations are meaningless for uncertain data, but many can be used for both crisp and uncertain data. Some operations may also be used in new ways or on new data types when the data is uncertain. Some new operations (and operations from other sources) are also introduced to deal with the uncertainty. For uncertain Boolean values, two methods are used. The simplest is a three-value logic, which has been used earlier. However, due to the definitions of the types, many functions may instead return the likelihood of the answer being true.

One potential problem with our model is that the mathematical complexity will make it a challenge to implement, and that simpler models might be better in certain cases. This is particularly true for the uncertain line model. The reason for the complexity of the uncertain line is that the probability distribution for an uncertain line is neither a probability density function like for points nor a probability distribution function like for regions, but something in between. This and other issues related to implementation of our model and similar models are discussed in chapter 4 and chapter 5.

Part of the mathematical complexity discussed in the previous paragraph may be easily removed from the model at the cost of a decrease in expressiveness. There is, for instance, no need to include a model for temporal data or temporal uncertainty in a given system unless one is actually needed. The probability functions may also be arbitrarily simple or complex. The simplest variant is the function with equal probability over the entire point or line, and with probabilities 1, 0.5 and 0 indicating the core, uncertain boundary and the outside of a region. The advantage of a simple function is that it is easier to

store and faster to compute. The advantages and disadvantages of models of different complexity are discussed in chapter 4.

The advantage of complex functions is increased expressiveness. Some operations, such as *existence*, will return only 1, 0.5 or 0 if the simplest option is used for the functions. This is the same as a three-valued logic. In some cases, the geologists making the measurements may make good educated guesses as to the probability function. Thus it would be an advantage to be able to store these like our more generic abstract model allows.

Chapter 4
Three Discrete Models for Uncertain Spatiotemporal Data

4.1. Introduction

In many cases, one does not have accurate measurements of the position and shape of a geographic or spatial object. However, one often knows roughly where the object may be, and how uncertain its position or shape is. When storing such objects in a spatial or spatiotemporal database, it is therefore important not just to be able to store the object, but to also store how uncertain the object is.

Spatial and spatiotemporal objects may be uncertain because the measurements needed to place the object accurately are too expensive, or because exact measurements are impossible.

> **Example 1:** When making a database over the soil types of different areas, there may be two kinds of indeterminacy according to [LAB96]. Firstly, there may be uncertainty. This means that the soil type changes abruptly without any visible sign on the surface. Secondly, there may be vagueness. This means that there is no clear boundary, and the two types are mixed in the boundary area.

Traditionally, vagueness has been modelled by using fuzzy sets, whereas uncertainty has traditionally been modelled using probability theory. Like chapter 3, this chapter will look mainly at uncertainty. The models presented

here may also be used for vagueness in the same way as the abstract model from chapter 3.

The timing of events may also be uncertain, for the same reasons as for spatial uncertainty.

When an animal type becomes extinct, it is hard to know exactly when this happened. One may know that in 1999 there were observations of the animals or at least signs of their presence, whereas in 2001 there are no signs of them at all, and they are presumed extinct. However, one does not know when in this time the last individual died.

The abstract model presented in chapter 3 is based on infinite point sets so that it can be conceptually simpler than a lower-level model. It may be seen as a high-level design for a database for uncertain spatiotemporal information. The goal of this chapter is to build on that model to create the next natural level design for such a database.

This chapter will present three distinct models for uncertain spatiotemporal information: A complex model that attempts to model almost all of the aspects that the abstract model covers, a medium complexity model that models only a subset of these aspects but is much easier to implement, and a simple model built to minimize the amount of storage required. These three models will be compared with respect to storage required, how easy it is to implement different operations on them, processing speed of some operations and modelling capabilities.

Unlike the abstract model, the models presented in this chapter will be based on a finite representation which can be stored in a computer. This means that they are much closer to an implementation than the model from chapter 3. They may become somewhat more complex, and they only manage to model a subset of the aspects that the abstract model manages. However, the goal of this chapter is to be able to model a larger subset than existing models at this level.

4.2. Related work

There are two basic discrete methods for storing spatial data, the raster and vector models. The raster model divides space into a partition (typically a grid) and stores one value in each cell. For vague data, [LAB96] and [Low94] present raster models in which a fuzzy membership value is stored in each cell. Fuzzy sets [Zad65] is a type of set in which the membership of an individual may be fuzzy, that is, it has a value between 0 and 1. The problem with this

kind of model is that the data volume quickly becomes very large if the spatial objects are to be stored accurately. The two raster models mentioned also cannot model lines or points. The advantage of such raster models is that the values of arbitrarily complex functions can be stored explicitly in the cells. Also, overlay operations in which the values from two functions are combined are much easier to compute for raster models than for vector models.

Vector models, on the other hand, store the boundary of a region as a set of line segments. This form of storage is more complex, but usually takes much less space. An example of a vector model for uncertain regions is presented in [Sch96]. In this model, the ROSE algebra, which is a vector-based representation built to avoid inconsistencies, is used as a foundation for a model for uncertain regions. However, you cannot store probability functions with this model. One approach for storing distinct probabilities in such a model is to store a number of different regions, one inside the other, where each successive region has a higher probability than the one before. This method is used to compute fuzzy intersection in [Sch01].

An early model for uncertain points and lines is presented in [Dut92]. In that model, an uncertain point is stored as a central point with a circular deviation. The probability of a point being at any one place follows a bi-Gaussian distribution over the deviation. Lines are made up of a series of such points. Each line segment may start at any possible position of the first point and end at any possible position of the second point. The probability of each of these potential line segments is the product of the probabilities of the two points being at those precise locations. The paper shows that the iso-lines of probability in such an uncertain line look like a snake that has eaten a lot of eggs, and the eggs are in different parts of its throat and stomach. This effect is shown in Figure 4.1, where the iso-line of probability density 0.5 has this characteristic appearance. The support of the uncertain line is also shown. The support is the area in which the line may possibly be.

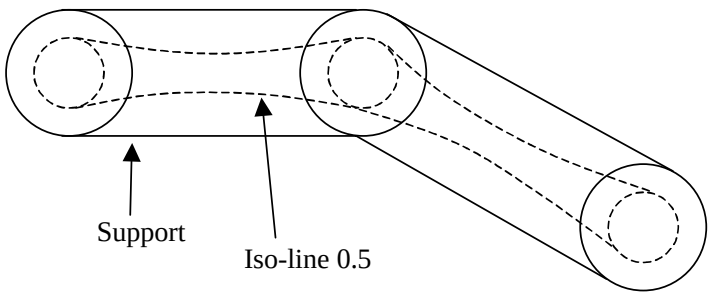


Figure 4.1 *Dutton's model of an uncertain line*

An attempt at a combination of raster and vector models is presented in [Sch00a]. This model resembles a raster model in that space is divided into a partition, and a fuzzy membership value is stored in each cell. Unlike normal raster models, the boundary of a region is stored by storing which vertices[1], rather than which raster cells, it goes through. Like any raster model, the boundary must follow the cell boundaries. This paper also shows how to implement fuzzy union, intersection and difference with this model.

A model for crisp[2] spatiotemporal objects has been presented in [FGNS00]. This paper uses a vector model for spatial objects and extends this model with time slices. The basic idea is that in each time slice, each spatial object evolves by simple rules. This is shown in Figure 4.2, where the number changes linearly inside each time slice. A point in this model moves along a straight line in each time slice, and may only change velocity between time slices.

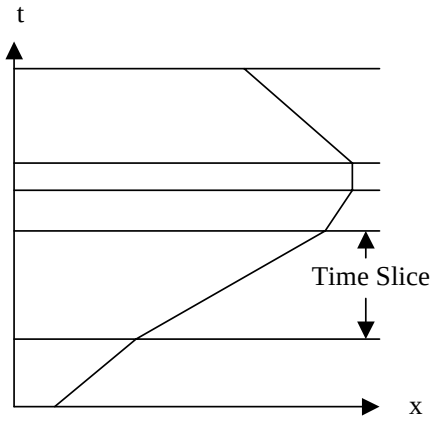


Figure 4.2 *Sliced representation of uncertain number*

A moving line segment may move and grow or shrink in a linear fashion, but may not rotate, and its speed of movement is the same throughout the time slice. The rule against rotation is there because a rotating line segment would yield a curved surface in space-time, whereas a non-rotating line segment would yield a straight one. Many operations are much easier to perform on straight planes than on curved ones. An example of how rotation may be implemented by non-rotating lines is shown in Figure 4.3.

Like other vector models, advanced types are built up from simpler ones, so that a region is built from several cycles which are built from several line

1. The vertices are the points in which two grid lines cross.
2. As opposed to indeterminate

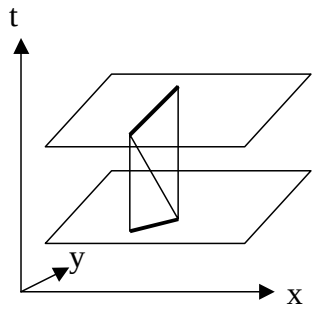


Figure 4.3 *Sliced representation of line segments*

segments. The moving types themselves are sets of disjoint time slices. A description of an implementation of the model from [FGNS00] can be found in [Rod00].

[TG01] discusses some of the problems in interpolating between snapshots to create the model presented in [FGNS00] and presents algorithms that solves many of them. This chapter will not focus on algorithms for such interpolation, but will also include some issues which become more complex with the presence of uncertainty, especially temporal uncertainty. The interpolation problem will be used to illustrate why the sliced representation becomes more complex in the uncertain case. This chapter will also discuss how the algorithms from [TG01] may be extended to the uncertain case.

4.3. Basis for the new model

As mentioned in Section 4.2, there are two basic modelling techniques for creating a discrete model: raster and vector. The vector approach is chosen in this chapter because rasters require much more storage space for regions, and lines may become inaccurate when stored in a raster format. A line in a raster model must be approximated to a series of pixels just like when a line is drawn on the screen.

Additionally, this chapter presents ways to store and compute probability functions in a vector model, so that a raster model is no longer needed to store exact probability values or fuzzy set values.

For the simple model, the points from [Dut92] will be used to store single points because they are simple to store. However, a different model for lines is developed in which computing the probability density of the line passing a certain point is somewhat simpler than in Dutton's model. Dutton's model is also unsuited to model uncertainty about the length of the line.

For the advanced model, the model presented in [Sch96] is used as a basis for regions. The advanced model for an uncertain region is based on the same concepts as those used in [Sch96], but it also includes other aspects like a system for modelling a probability function. This chapter adds lines, points, time and probability functions to this model.

For the medium complexity model, there is no previously published model which we can naturally make any comparisons with.

For the temporal aspects of the models, the methods from [FGNS00] will be used as a basis, although for some applications they will require significant modifications. However, in most cases the basic approach of splitting the objects into time slices will be retained.

The following types for crisp spatial data will be used to define the types in this chapter. These are taken from [FGNS00] with the exception of $D_{Segment}$ and D_{Line}. These two are different because a more specialized definition of lines is needed in the uncertain case[1].

Table 4.1 *Types for crisp spatial data*

Type name	Type definition
D_{Point}	A single point
D_{Points}	A set of points
$D_{Segment}$	A single line segment consisting of a start point and an end point
D_{Line}	A set of $D_{Segment}$ where each segment except possibly the two end segments share end points with the previous and the next segment, and where none of the segments may share interior points.
D_{Cycle}	A D_{Line} where the start point and end point are the same
D_{Face}	Area that has an outer cycle and a number of disjoint hole cycles
D_{Region}	Consists of a number of disjoint faces

The following terminology will be used for lines in this chapter. A line segment is a straight line going between two points. A curve is a single continuous line that does not intersect itself. A curve consists of a set of line segments. A line is a set of curves. Note that [FGNS00] does not use the concept of curves.

The abstract model from chapter 3 will be used as a basis for this discrete model. This means that the advanced model will try to model most of the as-

1. See Section 4.4, 5 and 6.

pects that the model from chapter 3 can model. The simpler models are of course more limited.

The basic idea from chapter 3 is that all uncertain objects, regardless of type, are known to be within a certain crisp region. It may also be known where the object is most likely to be. This is modelled as a function over the plane for all three types[1]. For points this is a pure probability density function, for regions it is a pure probability distribution function, and for lines it is a hybrid.

The uncertain point (A_{UPoint}) is modelled as a region with a probability density function indicating where the point is most likely to be. The uncertain curve (A_{UCurve}) is modelled as a core line with gradient lines crossing it, and probability functions for both of them. The probability distribution function along the core line represents uncertainty about the existence of the line and the length of the line. The probability density function along the gradient lines represents the fact that the exact location of the line is not known. These gradient lines must form a crisp face which is the area in which the line might possibly be.

The uncertain face (A_{UFace}) is modelled as a probability distribution function over the plane. The set of points with function value above 0 must form a crisp face.

In the abstract model from chapter 3 there are two types of functions, probability density functions and probability distribution functions. The probability distribution functions may be used in the discrete model as well, but there is a problem with the probability density functions.

A probability density function is defined as having an integral of 1, and is used for a continuous number line. However, in a computer the number line is never really continuous. Even if floating-point numbers are used, there is a minimum precision and therefore a minimum distance between subsequent numbers. This means that the correct choice is to store a probability mass function rather than a probability density function.

A probability mass function is a function which gives the probability that the number has a certain precise value, and the sum of all these values is 1.

The properties of these are defined by the following functions:

- Probability Mass Function:
  $ProbMass(P) \equiv (\forall x{:}P(x) \geq 0) \wedge \sum_x P(x) \leq 1$

1. Points, lines and regions.

- Probability Distribution Function:
  $ProbFunc(P) \equiv \forall x{:}(P(x) \geq 0 \wedge P(x) \leq 1)$

A simple way of storing a probability mass function that is shaped as a series of steps is described in [DS98].

There are several ways of representing coordinate values in a computer. The simplest of these is the integers, but there are other such systems, such as the dual grid presented in [LG00]. In the discussion of the types, the number system used for coordinate values will be referred to as *CVS*.

In chapter 3, the point and face types are modelled as functions over the plane. In the discrete case, this is not a feasible way of storing them for two reasons. First, such a function would quickly become very complex and the computer might have difficulties storing and computing them. Second, it would be next to impossible for the user to define such a function for each point or region. Therefore another method must be used. One simple method that has been used in many earlier models for vague and uncertain regions is to store two regions, one inside the other. The innermost region is the area in which the region is certain to exist, and the outermost region is the area where the region may be. It is known that the region is not outside the outermost region.

This method may be extended with a probability function describing how likely it is for the region to be in all the points in between the innermost and outermost regions. To make it simpler for the user to define this function, it should be one-dimensional rather than two-dimensional. User might choose between some predefined functions such as linear, Gaussian or step function, or they can be allowed to define their own functions.

Three different models for uncertain spatial and spatiotemporal information will be presented in this chapter. A summary of the properties of these models is given in Table 4.2. This table shows how much storage space the types use in the three models compared with the corresponding crisp types. It also shows how complex it is to compute set operations and probability values for the various types in the three models. Additionally, it shows how many of the three methods for dealing with uncertain time from Section 4.8.3 can be used with each model. A “no” in a row means that that operation or feature cannot be used on that data type in that model.

The next three sections will describe the three different models for uncertain spatial data that are all based on the principles described in this section. Section 4.7 will describe how to store and compute the probability functions. Section 4.8 and Section 4.9 will describe how to extend the three models to be

Table 4.2 *Comparison of the three models*

Type[1]		Model					
		Advanced		**Medium**		**Simple**	
Integer	**Storage**	Large	Stores probability of each possible value	3X	Stores support and probability mass function	2X	*Same as Medium* except assumes uniform probability
	Set. Ops	Simple		Simple		Simple	
	Prob. Func.	Simple		Simple		No	
	Temporal	All methods		All		All	
Range	**Storage**	7.5X	Core and support as separate "interval" objects	2.5X	Core and support in same interval object	2X	*Same as Medium* except assumes uniform probability
	Set. Ops.	Simple		Simple		Simple	
	Prob. Func.	Simple		Simple		No	
	Temporal	All		All		All	
Point	**Storage**	Large	Support as general face	5.5X	Support as distances at set angles	1.5X	Spherical support.
	Set. Ops.	Simple		Complex		Complex	
	Prob. Func.	Complex		Simple		No	
	Temporal	All		Special[2]		Special[2]	
Curve	**Storage**	3.75X	Support as general face	3X	Stores *CrossCurves* at arbitrary angles	1.75X	*Same as Medium* except *CrossCurves* have set angles.
	Set. Ops.	Simple		Complex		Complex	
	Prob. Func.	Complex		Simple		No	
	Temporal	All		Simple only		No	
Face	**Storage**	2X	Support as general face, core as general region	3X	As crisp face except uses uncertain curves	1.75X	*Same as Medium* except simple curves are used.
	Set. Ops.	Simple		Complex		Complex	
	Prob. Func.	Complex		Simple		No	
	Temporal	All		Simple only		No	

1. For uncertain reals and Boolean values, the three models are roughly the same. Uncertain Boolean values do not require more storage space than crisp ones. Uncertain reals require 3X space in the advanced and medium case and 2X space in the simple model.
2. See Section 4.9.2

spatiotemporal. Section 4.10 contains a description of the implementation of some spatial and spatiotemporal operations on the models described. Section 4.11 contains a discussion and comparison of the models presented in this chapter.

4.4. Advanced model for uncertain spatial data

The first model to be presented is the advanced one, because it is most similar to the model in chapter 3. This chapter also presents two simpler models. These are detailed in the next two sections and were created to address the problems with the advanced model. They are both less expressive than the ad-

vanced model. These three model sections will detail purely spatial models for uncertainty, without a time component. Section 4.7 will consider how to store probability functions and compute probabilities, Section 4.8 will describe how to extend the advanced model to become spatiotemporal and Section 4.9 will describe the same for the simpler models.

In all the modelling sections, the word *Core* will be used to mean the area with greatest probability. This is usually 1, but not necessarily. The probability in the core is always the probability that the object exists. This is stored separately. The reason for this is given in Section 4.4.2 and Section 4.7.

The word *Support* will be used to mean the area in which the probability or probability mass of the object is above 0. This is the area where the object might possibly be.

Section 4.4.1 deals with how to model non-spatial types. This is a necessary foundation for both the spatial and temporal types. The subsequent sections will look at points, lines and regions.

4.4.1. Base types

A single type for uncertain numbers was used in the abstract model. This is not a good solution in a discrete model, because in an actual database some numbers may be stored as integers and others as floating-point numbers. In the present model, two types of uncertain number are defined, the uncertain integer and the uncertain real number.

One way to store an uncertain integer is to store each value the uncertain integer might take as well as the probability that it takes that value. An example of such an integer is shown in Figure 4.4. This allows us to store an uncertain integer following any conceivable probability distribution. The database should have an integrity rule which disallows the storage of an uncertain integer with a total probability above 1. A crisp integer can be stored with just a single number and a probability of 1.

Definition 1: The **uncertain integer** is defined as follows:

$$DA_{UInteger} \equiv \Big\{ is \mid$$
$$is \subseteq \{(i \in Z, pi \in \Re^{+})\} \wedge$$
$$Finite(is) \wedge \sum_{is} is.pi = 1 \wedge$$
$$\forall a \in is, \forall b \in is, a \neq b \rightarrow a.i \neq b.i \Big\}$$

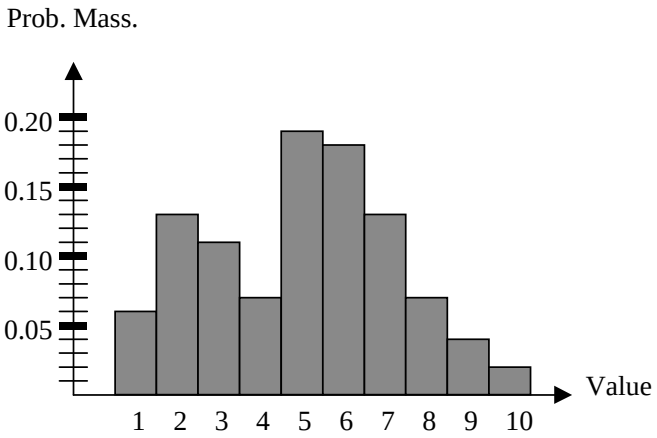


Figure 4.4 *Advanced uncertain integer*

Because this model stores one floating-point for each possible value, it will use (*m*+1)**n* times the storage space of a crisp integer, where *n* is the number of values the uncertain integer may take and *m* is how many times more storage space is needed to store a floating-point number compared to an integer. Due to this large increase in the storage required, a simpler model may be preferred in most cases.

For storing uncertain reals, one cannot use this approach, because it would yield a very large number of probabilities. The uncertain real should instead be stored as a probability mass function over the number line. One way to do this would be to store a starting point and an end point for the interval in which the real number may be, and a probability function over that interval as shown in Figure 4.5.

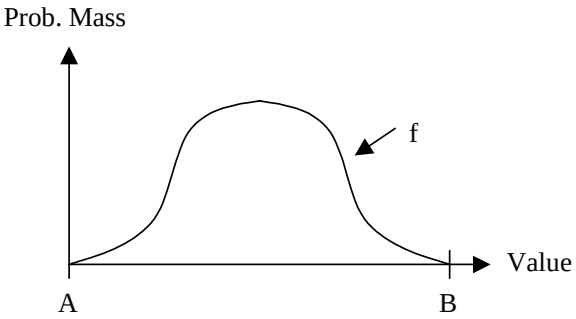


Figure 4.5 *Advanced uncertain real*

Definition 2: The **uncertain real** is defined as follows:

$$DA_{UReal} \equiv \{(a, b, pr) |$$
$$a \in \Re \wedge b \in \Re \wedge$$
$$pr{:}[a, b] \rightarrow \Re^{+} \wedge$$
$$a \leq b \wedge ProbMass(pr)\}$$

A crisp real may be stored by setting A=B and letting f be a function which is 1 for a single number and 0 for all others.

For the uncertain real, one must store two real numbers and a reference to the probability function used. Assuming this reference is the same size as a floating-point number, this means that an uncertain real takes three times as much storage space as a crisp real.

Chapter 3 describes two ways of handling functions which return Boolean values in the crisp case. If the function that returns the value is capable of determining the probability of the answer being true, that probability should be returned. Otherwise, a third "Boolean" value, *Maybe*, is returned when the result is uncertain. These two data types are defined as follows:

Definition 3: The **probability** is defined as a number between 0 and 1:

$$DA_{Prob} \equiv [0, 1].$$

This probability may be stored in a computer as a floating-point number.

Definition 4: The **uncertain Boolean** is defined as follows:

$$DA_{UBool} \equiv \{False, Maybe, True\}$$

If the crisp Boolean can be stored as a single bit, the uncertain Boolean needs twice as much storage space because it needs two bits. However, in many systems an entire byte is used to store a single Boolean and in those systems there is no additional storage space needed.

A type for an uncertain range, or set of intervals, is also needed. An uncertain range is a one-dimensional probability function that for each number gives the probability that the number is part of the range. An example of how the uncertain range could be stored is illustrated in Figure 4.6. In this figure, the light grey areas are the areas of uncertainty and the dark grey areas are the areas where the range is certain to be. In a computer, one probably only has some specific functions defined. Therefore, it is impossible to model the entire range as a single probability distribution function. A more plausible approach

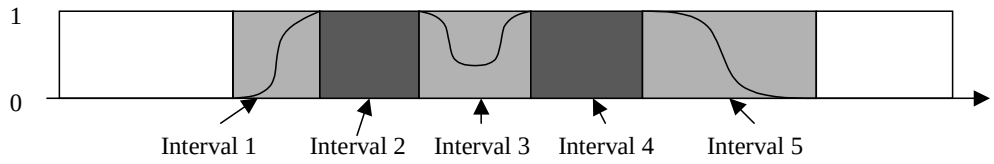


Figure 4.6 *Advanced uncertain range*

is to split the range up into several parts each of which can be modelled by a simple function.

One way to define the uncertain range from chapter 3 is to define each of the light or dark grey areas as separate intervals and define the range as a collection of such intervals. In this case, the range shown in Figure 4.6 consists of five such intervals. However, the core of this range consists of only two crisp intervals, and the support of this range is just a single crisp interval. The reason why the core and support yields a different number of crisp intervals is the fact that interval 3 from Figure 4.6 is uncertain and therefore is in the support but not in the core. To be able to model such uncertain intervals as interval 3, minimum and maximum probabilities must be included in the uncertain intervals.

Definition 5: The **uncertain interval** is defined as follows:

$$
\begin{aligned}
&DA_{UInterval(\alpha)} \equiv \{(sv, ev, min, max, if) | \\
&sv \in \alpha \wedge ev \in \alpha \wedge \\
&min \in DA_{Prob} \wedge max \in DA_{Prob} \wedge \\
&if{:}[sv, ev] \rightarrow [min, max]\}
\end{aligned}
$$

Definition 6: The **uncertain range** is defined as follows:

$$
\begin{aligned}
&DA_{URange(\alpha)} \equiv \{(IS, pe) | \\
&IS \subseteq DA_{UInterval(\alpha)} \wedge pe \in DA_{Prob} \wedge \\
&Finite(is) \wedge \\
&\forall a \in IS, \forall b \in IS, a \neq b \rightarrow Disjoint(a, b)\}
\end{aligned}
$$

The α in these definitions refers to the type that the interval or range is tied to. This can be one-dimensional types such as integer, real or time.

In this model, each uncertain interval takes 2.5 times as much storage space as a crisp interval (which requires two numbers) because two additional real numbers must be stored as well as a reference to a probability function (five numbers in total). Additionally, five such intervals are needed to store the range shown in Figure 4.6. For a single interval with uncertainty as to the length in both ends, three of the uncertain intervals defined here are needed.

This means that the storage space needed for an uncertain range is likely to be 2.5 * 3 = 7.5 times as high as that required for a crisp range.

4.4.2. Uncertain points

An uncertain point is a point for which the position is not known. It may not even be known if the point exists or not.

> **Example 2:** An animal with a radio transmitter may be tracked as long as it is in the range of the sensors, but if it moves outside this range, the people tracking the animal may only have a very general idea of where it is.

The uncertain point in the abstract model from chapter 3 is essentially a region with a probability density function indicating where in this region the point is most likely to be found. In most cases, this region will be connected, that is, be a single face. In the discrete model, this probability density function becomes a probability mass function. An easy way to define the probability mass function is to say that the value of the function is dependent on how far the position in question is from the edge of the face compared to how far it is from a central point. The central point is the point where the uncertain point has the highest likelihood of being. This means that a one-dimensional probability mass function can be used to store the probability distribution of the uncertain point. An example of an uncertain point is shown in Figure 4.7.

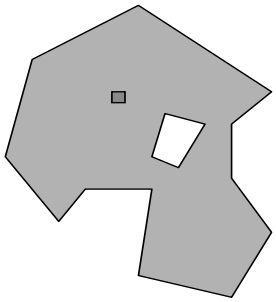


Figure 4.7 *Advanced uncertain point*

If one allowed a point to be in multiple disjoint faces, there would have to be one such central point in each face, as well as stored probabilities for how likely it is that the point is within any given face. Because allowing multiple faces makes the model significantly mode complex and is not necessary for most uncertain points, we choose to restrict uncertain points to a single face

The probability of the existence of a point may either be stored separately or be determined by the probability density function. In Section 4.7, it will be shown how the probability density functions may be stored and normalized so that they yield an integral value of 1 regardless of the size of the face. To avoid

making this process even more complex, the probability of the point existing is stored separately.

Definition 7: The **uncertain point** is defined as follows:

$$DA_{UPoint} \equiv \{(pf, cp, ps, pe) | \\ pf \in D_{Face} \wedge cp \in D_{Point} \wedge \\ ProbMass(ps) \wedge pe \in DA_{Prob} \wedge \\ Inside(cp, pf) \wedge \\ Increasing(ps)\}$$

The *Increasing* function returns true iff the input function always increases, that is, its derivative is positive. No further requirements on the function are necessary because the function has to be normalized for the area of the particular point anyway.

This definition allows for only one peak in the probability mass. This is because the probability mass function is increasing and because it increases when moving toward the central point.

One way to allow multiple peaks would be to allow multiple central points and one probability function tied to each. This would result in a somewhat more complex model and more complex mathematics because one would have to take the average of all these separate functions to get the actual probability values.

The face, which determines where the point may be, may take up much more storage space than a single crisp point because its representation may contain a large number of points. The storage space needed to store an uncertain point with this model may therefore be very high compared to that needed for a crisp point. To be able to model crisp points, the face must be allowed to be only a single point, and the probability function must be a function with a value of 1 in that point.

Set operations will need to return sets of points rather than individual points.

Definition 8: The **uncertain points** set type is defined as follows:

$$DA_{UPoints} \equiv \{UP \subseteq DA_{UPoint} | Finite(UP)\}$$

4.4.3. Uncertain lines

The uncertain curve as defined in chapter 3 consists of a core line and a set of gradient lines that cross the core line and that form a face together. The

core line is where the curve has the greatest likelihood of being. This arrangement with an infinite number of gradient lines is impossible to store in a computer, so an approximation must be used.

Instead of storing a set of gradient lines, the support is stored as a face. In this way, the curve gets a finite representation. This is the same as for the uncertain point. The gradient lines are computed when needed. The algorithm for doing this depends on how the probability functions are stored. This is discussed in Section 4.7. A face rather than a region is used because a curve should be continuous, and it cannot be continuous if the support consists of several disjoint parts. One example of an uncertain curve is shown in Figure 4.8. The support of this curve is the grey area.

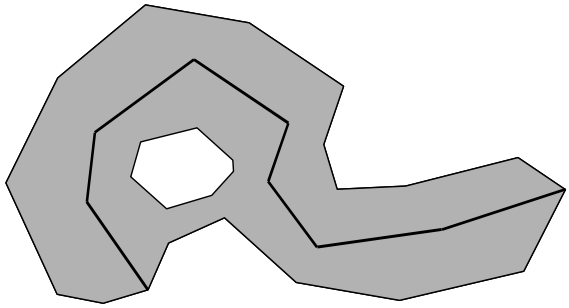


Figure 4.8 *Advanced uncertain curve*

The central curve is the thick line inside the grey area in Figure 4.8. This is stored in a similar fashion to a crisp curve with the additional requirement that any end points of the curve must also be on the boundary of the support.

The probability of existence of the curve may be stored together with the central curve by associating a probability function with each line segment. This works exactly like the uncertain range from Section 4.4.1 with a line segment corresponding to an interval as defined in Section 4.4.1, and the entire curve corresponding to a continuous range.

The uncertain line segment is stored with its two end points and a probability function over it. The segment itself is the straight line going between the two points.

Definition 9: The **uncertain segment** is defined as follows.

$$
\begin{aligned}
&D_{USeg} \equiv \{(sp, ep, min, max, sf) | \\
&sp \in D_{Point} \wedge ep \in D_{Point} \wedge \\
&min \in DA_{Prob} \wedge max \in DA_{Prob} \wedge \\
&sf{:}[sp, ep] \rightarrow [min, max]\}
\end{aligned}
$$

The probability of existence for the curve as a whole is stored separately. This is both to make it easier to compute, and to be similar to the uncertain point. This value should be multiplied to the probability function for the central line to get the actual probabilities.

The uncertain curve is therefore defined with a central line, which is a set of D_{USeg}, a face as support, a probability of existence and a probability function for the gradient lines.

Definition 10: The **uncertain curve** is defined as follows:

$$
\begin{aligned}
&DA_{UCurve} \equiv \{(sf, CC, pg, pe) | \\
&sf \in D_{Face} \wedge CC \subseteq D_{USeg} \wedge \\
&ProbMass(pg) \wedge pe \in DA_{Prob} \wedge \\
&\forall a \in CC \rightarrow Inside(a, sf) \wedge \\
&ContCurve(CC) \wedge \\
&\forall a \in CC, \forall b \in CC, a \neq b \rightarrow \neg Cross(a, b) \wedge \\
&\forall a \in Endpoints(CC) \exists b \in Points(sf) \rightarrow a = b\}
\end{aligned}
$$

Cross in this definition means that the two line segments cannot share points. *ContCurve* returns true iff the input set of line segments forms a continuous curve. This means that all the end points of the line segments except possibly for two must occur twice. The *Endpoints* function returns the end points in a set of line segments which occur only once. The *Points* function returns the points that are explicitly stored in the boundary of a curve or face[1]. The probability mass function *pg* is the function which determines the probability that the line passes through a particular point of the gradient line.

Note that the central line is permitted to be a cycle in this definition, which allows the modelling of uncertain cycles.

A crisp curve is stored by making *sf* equal to the central line. This is the only instance in which *sf* is allowed to be something other than a valid member of D_{Face}. The probability mass function is in this instance equal to 1 on the central line and 0 otherwise.

Each segment of the central line stores two probability values (two numbers) and a reference to a function (one number) in addition to the two end points (which consist of two numbers each). This means that they consume 7:4=1.75 times as much space as a crisp line segment. The outer cycle of the surrounding face takes twice as much space as a crisp line if it is stored with the same accuracy. If the support of the line has holes, these take more space. If

1. The points which the straight lines are drawn between.

each line has one hole on average, and each hole has as many border points as the core line, the uncertain line of this model takes 1.75(core line) + 2(outer cycle) + 1(hole cycle) =4.75 times as much storage space as a crisp line. It is most likely, however, that most lines do not have holes. In this case, the storage requires is 1.75+2=3.75 times that of a crisp line.

This method of defining uncertain curves is different from the method used in [FGNS00] for crisp curves. In their model, an uncertain curve is just a single line segment and the uncertain line is an arbitrary set of these. This approach cannot be used in the uncertain case. The shape of the support of the uncertain curve from the abstract model in chapter 3 is not necessarily related to the shape of the central curve. The support is also required to be a crisp face, which is a set of cycles. A cycle is a set of line segments which together form a closed line, and it is both easier and better to store a cycle as a closed line rather than as a set of unrelated line segments. Thus, the support of the uncertain curve cannot be divided in this way. When the support cannot be divided in this way, there is little point in using this method to divide the central curve.

An uncertain line is a set of uncertain curves. In the abstract model from chapter 3, the uncertain curves are required to be disjoint to ensure uniqueness of representation. When data about uncertain lines is stored, the uncertain curve should be used as it adequately covers the normal concept of a line, and the uncertain line should normally be reserved for temporary constructs and query results. Therefore, the restriction of disjointedness is not really needed in the discrete case unless one wants to model an uncertain graph. The latter is outside the scope of this thesis. Relaxing this requirement also makes the set operations much easier to implement for uncertain lines because it is no longer necessary to split curves that intersect or cross. Correctly splitting such a curve is impossible in some cases. One example of this is the example in Figure 4.9. In this case lines A and L may intersect, but do not necessarily intersect. Should they be split or not, and in that case where?

Definition 11: The **uncertain line** is a set of uncertain curves:

$$AD_{ULine} \equiv \{UC \subseteq DA_{UCurve} | Finite(UC)\}$$

4.4.4. Uncertain regions

Using a vector model to store indeterminate regions in a manner similar to the one described above for lines and points has been done in [Sch96]. The uncertain region model presented here will build on that model. The main extensions are storing a probability function and using the same methods for building point and line models. An uncertain face may be stored as a crisp face and a crisp region, where the region is contained in the face. The face is the

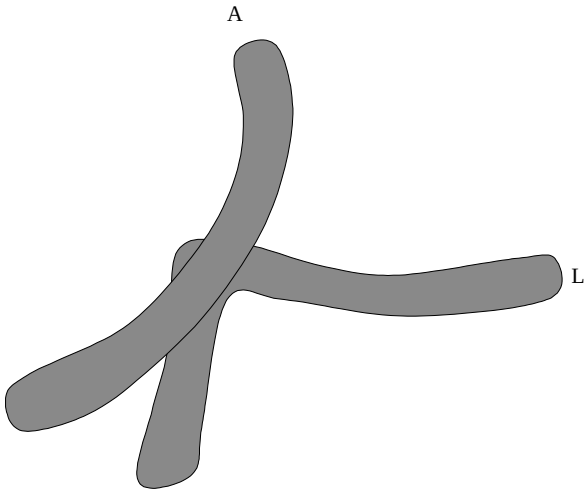


Figure 4.9 *Possibly intersecting curves*

support and the region is the core. The core is a region rather than a face because a face with multiple disjoint cores can be used to model uncertainty about the exact number of faces in a region. An example of an uncertain face is shown in Figure 4.10. In this example, the dark grey area is the region indicating the core, and the union of all the grey areas is the face indicating the support.

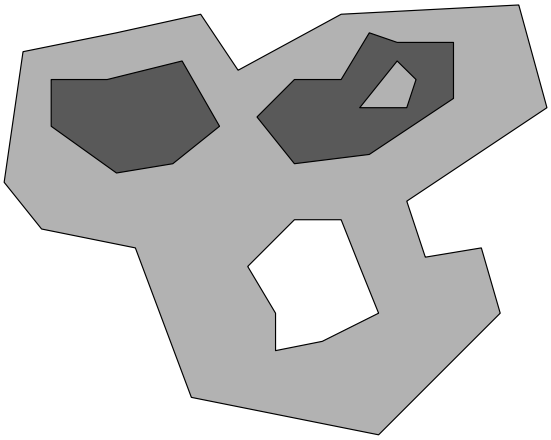


Figure 4.10 *Advanced uncertain face*

Additionally, the uncertain face should contain a probability function for determining the probability of the face existing in various parts of the area of the uncertainty. It should also contain a probability of existence.

Definition 12: The **uncertain face** is defined as follows:

$$DA_{UFace} \equiv \{(sf, cr, ps, pe) | \\ sf \in D_{Face} \wedge cr \in D_{Region} \wedge \\ ProbFunc(ps) \wedge pe \in DA_{Prob} \wedge \\ Inside(cr, sf)\}$$

Additionally, there should be a means to indicate the value of the probability function inside uncertain holes such as the one in the upper right corner of Figure 4.10. In this case, one might store a central point with either a probability value or a virtual distance from that point to the support.

A region is a set of disjoint faces. The uncertain region may be defined either through the face definition or directly using crisp regions:

Definition 13: The **uncertain region** is defined as a set of uncertain faces:

$$DA_{URegion} \equiv \{UF \subseteq DA_{UFace} | \\ Finite(UF) \wedge \\ \forall a \in UF, \forall b \in UF, a \neq b \rightarrow Disjoint(a, b)\}$$

To make it easier to compute the storage requirement of the uncertain region, an alternative definition is presented that defines the uncertain region only in terms of crisp regions.

Alternative specification: The uncertain region can also be defined as a core and a support region in a manner that closely resembles the DA_{UFace} definition.

$$DA_{URegion} \equiv \{(sr, cr, ps, pe) | \\ sr \in D_{Region} \wedge cr \in D_{Region} \wedge \\ ProbFunc(ps) \wedge pe \in DA_{Prob} \wedge \\ Inside(cr, sr)\}$$

From this alternative definition, one can see that storing an uncertain region essentially requires the storage of two crisp regions. This means that the storage requirement for an uncertain region is twice that of the crisp region. The function reference to the probability function takes minimal space compared to a crisp region. In Table 4.2, it says that faces require twice as much storage space. When an uncertain region takes twice as much space as a crisp one, the faces that it consists of should also take twice as much space as crisp faces.

4.5. Medium complexity model

The advanced model presented in Section 4.4 manages to model most of the aspects of the abstract model in chapter 3, but at the cost of increased complexity and increased storage space, especially for points. The advanced model would be good for applications which require the full power of the abstract model. However, for many applications a simpler model may be sufficient. This section presents a model that is much simpler than the advanced one, and the next section presents possible simplifications to make an even simpler model, while maintaining the goal of modelling all uncertain spatial data types.

4.5.1. Base types

The advanced model has a very costly but powerful model for uncertain integers. A cheaper model would be to use something like the advanced model for uncertain reals, that is, a probability function and an interval in which the number may be. The interval might be expressed as two crisp integers, and the probability function as a reference to the function. Such a reference does not need to take more space than a crisp integer.

Definition 14: The **uncertain integer** is defined as follows.

$$DM_{UInteger} \equiv \{(a, b, pi) |$$
$$a \in Z \wedge b \in Z \wedge pi{:}[a, b] \rightarrow \Re^{+} \wedge$$
$$a \leq b \wedge ProbMass(pi)\}$$

This form of uncertain integer takes three times as much space to store as a crisp integer because two numbers and a reference are stored rather than one number. This is far less than the advanced model, and this method can still store a wide range of functions.

The **uncertain real** is stored in the same way as in the advanced model.

If there is no need for uncertain holes in an uncertain interval, the uncertain interval and range may be defined in a different manner to save space. The uncertain interval may store four boundaries and one probability function, and the uncertain range is still a disjoint set of such intervals. The four boundaries are: start of support, start of core, end of core and end of support. Figure 4.11 contains an example of such a range containing two intervals. The light grey areas are the areas of uncertainty, and the dark grey areas are where the intervals are certain to exist.

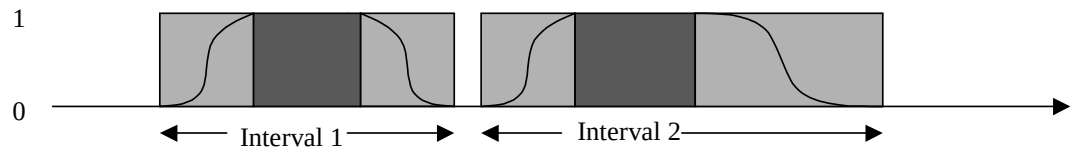


Figure 4.11 *Medium complexity range containing two medium complexity intervals*

Because a single interval now contains both core and support, the uncertain range does not need to contain nearly as many intervals, and the uncertain interval means what one intuitively expects it to mean. However, the definition of the interval becomes somewhat more complex.

Definition 15: The **uncertain interval** is defined as follows.

$$DM_{UInterval(\alpha)} \equiv \{(a, b, c, d, if) | \\ a \in \alpha \wedge b \in \alpha \wedge c \in \alpha \wedge d \in \alpha \wedge \\ ProbFunc(if) \wedge \\ a \leq b \leq c \leq d \wedge \\ a < d\}$$

The probability function is used between a and b and reversed between c and d. Between b and c, the probability value is 1. An alternative would be to store two different probability functions for the start and end of the interval. This would increase the storage cost for a single interval by +0.5. A single function is chosen because a given interval is likely to have similar behaviour in both edges.

The **uncertain range** in this model is defined exactly as in the advanced model, except that the medium complexity interval type is used instead of the advanced one.

For a single interval, one needs to store four border values instead of two, and also needs to store a reference to a probability function. Therefore, an uncertain interval takes (4+1):2=2.5 times as much storage space as a crisp interval. This is the same as in the advanced model. However, an uncertain interval with uncertainty about the length at both ends may be stored with just a single interval in this model, rather than three as in the advanced model. Thus a range consisting of such intervals costs 1/3 as much storage space as in the advanced model. However, for crisp intervals the two models are equally inefficient, because one interval of either type must be used to store a crisp interval, and both types of interval take the same amount of storage space.

Uncertain Boolean values and **probabilities** are stored the same way as in the advanced model.

4.5.2. Uncertain points

One major problem with the advanced model for uncertain points is that the storage space needed may be far larger than that needed for crisp points. If the database models mainly points and if storage space is an issue, models requiring less storage space may be needed. One way to limit the amount of storage space needed is to limit the number of points that make up the boundary of support of the uncertain point. One natural way of doing this would be to store the distance from the central point to the boundary of the support at certain predefined angles, such as every 45 degrees. An example of such an uncertain point is shown in Figure 4.12.

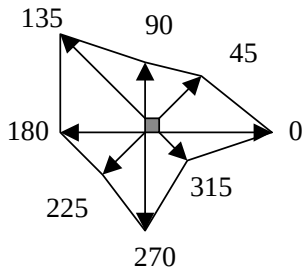


Figure 4.12 *Medium complexity uncertain point with eight angles.*

The probability mass values of the uncertain point can now be computed based on the relative distance from the central point and the edge of the support.

Definition 16: The **uncertain point** is defined as follows:

$$DM_{Upoint} \equiv \{(a_0 a_1 a_2 \ldots a_{N-1}, cp, ps, pe) \mid a_{i \mid i = 0 \ldots N-1} \in CVS \wedge cp \in D_{Point} \wedge ProbMass(ps) \wedge pe \in DA_{Prob} \wedge Increasing(ps)\}$$

In this formula, *CVS* is the number type used to represent the coordinates of D_{Point} values.

The disadvantage of this model compared to the previous one is that it cannot model holes, and that it can only model uncertain points in which there is a straight line from the central point to any point in the support.

Another problem with this model is that the results of spatial set operations such as finding the intersection of an uncertain point and a face[1] are not necessarily members of the base type. They are not members because the result

1. This is the part of the support of the uncertain point that is inside the face.

can have corners in different places than on the particular angles that are stored for the uncertain points. One solution is to store the normal point and to indicate that the probability mass function is the product of the normal probability mass function and the probability distribution function of the face. This problem does not occur in the advanced model because the face can be an arbitrary face in that model.

One advantage of this model compared to the advanced one is that the storage space needed is known and bounded. Storing a single distance for every 45 degrees yields eight numbers. Because the central point needs two and the probability mass function requires one, this means that the uncertain point takes 11 numbers to store, or 5.5 times as much as a crisp point. For a number of angles n, the uncertain point takes $((n/2)+1.5)$ times as much space.

The **uncertain points** set type is defined as in the advanced model except that it contains medium complexity uncertain points.

4.5.3. Uncertain lines

The storage cost for an uncertain curve in the advanced model is 3.75 times that of a crisp curve. This section will describe a model which takes somewhat less space. Another advantage is that it is very easy to compute probabilities for this model. The method for doing this will be described in Section 4.7. Some disadvantages are that the model presented here does not allow holes, and the shape of the support is partially determined by the shape of the central line. Another problem is that spatial set operations, such as finding the parts of an uncertain line that is inside a given region, may require some additional support. This problem is further discussed in Section 4.5.4 because it occurs a lot more often for medium complexity uncertain faces than for lines.

The basic idea is to store a central line, as well as crossing lines for each stored point along the central line. These crossing lines are equally long on each side of the central line, and determine the extent of the support of the line. These crossing lines are the “gradient lines” from Section 3.4.3 for the end points of each line segment. In the interior of the line segment, the gradient lines have an angle which is linearly interpolated between the two gradient lines at the end. The support of the uncertain curve is determined by taking straight lines between the ends of all the crossing curves. Straight lines are used to make it easier to run plane-sweep algorithms on the support.

One example of a line stored in this fashion is shown in Figure 4.13. One

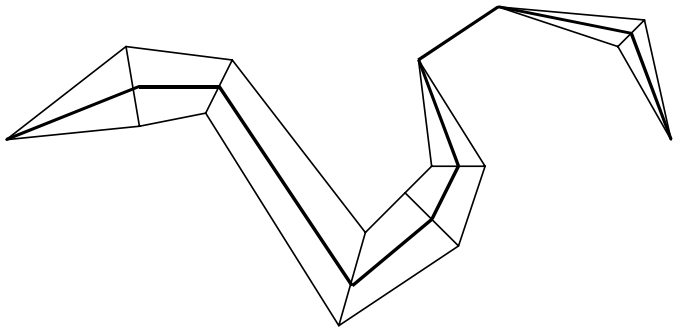


Figure 4.13 *Medium complexity uncertain curve*

can clearly see from this figure how the crossing lines determine the shape of the support of the uncertain line.

The only aspects that need to be stored about the crossing lines (hereafter called *CrossCurves*) are the length of the line and the angle between the line and the segment to which it belongs.

Definition 17: The **CrossCurve** is defined as follows:

$$D_{CCur} \equiv \{(a, b) | (a \in CVS \wedge b \in ANG)\}$$

In this formula, *ANG* is an angle. Angles are represented with floating-point numbers.

A line segment in this model may be defined as a single line segment of the central curve and the *CrossCurves* at each end of it. Each line segment contains a probability distribution function indicating how likely it is that the actual line exists in the various parts of the segment. The line segment also contains a probability mass function which applies along the *CrossCurves*. Storing the probability mass function in the segment rather than for the entire curve makes one able to use different functions for different parts of the curve. However, there will be discontinuities in the probability values if the function changes. Therefore, most curves should use a single function throughout the curve.

Definition 18: The medium complexity **uncertain segment** is defined as follows:

$$\begin{aligned} &DM_{USeg} \equiv \{(cc, bc, ec, pc) | \\ &cc \in D_{USeg} \wedge bc \in D_{CCur} \wedge ec \in D_{CCur} \wedge \\ &ProbMass(pc) \wedge \\ &CenteredOn(bc, cc.sp) \wedge CenteredOn(ec, cc.ep)\} \end{aligned}$$

CenteredOn means that the given point is on the middle of the *CrossCurve*. Note that in the advanced model, the uncertain curve as a whole cannot be split into uncertain line segments. The advanced uncertain segments are only parts of the central curve.

Definition 19: The medium complexity **uncertain curve** is defined as follows:

$$DM_{Ucurve} \equiv \{(SS, pe) | \\ SS \subseteq DM_{USeg} \wedge pe \in DA_{Prob} \wedge \\ ContCurve(ss) \wedge \\ \forall a \in SS, \forall b \in SS, a \neq b \rightarrow \neg Ccross(a, b)\}$$

Ccross for uncertain segments is true iff their central curves cross. In Figure 4.13 the *CrossCurve*s at the end have length 0. To model a crisp curve, all the *CrossCurves* should have length 0.

Compared to a crisp line, the *CrossCurves* and function references have an additional cost. A single *CrossCurve* contains two numbers. That means that the two crosscurves in the uncertain segment require four numbers. The advances uncertain segment requires seven numbers. The probability mass function requires one number. Because a crisp line segment can be stored with four numbers, an uncertain line segments takes 12/4=3 times as much space as a crisp one. This is only slightly better than for the advanced model.

The **uncertain line** is defined as in the advanced model except that it uses medium complexity curves.

4.5.4. Uncertain regions

One problem with both the abstract model from chapter 3 and the advanced model presented earlier is that the border of an uncertain region sometimes is not a valid uncertain line. This may be solved by defining the uncertain face in the same way as the crisp face with the exception that uncertain cycles are used instead of crisp ones. The uncertain cycle here is a special case of the uncertain curve in which the start and end points and crosscurves are the same and the curve has the same probability of existence over the entire curve. An example of such a face is shown in Figure 4.14. An uncertain hole in the core can be stored in this model by using an uncertain cycle which is not certain to exist.

Definition 20: The **uncertain cycle** is defined as follows:

$$DM_{UCyc} \equiv \{sc \in DM_{UCurve} | \\ IsCycle(sc.SS) \wedge ConstProb(sc.SS)\}$$

In this definition, *IsCycle* means that the set of uncertain segments forms a cycle. A set of uncertain segments forms a cycle iff both the central line and the outer lines form cycles. *ConstProb* means that all the line segments in the set have the same constant probability of existing.

Because the uncertain parts of the object are uncertain lines and these cannot have holes, this model cannot store holes in the support. It also cannot store a face with multiple core regions.

However, it is very easy to determine the probability function at individual points as well as iso-lines of probability. In Section 4.7 it will be shown that this is even easier for these regions than it is for medium complexity uncertain lines.

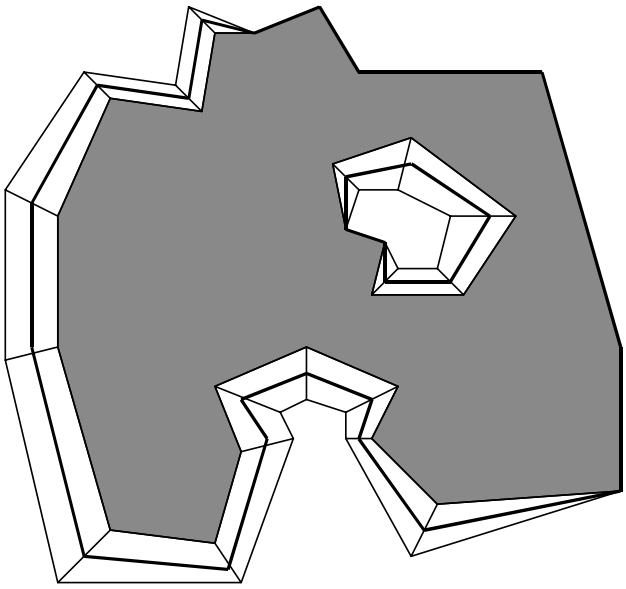


Figure 4.14 *Medium complexity uncertain face*

The uncertain face also uses a different kind of probability function than an uncertain curve. In the grey area enclosed by the cycles, the probability of existence is always 1. Inside the supports of the uncertain cycles, the probability of existence is the sum of the probability mass function of the uncertain curve taken from outside and inward. To avoid having to compute this, it might be better to store this sum directly rather than the probability mass function for the curves. If the function of the curves is also needed by the application, both may be stored together.

The definition of the uncertain face is taken from [FGNS00] and modified to use uncertain cycles.

Definition 21: The **uncertain face** is defined as follows:

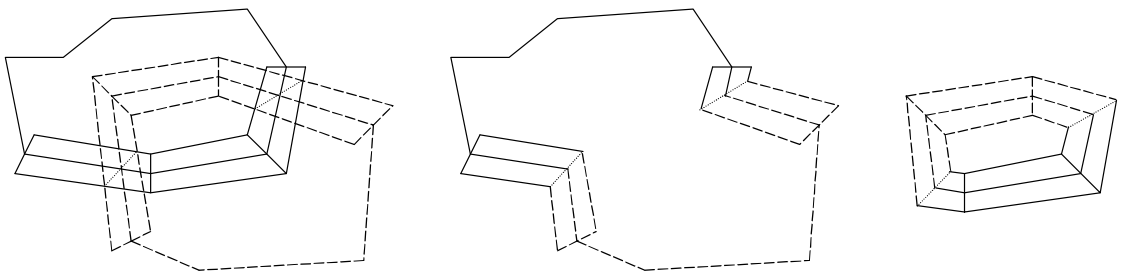
$$
\begin{aligned}
&DM_{UFace} \equiv \{(bc, HS, ps, pe) | \\
&bc \in DM_{UCyc} \wedge HS \subseteq DM_{UCyc} \wedge \\
&ProbFunc(ps) \wedge pe \in DA_{Prob} \wedge \\
&\forall a \in HS \rightarrow EdgeInside(a, bc) \wedge \\
&\forall a \in HS, \forall b \in HS, a \neq b \rightarrow EdgeDisjoint(a, b) \wedge \\
&bc.sc.pe \equiv 1\}
\end{aligned}
$$

In this definition, *EdgeInside* means that all the line segments of *a* are in the interior of the cycle defined by *bc* with 100 % certainty, and *EdgeDisjoint* means that the interiors of two cycles are certainly disjoint. The sum of a probability mass function is a probability distribution function. Therefore, *ps* is a probability distribution function. *Existence* is the probability that an uncertain object exists.

The definition of the **uncertain region** in this model is the same as Definition 13 except that medium complexity faces are used.

To make this model computationally closed under normal set operations, ways of dealing with uncertain curves that cross each other must be introduced. Figure 4.15 shows the union and intersection of two example objects. From this figure, one can see that the results of such set operations contain places in which one has to use just parts of some uncertain segments. The dotted lines from the figure shows where the segments are divided. This line also separates the segments that originated in the two curves.

However, this does not solve the problem when there is one or more *CrossCurves* inside the area in which the lines may possibly intersect as shown in Figure 4.16. This problem may be solved by adding a new type of "line segment" called a *CrossSet* to the uncertain curve. In Definition 19, the *CrossSet* behaves like a *MUSeg*.



a) Original objects **b)** Union **c)** Intersection

Figure 4.15 *Union and intersection of medium complexity regions*

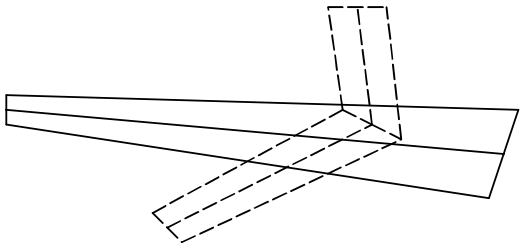
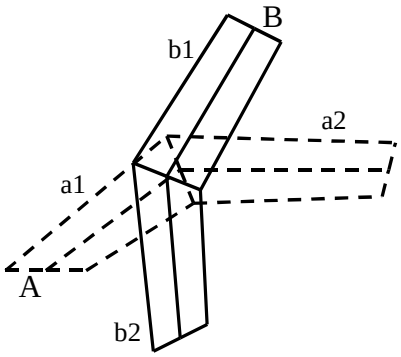


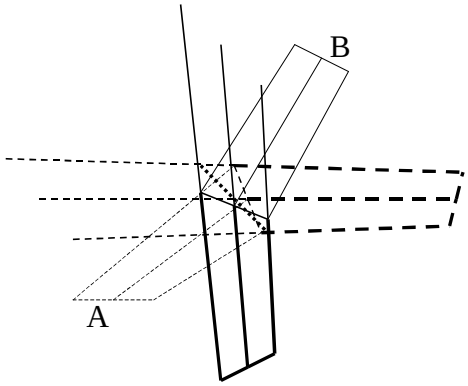
Figure 4.16 *Crossing lines with CrossCurve inside area of intersection*

The *CrossSet* contains all the uncertain segments that might possibly intersect the area in which the two curves may intersect. Figure 4.17 contains an example with four curves in the *CrossSet*. The *CrossSet* is computed as follows:

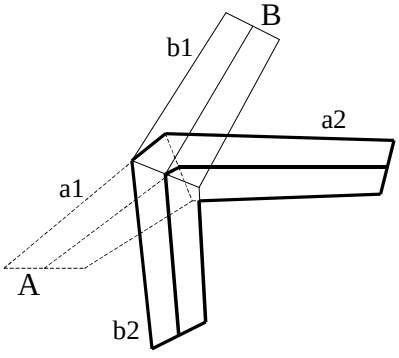
- Label the two curves as curve *A* and curve *B*. For each of these two curves, store all the segments that have supports that intersect the area in which the two curves may cross. Figure 4.17a contains two example curves that cross.



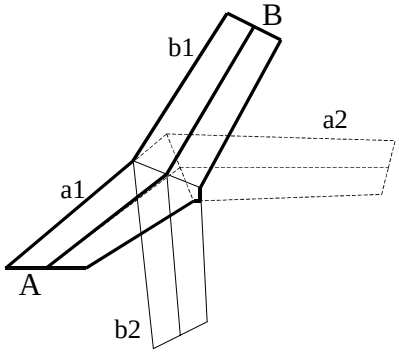
a) The two lines that cross.



b) The separating line between two segments.



c) Alternative result 1



d) Alternative result 2

Figure 4.17 *Constructing a CrossSet*

- For any pair of segments, where one is from curve A and the other from curve B, compute the separating line (the dotted line from Figure 4.15). This may require that one or both segments are extended beyond their normal end. In Figure 4.17b, the separating line between the two segments given with thick lines is computed. Because it is not certain that the two line segments intersect, they must be extended beyond their normal ends with the lines of medium thickness. The resulting separating line is the thick dotted line in the middle of the figure.

- On one side of the dividing lines use the line segment from curve *A*. On the other side, use the line segment from curve *B*. In Figure 4.17c and d, the thick lines indicate the two alternative *CrossSets* that can result from the two example curves.

Whether the result from Figure 4.17c or d is used depends on whether the *CrossSet* was the result of a *Union* or *Intersection* operation in the same manner as this is done for two normal line segments in Figure 4.15.

Definition 22: The **CrossSet** is defined as follows:

$$
\begin{aligned}
&D_{CSet} \equiv \{(A, B, SL) | \\
&A \subseteq DM_{USeg} \wedge B \subseteq DM_{USeg} \wedge \\
&SL \subseteq D_{Segment} \wedge \\
&\forall a \in A, \forall b \in B, \exists sl \in SL \rightarrow SepLine(sl, a, b)\}
\end{aligned}
$$

The function *SepLine* returns true iff the crisp line segment is a separating line between the two uncertain line segments.

One problem that needs to be solved is if two *CrossSets* cross. A simple solution to this problem is to approximate by creating a new *CrossSet* that uses only those line segments that extend outside the area in which the curves may cross and disregarding the two others, which contribute only small parts to the *CrossSet*. In Figure 4.17c, the two segments that extend outside are segments *a*2 and *b*2.

A more complex solution to the previous problem would be to allow the *CrossSet* to contain other *CrossSets* in addition to uncertain line segments. However, this will increase the storage requirement greatly and only give a minor increase in accuracy.

The increase in storage space for an uncertain region compared to a crisp region in this model is the same as for the uncertain curve. This means that it actually costs more storage space to store a medium complexity uncertain region than an advanced one. However, the basic uncertain segment stores some numbers that are not really necessary for uncertain regions. The probability of

existence for each end point in each uncertain segment is unnecessary because all probabilities of existence should be the same for all uncertain segments that are parts of the boundary of an uncertain face. The same reasoning applies to the reference to the probability function along the uncertain segment. If one does not store these, the storage space used is reduced from 3X to 2.25X. This is only slightly more than the advanced model uses.

However, for this slight increase in storage space, we gain the ability to compute alpha-cuts and the probability that a given crisp point is inside the region in an efficient and consistent manner. In Section 4.7 it will be shown that this is very difficult for the advanced model. Additionally, all but the first method for computing the probability functions for the advanced model increases the storage space required to even more than 3X.

4.6. Simple model

This section describes a very simple way of modelling uncertainty in spatial data. To save space, the simple model does not store probability functions but rather assumes a uniform probability distribution or probability mass. This makes computing the probabilities simple but does not allow the more accurate probability computations of the advanced and medium complexity models. This model may be used by those applications that do not require advanced features or that cannot afford much complexity.

4.6.1. Base types

The easiest way to store an uncertain number, regardless of type, is to store a single value, a deviation, and possibly a probability function. An example of such a number is shown in Figure 4.18. This example assumes a uniform probability mass. A number stored in this way takes up twice as much space as a crisp number. This definition covers the **uncertain integer** and **uncertain real**.

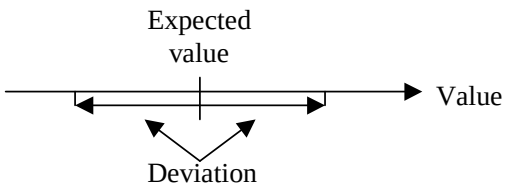


Figure 4.18 *Simple uncertain number*

Definition 23: The simple **uncertain number** is defined as follows:

$$DS_{UNumber(\alpha)} \equiv \{(ev, d) | ev \in \alpha \wedge d \in \alpha\}$$

In this definition, α is the type of number one wants to make uncertain. This type of uncertain number takes twice as much space as its crisp equivalent rather than three times for the medium complexity model. This assumes that the probability function is not stored. A function reference to a probability function would bring the storage cost of this type to the same level as that for the medium complexity model.

The only way to make the **uncertain interval** simpler is to remove the probability function and instead assume a probability of 0.5 in the area of uncertainty. This reduces the storage cost from 2.5 to 2 times as much as a crisp interval.

The **uncertain range** is the same as in the medium complexity model except that it uses simple uncertain intervals.

The **uncertain Boolean** and **probability** are the same in the simple model as in the medium complexity model.

4.6.2. Uncertain points

The simplest way to store an uncertain point is the way described in [Dut92]. This method stores a point as a central point with a circular deviation of a certain radius and assumes a probability function. (A Gaussian function is assumed in [Dut92].) A point stored in this fashion is shown in Figure 4.19.

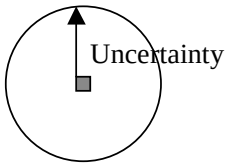


Figure 4.19 *Simple uncertain point*

Definition 24: Simple **uncertain point**:

$$DS_{UPoint} \equiv \{(a, cp) | (a \in CVS \wedge cp \in D_{Point})\}$$

This way of storing an uncertain point takes 1.5 times as much storage space as storing a crisp point because it needs one additional number and a point requires two numbers. A crisp point can be stored by setting *r* to 0.

The **uncertain points** set type is the same as in the medium complexity model except that it contains simple uncertain points.

4.6.3. Uncertain lines

The simple **uncertain curve** and **uncertain line** are based on the medium complexity ones. An alternative would be to store the uncertain line as a series of uncertain points as described in [Dut92]. This would yield a rather strange-looking line with bulges where each point is. However, if the uncertain line is measured as such points, this is a perfectly valid model.

Which of these two models is the most appropriate depends on the measurements used. If these are points, they could be directly stored in Dutton's model. However, if the model is derived in another way, our model is better for the reasons given in Section 4.3.

One natural restriction to the medium complexity uncertain curve and line is to require that the angle of the *CrossCurves* be the same with regard to both the incoming and outgoing line segments. In this case, the angle would not need to be stored. Additionally, one can save some space by not storing a probability function or a probability mass function and by storing a single number that indicates whether this segment is uncertain to exist or not rather than two probabilities in the end points. This would save five numbers for each line segment and make the storage required 1.75 times that of a crisp curve rather than three times. This is slightly more than the 1.5 times that a Dutton style uncertain curve would require, but we retain the ability to store uncertainty about the length of the curve, something that the Dutton model lacks.

4.6.4. Uncertain regions

The **uncertain cycle**, **uncertain face** and **uncertain region** are defined in an identical manner to the medium complexity one, except that simple uncertain curves are used. This means that the extra storage space used for an uncertain cycle, face or region compared to a crisp one is the same as for the simple uncertain curve. An extension to Dutton's model would be to store a region as a set of closed curves as in his model. The reason that we do not choose this is the same as for the simple uncertain line

4.7. Storing and computing probability functions

Different means of storing and computing probability functions for use in a vector model will be discussed in this section. The first section contains a discussion of how to store the probability functions. The second, third and fourth sections will describe various ways of computing the probability functions for the advanced model, and the last section will discuss how to compute the probability functions for the simpler models. This section will focus mostly on the

advanced model, because computing probabilities and iso-lines are more complex in this model than in the medium and simple models.

Another goal of this section is to find ways of using one-dimensional functions to compute the probabilities or probability densities rather than two-dimensional ones. This is done because one-dimensional functions are much easier to define for the user of the system and much easier to store and compute.

4.7.1. Storing probability functions

There are several alternatives as to how functions should be stored. The simplest alternative is to provide a set of predefined functions which are defined symbolically. A good set of functions might be the constant function for the cases in which nothing is known about the probabilities, the linear function, and the Gaussian function. If one needs sums (such as if one wants to store both the function for a medium complexity uncertain region and the function for the lines that make up its border), one might want to store the symbolic integrals of these functions as well. By storing these symbolic integrals along with the functions, they can be accessed fast and do not need to be computed.

Another alternative would be to let the user define the functions and store these as separate objects in the database. This would require the ability to store these functions somehow. A method for storing approximated probability mass functions in databases is described in [DS98]. In this method, each function is defined by a certain number of blocks with the same probability, but covering areas on the number line of possibly widely varying length. This means that the function is not stored symbolically, but that some values computed with the function are stored instead. These stored blocks together form a kind of step function.

Which of these methods one should choose depends on the data. If one needs detailed probability distributions for a few functions, the first method should be used because it allows more precise calculations. If one needs to be able to register a lot of different functions, especially if one wants the user to be able to register his/her own functions, the second approach should be used because it allows easy storage in the database. The second method also benefits from increased speed, as it is far cheaper to look up a table of blocks than it is to compute a complex formula on the fly.

The probability functions must also be scaled so that they always yield the proper sums regardless of the physical size of the object. This applies to the probability masses of points and along the gradient lines. The sum of the prob-

abilities of the point or line being at any of the possible coordinate values must be one[1]. For uncertain curves one can just divide the function results by the length of the curve. For an uncertain point, however, this is a problem because here the sum over the entire plane must be one, not just over a single line. Possible solutions to this problem are discussed in each method for computing probability functions.

The probability mass functions should be functions that accept input values from 0 to 1 and the sum of the probability of all possible positions must be 1. The actual distances should be scaled to be between 0 and 1. The values should be scaled to the number of possible values. [DS98] describes a way to do this for temporal uncertainty.

4.7.2. Computing values using distance from centre and edge

The following is a method for computing the values of the probability functions for all the spatial types in the advanced model. In this method, the relationship between the distance from the point to the support, *ds*, and the distance from the point to the core, *dc*, is used to compute the probability (for regions) or probability mass (for lines and points). For regions, this is the probability that the region contains this crisp point. For lines, this is the probability that the line goes through this crisp point. For points, this is the probability that the point is in this position. The value is computed using Equation 1. The distance to the core or support is the minimum distance that can be achieved by only going through points that are in the support of the object. Examples of these distances for objects with and without holes are shown in Figure 4.20.

Equation 1: $P(p) = F\left(\frac{ds}{ds + dc}\right)$

This approach is simple for computing the probability of a single point, but it has several disadvantages for other operations. First of all, it cannot be used reliably to compute iso-lines of probability. It is not feasible to compute each and every point along such an iso-line directly. An alternative would be to compute only some points and draw straight lines between them. The problem with this is that it may introduce inconsistencies. A point may be outside the alpha-cut returned, but when one asks about the probability of that particular point it may be on or slightly above the alpha-cut threshold. In Figure 4.21, the point *P* lies outside the computed alpha-cut but has an actual function value above the alpha-cut threshold. Additionally, there is no simple way to choose sample points which would yield small inconsistencies.

1. Because the existence is stored separately, this probability should not be less than one.

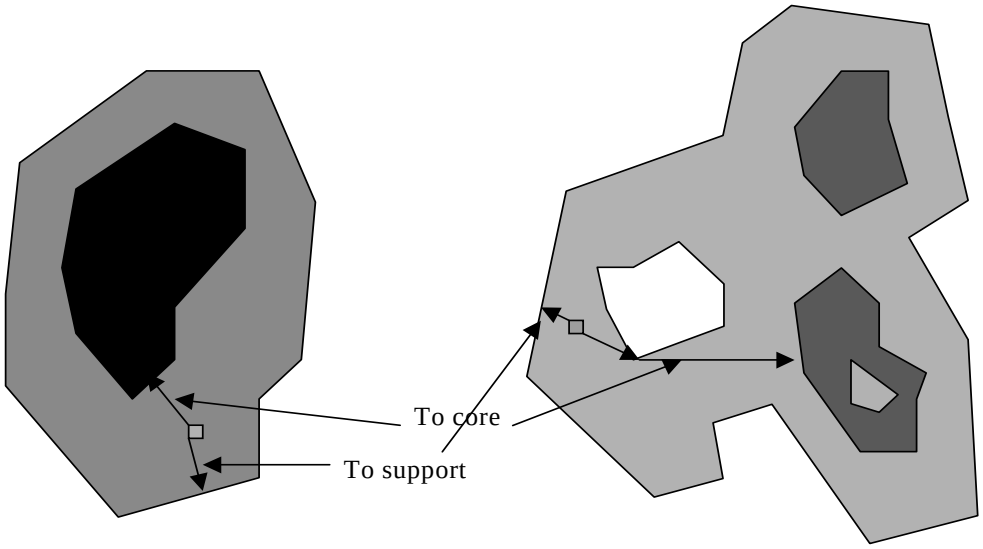


Figure 4.20 *Computing probabilities for uncertain region*

Another problem is that there is no easy way to compute the sum over all the potential points of an uncertain point to check whether it is above one or not. If the function is stored symbolically, an analytic integration may be performed. For a step function one can compute the areas with each value and sum it. If the function is not a step function and an analytic integration is impossible, it is virtually impossible to compute this sum. For a continuous function, an approximation may be computed by taking sample points, constructing a Voronoi diagram from them and the edge, and use the value in the point over the entire Voronoi cell containing the point. An alternative is to compute a set of iso-lines and use set values in each of the bands formed. The value used should be that which is in the middle between the two iso-line values. This essentially transforms the function into a step function.

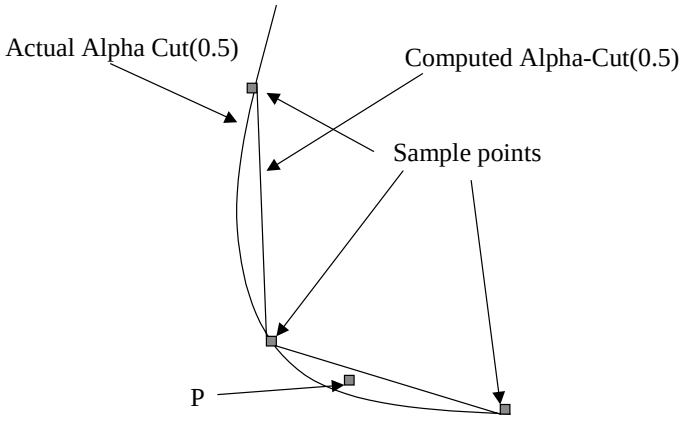


Figure 4.21 *Inconsistency of alpha-cut*

For lines one can get an approximate normalization by dividing the result of the function by *ds+dc*. This assumes that the gradient line through a given point follows a path consisting of straight lines. For instance, the gradient line that goes through point *A* in Figure 4.22 is assumed to follow two straight lines, one of which goes through point *B*. This is not a perfect normalization, however, because the gradient lines will not really follow this path. If one followed the path through *A*, point *B* would have a fairly high value, but if one followed the shortest path through point *B*, point *B* would have a value about as low as that of point *A*.

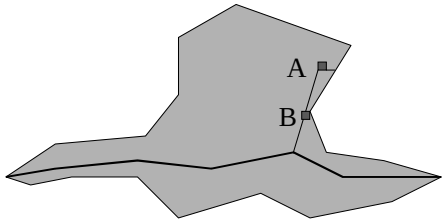


Figure 4.22 *Uncertain line with bulge in support*

This method of computing probability functions does not incur any additional storage cost.

This method of computing probability functions can be used for all spatial objects, but does not really support operations like alpha-cut well. One of the other methods should be used if possible.

4.7.3. Triangulation between core and support

If the spatial object does not have holes, and the support does not have concavities that are not shared by the core, a solution to the problems of the previous method is to create a triangulation between the core and the support like that shown in Figure 4.23. This may be done for regions by using the algorithm for creating a sliced representation found in [TG01]. For lines, a modified version of this algorithm may be used, and for points, lines may be drawn from all the corners in the support to the central point.

The reason that holes and concavities in the support cannot be handled is that some of the new lines that are inserted may be partially outside the object itself. There may be some cases with concavities in the core in which some lines will go through the core as well. In these cases, the algorithm will yield strange results.

The probability function is computed over the lines of the triangle. Figure 4.24 shows an example triangle containing a point. For the point *p* in Figure

4.24, draw a line that passes through both *p* and *a*. For a triangle with a line along the support, *a* is the point that is on the core. For a triangle with a line along the core, *a* is the point that is on the support. Then the function may be normalized so that the sum over this line from the core to the support is one and then computed for point *p* using Equation 1 as shown in Figure 4.24. For regions, the iso-lines only need to be computed along the triangle edges, and straight lines drawn between them. This will yield a consistent result. For any point along this straight line, this method of computing the function will yield the same result. This only works for lines when the probability functions are linear. This is because any other function would cause the iso-lines to become curved.

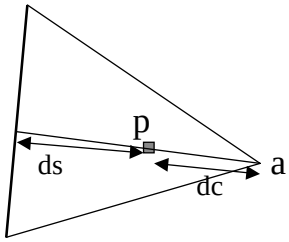


Figure 4.24 *Computing probability function for arbitrary point using the triangulation method*

For uncertain points, the function may be normalized by computing the sums in each triangle and summing all the triangles. If the function is a step function, the following method may be used to compute the sum of all the probabilities:

- Find the points corresponding to the start and end of each step of the function on the two triangle legs ending in the core point.

- Draw a line between these points.

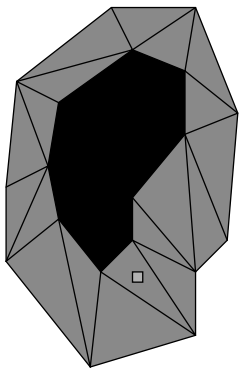


Figure 4.23 *Triangulation between core and support*

- Compute the area of the resulting polygon with four vertices.
- Multiply this area with the value of the function in this block.

Computing this for all the blocks yields the integral over that triangle. Repeat this for all triangles to get the integral of the function. This number should then be stored in the uncertain point so that the function values may be divided by it later.

One problem with this model for uncertain lines is that the function values in the points along the core line in which several triangles have their end points are undefined. This is because many lines meet at this point. These lines may have different lengths and may therefore be normalized differently. Therefore the functions along them will yield different values for the point along the central curve.

Another potential problem is that iso-lines of non-linear functions are not straight because of normalization. However, in Section 4.10.2 it will be shown that iso-lines are not that useful for lines and points, but what is useful is relative probability. This can be determined for lines and points in the same way as iso-lines for regions. Using relative probabilities also removes the problem of inconsistent probability values described in Section 4.7.2.

This model requires more storage space than the method from Section 4.7.2 because the additional lines in the triangles must be stored. A triangulation requires one additional line for each point in either the core or the support. These lines may be stored either as references in each point or as a list of references linking two points together. Assuming a reference takes as much space as a single number, this means that the lines take about half as much space as all the points on both the core and the support.

For points, this does not increase storage space as all the lines are assumed to go from the support to the central point. For lines, the triangulation between the core line and one side of the support would require one line segment for each line segment in the core line and support. Because the points are already stored, only two references are needed to store these lines. Therefore the triangulation between the core and one side of the support takes as much space as the core line, which takes as much space to store as a crisp line. Because this cost applies separately to each side of the support, the storage required for uncertain lines increases by +2X the space of a crisp line.

For regions, there must be a triangulation between the core cycles and the support cycles. The triangulation between these two cycles takes as much space as a single cycle with the same number of points as either the core or the sup-

port because there is one additional number per point and there are two points, one on the core and the other in the support. Therefore, the storage required for uncertain regions is +1X the space of a crisp region.

4.7.4. Storing boundary area as simplical complexes

The major problem with the second model is that it can neither store holes nor some concavities. This is because it may not be possible to create a triangulation that makes sense when there are holes in the support without corresponding holes in the core.

In some cases one may know the probabilities of the object existing in certain points precisely. This is the case in a lake with varying water level, because the probability that there is water there is directly related to the height of the point. In these cases one may use these points as additional base points for a triangulation as shown in Figure 4.25.

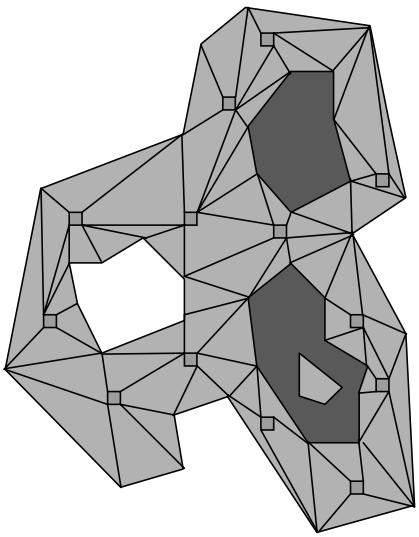


Figure 4.25 *Triangulation with extra points*

The boundary area (the area that is in the support but not in the core) is then stored as a set of triangles instead of with just the core and support. This means that the storage cost of this variant may be much higher than for the previous model, as there may be any number of measured points.

This approach solves the problem of holes if there are enough measured points. Enough in this sense means enough to create a triangulation that makes sense, that is, all triangle edges are inside the support of the object. Whether any given set of measured points is enough may be difficult to test without attempting to create the triangulation. This type of triangulation is created by

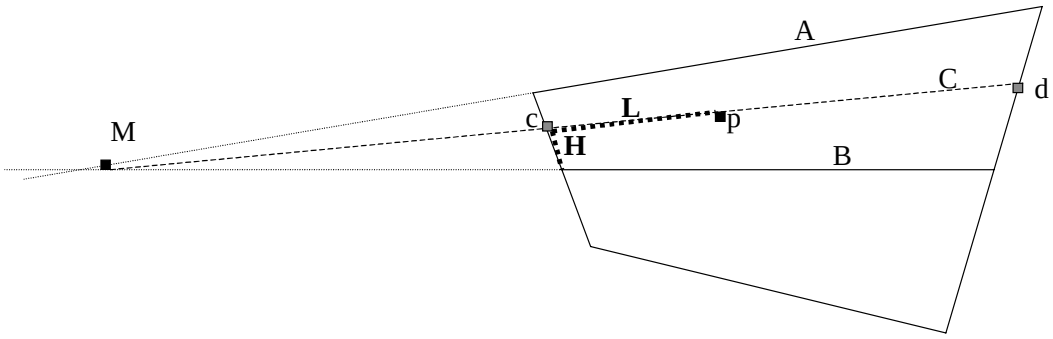


Figure 4.26 *Computing probability values for medium and simple uncertain segments*

drawing a line from each corner in the core or support to the closest point. This may be one of the additional base points or another point on either the core or the support, whichever the starting point is not on.

To use this approach for large data sets, the measured points must be available in advance. In many cases, such measured points are not available or the lack of measurements is itself a source of the uncertainty. In these cases this approach is not useful.

The user may create “measurement points” when storing data and use some other form of interpolation to get the values in those points. However, this requires a lot of user interaction with the system. It also requires a very skilled user because one has to know which points need to be computed. It is also not feasible for temporal data where one may have hundreds of snapshots for the same object, each of which may need its own measured points.

Another problem with this method is normalization. For uncertain points, the method described in Section 4.7.3 may be used. For uncertain lines, however, it is not so simple, because it may be impossible to determine where the gradient lines go. The function must be normalized by dividing it by the length of the gradient line, so if its length is unknown, the function cannot be correctly normalized.

4.7.5. Computing probability functions for medium and simple models

For the **lines** and **regions** in the medium complexity and simple models, computing the probability functions is easy. In the end points of each uncertain line segment, the probability is computed along the *CrossCurves*. In the interior, the function value for a point *p* should be computed as follows for regions:

1. Take the lines *A* and *B* (the central line) from Figure 4.26 and compute the point *M* in which they intersect.

2. Compute the line *C* from *M* to *p*.

3. Compute the point *c* or *d* where the line *C* crosses the *CrossCurves*.

4. Compute the distance *H* between the point *c* and the central curve *B* along the *CrossCurve*.

5. Find how long *H* is compared to the *CrossCurve*. This ratio determines the value of the probability distribution function for point *p*.

Two special cases need to be considered. The first is if the lines *A* and *B* are parallel, because in this case there is no meeting point *M*. In this case, the line *C* should be parallel to *A* and *B* and go through *p*.

The second is if the line *A* has length 0. In this case the line *A* becomes the point *a*. Compute the line *K* that passes through both *a* and *p*. *H* is now the distance from *p* to the line *B* along *K*, and the ratio from 5) uses the length of *K* instead of the length of the *CrossCurve*.

For uncertain lines, an additional step is needed to compute the probability function along the central curve. For a region, this is always 1 and therefore one does not need to compute it.

6. Find how long the distance *L* is compared to the distance between *c* and *d*. This ratio is used to find the probability that the curve exists at point *p*.

For lines, one also uses a probability mass function along the gradient lines rather than a distribution function. This mass function needs to be normalized to yield a sum of 1 regardless of the length of the gradient. Instead of using *H* and the length of the *CrossCurve* to determine the probability function, one has to find the length of the gradient that passes through *p*. The angle between this line and line *B* is linearly interpolated. Because the length of *L* is known, this ratio is easy to determine. One then needs to find where this gradient crosses *A* and *B*. Then the distance between *A* and *B* along the gradient and the location of *p* on the gradient is used to determine the value of the probability mass function.

This method of computing probabilities has several advantages compared to other models. In Dutton's model for uncertain lines, one would need to take the sum of the probabilities of all the possible lines that might cross *p* to determine the probability that the line was in *p*. This is much more costly than the procedure outlined above. In the advanced model there is the problem of

consistency from Figure 4.21, which is solved for the medium and simple models by this approach.

For **points**, the normalization in the medium complexity model is done as described in Section 4.7.3. In the simple model, one divides the probability mass function by the area of the support.

Computing iso-lines of the probability[1] is also simple in the medium complexity and simple models. The iso-line passing through point *p* in Figure 4.26 is line *C*.

4.8. Representing time and temporal uncertainty

An object may have uncertainty in the timing of events in addition to the spatial shape. This section will discuss how to represent time for uncertain spatial data as well as what the effect of uncertainty is on how spatiotemporal data should be represented.

4.8.1. The temporal types

There are two temporal types: Time instants and time intervals. Types for normal numbers and intervals have been described in Sections 4.4.1, 4.5.1 and 4.6.1. Time may be stored as either an integer or a floating-point number. In this model, time is stored as a fixed-precision number to make its representation similar to the representation of the spatial dimensions.

4.8.2. Storing spatiotemporal objects

In [FGNS00], spatiotemporal data is represented as a set of time slices. In each time slice the object evolves using a simple function. For a point, this simple function is linear movement. Thus a temporal point is stored as a set of disjoint time slices each of which contains a straight line in space-time. This line describes the linear movement of the point in time.

A moving line is a set of line segments, each of which consists of two end points. As noted in Section 4.3, this model does not allow rotating lines. Instead it stores a rotating line segment as two lines each of which becomes a single point at one of the ends of the time slice. Such anomalies are only permitted at the ends of the time slices, not inside them. A region is stored as a set of such lines. A method of interpolating between crisp regions to generate this storage format is described in [TG01].

1. Iso-lines of relative probability for lines and points.

The problem with using this approach for uncertain data is that the snapshots themselves may have temporal uncertainty, that is, one does not know precisely when they were taken. This means that either the sliced representation must be extended to deal with time slices with temporal uncertainty, or the temporal uncertainty must be eliminated by an interpolation algorithm so that the time slices may use crisp time instants as borders.

Storing temporal uncertainty for the object as a whole can be done in a similar manner as storing spatial uncertainty for uncertain lines. For each time slice, one can store an advanced uncertain interval (regardless of whether the advanced model is used for other types). This uncertain interval is considered a time interval and stores the start and end times for the time slice as well as the probability that the object exists at different times in that time slice.

4.8.3. Interpolating between snapshots with uncertain time

In this section, several possible algorithms are proposed to interpolate between snapshots with temporal uncertainty in such a manner that the resulting sliced representation can be stored using crisp time slices.

Note that all the algorithms from Sections 4.8.3.1 to 4.8.3.4 cause temporal uncertainty to be turned into spatial uncertainty. For instance, if one has a spatially crisp moving region, but does not quite know when the snapshots were taken, then the core of that region becomes smaller than its support because the support is a conservative estimate and the core is a progressive one. Thus a spatially crisp region with temporal uncertainty becomes a spatially uncertain region without temporal uncertainty, but possibly with an uncertainty about whether or not it exists at certain times.

4.8.3.1. Interpolating the support of uncertain objects by storing uncertain time slices

The support of all uncertain spatial objects is a face or region, which is composed of a set of cycles. Therefore, the support of a spatiotemporal version is a set of moving cycles. This subsection will therefore discuss how one can create a representation of an uncertain moving cycle given uncertainty about the times of the snapshots. Although interpolation is not the main subject of this chapter, it must be discussed in this context because the storage method used will depend on the interpolation method used.

When one knows for certain when the two snapshots were taken, the algorithm from [TG01] may be used to create the sliced representation from [FGNS00] of the object. With uncertain time, one must take into consideration

the various times the snapshots could possibly have been when creating the interpolation.

If one stores the temporal uncertainty for each time slice, one does not quite know where the lines of the border in space actually go. Figure 4.27 shows one example of two interpolations based on different times for snapshots 1 or 2. If one stores the temporal uncertainty in the time slices, the computer does not know which of these to use. Even for a non-temporal spatial query, the results would depend on the temporal uncertainty, and this aspect would have to be computed at query time.

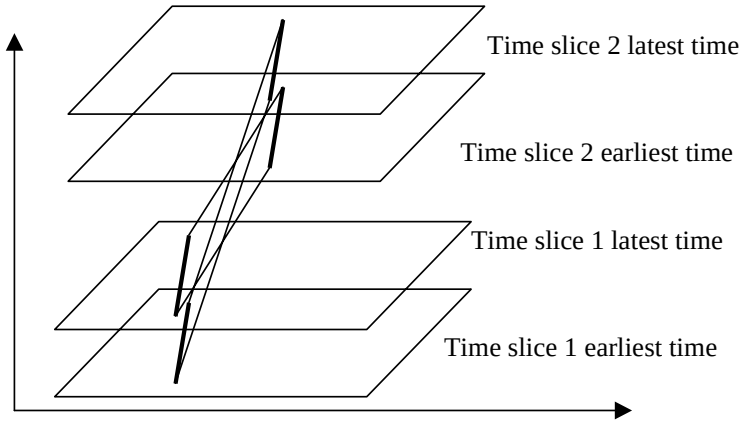


Figure 4.27 *Interpolations with different times*

As Figure 4.27 shows, the two interpolations become different at different time instants, but are both versions of the same interpolation. They are just displaced in time.

Because the support of the uncertain object should contain all points in which the object might possibly be, the support should be a conservative estimate of where the object may be. This means that it must include all points that are parts of the support for any combination of times for the snapshots.

To achieve this, we choose two sets of times, *A* and *B*. In *A*, both snapshots are at their earliest possible time. In *B*, both snapshots are at their latest possible time. In Figure 4.28 there is an example of two time slices with uncertain time, and the crisp time instants from each that would be included in *A* and *B*.

The snapshots themselves are really the same regardless of which time they are assumed to be in. Therefore the interpolations for *A* and *B* are the same except that they are displaced in time. In Figure 4.28a the two lines to the right

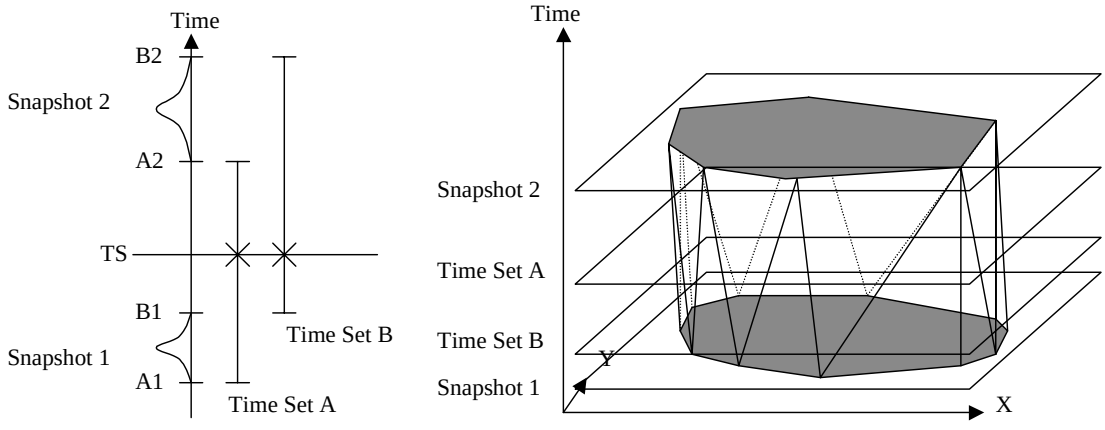
of the time axis are the crisp time intervals that are represented by *A* and *B*. For a given time instant *TS*, one checks where the time instant is on these lines. The relative position of *TS* on each of those two lines is then converted into a number between 0 and 1. This number is where the snapshots corresponding to time sets *A* and *B* should be taken in the sliced representation between the time slices. This is shown in Figure 4.28b.

However, because the snapshots could also be at intermediate times, simply taking the union of these two is not enough. This is because there may be concavities in this union that the object may have been in if the snapshots actually are from intermediate times.

Label the time instant in the sliced representation given by time set *A* as t_A and the instant given by time set *B* as t_B. To check for this movement, one can look at the movement of the points that make up the border of the sliced representation in the time from t_A to t_B and remove any concavities that any of these points move through in this time. These concavities may be closed because if the point has moved through it, the support of the object is there in some combination of times of the two snapshots. One example of this process is shown in Figure 4.29.

The following procedure may therefore be used to find the support of an uncertain spatiotemporal object at a given crisp time instant.

- Create a normalized[1] interpolation between the two snapshots when they are first stored. All the possible interpolations are versions of this



a) Time sets *A* and *B*. **b)** Sliced representation with indicated times.

Figure 4.28 *Interpolating with between snapshots with temporal uncertainty*

1. Normalized here means that the interpolated version starts at time 0 and ends at time 1.

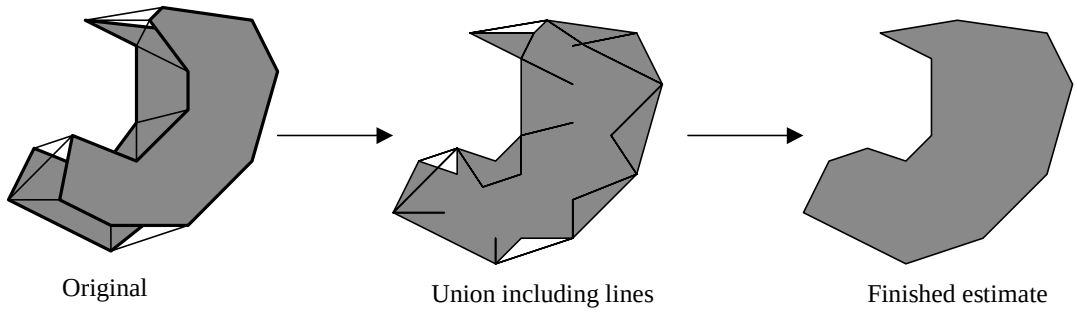


Figure 4.29 *Creating conservative estimate for support*

one with different start and end times because one always interpolates between the same snapshots.

- At query time, find the normalized times corresponding to the following two cases:
  - Both snapshots are at the earliest possible time (t_A)
  - Both snapshots are at the latest possible time (t_B)
- Take the part of the sliced representation that goes between times t_A and t_B.
- Remove the time dimension from all the lines, including those that depict movement of points[1] and are not lines in any of the snapshots.
- Remove all lines and parts of lines that are inside the resulting object.
- The remaining line segments form the outline of the support.

In this method a lot of computation is delayed to query time. This means that it is faster to store the data, but slower to query them. For a write-optimized database, this would be a good solution. However, most databases are read-optimized because there are many more queries than writes.

Only time instants, not time intervals can be queried when using this method. This is because the computation would then have to be performed for all the possible time instants in the time interval.

It is therefore usually better to use an interpolation algorithm that eliminates the need to store temporal uncertainty in each time slice. The time slices will then be crisp, and few computations about temporal uncertainty need to be done at query time.

1. See Figure 4.30

4.8.3.2. Interpolating the support of uncertain objects by taking conservative estimates

One solution is to create a time slice which is a conservative estimate of the real object. This is done by first assuming that all the snapshots are at the expected time and then changing the snapshots so that they are conservative estimates of the real snapshot. This can be based on the following procedure.

- For each snapshot there is a period of uncertainty. Take the following interpolations:
  - This snapshot and the earlier snapshot with the snapshots at the latest possible time.
  - This snapshot and the later snapshot with the snapshots at the earliest possible time.
- Take the combination of the first interpolation at the earliest time of this snapshot, the second interpolation at the latest possible time, and the snapshot itself using the method described in Section 4.8.3.1.
- Use this value for the entire period of uncertainty
- Interpolate between these new snapshots in the areas of uncertainty.

The main difference between this method and the method in Section 4.8.3.1 is that the estimates of where the object might be are constructed when the data are stored rather than when the user asks a query.

This method gives a conservative estimate of the support of the object. The advantage of this method compared to the one from Section 4.8.3.1 is that

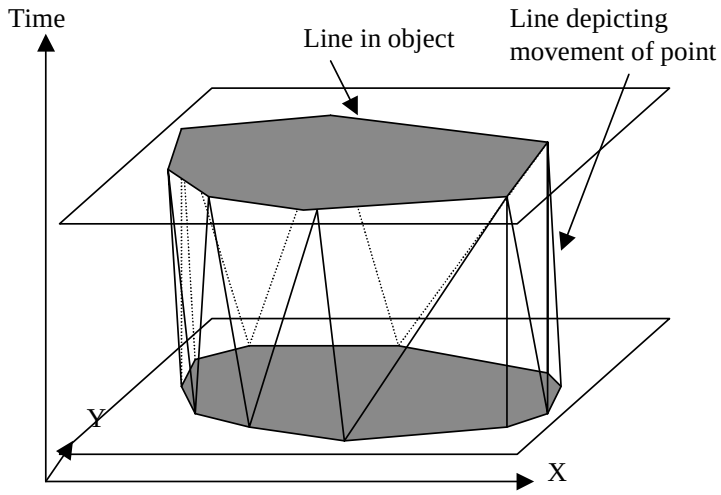


Figure 4.30 *Different types of lines in 3D polyhedral version of sliced representation*

it requires much fewer computations at run-time, and that it can be used for uncertain time intervals as well as time instants. The disadvantage is that the estimate is less accurate, and that this method requires more storage space. An analysis of the storage requirements for this approach is given in Section 4.8.4.

4.8.3.3. Interpolating the support of uncertain objects by assuming constant values

A very simple method which might be good enough in certain cases is to use the snapshot itself as the object value in the entire period of uncertainty, and use the normal interpolation method for objects without temporal uncertainty between the snapshots. This method is an almost trivial extension to a system which already uses the sliced representation, but is less accurate than either of the other two representations because it does not guarantee that the support of the object contains the object. In most cases this discrepancy is really small, but if the object changed significantly and the periods of uncertainty are large compared to the periods between the snapshots, this method yields very inaccurate results.

Note that none of these three algorithms work if the times of the snapshots are so uncertain that the periods of uncertainty overlap. To do this with any degree of accuracy, one would have to consider not only the previous and next snapshots, but potentially others as well. When interpolating between two snapshots, one would have to consider all snapshots that have periods of uncertainty that overlap with the periods of uncertainty of either of the two snapshots under consideration.

4.8.3.4. Interpolating the core of an uncertain object

The core of an object is often a different type of object than the support. For instance, the support of an uncertain curve is a crisp face while the core is a crisp line. Therefore other techniques must be found to interpolate between the cores.

The easiest case is the uncertain **region** because its core is also a region and therefore the algorithm for support can be used. The only difference is that the core should be a progressive estimate instead of a conservative one. This is because a point should only be a member of the core if it is part of the core for all possible times of the two snapshots.

This progressive estimate can be produced using any of the methods from Sections 4.8.3.1 to 4.8.3.3. However, the progressive estimate for the methods described in Section 4.8.3.1 and Section 4.8.3.2 must be computed differently. Instead of taking the union of times A and B, one must compute the intersec-

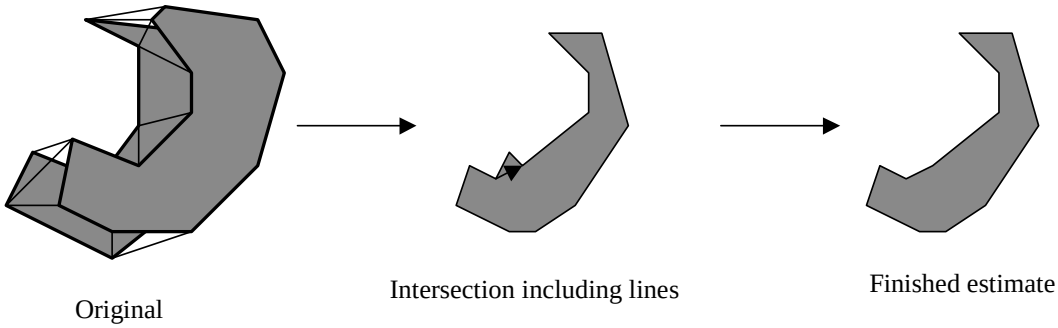


Figure 4.31 *Creating progressive estimate for the core of an uncertain region*

tion. Rather than closing concavities by adding to the interpolation, one must remove outcroppings by subtracting from the interpolation. Any point moving through an outcropping shows that at some intermediate set of times the outcropping was not part of the core. One example of this process is shown in Figure 4.31.

The following procedure may therefore be used to find the core of an uncertain region:

- Let *A* and *B* be the two interpolations described in Section 4.8.3.1.
- Let *C* be the intersection of *A* and *B*.
- Remove the time dimension of all the lines as described in Section 4.8.3.1.
- For any of these lines crossing *C*, do the following:
  - Go along the crossing line in the direction of time.
  - Check the curvature of *C* at the end points of the line.
  - If it curves to the left, remove the part of *C* that is to the left of the line.
  - If it curves to the right, remove the part of *C* that is to the right of the line.

After this procedure, *C* is the progressive estimate of the polygon.

For the cores of **lines** and **points**, one does not have conservative and progressive estimates. For a point, the simplest solution is to use average values. That is, use the expected time of the snapshot when interpolating the central point. Because the support is a conservative estimate, one is always guaranteed that this central point is inside the support.

A more complex solution is to take the average of the core as it would have been with the snapshots at different times. This works like the method described in Section 4.8.3.1 except that the average between the two values is used rather than their combination.

The average is constructed using a modified version of the algorithm for interpolating between two regions from [TG01] and then using the value in the middle rather than trying to find a progressive estimate.

These solutions for uncertain points can be used for uncertain curves as well with one modification. The central curve of an uncertain curve extends all the way to the edge of the support in the definitions of the uncertain curve. However, when the support is a conservative estimate and the central curve uses the expected times, it is no longer guaranteed to do this. Therefore the last segments must be enlongated until they reach the edge of the support. Alternatively, an additional line segment may be inserted at the end and the probability function for the segments adjusted accordingly.

All the methods described in this section assume that the support is the conservative estimate generated by one of the methods from Sections 4.8.3.1 and 4.8.3.2. Because the method in Section 4.8.3.3 does not produce a conservative estimate, one is not guaranteed that the core is always inside the support if one combines the methods described here with that method. The best method to use in that case is to use the method from Section 4.8.3.3 for the core as well as for the support. Because the method from Section 4.8.3.3 does not depend on an algorithm for interpolating regions, it can be used equally well for lines and points.

4.8.3.5. Interpolating the probability function

When creating an interpolation between two snapshots with uncertain time, one also needs to interpolate the probability values of the points in between the snapshots.

The mathematically correct solution would be this: Define a function as follows: For each possible set of time instant in which snapshots 1 and 2 each could have been generated, compute the probability of the point under consideration being inside the region or on the line or point. Multiply this by the probability masses of snapshots 1 and 2 each being at those time instants. Take the symbolic double integral of the resulting function over all the time instants in which each of the snapshots 1 and 2 could possibly be.

This is a very complex solution and works only if the computer knows the symbolic integral of this compound function and can compute a mathematical

function indicating whether the point is inside the polygon for a given combination of times or not. In most cases the probability function will not change between snapshots. A simpler solution is to use the normal method for computing the probability function for the non-temporal version of the object under consideration for the estimated value[1] at that time instant.

A particular problem that occurs when creating the representation of a crisp region with uncertain time is how to compute the probability values. Using the mathematically correct method above, the probability function becomes an advanced symbolic integral. The second method is not usable because crisp objects do not necessarily have probability distribution functions. A possible solution would be to assume a simple function such as a linear function in this case.

4.8.4. Storing the sliced representation on disk

Storing the interpolated version of an uncertain spatial object using the methods from either Section 4.8.3.1 or Section 4.8.3.3 does not require any additional storage space. For the method from Section 4.8.3.1, the normalized interpolation as well as the uncertainty in the timing of each snapshot are stored. This time uncertainty requires storing an uncertain number. Comparing the storage required for an uncertain number with that required for the other types, one finds that this takes negligible space compared to all spatial objects in all models except simple uncertain points. In the simple model, an uncertain number requires two numbers to store, and an uncertain point requires three numbers to store.

The method from Section 4.8.3.3 takes no more space than storing a representation without temporal uncertainty. In the time period in which a given snapshot may have been generated, the snapshot is assumed to be static, so no additional information about these periods needs to be stored.

The method from Section 4.8.3.2 will likely cause the storage requirement to be between 2 and 3 times as much as storing an object without temporal uncertainty. First of all, the interpolated versions used to construct the conservative estimate will contain twice as many points as the snapshots themselves. This is due to the fact that the sliced representation does not tolerate rotating line segments and all segments that rotate become two segments in the intermediate versions. Second, the estimate itself is the combination of two such interpolations as well as the snapshot itself. In the ideal case, much of the original snapshot is used, so the storage space is only slightly increased. In the worst

1. Estimated using any of the methods from Section 4.8.3.1 to Section 4.8.3.4.

case, large portions of both interpolations will be used, which means that storage space is increased four times. In the average case, however, slightly more than twice as much space will be used.

For this method it might also be a good idea to store the actual snapshots in addition to the spatiotemporal structure. This enables one to reinterpolate if additional snapshots are inserted, and also enables the user to ask queries regarding the snapshots as well as the derived spatiotemporal structure. Storing the two snapshots takes half as much space as storing a time slice, which means that 0.5X space is added.

A small disadvantage of this is redundancy, as the database system must check that the snapshots are not updated without also updating the time slices and vice versa. In most cases, the user should only be allowed to update the individual snapshots. After such an update, the database should automatically perform a reinterpolation of the object around the updated snapshots.

For the method in Section 4.8.3.1, the storage requirement is the same as it would have been without temporal uncertainty because the handling of temporal uncertainty is done at query time, not storage time.

4.9. Storing spatiotemporal data with the simpler models

This section describes how the specific data types discussed in Sections 4.5 and 4.6 can be stored as temporal data using the time slice model for storing spatiotemporal data. The spatial data types from the advanced model from Section 4.4 can be stored using the techniques described in Section 4.8 directly. The other models do not need quite as complex storage structures.

4.9.1. Base types

Integers can only change discretely. Crisp integers are therefore constant in each time slice in [FGNS00]. Uncertain integers may also be modelled in this way, and this would ensure consistency with the crisp integers. However, uncertain integers can also be modelled in other ways. For instance, the probability values of each possible number may be linearly interpolated between the two snapshots.

For a crisp integer, this would correspond to a linear interpolation of the value, rounded to the nearest integer value. For the medium and simple models this interpolation is computed by taking a weighted[1] average of the probability functions in each snapshot.

The easiest method for handling temporal uncertainty is to use the snapshot value in the entire period of uncertainty, and to use interpolated values between the snapshots. By using this method, temporal uncertainty does not cost any additional storage space.

Alternatively, the border values of the support may be interpolated. Any of the algorithms presented in Section 4.8.3.1 to Section 4.8.3.3 may be used to interpolate the support of uncertain **numbers**, **intervals** and **ranges** from all the three models. For the core of the uncertain number and interval, any of the methods from Section 4.8.3.4 that apply to corresponding spatial types[1] may be used.

For the advanced uncertain integers, the equivalent of the combination algorithm from Section 4.8.3.1 would be to take the average probability for each value for the two interpolations. Because the border values move linearly between the two interpolations along a single line (the number line), there is no need to consider intermediate values.

Uncertain **Boolean** values are like uncertain integers: They may either be constant or linearly interpolated. A linear interpolation yields “Maybe” in all cases except where both the time slice immediately before and immediately after are both “No” or both “Yes”. To save space, all time slices with the same interpolated value may be joined into one. This takes O(n) time, where n is the number of snapshots.

Some uncertain Boolean values are the results of functions rather than fixed values. One may get more accurate results for these by storing the functional relationship and executing it for the time instant one wants rather than interpolating the Boolean value.

Probabilities may be interpolated in one of two ways. If there is an exact function that produces the number, that function may be computed for any time instant to yield a value. If there is no such function, the value may be linearly interpolated between the two snapshots. If there is temporal uncertainty, the probabilities should be interpolated as if each time instant lies where it has the greatest probability density. The normal interpolation methods for uncertain numbers cannot be used in this case because probabilities and degrees are defined as crisp numbers.

1. Weighted by the time distance of the point to each snapshot divided by the time distance between the snapshots.
1. Point for number, Region for interval.

4.9.2. Uncertain points

For the medium complexity and simple uncertain points, the following method may be used: First, the central point is interpolated as described for the core of an uncertain object in Section 4.8.3.4. Second, each of the distance values are interpolated as uncertain real numbers. This algorithm ensures that the interpolation remains a valid instance of the data type. A problem here is that it may lead to rotating line segments for the medium complexity model, and the sliced representation from [FGNS00] does not allow this.

One possible solution for this is to split the rotating line segment into two line segments as is normal for the [FGNS00]-model. The problem with this is that the intermediate versions are no longer members of the original type, but must at least partially be described with the advanced model for uncertain points. Therefore there is a choice between allowing rotating line segments or allowing the intermediate values to be something other than members of the normal type, which means that a query on an interpolated value must return another type than the one used to construct the interpolation.

This means that the medium complexity uncertain point is not well suited to spatiotemporal information. Because the advanced model for points does not suffer from these problems, it should be used instead, unless one cannot afford the storage space required.

The algorithms for combining two interpolations from Section 4.8.3.1 and 4.8.3.2 do not work with the simple and medium complexity models for uncertain points because one is not guaranteed that the results of the interpolation algorithms in those two sections can be represented as a medium complexity or simple uncertain point. Therefore, only the strategy from Section 4.8.3.3 for interpolating with temporal uncertainty can be used with the simple and medium complexity models. To ensure a consistent treatment of the core and the support, the same method should also be applied to the core.

4.9.3. Uncertain lines

The medium complexity line is fairly simple to extend to the spatiotemporal case. Each line segment in a crisp curve has become a trapezium with a central line segment and two outer line segments. Three examples of these are shown as light grey areas in Figure 4.32. The uncertain line segments may be matched to a single crossing line in the other uncertain curve as shown in the figure. Then take each of the outer lines as well as the core line of the uncertain line segment and match each of these lines to a point on the crossing line. The central line is matched to the central point and the outer lines are matched in

such a fashion that the triangles they form do not cross each other. These three lines and the points they are matched to form triangles to be used in the sliced representation from [FGNS00].

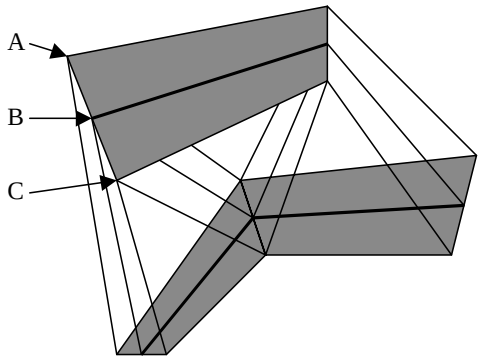


Figure 4.32 *Interpolating medium complexity uncertain line*

This yields a valid sliced representation because neither the segments in the central line nor the segments in the borders of the support rotate. The fact that the crossing lines rotate is not important because these are not returned in queries, but are just aids in storing and computing.

It is in fact impossible to create a spatiotemporal version of the uncertain curve in which these crossing lines do not rotate. Because the angles of the crossing lines are normally not the same in the two uncertain curves, and because a line segment in one snapshot normally has to be matched to a crossing line in the other, the crossing lines may have to rotate.

The intermediate values in this interpolation are also valid members of the non-temporal type. Points A, B and C begin and end on a straight line segment, and they are equally far apart. This is required by the non-temporal model. Because they move linearly between these two positions, they are also on a straight line in all the interpolated time instants between the two snapshots, and they are always equally far apart.

The interpolation method described in this section does not allow the uncertain line segments to be matched to each other unless the grey area in each has exactly the same shape or one allows rotating line segments in the sliced representation. To avoid rotations altogether, line segments are always matched to crossing lines even if the central line segments are parallel.

Note that this method does not work for the simple model where the angles of the crossing lines are fixed by the angles between the two central lines. This is because the crossing lines rotate through the interpolation.

Because these line segments must rotate, it is impossible to create a spatiotemporal version of this model which guarantees a valid member of the non-temporal model for all time instants.

One problem with both the medium and simple models for uncertain lines is that the combining strategy described in Section 4.8.3.1 does not produce results of the non-temporal types. Because the method from Section 4.8.3.2 also relies on these combining strategies, it cannot be used either. This means that only the strategy from Section 4.8.3.3 may be used to interpolate the shape of an uncertain line when the time the snapshots were taken is uncertain. Because the core and the support are stored together in the medium and simple uncertain lines, this method must be applied to the entire object, not just the support.

4.9.4. Uncertain regions

For the simple and medium complexity models, a region is just a collection of uncertain lines. These are modelled and interpolated as described in the previous section.

4.10.Examples of operations

In this chapter, we will describe algorithms for three operations, *Inside*, *Alpha_Cut*, and *Intersection* using the data models described in this chapter. The actual implementation and how to implement more operators is briefly described in chapter 5.

In the algorithms presented here, the type definitions from Table 4.3 will be used.

4.10.1.Inside

The *Inside* operation is used as an example of how to implement a spatiotemporal operator using these models. Although this operation is described for the spatiotemporal case and the other two are described for the spatial case only, we choose to take *Inside* first because it is the easiest. The *Inside* function test whether a given spatial object *A* is inside a given region *B*. In Figure 4.33, some examples of the results of the *Inside* operator for an advanced uncertain point and an advanced uncertain region are shown. This figure uses a purely spatial representation rather than a spatiotemporal one to improve readability.

The algorithm that is given in this section works for all three data models and for all the presented approaches for interpolating with uncertain time. The differences between these approaches lie inside the *intersection* operations and

Table 4.3 *Type designators*[1]

Type designator	Type
N	Number
B	Boolean
I	Time Interval
Po	Point
C	Curve
F	Face
Re	Region
S	Any spatial type
UNIT(X)	One time slice of X
MOV(X)	Moving (set of time slices) X
CX	Crisp X

1. All these type designators are for uncertain types except for CX.

how the *UNIT*(*X*) objects are computed. A time slice of a variable is referred to as a *unit* of that type in the descriptions. The following implementation works if *A* is an uncertain point.

Algorithm *Inside*(*mp, mr*)

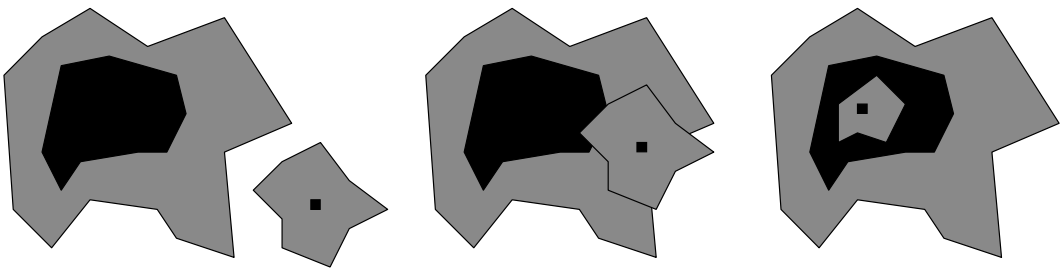
Input: A *MOV*(*Po*) (mp) and a *MOV*(*Re*) (mr)

Output: A *MOV*(*B*) telling for each time instant whether the moving point is inside or outside the moving region

Method:

let $mp = \{up_1, \ldots, up_n\}$ such that the list is ordered by time intervals

let $mr = \{ur_1, \ldots, ur_n\}$ such that the list is ordered by time intervals



Inside = *No* Inside = *Maybe* Inside = *True*

Figure 4.33 *Examples of results of the Inside operator applied to a point and a region*

```
    let ri be an initially empty set containing the following:
        {(i, up, ur) | i ∈ I ∧ up ∈ Po ∧ ur ∈ Re}
    tups := mp
    turs := mr
    while tups ≠ ∅ ∧ turs ≠ ∅ do
        pst := Member_Lowest_Start_Time(tups)
        rst := Member_Lowest_Start_Time(turs)
        psvt := Valid_Time(pst)
        rsvt := Valid_Time(rst)
        if Overlaps(psvt, rsvt) then
            tvt := Intersection(psvt, rsvt)
            tri := (tvt, pst, rst)
            ri := ri ∪ {tri}
        end if
        if End(psvt) ≤ End(rsvt) then
            tups := tups \ {pst}
        else
            turs := turs \ {rst}
        end if
    end while
    let ub be an initially empty set of UNIT(B)
    let mb be an initially empty MOV(B)
    let ri = {ri_1, ..., ri_n}
    for each ri_i do
        ub := Upoint_Uregion_Inside(ri_i.up, ri_i.ur)
        mb := Concat(mb, ub)
    end for
    return mb
end Inside
```

The function *Concat* adds new time slices to an existing sliced representation. It also merges adjacent time slices with the same value (Boolean values are constant in each time slice). The function *Member_Lowest_Start_Time* returns the time slice with the smallest start time in a set of time slices. The *Valid_Time* function returns the time interval in which a particular time slice is valid. The *Overlaps* function returns *True* if the two time intervals overlap and *False* otherwise. The *Intersection* function returns the time interval in which the two given time intervals overlap. The *End* function returns the time instant in which a given time interval ends.

Note that this algorithm for *Inside* is exactly the same as the one presented in [FGNS00]. This means that the entire difference lies in the *Upoint_uregion_inside* algorithm.

A point is only known to be inside a region if the support of the point is inside the core of the region and both point and region are certain to exist. Otherwise, if the supports of the point and region intersect, then the point may be inside the region.

Algorithm *Upoint_Uregion_Inside*(*up*, *ur*)
Input: A *UNIT*(*Po*) (*up*) and a *UNIT*(*Re*) (*ur*)
Output: A set of *UNIT*(*B*) representing when *up* was inside *ur* during the time interval in which both are valid
Method:
let *up* = (*ip*, *cp*, *sp*)[1]
let *ur* = (*ir*, *cr*, *sre*)[2]
$i := ip \cap ir$
$sir := sp \cap sre$
sra := *DefTime*(*sir*)
let $sra = \{sr_1, \ldots, sr_n\}$
let *ub* be an initially empty set of *UNIT*(*B*)
for each sr_i **do**
$ub := ub \cup (sr_i, Maybe)$
end for
cins := *Inside*(*sp*, *cr*)
let $cins = \{cb_1, \ldots, cb_n\}$
for each cb_i **do**
Replace(*ub*, cb_i, *True*)
end for
$isr := i \setminus sra$
$isr := isr \setminus DefTime(cr)$
let $isr = \{isr_1, \ldots, isr_n\}$
for each isr_i **do**
$ub := ub \cup (isr_i, No)$
end for
return *ub*
end *Upoint_Uregion_Inside*

The function *Replace* adds a Boolean unit with the second input as the time interval and the third input as the truth value to the set of Boolean units given in the first input. In the case of overlaps, the part of the overlapping unit from the first input is removed and the unit from the second and third inputs is inserted in its place. This may cause the overlapping unit from the first input to be split into several parts. The function *DefTime* returns the uncertain range of

1. *ip* is the time interval in which this point is valid. *cp* is the central point. *sp* is the support of the point.
2. *ir* is the time interval in which this region is valid. *cr* is the core of this region and *sr* is the support of this region.

times in which the input object exists. The function *Inside* returns a moving crisp Boolean value that is true when the first crisp region is inside the second.

The *Upoints_Uregion_Inside* algorithm needs to take the intersection of two polyhedra, the support of the moving point unit and the support of the moving region unit. Taking the intersection of two general polyhedra is a very expensive operation. However, in this case we know that two sides are flat and parallel to each other. This knowledge may be exploited to create a simpler algorithm. However, this is outside the scope of this thesis. An educated guess as to the running time of such a function would be the running time of the intersection of one general and one convex polyhedron. This has a running time of O(n*log(n)) according to [DMY93], where n is the number of points in the two polyhedrons plus the number of points in the result. The maximal number of times the *Upoints_Uregion_Inside* algorithm is run is *up* + *ur*, where *up* is the number of time slices in *mp* and *ur* is the number of time slices in *mr*.

4.10.2. Alpha_Cut

The *Alpha-Cut* operation is an example of an operation that uses the probability function of an object. This function will have different implementations depending on which data type it is applied on and which mode for storing the probability functions is used. Three examples are given here, two for the advanced model with different ways of storing the probability function, and one for the simpler models. All of these versions deal with uncertain regions. One of the methods can also deal with uncertain lines.

For uncertain lines and points, an operation which uses an absolute threshold is of relatively little use as the amount of the object it will return is strongly dependent on the size of the support of the line or point. This is due to the normalization required to make the sum over the line or area equal to one. For a point that is certain to exist and has a large support, *Alpha_Cut*(*p*, 0.5) might return nothing, while if the same point had a small support the same operation might return almost the entire support of the point. An example of this phenomenon is shown in Figure 4.34. This figure shows two uncertain reals, both of which are certain to exist because the integral of their probability mass functions are 1. However, for the real number *A* the alpha-cut(0.7) operation yields a small interval while for *B* it yields nothing because the highest probability mass for that real is 0.5.

A more useful variant of this operation is the one that uses a **relative** weight rather than an absolute one. This means that the parameter to the operation does not give a specific threshold, but a multiplier to the highest value the function has. This will cut a similar amount of the support from points with

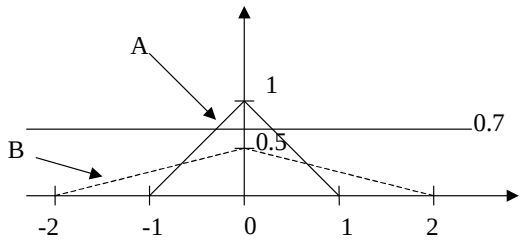


Figure 4.34 *Absolute alpha-cuts for uncertain reals*

both large and small supports as long as they use the same probability functions. In the example in Figure 4.34 *B* would have an alpha-cut(0.7) that is twice as long as that of *A* because the support of *B* is twice as long as the support of *A*. This type of alpha-cut is also easier to implement because it does not need to be concerned with the normalization of the function. This behaviour is caused by the normalization required to make the probability sum 1, and this therefore applies to both points and lines, but not regions.

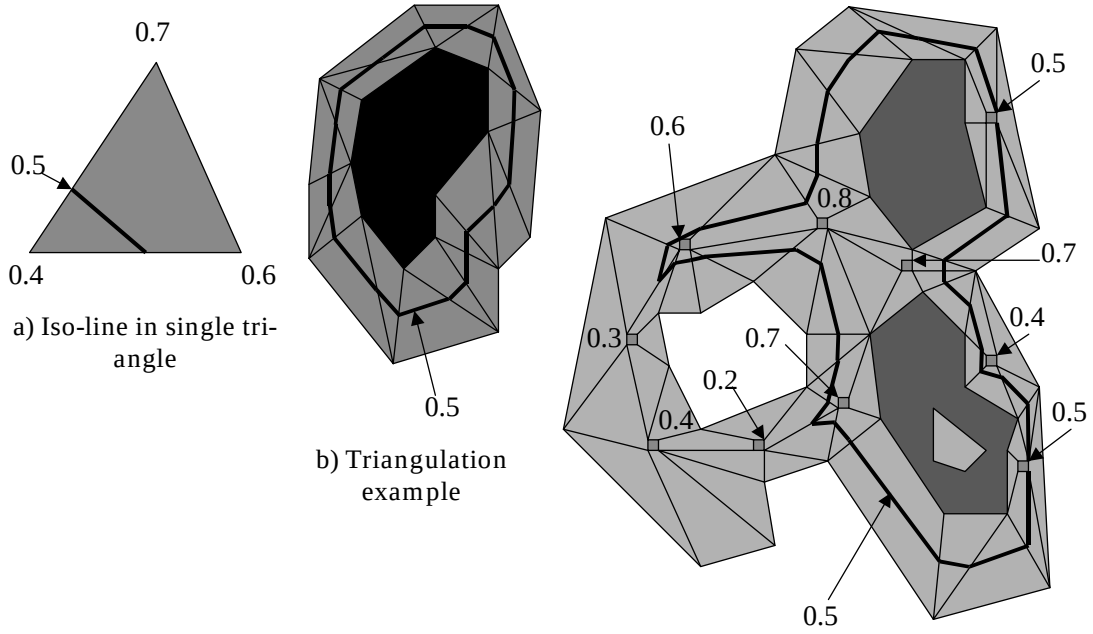
All these implementations assume that the probability function either rises or is constant until it reaches the core. That means that there are no additional peaks at lower values.

4.10.2.1.Alpha-cut, Advanced model, Region, Distance to centre and edge

This method for computing the probability functions has the problem that to accurately compute the alpha-cut, one would have to compute an infinite number of values. Because this is not feasible, a set of sample points must be computed. The advanced model also gives no really good method for finding a point with value “x”. One strategy is to take a number of random points in the support, compute their value, and use that information to compute the value of the alpha-cut. This can be done by constructing a triangulation as described in Section 4.7.4 from the sample points and computing the alpha-cut using that triangulation with the algorithm described in Section 4.10.2.2. The basic problem with this method is that it will yield an inaccurate result.

4.10.2.2.Alpha-cut, Advanced model, Region, Triangulation or Simplical Complexes

This algorithm will base itself on the “storing boundary areas as simplical complexes” method from Section 4.7.4. This method can also be applied to the “triangulation between the core and the support” method from Section 4.7.3, but an implementation made specifically for it would be somewhat simpler as all the lines between the core and the support would have probability values from 0 to the maximum.



a) Iso-line in single triangle

b) Triangulation example

c) Simplical Complexes example

Figure 4.35 *Alpha-cut for advanced model with different representations of the probability function*

The algorithm basically works as follows. First, one computes where on the edges of each triangle the probability function is equal to the input value. The probability function is equal to the input value in either no segments or two segments if the probability function is increasing. Then, one must draw a line between these two points. One example of this with input value 0.5 and a linear function is shown in Figure 4.35a. This process is applied to all the triangles of a face represented by the *Triangulation* and *Simplical Complexes* approach respectively in Figure 4.35b and c. The area that is inside the support but not inside the core is referred to as the *boundary* in the algorithm.

Because the alpha-cut of a continuous function which rises from a support to a core will be a cycle or a set of cycles, this algorithm assumes that the individual lines returned by the function *AlphaLine* will form a set of cycles together.

In this algorithm, a type for a tree of cycles in which each node spatially contains all its children is needed.

$$C_{Tree} \equiv \{(n, C) | n \in D_{Cycle} \wedge C \subseteq C_{Tree}\}$$

Additionally, the crisp face is assumed to be defined as follows:

$$D_{Face} \equiv \{(oc, HC) | oc \in D_{Cycle} \wedge HC \subseteq D_{Cycle}\}$$

Algorithm *Alpha_Cut_A*(*ur, v*)
Input: A *R*(*ur*) and a *CN* (*v*)
Output: A *CRe* containing all points in *ur* with probability value greater than *v*
Method:
let *cl* be an initially empty set of line segments
let *tri* be the set of triangles that makes up the boundary of *ur*.
let $tri = \{tri_1, \ldots, tri_n\}$
for each tri_i **do**
$cl := cl \cup AlphaLine(tri_i, v)$
end for
let *roots* be an initially empty set of C_{Tree}
let *sc* be a cycle constructed from the line segments in *cl*
let $sc = \{c_1, \ldots, c_n\}$
for each c_i **do**
InsertCCNode(c_i, *roots*)
end for
let *fs* be an initially empty set of crisp Faces
AddFaces(*roots*, *fs*)
let *cr* be a crisp region containing the faces in *fs*
return *cr*
end *Alpha_Cut_A*

Procedure *AlphaLine*(*tri, v*)
Input: A triangle (*tri*) and a threshold value (*v*)
Output: A set containing either zero or one line. The line is the line with value *v*
Method:
let *tri* = (l_1, l_2, l_3)
for each l_i in *tri* **do**
if the entire line l_i has the value *v* **then**
return l_i
end if
end for
let *ps* be an initially empty set of points
for each l_i in *tri* **do**
if there is a point *p* on line l_i that has value *v* **then**
$ps := ps \cup p$
end if
end for
if *Card*(*ps*) = 2 **then**
l := the straight line between the two points in *ps*
return *l*
end if

```
    return ∅
end AlphaLine

Procedure InsertCCNode(cyc, nl)
Input: A D_Cycle (cyc) and a set of C_Tree (nl)
Output: This procedure alters the nl input variable by inserting cyc in the
    right-hand node of the right-hand member of nl or adding a new root to
    nl.
Method:
    ins := False
    let nl = {n_1, ..., n_m}
    for each n_i do
        if Inside(cyc, n_i.n) then
            InsertCCNode(cyc, n_i.C)
            ins := True
            break
        end if
    end for
    if not(ins) then nl := nl ∪ {cyc}
end InsertCCNode

Procedure AddFaces(nl, fs)
Input: A set of C_Tree (nl) and a set of CF (fs)
Output: This procedure alters the fs input variable by adding faces con-
    structed from the trees in nl.
Method:
    let nl = {n_1, ..., n_m}
    for each n_i do
        let tf be a crisp face
        tf.oc := n_i.n
        let n_i.C = {c_1, ..., c_n}
        for each c_j do
            tf.HC := tf.HC ∪ {c_j.n}
            if c_j.C ≠ ∅ then AddFaces(c_j.C, fs)
        end for
        fs := fs ∪ {tf}
    end for
end AddFaces
```

The function *Card* returns the number of elements in a set. The function *Inside* returns *True* if the first input cycle in inside the second input cycle and *False* otherwise.

The running time of this version of Alpha-cut is proportionate to the number of triangles in the representation of the support because the *for* statements run once for each line, and there are three lines in each triangle. The checking in each *for* statement takes constant time. The *Inside* tests in *InsideCC-Node* must be run once for each cycle. However, one already knows that either one cycle is completely inside the other or it is completely outside. This means that one only needs to test a single point of one polygon against the other polygon. Testing this only takes O(log(*n*)) time, where *n* is the number of points in the polygon according to [BKOS98].

4.10.2.3.Alpha-cut, Medium and Simple models, Lines

To compute the alpha-cut operator for the simpler models, one must find the line segments with the given probability value for each segment of the core line. Figure 4.36 shows the same alpha-cut for a medium complexity or simple uncertain line segment as Figure 4.35 shows for the advanced model.

For an uncertain line, the alpha-cut consists of two segments for each segment of the core line because the probability function is highest at the core line and then falls towards both ends.

The algorithm below is for uncertain curves, using the definition of alpha-cut from the start of Section 4.10.2[1]. This algorithm also assumes that the functions are linear for both the gradient lines and the core line. This is because the alpha-cut becomes a set of straight line segments if the functions are linear. If these functions are both constant, *Alpha_Cut* becomes trivial and if the functions are non-linear, the alpha-cut becomes a curved line. A step function will result in a line with many small segments, some being straight in the same direction as a linear function would have, and others being straight in a direction towards or away from the central curve.

The alpha-cut for an uncertain line is the union of the crisp regions resulting from the alpha-cuts on the individual curves.

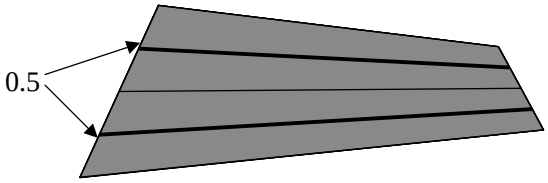


Figure 4.36 *Alpha-cut for the medium and simple models*

1. That is, it uses relative threshold rather than absolute ones.

Algorithm *Alpha_Cut_MS*(*uc*, *v*)
Input: A *C* (*uc*) and a *CN* (*v*)
Output: A *CRe* which contains the alpha-cut of the curve *uc* at the threshold value *v*
Method:
let *sc* be an initially empty set of line segments
let $uc = \{us_1, \ldots, us_n\}$
let *rr* be an initially empty *CRe*
for each us_i **do**
let us_i = (*cc*, *bc*, *ec*, *pc*)
let p_{11} and p_{12} be the points on *bc* where the function
$pc(p_{1x}) \cdot cc.sf(cc.sp) = v$
let p_{21} and p_{22} be the points on *ec* where the function
$pc(p_{2x}) \cdot cc.sf(cc.ep) = v$
If p_{11} or p_{12} does not exist **then**
let *p* be the point on *cc* where *cc.sf*(*p*) = *v*
let p_{11} and p_{12} both be equal to *p*
end if
If p_{21} or p_{22} does not exist **then**
let *p* be the point on *cc* where *cc.sf*(*p*) = *v*
let p_{21} and p_{22} both be equal to *p*
end if
If p_{11} and p_{21} exist and $p_{11} \neq p_{21}$ **then**
let *s* be the line segment that goes from p_{11} to p_{21}
sc := $sc \cup s$
let *s* be the line segment that goes from p_{12} to p_{22}
sc := $sc \cup s$
end if
if $p_{21} = p_{22}$ **then**
cy := *Construct_Cycle*(*sc*)
rr := $rr \cup \{Construct\_Face(cy)\}$
let *sc* be an initially empty set of line segments
end if
end for
return *rr*
end *Alpha_Cut_MS*

In this program, the function *Construct_Cycle* constructs a cycle based on the line segments given as input. The line segments in *sc* will form either a cycle, a line or two lines. In the first case, constructing a cycle is trivial. In the second case, an additional line segment is inserted between the ends of the line to make it a cycle. In the third case, two line segments are inserted between the ends of the two lines in such a way that the two new segments do not cross.

Construct_Face constructs a crisp face with the given cycle as the outer cycle and with no holes.

This version of the alpha-cut function takes time that is proportionate to the number of line segments that the line is made up from. The outer *for* statement runs once for each line segment and the inside takes constant time.

For uncertain regions, the algorithm only needs to return one segment for each segment of the core line, and it must test each cycle in the region separately. There is only one line segment because the function is highest on the side of the uncertain line segment that borders the core.

4.10.3. Intersection

The *Intersection* operation is a very commonly used set operation. This operator determines the area in which two objects intersect. One problem that needs to be solved to make this function work is how to calculate the probability that the intersection between two objects exists when the supports intersect but the cores do not. If the cores intersect, then the probability that the intersection exists is the probability that both object exist. However, if only the supports intersect, the probability that they intersect is lower.

If the functions used are step functions, a possible solution is described in [Sch01]. A step function in this context is a function that is increasing and that increases in only a finite set of discrete points. In all other points, it is constant. The solution for two objects *A* and *B* is to compute the alpha-cut regions for all the points at which the function jumps to a higher value. Then one computes the intersection of each possible alpha-cut of *A* with each possible alpha-cut of *B*. The probability that each such combination is the actual intersection is the product of the probabilities that the two alpha-cut values are the true values of the two regions. This is generally the height of the step up to this value from the previous one. The final probability is the sum of the probabilities of all the combinations that intersect. One problem with this approach is that it takes a lot of time, because one has to take n^2 crisp intersections where n is the number of steps in the step function.

If the function is not a step function, the result can only be approximated. An accurate result could only be obtained through creating an area integral over the area of the result of the intersection operation. One method which might produce good results in some cases would be to compute the centre of gravity of the intersection and compute the probability value there. However, this does not always yield a good result. In fact, one is not guaranteed that the centre of gravity is inside the object.

In all the models, one knows that the two cores are closest to one another at a place where at least one of the cores has a corner[1]. Therefore, one method for checking where the highest value is, is to check the distance from each corner in each core to the other core. Then draw a line from the corner that is closest to the other line so that this line is as short as possible. Take the probability value of the intersection at the point in the middle of this line. Use this value as the probability that the intersection exists. This is the optimal value if the probability function for the region is linear, and also works well for many other types of functions. It may not work well for step functions, but for those the method described in [Sch01] may be used.

If the functions are simple enough[2], one can also estimate the probability of existence by computing the one-dimensional symbolic integral that checks whether the intersection exists at a particular line. This line is typically the shortest line between the cores of the two regions. This is the same as checking the existence of the intersections of two intervals. To check this, one must check whether the upper border of interval *A* is greater than the lower border of interval *B*. The probability of this is $\int_i \int_i (a > b) \cdot P(a) \cdot P(b) dbda$, where $P(a)$ is the probability density that the border of *A* goes at *a*, and *i* is the interval in which they may possibly intersect. For a linear probability distribution function, the corresponding density function is constant. For the example in Figure 4.37, this symbolic integral ultimately becomes the following formula:

Equation 2: Probability of overlap between intervals with linear probability distribution:

$$P = P(b)(e-d) + (\tfrac{1}{2}f^2 - ef + \tfrac{1}{2}e^2)P(a)P(b) + P(a)(g-f)$$

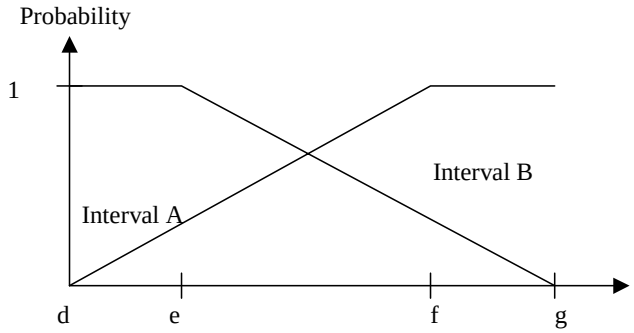


Figure 4.37 *Finding the probability of intersection of two intervals*

Another problem is to compute the probability function for the resulting object. An easy and correct way to do this is to store a link to the source objects

1. A registered point in the core cycle.
2. That is, simple enough that computing its symbolic integral is easy.

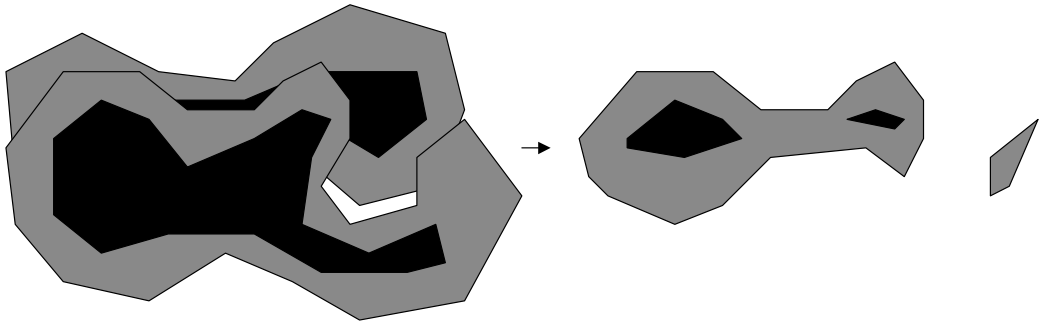


Figure 4.38 *Intersection of two advanced regions*

instead of constructing a new function. Every time the value of the function is requested, the value is computed for that point for the source objects and then multiplied together to get the probability value of the result object. The problem with this approach is that parts of the parent objects must be fetched every time the probability function of the intersection is used.

If this extra data fetching is unacceptable, one may simply use the function of one of them[1] for the result object. This is less accurate, but may yield an acceptable approximation. Which of these two methods to use is a trade-off between increased accuracy and faster data retrieval.

4.10.3.1.Intersection, Advanced model, Regions

To take the intersection of two regions in the advanced model, one must take the intersection of the cores with each other and the supports with each other. Then, one has to combine these to form legal uncertain faces, which make up the resulting region. One example of this is shown in Figure 4.38. These regions consist of only one face each, but the resulting region consists of two faces. One of these faces is not certain to exist. Note that in this particular case, using the centre of gravity to compute the probability value is the best, because the face is convex and the shortest line between the two cores that passes through the face passes through one of its corners.

Algorithm *Intersection_A*(*r1*, *r2*)
Input: Two *Re*'s (*r1* and *r2*)
Output: A *Re* which is the intersection of the two input regions
Method:
let *crc*1 and *crc*2 be initially empty crisp regions
let *crs*1 and *crs*2 be initially empty crisp regions
let $r1.UF = \{r1f_1, \ldots, r1f_n\}$

1. One approach is to choose the one that would give the easiest computations. If there is no difference, a good rule would be to choose the function of the first input.

```
    let r2.UF = {r2f_1, ..., r2f_m}
    for each r1f_i do
        crc1 := crc1 ∪ r1f_i.cr
        crs1 := crs1 ∪ {r1f_i.sf}
    end for
    for each r1f_i do
        crc2 := crc2 ∪ r2f_i.cr
        crs2 := crs2 ∪ {r2f_i.sf}
    end for
    let crc be an initially empty crisp region
    crc := crc1 ∩ crc2
    crcs := Decompose(crc)
    let crs be an initially empty crisp region
    crs := crs1 ∩ crs2
    crss := Decompose(crs)
    let res be initially empty an uncertain region
    let crcs = {crcs_1, ..., crcs_m}
    let crss = {crss_1, ..., crss_n}
    for each crss_i do
        let res_i be an initially empty uncertain face
        res_i.sf := crss_i
        for each crcs_j that is inside crss_i do
            res_i.cr := res_i.cr ∪ crcs_j
        end for
        let res_i.ps be a reference to the product of the probability functions of
            the source faces
        if res_i.cr ≠ ∅ then
            res_i.pe := 1
        else
            let f1 and f2 be the faces in the original objects from which res_i
                were created.
            let l be the shortest line from f1.cr to f2.cr that goes through res_i
            p := Prob_Comp(f1, f2, l)
            res_i.pe := p
        end if
        res := res ∪ res_i
    end for
    return res
end Intersection
```

In this program, the function *Prob_Comp* computes the probability that the two faces given as the first and second input intersect in the area that the line given in the third input passes through. This can be done using any of the

methods given in Section 4.7. The function *Decompose* returns a set of the components of the input type. For a region, it returns a set of faces.

The running time of this operation is the running time of its longest component. These components are:

- Constructing *crc*1 and *crc*2: O(*f*), where *f* is the total number of faces in both regions, because all *f* faces must be extracted. Constructing *crs*1 and *crs*2 has the same complexity, but there are fewer support faces than core faces. This also applies to the next point.

- Constructing *crc*: O(p*log(p) + k), where p is the total number of points in the cores of both input regions and k is the number of crossing points. This is the running time of the crisp *Intersection* operator between two arbitrary polygons according to [BKOS98].

- Constructing *res*: As written in the pseudocode, this operation takes O(s*c*p_{sc}*log(p_{sc})) where s is the number of faces in the support, c is the number of faces in the core and p_{sc} is the average number of points in each support and core face, because all the faces in the core must be checked with an *Intersection* against all faces in the support. However, with the use of a spatial index, this could be reduced to O(s*log(s) + c*log(s) + c*p_{sc}*log(p_{sc})) by indexing the support faces and querying this index when inserting the core faces.

4.10.3.2.Intersection, Medium and Simple models, Regions

For medium and simple regions, the intersection of two region objects is the intersection of all its faces. The function for the intersection of two single faces is given here. To find the intersection of an entire region, find which faces intersect at all, compute the intersections of those and put all the resulting faces in the resulting region.

Algorithm *Intersection_Face_MS*(*f1, f2*)
Input: Two *F*'s (*f1* and *f2*)
Output: A *Re* which is the intersection of the two input faces
Method:
s1 := *Support*(*f1*)
s2 := *Support*(*f2*)
sr := *Intersection*(*s1, s2*)
if ($sr = \emptyset$ **return** $\emptyset$
let *iseg*1 be an initially empty set of uncertain line segments.
for each uncertain line segment seg_i in *f*1.*bc* **do**
if *Intersection*(*Support*(seg_i), *sr*) != $\emptyset$ **then**
$iseg1 := iseg1 \cup \{seg_i\}$

```
    end if
end for
let iseg2 be an initially empty set of uncertain line segments.
for each uncertain line segment seg_i in f2.bc do
    if Intersection(Support(seg_i), sr) != ∅ then
        iseg2 := iseg2 ∪ {seg_i}
    end if
end for
for each s_i ∈ iseg1 that crosses a member in iseg2 do
    let cs2 be a set containing all the segments in iseg2 that cross s_i
    let cs1 be a set containing all the segments in iseg1 that cross at
        least one member of cs2
    iseg2 := iseg2 \ cs2
    iseg1 := iseg1 \ cs1
    iseg1 := iseg1 ∪ Generate_Crossing(cs1, cs2)
end for
for each s_i ∈ iseg2 that crosses a member in iseg1 do
    let cs1 be a set containing all the segments in iseg1 that cross s_i
    let cs2 be a set containing all the segments in iseg2 that cross at
        least one member of cs1
    iseg2 := iseg2 \ cs2
    iseg1 := iseg1 \ cs1
    iseg1 := iseg1 ∪ Generate_Crossing(cs1, cs2)
end for
let hs be an initially empty set of uncertain cycles
cr := Intersection(Core(f1), Core(f2))
for each hole h_i in f1 do
    if Intersects(h_i, cr) then
        hs := hs ∪ {h_i}
    end if
end for
for each hole h_i in f2 do
    if Intersects(h_i, cr) then
        hs := hs ∪ {h_i}
    end if
end for
for each h_i ∈ hs do
    for each h_j ∈ hs do
        if Intersects(h_i, h_j) then
            hs := hs \ {h_i, h_j}
            hu := h_i ∪ h_j
            hs := hs ∪ {hu}
        end if
    end for
```

```
    end for
    let rr be an initially empty uncertain region
    c := Construct_Cycles(iseg1 ∪ iseg2)
    let c = {c_1, ..., c_n}
    for each c_i do
        let rf be an uncertain face
        rf.bc := c_i
        rf.HS := the members of hs that are inside c_i
        rf.ps := f1.ps
        if Core(rf) ≠ ∅ then rf.pe := 1 else
            rf.pe := Probability_of_Existence(rf)
        end if
        rr := rr ∪ {rf}
    end for
    return rr
end Intersection_Face_MS
```

In this algorithm, the function *Generate_Crossing* handles line segments that cross. It takes two sets of line segments that cross each other as input. If the sets contain only one element each, it generates two new line segments that do not cross as shown in Figure 4.15. Otherwise, it generates and returns a *Cross-Set*.

The function *Construct_Cycles* returns a set of all valid uncertain cycles that can be formed from a set of uncertain line segments.

The predicate *Intersects* returns *True* iff the two faces given as inputs intersect and *False* otherwise.

The function *Probability_of_Existence* computes the probability that a given uncertain object exists. It may be implemented using any of the methods described in the beginning of Section 4.10.3.

This algorithm requires that one finds the support and core of the regions and intersects them. Of these operations, the intersection is the only one with greater than linear running time. It runs in $O(p^*\log(p) + k)$ time, with p and k defined as in Section 4.10.3.1.

4.11. Discussion

This chapter has presented three different discrete models for storing uncertain spatial data. These models have been extended to store spatiotemporal data. For the advanced model, three different ways of storing the probability function have been described.

The advanced model has the advantage that it is capable of modelling almost as many cases as the abstract model presented in chapter 3. However, the advanced model requires much more storage space than the other two models, especially for points. It also is the only model that requires special considerations for storing the probability function. The medium complexity and simple models are capable of storing and computing the probability functions in a simple way, but the advanced model is not. Only the *distance to centre and edge* method from Section 4.7.2 for computing the probability function can be used universally, and in Section 4.10.2.1 it is shown that the alpha-cut operator cannot be properly implemented for this method. The *triangulation between core and support* method from Section 4.7.3 cannot be used when there are holes, and the *simplical complexes* method from Section 4.7.4 can only be used when the probability function actually has been measured in certain points or the user is skilled enough to choose appropriate points for which the probability function is computed. The advantages of the advanced model is that it can use more advanced and accurate algorithms for handling temporal uncertainty, and that set operations are much easier to implement for this model than for the other two. Additionally, this model is much closer to the abstract model in chapter 3 than the other two.

The medium complexity and simple models are fairly similar. One problem with them is that they cannot model holes in lines or the support of regions. They also cannot model faces with disjoint cores but single support. Thus they are less expressive than the advanced model. They also have the problem that a special construct called the *CrossSet* has to be defined to make them computationally closed for regions. The advanced model does not need any special considerations for this.

One problem with the simple line model is that it does not support interpolation in time. This is because the angles between the crossing curves and the core line are fixed. This means that the linear interpolation algorithm described in Section 4.9.3 for the medium complexity model cannot be used for the simple model. For points the simple model is less expressive, but requires less storage space. The simple model for points also does not fit into the sliced representation because the support of the point is a circle and not a set of straight line segments. This may be solved by making it into a set of such line segments approximating a circle (such as a hexagon or octagon where the corners are at the given distance).

Compared with earlier models, the major advantage of the new models for spatial data is that they can model all the normal spatial data types. [Dut92] only describes modelling points and lines, although it is simple to extend his

model to regions. An advantage of the medium and simple models presented here over the model from [Dut92] is that it is much easier to compute the probability values. In [Dut92], only the points have a probability function, and to compute the probabilities of the line passing through a given point, one has to take the integral of all possible placements of the two end points which would yield a line passing through the given point. This integral is infeasible to perform in practice in a database system, although it could be done numerically in a simulator in which one has a lot of time available.

The different advantages and drawbacks of the three models show that none of the three models are obviously superior to the others. One should choose which to use based on what data one has and which functionality and operations one needs.

The advanced model presented here is really an extension to the work presented in [Sch96]. The additions are the ability to model points and lines as well as the ability to store and compute a probability function. Some work has been done on fuzzy membership functions in [Sch01], although this work concentrates on step functions rather than continuous functions. However, it seems that some operations, such as *Intersection* are much easier to compute for step functions than for continuous functions.

In this chapter, various operations have been defined with different probability functions as examples. This shows that different probability functions are easy to handle for various types and operations. When choosing what type or types of functions to use, one might keep this in mind. In general, checking the existence of the intersection of two or more uncertain regions is easiest with step functions, while finding the alpha-cut is easiest with a linear function. Finding the alpha-cut from a step function is easy for regions, but is somewhat more complex for lines.

Storing spatial uncertainty (or vagueness) takes more space than only storing crisp data. How much space is demanded depends on the model chosen. In the advanced model, uncertain regions require twice as much space, lines 3.75 times as much, and points many times as much. Thus, for a database containing mainly regions, the cost is doubled, whereas for a database containing mostly points, it is increased enormously. For the simple model, uncertain regions, lines and points take 50 % more space than crisp ones. This is a much smaller increase, but still significant. In the model presented in [Dut92], points and lines take twice as much space as crisp points and lines, because the radius of the spheres for each point must be stored. We assume here that the probability function used for the points is predetermined.

One problem with all the data types presented here is that they are much more complex than the corresponding crisp types from [FGNS00]. This also causes the implementation of the operations to be run on these data to become more complex. However, the simple model is less complex than some other models for uncertain spatial or spatiotemporal data that have been published. The probabilities are much easier to compute than in Dutton's model, and the region model is not much more complex than a model for a crisp region

The time taken to process data depends on the operation. Plane-sweep algorithms typically take O(n*log(n) + k) time where k is the number of intersection points according to [NP82]. However, these do not usually run on all the points at once. For regions with only spatial uncertainty, they run on the core region and the support region separately. Therefore the increase in processing cost is proportional to the increase in data stored. This is also true for uncertain lines, although the support here normally takes twice as much space as a crisp line. This means that an operation on the core will take somewhat longer compared to the size of the line than operations on crisp lines. Only for points is there an increase in processing cost significantly greater than the data increase, because operations that before only had to operate on a single point now have to operate on the support face of the point.

Storing the probability function also requires space if one of the triangulation models are used. For the *Triangulation between core and support* method, it takes +100 % storage space compared to an object without probability function support, whereas the *Simplical Complexes* method takes more than that. These data only need to be fetched when the probability function is used, but many operations use this function. All the operations from Section 3.6 in which a probability value or a degree of truth is returned need the probability function.

Temporal uncertainty also costs additional storage space. For the model presented in Section 4.8.3.3, the storage space increase is minimal as the only additional information is the duration of the time in which the snapshot itself is used as the object value. This is also true for the approach from Section 4.8.3.1, but here the temporal interpolation is done at query time, which significantly increases the query cost. The method from Section 4.8.3.2 replaces the snapshots themselves by interpolations. Because these are created using the sliced representation method, they take at most twice as much storage space as the original snapshots, and they are likely to take close to twice as much. The only time in which an interpolated line segment does not take twice as much space as the snapshot is when the lines are parallel, and that almost never happens in practice. Because lines, regions and advanced points consist of line segments, this increase applies to all the advanced types, which are the only types for

which the method from Section 4.8.3.2 can be used. In the method from Section 4.8.3.2, processing cost increases more than the storage cost because the O(n*log(n)) procedures have to run on polygons with a greater number of vertices than the originals.

Chapter 5
Implementation of the Medium Complexity Discrete Model

The goal of any computer science design process should be to create a design that can be implemented. To test the implementability of the model and to determine whether there were any major gaps in the design that were not found already, parts of the medium complexity model from Chapter 4 have been implemented as a set of Java classes. Some tests have also been run on these. The medium complexity model was chosen because it is fairly easy to implement and because if it can be implemented, so can the simple model as the simple model is a straightforward simplification of the medium complexity model. The implementation was constructed by first implementing and testing the simple types and then extending this to more complicated types. For instance, the medium complexity uncertain curve is used as part of the medium complexity uncertain face. This chapter deals with the implementation and how it was done. Some test results are also included.

One example of an application of this model is this: Imagine you have scientists who are driving around making measurements in the Sahara desert to determine the extent of underground water reservoirs. The scientists themselves are uncertain points due to the imprecision of the positioning system that they use. The roads are uncertain lines because the roads in the Sahara desert are more like routes that shift as the sand dunes move than paved roads. The water reservoirs that the scientists are studying are uncertain regions because they are located deep underground and it is therefore not feasible to do more than a few measurements at each site. The scientists therefore lack the

necessary information to define them precisely.

5.1. Implementation environment

This implementation of the data types was made in Java. This language was chosen because it is easy to use and relatively common. It can also be run on almost any platform. It is possible to use Java to program plug-in modules for several database systems, so there is no need to change the programming language used to integrate these classes into a database. For increased performance, one might want to convert them to C++. As Java is closely related to C++, such a conversion should not be too difficult.

5.1.1. The Java object model

Java is an object-oriented language. Programs are divided into separate classes, which contain a set of operations and encapsulate some data. Because the model from Chapter 3 was designed as a set of abstract data types, it is easy to translate this model to an object-oriented language. Each ADT becomes a separate class. Each class implements the operations that can be applied to that type.

Java supports only single inheritance of functionality, that is, an object can inherit code from only one parent. However, it supports multiple inheritance of interfaces. An interface in Java is a collection of method declarations without method bodies. An interface may also contain constants, but not variables. A Java class can inherit from any number of interfaces. A class inheriting from an interface must contain the implementation of the methods specified in the interface. Interfaces are considered types, and one can have variables of these types. A variable of an interface type can contain an instance of any object that implements that interface.

5.1.2. The Java collections framework

The Java collections framework is used in this implementation to store sets of individual types. The collection classes are used internally in the implementations of the set types and not directly. The classes in the collection framework are implemented to work with generic objects. Therefore, any type checking or checking for such things as spatial overlap must be implemented by the classes for the set types.

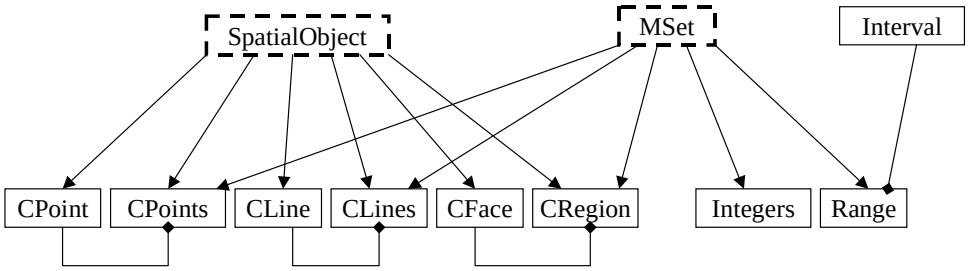


Figure 5.1 *Type hierarchy for the crisp types*

5.1.3. Serialization in Java

Java contains a mechanism in which objects may be stored and retrieved from disk. To indicate that an object can be stored on disk, it should implement the **serializable** interface. This interface contains no methods. Java contains standard methods for storing objects on disk. All the types in this implementation implement the serializable interface, but this functionality has not been tested yet.

5.2. Program design

This section contains a description of how the program is designed. Class hierarchies for both the uncertain types and the crisp ones on which they are based are given as well as a brief description of the testing environment. Further information on the structure and functions of the program can be found in Appendix B.

5.2.1. Class hierarchy

This implementation was designed so that each of the uncertain types described in Chapter 3 is a separate class. This is also true of the corresponding crisp types. Additionally, a class hierarchy was created so that one might be able to use a spatial object even without knowing exactly what kind of spatial object it is. The class hierarchy used is shown in Figure 5.1 and Figure 5.2. The boxes with dashed lines indicate interfaces, the rest indicate classes. The normal arrows indicate inheritance, while the lines that end in squares indicates that an object of the type with the square contains objects of the type without the square.

The **SpatialObject** interface works as a supertype for all spatial objects. The **MSet** type works as a superset for all set types. This is called MSet and not just Set because that name is already in use in Java. The MSet type is needed because the Java collections framework uses different names for its methods and

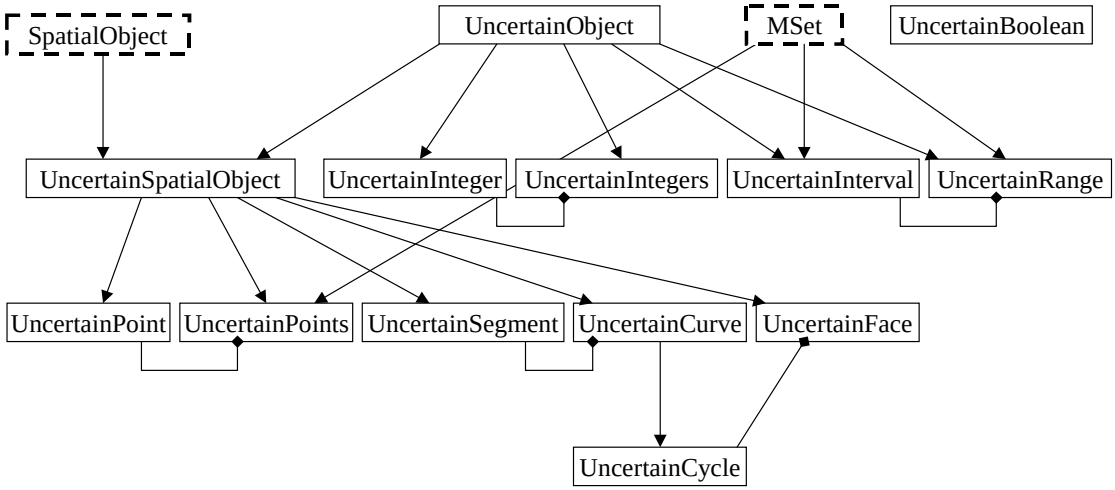


Figure 5.2 *Type hierarchy for the uncertain types*

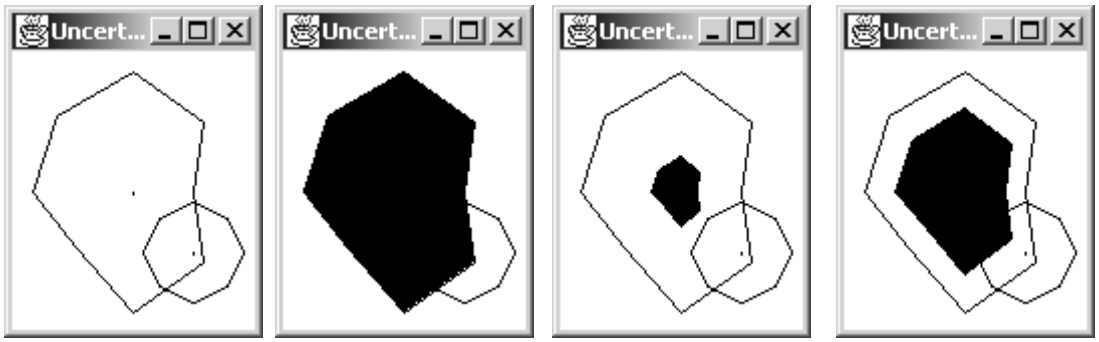
does not contain all the methods that the model presented in Chapter 3 specifies. The **UncertainObject** class is the superclass of all the uncertain types, and defines all the functions that should be implemented by all of them. It also contains the probability of existence variable because that is also common to all uncertain objects. The **Uncertain Boolean** is not a subclass of this class because some of the operations that are meaningful for the other uncertain objects, like *Accuracy*, are not meaningful for uncertain Booleans. Uncertain Booleans also lack uncertainty about existence, because this would make the logic that they define much more complex.

The **integer** type was used for coordinate values to make the implementation simple. This means that there may be small inconsistencies like that in Figure 5.4e in which the alpha-cut face ends in a point slightly outside the core line when the end should have been on the core line. This problem may be solved for spatial data by for instance implementing the ROSE algebra from [GS95] or for spatiotemporal data by using dual grids from [LG00].

5.2.2. Test system design

Each of the basic types has been tested separately. The basic types are interval, range, point, curve, face and uncertain versions of these. These have been tested using separate test programs. These consist of one main class that contains the actual test code. The curve, face, uncertain curve and uncertain face use a graphical interface to enter the types, the others use a text interface. Additionally, the test programs include a common class that can display several spatial object and guarantee than one of them is at the top.

The spatial objects all contain a **draw** method, which takes a graphics context as input and draws a representation of the object in that context. Calling this method in all objects to be displayed is enough to get them on the screen.



a) Two uncertain points **b)** Support of first point **c)** Alpha-Cut(0.7) **d)** Alpha-cut(0.3)

Figure 5.3 *Uncertain point example*

5.3. Data type examples

In this chapter there are examples of tests of the uncertain point, uncertain curve and uncertain face types. These three were chosen because they represent the individual types. The uncertain points and uncertain line set types are little more than regular sets, with no checks for overlapping. The uncertain region type has not been fully implemented yet.

All the types presented here use linear probability functions. The uncertain curves use linear probability functions both for the central curve and for the gradient lines. This is because the algorithms used for determining alpha-cut for curves depend on that assumption. A linear probability function also allows the test to show that the alpha-cut operation yields different results for different input values. The constant function would not have allowed this.

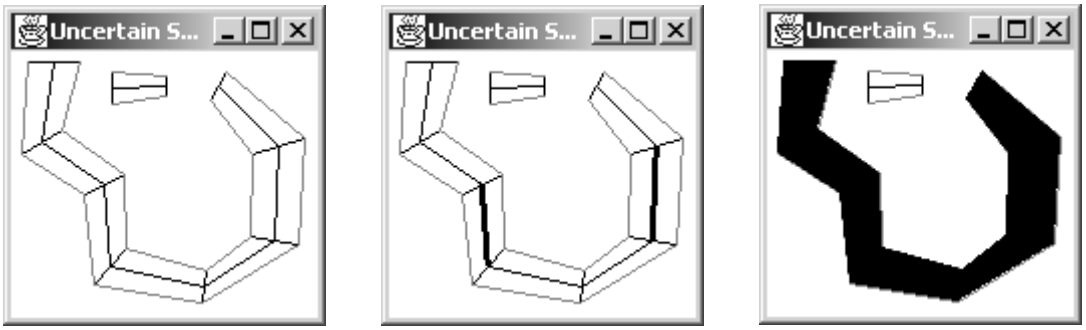
5.3.1. Uncertain point

The uncertain points shown in Figure 5.3 have linear probability functions and eight corners. The core of the points are the dots inside the polygons. The first point, which is the point the operations are run on, is somewhat irregular in shape, whereas the second point has an almost circular support with the core at the centre. Both the points are certain to exist.

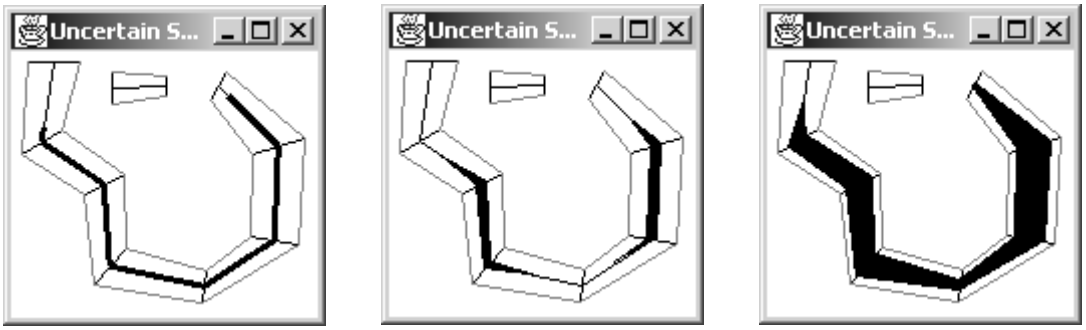
The number of corners is a constant defined in the uncertain point type. This means that all uncertain points have the same number of corners.

The return values of the other operations are as follows:

- A equals B: Maybe



a) Two uncertain curves **b)** Core of first **c)** Support of first



d) Expected value of first **e)** Alpha-Cut(0.7) **f)** Alpha-Cut(0.3)

Figure 5.4 *Uncertain curve example*

- A More Accurate Than B: False
- Resemblance between A and B: 0.12
- Area of A: 6876.6 pixels

The following source data was given to the program to produce the sample points:

Point A: Core X: 60, Core Y: 70, corners: 30, 50, 60, 40, 50, 55, 60, 50.

Point B: Core X: 90, Core Y: 100, corners: 25, 25, 25, 25, 25, 25, 25, 25.

5.3.2. Uncertain curve

Figure 5.4 shows two uncertain curves. Curve *A* has many segments and curve *B* has only one segment. Both curves have probability 1.0 of existing. The black line is the central line, the black line segments crossing the black line are the crosscurves and the grey lines are the boundaries of the support of the curve. The points along it have the following probabilities of existing, from left end to right end: 0.0, 0.6, 1.0, 1.0, 0.5, 1.0, 1.0, 0.4. The figure shows the results of various operations performed on the uncertain curve *A*.

In Figure 5.4a, the long black line is the central curve, the black line segments that go through it are the crosscurves, and the grey lines that go between the crosscurves indicate the support of the curve.

The core and expected values are indicated by thick lines in Figure 5.4b and 5.4d respectively. The support and alpha-cuts are indicated by black areas. The alpha-cut function returns a crisp region, which can contain any number of crisp faces.

Because a graphical user interface was used to draw the uncertain curve and face examples, it is impossible to provide the exact input to the program for either curves or faces. For curves, many of the non-spatial operations have not been implemented yet.

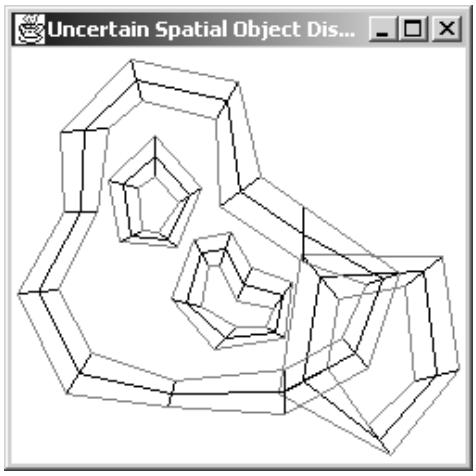
The uncertain curve was at first implemented as a collection of lists. There was a list of core lines, a list of crosscurves, a list of probabilities and a list of probability functions. This would have been the most efficient implementation for uncertain curves if it was not for the needs of the uncertain region. According to Section 4.5.4, one may need *CrossSets* to represent the results of set operations on uncertain regions. Because of this, the uncertain curve was changed so that it was just a set of uncertain segments instead. The uncertain segments contain the core line, crosscurves, probabilities and probability function. The function for inserting new segments into the uncertain curve makes sure that the crosscurves of consecutive segments are the same. The *CrossSet* is a subclass of the uncertain segment. This also means that uncertain cycles, which by definition are closed, must continuously update the closing segment as new segments are inserted.

5.3.3. Uncertain face

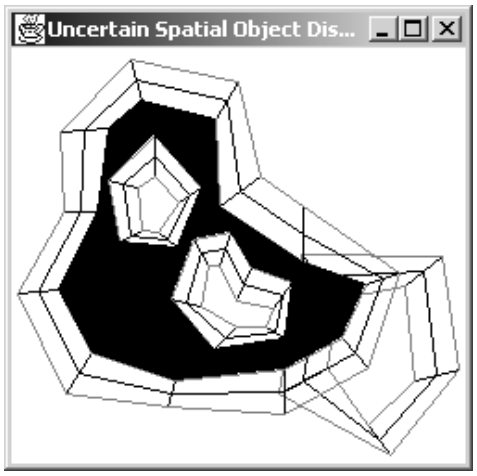
The example in Figure 5.5 contains two faces, both of which are certain to exist. Face *A* contains two holes and face *B* has no holes. The topmost hole is certain to exist, and the bottom-most hole has a probability of 0.6 of existing. Because the borders of the uncertain face in the medium complexity model is an uncertain curve, the borders are drawn as uncertain curves. In those of the examples from Figure 5.5 that contain large black areas, these are the crisp faces that result from the operation mentioned below the example.

Other implemented operations on the two faces in the example give the following results:

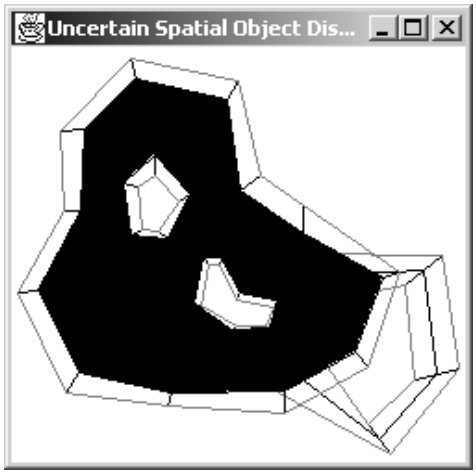
- Accuracy of A: 0.422



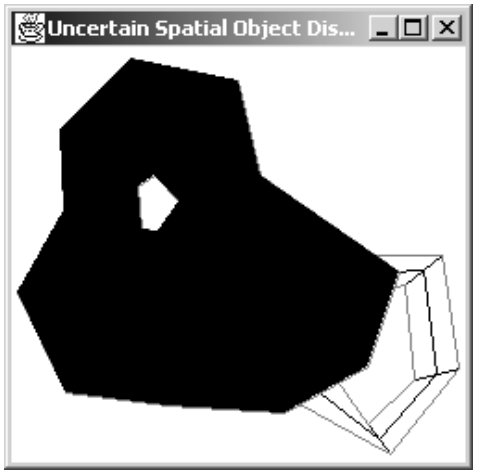
a) Two example faces



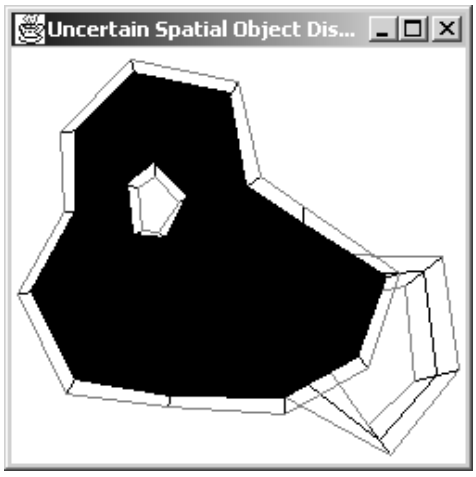
b) Core of first



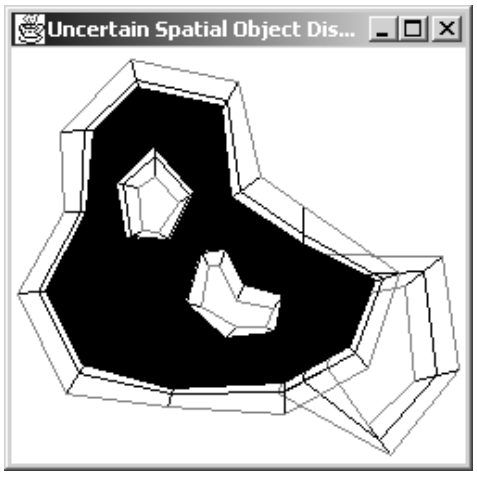
c) Support of first



d) Expected value



e) Alpha-cut(0.3)



f) Alpha-cut(0.8)

Figure 5.5 Uncertain face example

- A equals B: False
- A More Accurate Than B: False
- Resemblance of A and B: 0.098
- Area of face A: 27355.75 pixels

Implementing union and intersection as max and min operations (like in fuzzy set theory) is much simpler than a correct probabilistic implementation. For instance, a very simple algorithm can be used for overlapping holes if one used max and min. This allows us to extend the medium complexity model for uncertain faces so that it also handles uncertain holes that may overlap each other or that overlap with the border curve.

When taking the alpha-cut using fuzzy set mathematics, check the alpha-cut of the support face as well as the alpha-cut of each hole. This is alpha-cut(threshold / prob.exist - 1.0 / prob.exist + 1.0). If this formula is below 0, the alpha-cut does not exist. Then take the alpha-cut of the support and do a spatial set subtraction of all the alpha-cuts of the holes with it. The result is the alpha-cut of the face as a whole. The support is alpha-cut(0.0), the core is alpha-cut(1.0), and the expected value is alpha-cut(0.5).

To do the same with probabilistic computations of the intersection operation would require that for each point one had multiplied the probability of the object existing and 1-p, where p is the probability of the hole existing. If the result is lower than the alpha-cut, the point is in the alpha-cut, otherwise it is outside. This computation is infeasible to perform in practice.

5.4. Implementation of set operations

There are three types of set operations: Regular set operations, number-line operations and spatial set operations. The two last are closely related but different from the first type. These three types of set operations have to be distinguished because they are implemented very differently. They also have somewhat different semantics, so that the regular set union of two regions may be different from the spatial set union of the same regions.

5.4.1. Regular set operations

A regular set operation checks each individual member of a set for union, intersection, minus or containment in the set. Regular set operations are used when the set is finite and discrete.

5.4.2. Number-line set operations

These operations are used for sets containing a potentially infinite amount of numbers. A number-line containment operation would check if one of the intervals in a range contained the specific interval, that is, that it covers the same area of the number line and potentially more. A regular set containment operation, however, would check if the set contained that specific interval by using the equals() operation that is built into Java. This operation returns *True* if the two objects are equal and *False* otherwise. Because intervals are equal when they have the same boundaries, the regular set containment operation would only return *True* if the range contained that specific interval, but would return *False* if it contained an interval that contained the input interval.

5.4.3. Spatial set operations

These are like the number-line set operations, except that they apply to spatial objects. A regular set union of two regions could just be implemented by using the Java collection framework and would give no guarantees that the faces in the result region are disjoint. A spatial set union, on the other hand, checks for intersection of the faces and combines those faces that do intersect. The spatial set union thus helps to maintain the requirement on regions that the faces are disjoint.

Another example of the difference is containment. A regular set containment operation on a region and a face would check if the region contains that specific face, whereas a spatial set containment operation would check if the region contains a face that covers the entire area that the input face covers.

5.5. Core and Support for uncertain faces

The outer boundary of an uncertain face is an uncertain cycle. The support of such a cycle is a ring-shaped region that is bounded by a crisp cycle both on the outside and the inside. The outside cycle bounds the support of the uncertain face, and the inner cycle bounds the core of the uncertain face. However, the uncertain curve does not contain any means to compute this. Therefore, the uncertain cycle subclass computes which cycle is the inner and which is the outer when they are needed. This computation simply checks which is inside the other.

This computation is also needed to compute alpha-cuts for uncertain faces. The function along the crosscurves should be 1 on the core cycle and 0 on the support cycle, so one must make sure that the function produces 0 on the

support cycle and 1 on the core cycle. To do this, one must know which is which.

Chapter 6
Discussion and Evaluation

This chapter contains a discussion of some of the aspects of this work. It also contains an evaluation of the work. This evaluation looks on how well the work fulfils the requirements and answers the research questions posed in Chapter 1. There is also a comparison of this work with earlier work, and a table showing where I got the inspiration for some of the features of the models presented in this thesis.

6.1. Discussion

This section will discuss some issues concerning the models presented in this thesis. It extends the discussion sections in Chapter 3 and Chapter 4.

6.1.1. Probability functions

In the abstract model from Chapter 3, the probabilities of the objects being at various places in the area of uncertainty may be computed with an arbitrary probability function. In the discrete models from Chapter 4, it is shown that relatively simple functions like the step function and the linear function give much easier computations. Thus, there is a trade-off between accurate measurements of probability and ease of computation.

Cartographers have developed various statistical techniques to estimate the probabilities of objects. For instance, [Edw94] contains a method for statistically estimating a fuzzy boundary based on different interpretations of the same feature. These estimates may form 2D probability distributions. Although Chapter 4 assumes 1D probability functions, there is no problem using a 2D function instead. These estimation techniques often produce results in which a probability of 0 is never reached, or is reached only very far from where the ob-

ject is expected to be. Thus, rather than taking the 0 probability boundary as the support, a good approximation would be to use a number slightly higher, such as the 5 % probability.

Hence being able to handle sophisticated functions allows the model to be more accurate. The problem with this is set operations and the alpha-cut function. For a spatial set operation, one ideally has to multiply the probability functions of both objects. This may be achieved by storing the product as the function for the new object.

Another problem, which is made worse by the previous one, is that the actual alpha-cut of an object may consist of curved segments rather than straight ones. Computing these curved segments is impossible in practice, so an approximation must be made. However, such an approximation leads to the inconsistency problem described in Section 4.7.2. Thus, using the alpha-cut operation with a complex function type may lead to inconsistencies.

Simple functions are also easier for the user to define if the user is not very skilled. They even cause the implementation to become much easier. The easiest cases from an implementation viewpoint is linear or step functions combined with the rules from fuzzy set theory rather than those from probability theory for set operations. Intersection in fuzzy set theory requires taking the minimum of the two values while in probability theory it requires taking the product. In Section 5.3, it is shown that the medium complexity uncertain face can model some additional cases if fuzzy set mathematics is used.

Probability theory was chosen as a basis for the models rather than fuzzy set theory because it allows a more accurate representation of reality, and it provides much more powerful tools. For instance, fuzzy set theory has no equivalent of probability densities or masses.

6.1.2. Uncertainty vs. vagueness

The models described in this thesis have primarily focused on modelling uncertainty, that is when one does not know whether an object exists or exactly where it is. Another aspect of spatial indeterminacy is vagueness, that is when objects are imprecisely defined or when there is a gradual rather than abrupt change. See Section 1.1 for examples.

Parts of the framework presented in this thesis may also be used to model vagueness, although this framework could not distinguish uncertainty from vagueness. A vague region behaves much like an uncertain region. In fact, [Sch99] models vague regions in much the same way as uncertain regions are modelled in chapter 3. Vague lines and points would not need a support area

Table 6.1 *Comparison of models*

Aspect	Abstract	Advanced	Medium	Simple
Arbitrarily shaped support	Yes	Yes	Region Only	Region Only
Support for probability functions	Yes	Yes[1]	only 1D	No
Consistency in probabilities	N/A	No	Yes	N/A
Arbitrary holes	Yes	Yes	No	No
Normalization	Yes	Complex	Simple	N/A
Set operations	Yes	Simple	Complex	Complex
Storage required	N/A	Large	Roughly 3X	Roughly 2X
Temporal extension possible	Yes	Yes	Yes	No

1. Only 1D is described, but 2D is also possible

because one already knows where the object is. A vague line would only require a fuzzy membership function over the line. The probability function over the central curve could be used for this purpose. Likewise, the probability of existence could be used for this purpose for vague points.

6.1.3. The three discrete models

In Chapter 4, three different discrete models for uncertainty in spatial and spatiotemporal information were described. Table 6.1 shows some relevant properties for the abstract model and each of the three discrete models. The storage requirement is based on a database containing roughly equal numbers of points, lines and regions. The table clearly shows that none of the discrete models is always better than the two others. Each of them has its own strengths and weaknesses. The advanced model handles arbitrary supports for all types and handles set operations easily. The medium complexity model handles probability functions consistently and requires less storage space, but set operations are more difficult to implement. The simple model requires the least amount of storage space, but even it requires twice as much as the [FGNS00] model for crisp spatiotemporal objects.

6.2. Evaluation

The chosen approach and the methods used will now be evaluated with respect to the research questions and requirements that were presented in Section 1.2 and Section 1.5 respectively.

6.2.1. Fulfilment of requirements

This subsection lists all the requirements from Section 1.5 with an explanation of whether they are fully satisfied and for which models and types they are fully satisfied.

R1 *One must be able to tell whether an object is uncertain or not*

This requirement is satisfied by all the models. For regions, one can determine if the support and core are equal. In that case, the object is crisp. In all other cases it is uncertain. For lines and points, one can determine whether the support has an area. If it does, the object is uncertain. If it does not, one must check the probability of existence as well as - for lines - the probabilities along the line segments. If all of these are 1.0, the object is crisp. In all other cases it is uncertain.

R2 *One should be able to say something about how uncertain an object is*

Chapter 3 describes the *Accuracy* operator that returns this for uncertain regions. For uncertain lines, one may check the width of the support at various points long the line, and for points one might check the size of the support. The *Accuracy* operation has been implemented for uncertain faces.

R3 *One must be able to say where the object certainly is not*

All the types in all the models include a support, which is defined such that the object is certainly inside it. This means that the object is certainly not outside the support.

R4 *One must be able to say where the region certainly is*

Regions in all the models contain a core that is defined as the area in which the region certainly is.

R5 *One should be able to compute the probability that the object overlaps or is inside a given area*

This requirement is satisfied by the abstract model, but only partially satisfied by the discrete models. For the simple discrete model, this cannot be done as there are no probability functions in this model. For the other two dis-

crete models, the results can often only be approximated. See Section 4.7 and Section 4.10.3 for details.

R6 *One should be able to compute the probability that a given crisp point is inside an uncertain region*

In the abstract model and in both the advanced and medium complexity discrete models, this is possible through computing the probability function at that crisp point. In the simple discrete model, this probability is either 1.0, 0.5 or 0.0.

R7 *The model should be able to handle all numerical and spatial data types*

The abstract model defines numbers, intervals, Boolean values, points, lines and regions. The discrete models also define these types, except that the number type is divided further into integers and floating-point numbers.

R8 *The model should be able to store objects with temporal as well as spatial uncertainty*

Both the abstract model and the advanced discrete model allow this. In the medium complexity discrete model there are some problems with storing temporal uncertain points, and the simple discrete model cannot be extended to store spatiotemporal objects.

R9 *One should be able to get crisp versions of uncertain objects with varying degrees of confidence*

All the models contain the *Alpha-cut* operation, which does this.

R10 *One should be able to compute all operations that can be run in standard spatiotemporal models like* [GBE+00]

Chapter 3 defines the semantics for all the operations that are defined in [GBE+00] for uncertain spatial and spatiotemporal data. However, some of these operations cannot return meaningful results when there is uncertainty. These operations are listen in Section 3.6.1.1 and Section 3.6.2.1.

6.2.2. Answering the research questions

This section describes the answers to the five research questions from Section 1.2.

Q1 ***Current Situation:*** *Are there efforts that already have answered the main question or addressed a part of it?*

There are overviews of the current situation in Chapter 2 as well as in the introductory sections in Chapter 3 and Chapter 4. From these overviews, it becomes clear that there is already work that answers parts of the research questions. Typically, the previous efforts have concentrated on one particular data type and constructed a model for that. Examples are [Sch96] for regions and [Dut92] for points. An exception to this is the abstract model for vague spatial data from [Sch99], which includes all three spatial data types and uses the same framework for each. However, [Sch99] tries to model vagueness whereas this thesis tries to model uncertainty.

Q2 ***Requirements:*** *What are the requirements for a system which answers the research question?*

The requirements in Section 1.5 were set up based on some of the aspects of uncertain spatial data that users would find most vital to know. Some of the requirements also ensure that the new model does not lose too many of the capabilities that have been developed for models for crisp data.

Q3 ***Abstract Solution:*** *How should uncertainty in spatial data be modelled in general?*

The main idea has been that all uncertain spatial objects, regardless of type, should store a region that indicates where the object might possibly be. This idea was behind all the developments in Chapters 3, 4 and 5.

Q4 ***Discrete Solution:*** *How should the abstract solution be implemented in a computer?*

Three different solutions to this question were given in Chapter 4. The advanced model was based on storing a discrete region or face with each uncertain spatial object. The medium complexity model was developed to make computing the probability function simple. The simple model was further developed from the medium complexity one to see how much storage space could be saved by removing various capabilities.

Q5 ***Evaluation:*** *How well does this research solve the problems, and how do the solutions presented compare to previous work?*

The rest of Chapter 6 addresses this question.

6.2.3. Limitations of the chosen approach

There are some special cases that have not been adequately covered by the model. One is an uncertain point with a linear support rather than a region support. An example of this would be a car that is known to be on a certain

road (a crisp line), but one does not know precisely where. This may be covered by the abstract model by saying that a line is a degenerate face.

Another example is a line that is uncertain only in parts of it. This is handled in the abstract model by using several lines, some that are crisp and some that are uncertain. In the abstract model, partially crisp lines cause the problem that the probability function goes from being finite to being infinite in the point in which the line becomes crisp. In the discrete models, this is not a problem because probability mass functions are used. The probability mass function would become 1 in the same point.

A further limitation is that the uncertain line models are complex and therefore hard to implement. The implementation used the medium complexity line model. An implementation of the advanced line model might become a lot harder, especially if probability functions are used.

6.2.4. Advantages of the chosen approach

The major advantage of the chosen approach is that it models all uncertain spatial data in roughly the same manner. Thus it is easy to see how they will interact with each other. Most previous work has concentrated on a single type or at most two types. It is also straightforward to extend the more advanced models to model temporal as well as spatial aspects of the data. The temporal extensions presented in this thesis seem to be the first attempts at representing uncertainty that is temporal as well as spatial. This approach also makes it possible to store probability functions that indicate the likelihood of an object being at particular places.

Although a full test of the entire medium-complexity model has not been performed, the preliminary testing of the implementation described in Chapter 5 indicates that the models designed in this thesis work in practise and that the approaches taken and choices made during this work seem appropriate for the challenges that this thesis tries to address.

6.3. Comparison with other work

This thesis builds on a number of ideas from previous work. An overview of the main ideas employed and their inspiration is listed in Table 6.2.

Compared to [Dut92], this work has had a slightly different focus. While [Dut92] focused on measurement errors and digitization error, this work also attempts to model uncertainty due to incomplete knowledge. Therefore, the abstract model for uncertain points allows the point to be inside a general face

***Table 6.2** Inspiration of features*

Feature	Inspiration	Extension
Region as support for uncertain spatial objects	[Sch96]: Region as support for vague regions. [Dut92]: Circular support for points.	Region as support for all uncertain spatial objects, not just uncertain regions. Non-circular support for uncertain points.
Spatial types as probability functions over the plane	[Sch99]: Fuzzy types as membership functions over the plane.	Use of probability theory instead of fuzzy set theory. Use of probability densities and probability masses.
1D probability functions in discrete models	[DS98]: Probability distributions for temporal uncertainty.	Use of 1D functions for 2D data types.
Temporal uncertainty as uncertainty about existence over time	[DS98]: Uncertain time intervals.	Use of this concept for spatial objects. Connecting temporal uncertainty with uncertainty about existence.
Appropriate operation of uncertain data	List of operations in [GBE+00].	Evaluation of the operations with uncertain data.
New operations for uncertain data	Operations for vague spatial data and fuzzy set theory [Zad65].	Extension to spatiotemporal data.
Triangulation approaches to computing probability functions	Triangular irregular networks used for elevation models [Hell90].	New usage: Used for computing probabilities in border area.

and not just a circle. Additionally, the medium complexity and simple discrete models for uncertain lines also have a much easier way of computing probability densities for lines that Dutton's model.

Compared to Schneider's various models ([Sch96], [ES97], [Sch99] and [Sch00a]) the approach presented in this thesis can model more types of data because it can model temporal as well as spatial data. Schneider also focuses on vagueness while this thesis has focused on uncertainty. The difference in focus has caused the point and line models proposed here to become very different from those presented in [Sch99]. The types presented here model uncertainty about the position of the object, while the types in [Sch99] model uncertainty in whether a point or line belongs to a given concept.

For regions, the abstract model is fairly similar to the regions from [Sch99]. In both models the region is a function over the plane that for every point in the plane can yield a value between 0 and 1. The difference concerns what the function represents. In [Sch99], the function represents uncertainty in classification (vagueness), whereas in the models from this thesis the function refers to uncertainty in where the border lies.

Further, in the advanced discrete model, regions are very similar to the regions from [Sch96]. Both approaches see uncertain regions in the same way: with a core in which the region is certain to be and a border area in which the region may possibly be. The advanced discrete model has added the ability to compute probability values for various parts of the border region.

Handling temporal uncertainty was inspired by [DS98]. The time that a spatial object exists (the *DefTime* operation) is a time range in that paper. My extension is to store such a time range with a spatial object and to combine it with the sliced representation. Like an uncertain line or an uncertain range, a time slice contains a probability function indicating the probability that the object exists at various times in that time slice.

All the operations from [GBE+00] have been evaluated for use in the uncertain case as part of this work. However, as given in Section 3.6, not all of these operations are useful when there is uncertainty. This thesis also presents some new operations that can be used with uncertain spatial and spatiotemporal data. Some of these, like the *Core, Support* and *Alpha-Cut* operations, have been described previously for fuzzy set theory.

The triangulation approaches to storing and computing probabilities in the advanced discrete model from Section 4.4 were inspired by the triangulation approaches for storing elevation models in geographic databases.

Chapter 7
Conclusions and Future Work

This chapter summarizes the conclusions reached in this thesis and points out directions for possible future research.

7.1. Conclusions

An abstract model for spatial and spatiotemporal data has been described in this thesis. Three different discrete models have been built based on this abstract model. The medium complexity discrete model has been partially implemented. This shows that it is possible to create an implementable model for uncertainty in spatial and spatiotemporal data.

However, the analysis of storage requirements in Chapter 4 shows that uncertain data requires significant amounts of storage space. Even the simple discrete model, which was made to minimize the required storage space, increases storage space by 1.75 times for curves and faces, and by 2 times for points and base types compared to similar models for crisp data. In the advanced discrete model, faces require 2 times as much space, curves 3.75 times as much space and points many times as much space as the corresponding crisp types. This shows that there is a trade-off between modelling capability and storage space needed. The more computationally complete models require more storage space.

Another conclusion from Chapter 4 is that there is a trade-off between modelling capability and how easy it is to compute probability functions consistently. For the advanced discrete model, special additions must be made if one requires consistent results from the probability functions. Some of these additions cost extra storage space and impose limitations on the modelling ca-

pabilities. The medium complexity discrete model does not require such special considerations, and the simple discrete model does not store probability functions.

In Chapter 5, it was shown how much easier it is to implement certain operations if one uses the mathematics from fuzzy sets rather than the mathematics from probability theory. The difference stems from the fact that fuzzy set theory uses the minimum operation for intersection, while probability theory uses multiplication. Likewise, fuzzy set theory uses the Maximum operation for union, while probability theory uses the formula $a + (1 - a) \cdot b$.

7.2. Future work

Although this thesis presents a complete approach, this may be extended with further capabilities. The implementation may also be extended.

7.2.1. Model extensions

- *Modelling uncertain fields.* The thesis discusses how to model uncertainty with object data. Object data is data about specific objects, which may be vague or uncertain, but are nonetheless distinct. Fields, on the other hand, are functions over the entire plane or space under consideration, and do not represent specific objects. However, fields can also be uncertain due to many of the same factors as for objects.

- *Three-dimensional data.* The models presented here work for two-dimensional data, but there are further problems if one wants to model uncertainty with objects in three (or more) dimensions.

- *Granularities/Mass functions in discrete models.* Using a probability mass function normally requires that one knows how many possible values there are. For spatial data, this is possible if one uses integers or another fixed point system for numbers. However, floating-point numbers are also often used. Further, the dual grid approach from [LG00] uses numbers of different precisions. This makes computing the number of possible values difficult. How to handle this consistently is still an open question.

- *Uncertain points with linear support.* Uncertain points are assumed to have a crisp face as support in this model. However, sometimes an uncertain point may have a crisp line as support. In the abstract model, one may treat a line as a degenerate face. How this will affect the discrete model is also an open question.

- *Modelling relations between uncertain objects.* The model described in this thesis treats single objects. However, in a real database objects are often connected to each other. An uncertain line might be the border between two uncertain regions, or a set of uncertain regions might form an uncertain partition. In Section 3.6.1.1 it is mentioned that one cannot determine that two uncertain regions have a common border unless this fact is explicitly stored. One might extend the model described in this thesis so that it is capable of storing such relationships as well as individual objects.

7.2.2. Implementation extensions

- *Complete implementation of the medium complexity model*. The implementation described in Chapter 5 is not complete. The uncertain region type, as well as several operations for uncertain curves, have not been implemented. Further, the implementation does not cover the temporal part of the approach.

- *Implementation of the advanced model.* To test how implementable the models presented here are, the advanced discrete model should also be implemented. The advanced model is potentially much more difficult to implement than the medium complexity model.

- *Implementation of a data blade.* As a further test, the implementation should be extended with an integration into an existing database system. This has not been done due to constraints in both time and resources.

7.2.3. Other extensions

- *Query language extensions*. This thesis has presented ways to model uncertainty in spatial and spatiotemporal data. However, the query language of a database may also have to be extended to enable the user to fully utilize the new models.

Appendix A

Creating Representations for Continuously Moving Regions from Observations

By **Erlend Tøssebro** and **Ralf Hartmut Güting**[1]

Abstract. Recently there is much interest in moving objects databases, and data models and query languages have been proposed offering data types such as *moving point* and *moving region* together with suitable operations. In contrast to most earlier work on spatiotemporal databases, a moving region can change its shape and extent not only in discrete steps, but continuously. Examples of such moving regions are oil spills, forest fires, hurricanes, schools of fish, spreads of diseases, or armies, to name but a few.

Whereas the database will contain a "temporally complete" representation of a moving region in the sense that for any instant of time the current extent and shape can be retrieved, the original information about the object moving around in the real world will most likely be a series of observations ("snapshots"). We consider the problem of constructing the complete moving region representation from a series of snapshots. We assume a model where a region is represented as a set of polygons with polygonal holes. A moving region is represented as a set of *slices* with disjoint time intervals, such that within each slice it is a region whose vertices move linearly with time. Snapshots are also given as sets of polygons with polygonal holes. We develop algorithms to interpolate between two snapshots, going from simple convex

1. Praktische Informatik IV, Fernuniversität Hagen, D-58084 Hagen, Germany

polygons to arbitrary polygons. The implementation is available on the Web.

A.1. Introduction

Databases have for some time been used to store information on objects which have positions or extents in space. There are also many applications of databases which store information about how such objects change over time. Spatial objects that move or change their shape over time are often referred to as moving objects. In [GBE+00] an abstract model for representing moving objects in databases is described. In an abstract model, geometric objects are modeled as point sets. For continuous objects like lines or regions, these point sets are infinite. This means that these models are conceptually simple, but cannot be directly implemented. A discrete model, on the other hand, can be implemented but is somewhat more complex. A discrete model for spatiotemporal objects, which builds on the abstract model in [GBE+00], is described in [FGNS00].

Early research on spatiotemporal databases concentrated on modeling discrete changes to the database. Examples of such models can be found in [Wor94], [CG94], and [PD95]. More recent research also addresses the dynamic aspect, that is, that objects may change continuously without explicit updates. One example of such a model is presented in [SWCD97]. However, this model covers only the current and expected near future of the objects, and not the histories of the objects, and it also does not deal with moving regions. Constraint databases can also be used to describe such dynamic spatiotemporal databases. One study of constraint databases which explicitly addresses spatiotemporal issues is [CR97]. [CR99] contains a framework in which all spatiotemporal objects are described as collections of *atomic geometric objects*. Each of these objects is given as a spatial object and a function describing the development of this object over time. For the continuous functions, affine mappings (allowing translation, rotation and scaling) and subclasses of these are considered. However, to the authors' knowledge, [GBE+00] and [FGNS00] describe the only comprehensive model describing spatiotemporal data types and operations.

The model in [FGNS00] describes a way to represent continuously moving, amorphous objects in a database in such a manner that it is possible to produce a "snapshot" of the object at any time within the time interval in which it exists. However, most data about moving objects will come in the form of snapshots taken at specific times. This paper addresses the problem of creating this type of representation from a series of snapshots of a moving amorphous region. Important types of such regions in the real world would be oil spills, forest fires, fish schools, and forests. (Forests change continuously because of

deforestation, climatic changes, etc.).

This problem is similar to the problem of interpolating or blending shapes, which has been studied in the computer graphics community, because both problems involve creating plausible in-between shapes at any time between the two states given. One example of such a shape interpolation algorithm is given in [SG92]. This algorithm was created to solve the problem of creating a smooth blending between two figures in an animated movie. A comparison between the algorithm given in [SG92] and our algorithm is given in Section A.8.

A problem which occurs when the moving region consists of several disjoint parts is to discover which part in the first snapshot corresponds to which part in the second snapshot. Because the region parts may have changed both their positions and shape, it may not be obvious to a computer which of them to match. One region part may also have split into two between the two snapshots.

In Section A.2 the representation of regions and moving regions from [FGNS00] is described. Section A.3 then introduces the basic algorithm for building this representation for convex regions. In Section A.4, a way of representing a non-convex area as a tree of convex areas with convex concavities is described. This structure is later used to apply the technique described in Section A.3 for non-convex regions. Section A.5 describes strategies for discovering which regions, or components of regions, in one snapshot correspond to which regions in the other snapshot. This is important both for creating representations for multi-component regions and for matching parts of the tree representation of Section A.4 correctly. Section A.6 describes the algorithm for interpolating between arbitrary polygons; an important subproblem is the matching of concavities between snapshots. In Section A.7, the quality of the results for different types of regions is discussed. Section A.8 is a comparison between our work and [SG92], and Section A.9 contains the conclusions to this paper.

A.2. Representing Regions and Moving Regions

In this section we review the structure and representation of static and moving regions defined in [FGNS00], since this representation needs to be created by our algorithms. We start by considering a (static) region, as a moving region needs to be consistent with it. Indeed, a moving region, evaluated at any instant of time, yields a region.

A *region* may consist of several disjoint parts called *faces*, each of which

may have 0 or more holes. At the discrete level, the boundaries of faces as well as holes are described by polygons. Hence a region looks as shown in Figure A.1.

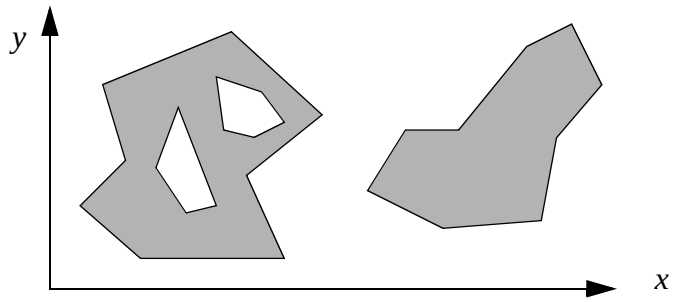


Figure A.1 A region

This structure is defined in terms of *segments*, *cycles*, and *faces*. We sketch the structure of the formal definitions in [FGNS00]; more details can be found there.

$Seg = \{(u, v) \mid u, v \in Point, u < v\}$

A *segment* is just a line segment connecting two points which need to be distinct.

$Cycle = \{S \subset Seg \mid ...\}$

A *cycle* is a set of line segments forming a closed loop which does not intersect itself, hence it corresponds to a simple polygon.

$Face = \{(c, H) \mid c \in Cycle, H \subset Cycle,$ such that $...\}$

A *face* consists of a cycle c defining its outer boundary, and a set of cycles H defining holes. These holes must be inside the outer cycle, and must be pairwise disjoint. H may be empty.

$Region = \{F \subset Face \mid f_1, f_2 \in F \wedge (f_1 \neq f_2) \Rightarrow edge\text{-}disjoint\,(f_1,f_2)\}$

A *region* is a set of disjoint[1] faces.

A *moving region* is described - like the other “moving” data types in [FGNS00] - in the so-called *sliced representation*. The basic idea is to decompose the temporal development of a value into fragments called *slices* such that within a slice this development can be described by some kind of “simple”

1. Edge-disjoint means that two faces may have common vertices, but must otherwise be disjoint (i.e., they may not share edges).

function. This is illustrated in Figure A.2.

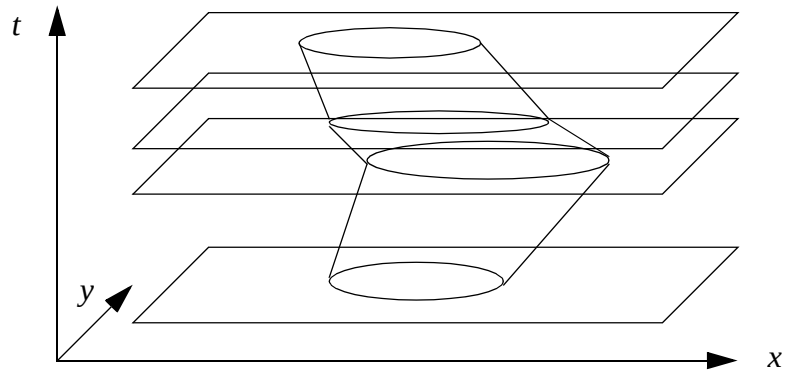


Figure A.2 Sliced representation

Hence each slice corresponds to a time interval; the time intervals of distinct slices are disjoint. For a moving region, the "simple function" within a single slice is basically a region (as defined above) whose vertices move linearly in such a way that at any instant of time within the slice a correct region is formed. Such a slice is shown in Figure A.3.

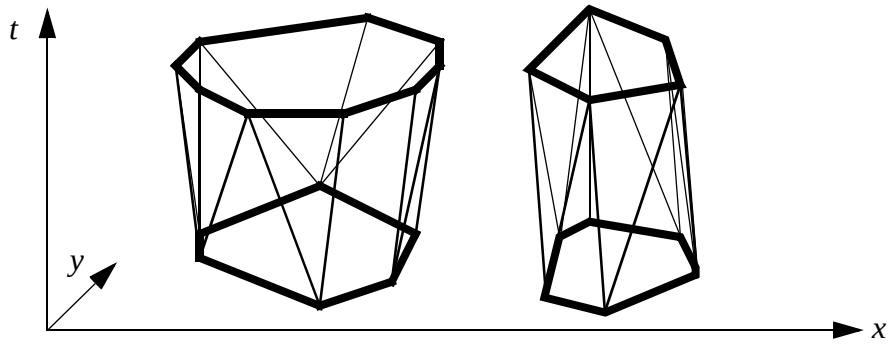


Figure A.3 A slice of a moving region representation

The structure represented within a single slice of a moving region is called a *region unit*. This structure is defined bottom-up in terms of *moving points*, *moving segments*, *moving cycles*, and *moving faces* analogously to the definition of a region. Again we sketch the formal definitions from [FGNS00].

$MPoint = \{(x_1, x_1, y_0, y_1) \mid x_0, x_1, y_0, y_1 \in \text{real}\}$.

A moving point is given by four real coordinates. The semantics of this four-tuple, that is, the function for retrieving the position of the moving point at any point in time is

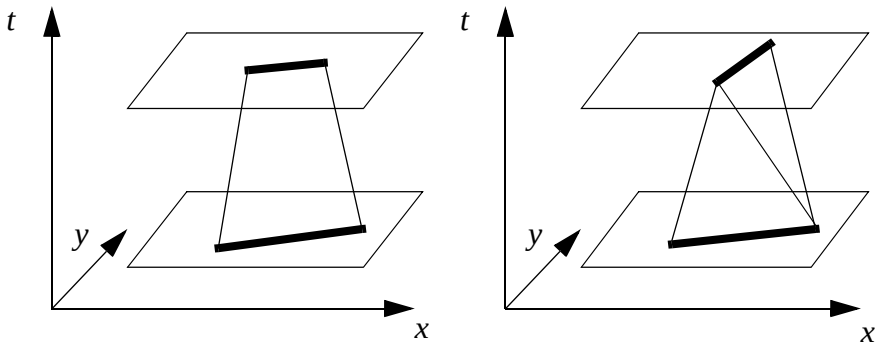
$$p(t) = (x_0 + x_1 \cdot t, y_0 + y_1 \cdot t)$$

In the three-dimensional (*x, y, t*)-space, a moving point forms a straight line.

A moving segment is defined by:

$$MSeg = \{(s, e) \mid s, e \in MPoint, s \neq e, coplanar(s, e)\}$$

A moving segment consists of two moving points which are coplanar, i.e., lie in the same plane in the (*x, y, t*)-space. Hence in 3D a moving segment is a



a) A moving segment

b) Two moving segments representing a rotating line segment

Figure A.4 Moving line segments

trapezium (Figure A.4a). The segment may degenerate at one end of the time interval into a point, hence we may have a triangle in the 3D space. This means that a moving segment cannot rotate as time passes. One can create a (rough) representation for a line segment which rotates by creating two moving segments, each of which is the line segment in one snapshot and becomes a point in the other (Figure A.4b).

$$MCycle = \{(s_0, ..., s_{n-1}) \mid n \geq 3, s_i \in MSeg\}$$

An *MCycle* is the moving version of the *Cycle*. It contains a set of moving line segments. None of these may intersect in the interior of the time interval in which the *MCycle* is valid. The *MCycle* must yield a valid *Cycle* in all instants in the interior of the time interval.

$$MFace = \{(c, H) \mid c \in MCycle, H \subset MCycle\}$$

This is a moving version of the *Face*. The *MFace* must yield a valid *Face* in all time instants in the interior of the time interval.

$$URegion = \{(i, F) \mid i \in Interval, F \subset MFace \text{ such that } ...\}$$

A region unit consists of a time interval and a set of moving faces such that evaluation at any instant of time in the interior of the time interval yields a valid region value.

A.3. The Easy Case: Interpolating Between Two Convex Polygons

The problem is now to compute from a list of region snapshots a moving region representation. This reduces to the problem of computing a region unit from two successive snapshots.

In this section we first consider the most simple case of the problem which occurs if each of the two snapshots is a single convex polygon without holes. In this case one can apply an algorithm that we call the “rotating plane” algorithm. It can be described as follows. Input are two convex cycles at different instants of time.

To create one moving segment, start with a segment *s* in one of the polygons and create a plane perpendicular to the time axis through it. Then rotate that plane around segment *s* until it hits a segment or a point from the other polygon[1]. If in the other polygon there exists a segment *s*’ which is parallel to *s*, then the plane will hit this segment, and the algorithm will create a proper trapezium-shaped moving segment between *s* and *s*’. If there is no parallel segment, then the plane will hit a point *p*. Then a degenerate moving segment will be created which starts out as the original segment *s* and ends as point *p*, thus forming a triangle in space-time.

This algorithm can be implemented in a computer in the following fashion: Take the segments in both polygons and sort them according to their angle with respect to the *x*-axis (for instance). Then go through the two lists in parallel, starting with the segment with the smallest angle in either list. For a given segment check the next segment in the other list. If the angle of this segment is equal to the angle of the chosen segment, create a proper moving segment connecting the two and mark both segments as done. If the angle is different, take the first point in the other segment, use it as the second “segment”, and mark only the chosen segment as done. After the moving segment is formed, take the unmarked segment from either list with lowest angle as the next segment.

An example of the matchings generated by this algorithm is given in Figure A.5. Because the angle of segment *c* is greater than the angle of segment *a*,

1. It should be rotated in such a direction that the part which moves towards the other object hits the other object on the same side as the segment is on the first object.

and less than the angle of *b*, the segment *c* is matched to the point between segments *a* and *b*.

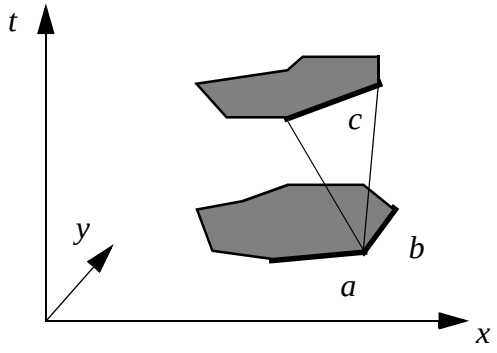


Figure A.5 Example of matching created by the rotating plane algorithm

We now give a more formal description of this algorithm (Figure A.6). The representation of a line segment (*Seg*) is extended to contain an angle as well as the two end points. Also a function *make_moving_point* (Figure A.7) is used to create a moving point from two static points.

Computing the angles between all segments and the *x*-axis takes O(*n*) time, where *n* is the total number of segments. Finding the segments with the lowest angle can also be done in O(*n*) time. Assuming the segments in the two snapshots are already ordered so that adjacent segments are also neighbours in the list, finding the next segment with the lowest angle can be done in constant time (test the next segments in both snapshots and use the smaller one). Adding a new moving segment to the result *r* can also be done in constant time. Because both of the last two operations must be performed once for every segment, the total time for them is O(*n*). Therefore, this algorithm takes O(*n*) time. Note that in the implementation the removal from *um* and checking for membership in *um* is done by modifying or checking a variable associated with each line segment rather than by physically removing or checking in a set. This also applies to the other algorithms below which use a set of unmarked objects.

If the segments are unsorted or sorted by a different criterion than ordering along the border of the cycle, sorting them by angle takes $O(n \cdot \log(n))$ time, and hence the running time of the algorithm will grow to $O(n \cdot \log(n))$.

Theorem 1: Given two convex cycles c_1 and c_2 at times t_1 and t_2, algorithm *rotating_plane* computes a region unit connecting these two cycles. If the two cycles consist of a total of *n* segments and the cycles are represented in (e.g. clockwise) order, then the algorithm requires O(*n*) time. If the two argument cycles are not given in order, then $O(n \cdot \log(n))$ time is required.

algorithm *rotating_plane*(s_1, s_2, t_1, t_2)
input: Two convex *cycles*, s_1 and s_2, which represent snapshots of the moving cycle at the distinct times t_1 and t_2, respectively.
output: An *mcycle* which yields the two *cycles* at the given times.
method:
  let $s_1 = \{s_{1,1}, ..., s_{1,n}\}$; let $s_2 = \{s_{2,1}, ..., s_{2,m}\}$;
  let *um* be a list of *Segs*; *um* := $\emptyset$;
  for each $s_{i,j}$ **do**
    compute the angle between $s_{i,j}$ and the *x*-axis, and store it in $s_{i,j}$.*angle*;
    um := *m* $\cup$ $\{s_{i,j}\}$
  end for;
  MCycle r := $\emptyset$;
  while (*m* $\neq \emptyset$) **do**
    l_1 := the $s_{i,j}$ with the lowest angle, $_{i,j} \in$ *um*;
    l_2 := the $s_{k,l}$, $\neq i$, with the lowest angle, $_{k,l} \in$ *um*;
    if no such l_2 exists **then**
      l_2 := the $s_{k,l}$, $\neq i$, with the lowest angle
    end if;
    if ($_1 \in s_1$) **then**
      let $l_1 = (a, b)$; let $l_2 = (c, d)$
    else let $l_1 = (c, d)$; let $l_2 = (a, b)$
    end if;
    let mp_1 and mp_2 be *MPoints*;
    if (angle of l_2) = (angle of l_1) **then**
      mp_1 := *make_moving_point*(a, c, t_1, t_2);
      mp_2 := *make_moving_point*(b, d, t_1, t_2);
      um := *um* \ $\{l_1, l_2\}$
    else
      mp_1 := *make_moving_point*(a, c, t_1, t_2);
      mp_2 := *make_moving_point*(b, c, t_1, t_2);
      um := *um* \ $\{(a, b)\}$
    end if;
    MSeg ms := (mp_1, mp_2);
    r := $\cup$ $\{ms\}$
  end while;
  return *r*
end *rotating_plane*

Figure A.6 Algorithm rotating_plane

A problem with this interpolation method is that it is poor in handling rotation. If a long, thin object rotates 90 degrees between snapshots, the interpolation in the middle between them will be more or less quadratic, and will probably have a much larger area than the object has in either snapshot. For this reason, one must ensure that the snapshots are so close to each other in time that only a small amount of rotation has happened between them.

```
function make_moving_point(a, b, t0, t1)
input: Two points, a and b, and two distinct times t0 and t1.
output: A moving point which is at a at time t0 and at b at time t1.
method:
    dx := (b.x - a.x) / (t1 - t0);
    dy := (b.y - a.y) / (t1 - t0);
    mp := (a.x - x · t0, dx, a.y - y · t0, dy);
    return(mp)
end make_moving_point
```

Figure A.7 Function make_moving_point

So far we can handle a single convex polygon in both snapshots, the most simple case. Two major problems remain:

1. Treating concavities.
2. Treating regions with more than one face. Here the problem is to match faces from the first snapshot correctly with faces from the second snapshot. Another version of this problem is one face with several holes. One face with one hole can be treated by interpolating separately between the outer cycles from the two snapshots and the two hole cycles and then subtracting the "moving hole" from the "moving outer cycle". But if there are several holes, the algorithm must discover which holes correspond.

These problems are addressed next.

A.4. Representing Non-Convex Polygons by Nested Convex Polygons

We now focus on treating a region which still consists of a single face without holes, i.e., a single cycle, but which needs not be convex any more. The basic idea is to reduce this problem to the previous one by viewing a non-convex polygon as being composed recursively from convex components.

This section first describes a representation in which a general cycle is stored as nested convex polygons. The second subsection describes an algorithm for generating this representation from a *Cycle*.

A.4.1. The Convex Hull Tree

This is a way to store arbitrarily shaped regions by storing convex regions with convex holes. These convex regions and convex holes may then be treated independently by the rotating plane algorithm, allowing it to work for objects

with concavities as well.

In an abstract view of the convex hull tree, each node *p* represents a convex cycle *c* without holes. Each descendant *d* of *p* represents a hole to be cut out from *c* to form the cycle represented by the subtree rooted in *d*. This general method may be used both for storing real holes and for storing concavities in the object. A concavity can simply be represented by a hole which includes a part of the boundary of the cycle. See Figure A.8.

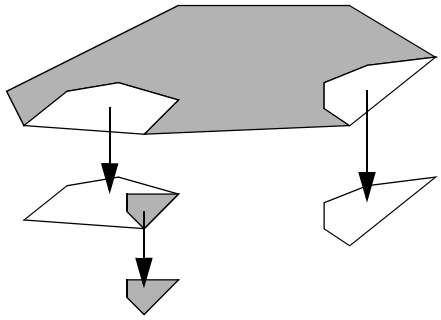


Figure A.8 A convex hull tree

In the implementation of the convex hull tree, a cycle is stored in the following manner: Each node contains a list of line segments representing the convex hull of the cycle. For each of the segments in this representation which were added to make the cycle convex, a link to a child node is stored. This child contains the convex hull of the area which should be extracted to get the real cycle. If the extracted area contains concavities itself, then the child will have children of its own with extracted areas.

An example of a cycle with several concavities and a convex hull tree representation of this cycle is shown in Figure A.9. In this figure, the cycle itself is represented by the thick lines. The segments of medium thickness were added to make it convex. The other segments were added to make nodes further down in the tree represent convex areas. The top node of the tree representation to the right in the figure contains the segments of the convex hull. The line style is the same in the nodes as in the drawing of the region.

This structure as it is described here cannot store holes, because a hole is not connected to a segment in the parent node. However, one could permit the root[1] node to have links to subnodes which are not connected to any particular

1. This should not be permitted for nodes other than the root. If the hole is in the object itself, it should be linked to the root. If the hole is in a concavity, then the object is no longer a single region, but several disjoint regions.

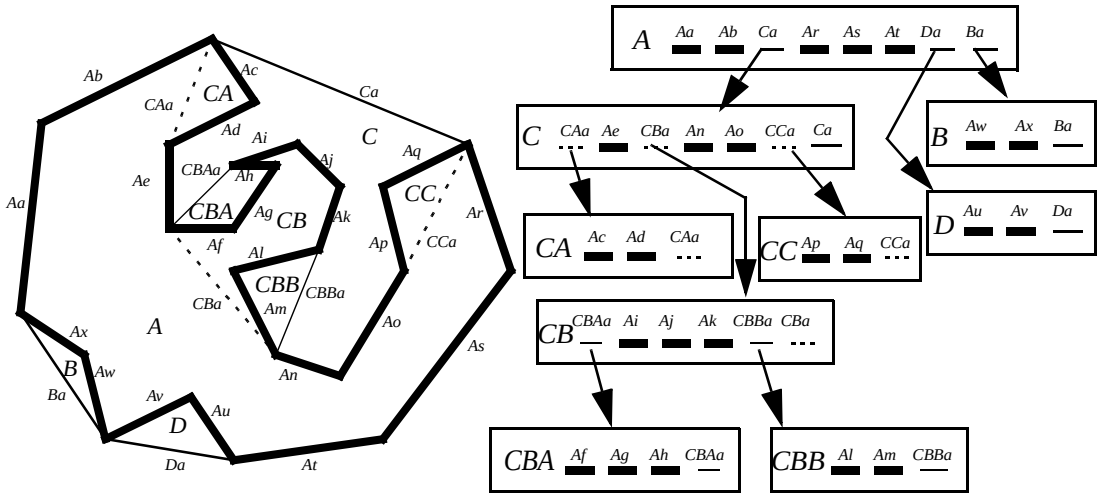


Figure A.9 A region with concavities and its convex hull tree representation

segment. These would then represent holes.

A.4.2. Computing a Convex Hull Tree from a Polygon

To build a convex hull tree for an arbitrarily shaped polygon, use the following steps:

1. Start at the root node and the entire polygon.
2. Create the convex hull of the polygon.
3. Store a segment list representation of the convex hull into the node.
4. For each of the segments which were added to make the polygon convex, create a new node.
5. For each of these holes with new nodes, go to step 2.

The algorithm for building a convex hull tree (Figure A.10) uses two new types, *CHTNode* and *CHTLineSeg*. *CHTLineSeg* is a line segment (*Seg*) which in addition to the two end points may store a link to a child *CHTNode*. The *CHTNode* type is the same as the *Cycle* type, with the exception that it stores *CHTLineSegs* instead of normal line segments.

Our implementation uses the Graham scan from [Gra72] to compute the convex hull in $O(n \cdot \log n)$ time for a given polygon with n vertices. This must be performed once for the whole object and once for each concavity. Because the number of vertices in all the concavities at each level of the tree is less than or equal to n, the total time for computing convex hulls is bounded by

```
algorithm build_convex_hull_tree(polygon)
input: A Cycle polygon.
output: A CHTNode which is the root of the convex hull tree for polygon.
method:
    CHTNode cl := ∅;
    Cycle ch := the convex hull of polygon; let ch = {cs_1, ..., cs_n};
    for each cs_i ∈ ch do
        if (cs_i ∉ polygon) (that is, it was added to make the polygon convex)
        then
            cp := cs_i and the segments in polygon which were replaced by cs_i;
            cch := build_convex_hull_tree(cp)
        else
            cch := ⊥
        end if;
        cl := cl ∪ {(cs_i.u, cs_i.v, cch)}
    end for;
    return cl
end build_convex_hull_tree
```

Figure A.10 Algorithm build_convex_hull_tree

$O(dn \cdot \log n)$, where d is the depth of the convex hull tree.

The line segments in the convex hull will be returned in counterclockwise order by the procedure for computing the convex hull. If the line segments in the given polygon are also ordered in this way, discovering which segments from the convex hull are not in the original region and discovering which segments they have replaced can be done in linear time by going through both lists in parallel, and testing for equality. When the two segments are not equal, go through the list from the original polygon and put segments into a separate list L until a segment with end point equal to the end point of the segment from the convex hull is found. List L will then contain the segments which were replaced by the segment in the convex hull. The only problem with this algorithm is finding where in the two lists to start, because the starting segment must be in both sets. This can be done by marking which segments are in the convex hull and which are not during the construction of the convex hull, and then testing the lines in the region beginning with the first until one is found which is on the convex hull. This takes $O(n)$ time. Finding which element of the hull is equal to this segment can then also be done in $O(n)$ time. Marking whether the segments are in the convex hull or not does not change the asymptotic running time of the Graham scan. Because this linear running time is less than the time taken by the Graham scan, the running time of the entire algorithm is equal to the running time of the Graham scan.

Theorem 2: For a given polygon with n vertices, the convex hull tree can be built in $O(dn \cdot \log n)$ time, where d is the depth of the resulting tree.

To recreate the polygon which is represented by a convex hull tree, start with the root node and do the following:

- For each segment in the node which does not have a child, return that segment.
- For each segment in the node which has a child, go to that subnode and use this procedure on that node.

A.5. Matching Corresponding Components

We now address the problem of matching components in one snapshot with components of the other which comes in three flavors:

- Given observations of a moving region consisting of several faces, which faces in the older snapshot correspond to which faces in the newer one?
- Given a moving face with several holes, which holes in the old snapshot correspond to which holes in the new?
- Given a moving face (cycle) with concavities and two snapshots of it, which concavities in the old and new snapshots correspond to each other?

Figure A.11 illustrates the problem. It becomes aggravated by the fact that components may split or merge between snapshots.

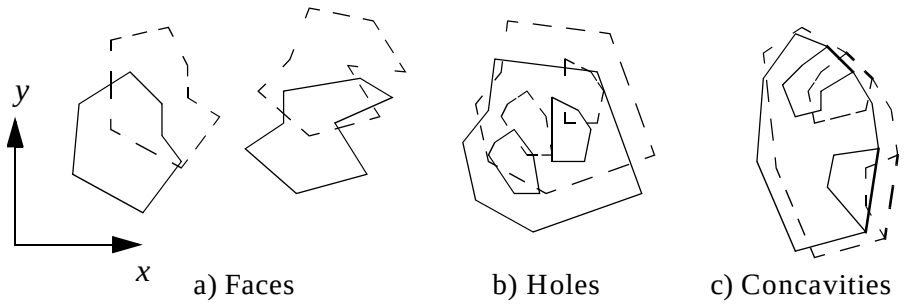


Figure A.11 Matching components of moving region observations

In all three cases we need to find matching pairs of cycles (i.e., simple polygons). From now on we assume that two sets of cycles C and D are the given input for this problem.

A.5.1. Requirements for Matching

Before discussing strategies for matching, we should understand the quality criteria for such strategies.

1. It seems obvious that matching should work correctly for any component that has not moved at all.
2. Components that have moved a small distance relative to their size should be matched correctly.
3. It should be possible to match components that have had minor changes to their shape and size.
4. A matching algorithm should discover that a component has been split into fragments or merged from them.
5. A matching strategy should offer criteria to judge the quality of observations. In other words, it should allow one to decide whether two successive snapshots are close enough in time, or too far apart.

Generally, it seems reasonable to require that a matching strategy is guaranteed to produce correct matchings for the components of a moving region if the frequency of observations is increased. This can be formulated a bit more precisely as follows:

Definition 3: Let *mr* be a moving region with several components, and let S_1 and S_2 be two observations of it at times t and $t + \Delta t$. A matching strategy is called *safe*, if it is guaranteed to produce a correct matching of the components of *mr* if $\Delta t \to 0$. In other words, there exists an $\varepsilon > 0$ such that the matching is correct for all $\Delta t \leq \varepsilon$.

A.5.2. Strategies for Matching

Strategies for matching include the following:

1. *Position of centroid*. For each cycle, compute its centroid (center of gravity). This transforms each set of cycles into a set of points. A closest pair in the point sets *C*' and *D*' is a pair of points (*p, q*) such that *q* is the point in *D*' closest to *p* and *p* is the point in *C*' closest to *q*. For each closest pair, match the corresponding cycles.
2. *Overlap*. For each pair of cycles *c* in *C* and *d* in *D* compute their intersection area *u* and take the relative overlap, that is, *overlap*(*c, d*) = *size*(*u*) / *size*(*d*) and *overlap*(*d, c*) = *size*(*u*) / *size*(*c*). The *overlap* relationship can be represented as a weighted directed graph (i.e. if *overlap*(*c, d*) = *k*, for *k* >

0, then there is an edge from *c* to *d* of weight *k)*. Then there are several options:

a. *Fixed threshold.* Introduce a threshold t (e.g. $t = 60\%$). Two cycles *c* and *d* match if $overlap(c, d) > t$ and $overlap(d, c) > t$.

b. *Maximize overlap.* For all cycles (nodes) order their outgoing edges by weight. For a node *c* let $succ_1(c)$, ..., $succ_n(c)$ be its ordered list of successors. Match *c* with *d* if $d = succ_1(c)$ and $c = succ_1(d)$.

So far we have considered the matching of single cycles. However, the overlap graph allows us to recognize in a natural way transitions where cycles split or merge. See Figure A.12. Here *c* splits into *d, e,* and *f* (or is a merge of *d, e,*

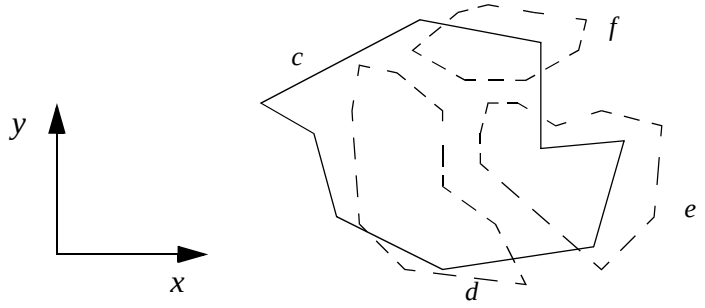


Figure A.12 Cycle c splits into three cycles d, e, and f

and *f*). This can be deduced from the fact that for each of the three fragments the overlap with *c* is large (above 50 %, say) whereas for *c* the overlap with either *d, e,* or *f* is relatively small, but the *sum* of their overlaps is large. This leads to strategies for matching a cycle with a set of cycles:

c. *Fixed threshold, set of cycles.* As in (a), introduce a threshold t (e.g. $t = 60\%$). Match *c* with $\{d \in D \mid overlap(c, d) > t\} \cup \{d \in D \mid overlap(d, c) > t\}$.

d. *Maximize overlap, set of cycles.* Order outgoing edges by weight as in (b). Match *c* with $\{succ_1(c)\} \cup \{d \in D \mid c = succ_1(d)\}$.

What is a good strategy in the light of the requirements of Section A.5.1? Using the centroids, although simple, is not a safe strategy. This is because centroids may lie outside their cycles so that centroids even of disjoint cycles may coincide. This can lead to entirely wrong matchings. The overlap techniques are safe because overlaps approach 100% for region observations when $\Delta t \to 0$. Of course, snapshots have to be close enough to ensure reasonable results.

In the remainder of this paper, we will restrict attention to considering a single cycle with concavities, represented in a convex hull tree. The full paper [TG02] covers the general case with multiple faces and holes. However, the techniques for matching components are already needed in the restricted case for matching concavities in two snapshots of a single cycle. Also, we need to treat transitions such as the splitting / merging of concavities.

A.5.3. Matching Two Convex Hull Trees

To support the matching of concavities, we compute for two given convex hull trees an overlap graph. Its nodes are the nodes of the convex hull trees; to store the edges, the data structure for nodes is extended to store also a set of pointers to other nodes; each pointer has an associated weight indicating the overlap.

type *OverlapEdge* = { (*node*, *weight*) | *node* ∈ *CHTNode*, *weight* ∈ *real*}
CHTNode **subtype** *CHTNodeWO* = { (..., O) | O ⊂ *OverlapEdge*}

In the description of algorithm *compute_overlap_graph* (Figure A.13) we assume that the two argument convex hull trees have been constructed using nodes of type *CHTNodeWO* ("convex hull tree node with overlap") and that in each node the set *O* of overlap edges has been initialized to the empty set. This is a trivial modification of algorithm *build_convex_hull_tree*.

The algorithm traverses the tree, computing the overlap for pairs of nodes of different trees at the same level whose parents overlap. If the two nodes overlap at a percentage higher than *criterion*, then the nodes are linked.

The intersection of two convex polygons with l and m edges can be computed in time $O(l + m)$ (see e.g. [PS85, Theorem 7.3]). If the two polygons represented in the convex hull trees have a total of n edges, then the running time for *compute_overlap_graph* can be bounded by $O(d \cdot f^2 \cdot n)$, where d is the depth of the tree and f the maximal fanout, since on each level of the tree there are less than n edges and overlap computation is called for each combination of f sons of a node. – Our implementation described in Section A.7 uses a function for computing the intersection of two polygons that comes with *java 1.2* (*java.awt.area*) and the authors do not know what algorithm is used there.

A.6. Interpolating Between Two Arbitrary Polygons

We are now ready to address the problem of interpolating between two general, possibly non-convex polygons. We assume these polygons are repre-

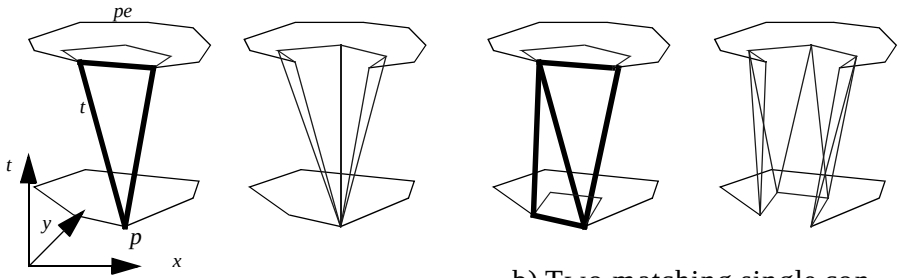
algorithm *compute_overlap_graph*(cht_1, cht_2, *criterion*)
input: Two convex hull trees cht_1 and cht_2 with nodes of type *CHT-NodeWO* and the real number *criterion*, which controls how much two convex hull tree nodes must overlap to be considered a match.
output: cht_1 and cht_2 are updated to contain overlap edges for matching pairs of nodes.
method:

```
overlap := intersection(cht1, cht2); // intersection of convex polygons
                                     // in the roots
overlap1 := (area(overlap)/area(cht1))*100;
overlap2 := (area(overlap)/area(cht2))*100;
if (overlap1 > criterion) and (overlap2 > criterion) then
    OverlapEdge oe1 := (cht2, overlap1);
    OverlapEdge oe2 := (cht1, overlap2);
    cht1.O := cht1.O ∪ {oe1}; cht2.O := cht2.O ∪ {oe2};
    for each son s1 of cht1 do
        for each son s2 of cht2 do
            compute_overlap_graph(s1, s2, criterion)
        end for
    end for
end if
end compute_overlap_graph
```

Figure A.13 Algorithm compute_overlap_graph

sented by convex hull trees for which the overlap graph has been computed.

The basic idea is, of course, to use the *rotating_plane* algorithm from Section A.3 on each matching pair of nodes of the two convex hull trees. Let us consider what can happen for a concavity from one snapshot to the next.



a) Unmatched concavity

b) Two matching single concavities

Figure A.14 Transitions for cancavities

The first case (see Figure A.14 a)) is that the concavity does not find a "matching" partner in the other polygon. In this case we consider the trapezium *t* involving its parent edge *pe* which is most likely a triangle (drawn fat in Figure A.14). All the edges of the concavity are connected by triangles with the

point *p* in the other polygon in which triangle *t* ends.[1] So the concavity appears to spring from *p* or to disappear into *p* depending on which snapshot is first in time.

Technically, trapeziums are first constructed for the two convex outer polygons, which includes the creation of *t*. Then, trapeziums (triangles) are constructed for the concavity, including its parent edge, so that *t* is created once more. Then the union is formed of the first set and the second set of trapeziums, *subtracting their intersection*. This leads to the complete removal of trapezium *t*.

The second case (Figure A.14 b)) is that there is a single matching partner for the given concavity in the other polygon. Then trapeziums are constructed recursively for the two concavities. Again, this also yields the trapeziums involving the parent edges of the two concavities so that these can be removed from the result when forming the union with the trapeziums from the next higher level.

The third, most involved case occurs if the concavity matches more than one concavity in the other polygon (Figure A.15).

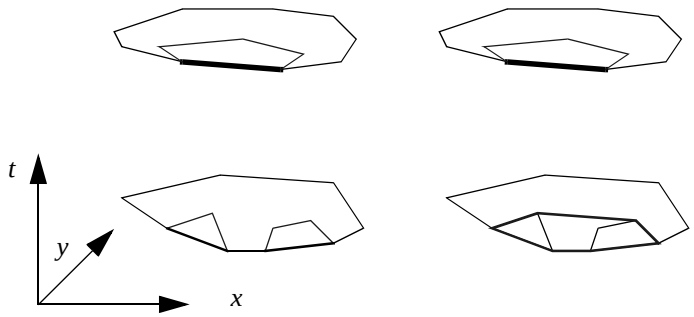


Figure A.15 Transitions for concavities: one concavity matches several concavities

In this case, before the interpolation is performed, the set *C* of concavities matching the one concavity is first joined into a single convex polygon. This is done as a transformation on the convex hull tree, which is illustrated in Figure A.16.

The algorithm for performing the transformation shown in Figure A.16 is called *join_concavities* (Figure A.17). It uses a function *recreate_polygon*

1. If *t* is indeed a trapezium which happens if there is a segment *s* parallel to *pe* in the other polygon, then one of the end points of *s* is selected arbitrarily to play the role of *p*.

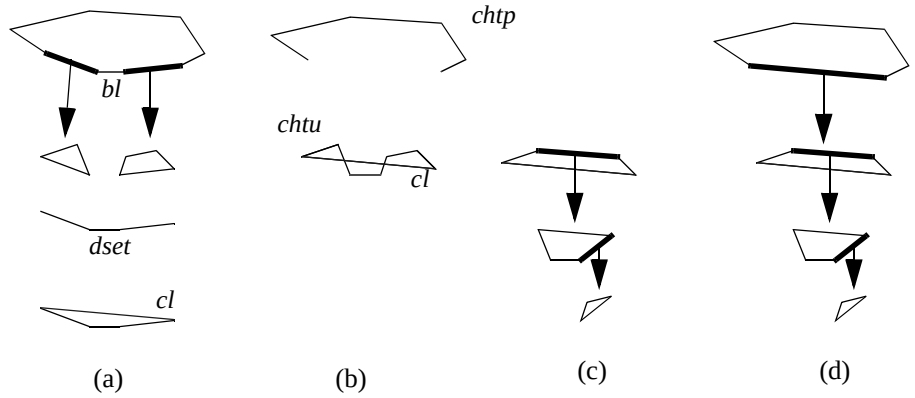


Figure A.16 Rebuilding the convex hull tree to join concavities

algorithm *join_concavities*(*chts*, *chtp*, *bl*)
input: A set of convex hull tree nodes with overlap graph, *chts*, which represents the concavities to be joined, the convex hull tree node with overlap graph *chtp*, which is the parent node of the nodes in *chts*, and a set of lines, *bl*, which represents the lines between the concavities.
output: A single convex hull tree node which is the union of the others.
method:
let *chtu* be an empty set of line segments;
for each $ht \in chts$ **do**
chtu := *chtu* ∪ *recreate_polygon*(*cht*)
end for;
let *dset* be an empty set of line segments;
dset := *dset* ∪ *bl*;
for each $\in chtu$ **do**
if *l* ∈ *chtp.S* **do**
chtu := *chtu* \ {*l*};
chtp.S := *chtp.S* \ {*l*};
dset := *dset* ∪ {*l*};
end if
end for;
let *cl* be the line segment that needs to be added to *dset* to make it a cycle;[1]
chtu := *chtu* ∪ *bl*;
chtu := *chtu* ∪ {*cl*};
let *res* be the cycle formed by the line segments in *chtu*;[2]
resch = *build_convex_hull_tree*(*res*)
let *rlp* be a *CHTLineSeg* containing the line segment *cl* and a reference to the *CHTNodeWO resch*;
chtp.S := *chtp.S* ∪ {*rlp*};
return *resch*
end *join_concavities*

Figure A.17 Algorithm join_concavities

1. *dset* now contains all the line segments in the parent that point to the cycles that should be joined. It also contains the lines between them. Because the lines in the parent form a convex polygon, adding only a single line makes this collection of line segments a cycle.
2. Note that the line segments in *chtu* are not necessarily a cycle, because the line *cl* may cross some of the other lines. The implementation contains code that handles this particular case in all functions that normally take cycles as input. The implementation always ignores the line which a node has in common with the parent.

(Figure A.18) implementing the strategy for reconstructing a polygon from a convex hull tree sketched at the end of Section A.4. Some of the notations used in *join_concavities* are shown in Figure A.16.

---

algorithm *recreate_polygon*(*cht*)
input: A convex hull tree, possibly with overlap graph, *cht.*
output: The cycle represented by *cht.*
method:
  let *res* be an empty set of line segments;
  for each *ls* ∈ *cht.S* **do**
    if (*ls* contains link to child node) **then**
      res := *res* ∪ *recreate_polygon*(*ls.child*)
    else
      res := *res* ∪ {*ls*}
    end if
  end for;
  return *res*
end *recreate_polygon*

Figure A.18 Algorithm recreate_polygon

Finally, the overall algorithm for interpolating between two polygons is given in Figure A.19, Figure A.20, and Figure A.21. The strategy for matching

---

algorithm *create_moving_cycle*($poly_1$, $poly_2$, t_1, t_2, *criterion*)
input: Two polygons, $poly_1$ and $poly_2$ represented as *Cycles*, two times, t_1 and t_2, representing the times when $poly_1$ and $poly_2$ are valid, and a *criterion* specifying how much overlap is required to consider two objects to match.
output: An *MCycle* resulting from the interpolation of the two polygons.
method:
  cht_1 := *build_convex_hull_tree*($poly_1$);
  cht_2 := *build_convex_hull_tree*($poly_2$);
  compute_overlap_graph(cht_1, cht_2, *criterion*);
  return *trapezium_rep_builder*(cht_1, cht_2, t_1, t_2)
end *create_moving_cycle*

Figure A.19 Algorithm create_moving_cycle

algorithm *trapezium_rep_builder*(cht_1, cht_2, t_1, t_2)
input: Two convex hull trees with overlap graph represented by their roots, cht_1 and cht_2, and two times, t_1 and t_2, when the polygons represented by cht_1 and cht_2 are valid.
output: An *Mcycle* resulting from the interpolation of the two polygons.
method:
  $children_1$:= the children of cht_1; $children_2$:= the children of cht_2;
  MCycle mc := *rotating_plane*(cht_1, cht_2, t_1, t_2);// *convex hull tree node*
                                                  // is a subtype of *cycle*.
  um := $children_1 \cup children_2$; // "unmatched children"

  //Step 1: Find partners in cht_2 for children in $children_1$
  for each *child* $\in$ $children_1$ **do**
    ol := the list of concavities that overlap *child* (according to the overlap graph);
    // restrict *ol* to concavities for which *child* is the maximally overlapping one
    for each *c* $\in$ *ol* **do**
      col := the list of concavities that overlap *c*;
      if not (*child* is the element of *col* with greatest overlap) **then**
        ol := *ol* \ {*c*}
      end if
    end for;
    if *ol* $\neq \emptyset$ **then**
      lsbc := {the line segments that lie between the concavities in *ol*};
      concavity := *join_concavities*(*ol*, cht_2, *lsbc*);
      cr := *trapezium_rep_builder*(*child*, *concavity*, t_1, t_2)
      mc := (*mc* $\cup$ *cr*) \ (*mc* $\cap$ *cr*);
      um := *um* \ {*child*}; *um* := *um* \ *ol*;
    end if
  end for;

Figure A.20 Algorithm trapezium_rep_builder, Part 1

concavities is actually a mixture of strategies 2c and 2d: The overlap graph is constructed applying a fixed threshold (*criterion*). But then a concavity *c* is matched to all concavities connected by an overlap edge for which it is the maximally overlapping concavity.

The analysis of the complexity of this algorithm is a bit more involved and for lack of space omitted here. In the full paper [TG02] an upper bound of $O(d^2\ n \log n)$ is derived, where d is a bound on the depth of the convex hull trees and n the total number of edges of both polygons.

```
    //Step 2: Find partners in cht_1 for yet unmatched children in children_2
    for each child ∈ (children_2 ∩ um) do
        ol := the list of concavities that overlap child (according to the
            overlap graph);
        for each c ∈ ol do
            col := the list of concavities that overlap c;
            if not (child is the element of col with greatest overlap) then
                ol := ol \ {c}
            end if
        end for;
        if ol ≠ ∅ then
            lsbc := {the line segments that lie between the concavities in ol};
            concavity := join_concavities(ol, cht_1, lsbc);
            cr := trapezium_rep_builder(child, concavity,  t_1, t_2);
            mc := (mc ∪ cr) \ (mc ∩ cr);
            um := um \ {child}; um := um \ ol;
        end if
    end for;

    //Step 3: Connect still unmatched children with points (Figure A.14 (a))
    for each child ∈ ((children_1 ∪ children_2) ∩ um) do
        li := the line in the parent containing the pointer to child;
        ml := the moving line segment in mc which contains li;
        cp := one of the points in ml but not in li;
        for each line segment l in recreate_polygon(child) do
            ms := a moving line segment connecting l as and cp (a triangle);
            mc := mc ∪ {ms}
        end for;
        change ml such that it no longer contains li, but only one end point
        from li. If this turns it into a moving point, remove it entirely
    end for;
    return mc
end trapezium_rep_builder
```

Figure A.21 Algorithm trapezium_rep_builder, Part 2

A.7. Experimental Results

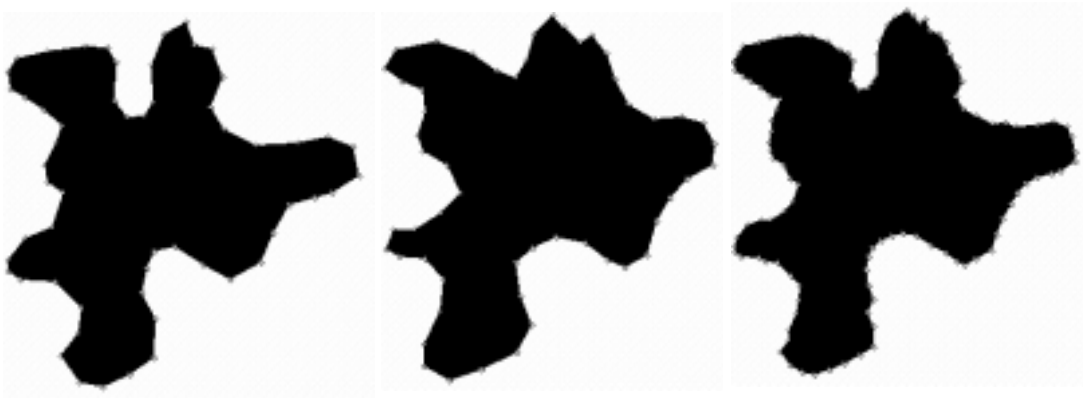
All algorithms described in this paper have been implemented in Java. The implementation is available on the Web at `http://www.idi.ntnu.no/~tossebro/mcinterpolator/interpolator.html`. There is an applet that allows one to interactively enter two snapshots and then see the interpolation, then a version for download that creates from two snapshots a VRML file which can be studied through a VRML viewer. The documented source code is also available.

The experimental results described next have been derived from this implementation. Matching multiple regions and joining separate regions (discussed in the full paper [TG02]) have not been implemented yet. The current program also has no support for holes which are not concavities. The implementation works for all regions which remain in one piece and do not move much relative to one another. (The more movement or rotation there is, the lower the quality of the resulting triangle representation will be.) The representation created by the algorithm was passed to the extensible database graphical user interface created by Miguel Rodríguez Luaces, which used it to create interpolated values between the two snapshots. All the interpolations shown in this document were created by this program.

An extension which handles multiple regions, regions with holes and regions which split and merge is planned to be built on top of the existing program.

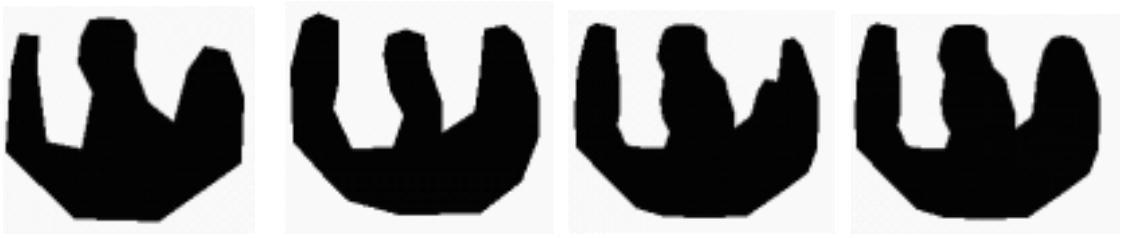
For the artificial test cases which were used to test the program for bugs, the results have in most cases become fairly good, such as in Figure A.22. However, the algorithm is very sensitive to overlap, and the smaller concavities may be erroneously matched to points if they have moved a large distance relative to the size of the concavity. This problem may be reduced by reducing the threshold overlap, that is, how much should two concavities overlap to be considered to match. The danger of reducing this criterion is that concavities might be matched erroneously if they overlap by a small percentage (The program always matches the object to the first object or combination of objects which match by more than the criterion). The overlap percentage was lowered several times during testing. The first tests were conducted with an 80% overlap requirement, while the last tests used a requirement of 10%. 5% may be even better in some cases, but with such a small overlap criterion, there is a danger of matching the concavities wrongly due to small overlaps with other concavities. Note that this problem is more likely to occur for high snapshot distances. With a very small snapshot distance, the concavities have moved little, and overlaps between “non-matching” concavities will be unlikely. With a greater snapshot distance, these overlaps may be significant. Figure A.23 shows two interpolations, one with a criterion of 40% and one with a criterion of 10%. The one with 10% clearly looks better than the one with 40%, especially the right part of the figure.

Another problem which has occurred in a few cases is that the convex hull trees have become slightly different for very similar regions, causing some strange behavior by the matching algorithm. A specific example of this is shown in Figure A.24, where changing the position of a single point causes a



First snapshot Second snapshot Interpolated value

Figure A.22 Interpolation of regular object



First snapshot Second snapshot Interpolation, overlap criterion 40%. Interpolation, overlap criterion 10%.

Figure A.23 Test object with original snapshots and interpolated values

line which previously belonged to the convex hull of the region to instead belong to the concavity. In this particular example, concavity *a* will be matched to point *b*. When the larger concavities are first matched, the thin line in concavity *a* will be matched to point *b* by the rotating plane algorithm. When concavity *a* is then added, and no matches are found for it, all the lines in it will be matched to point *b*. This interpolation artifact is clearly seen in the interpolations shown in the right part of Figure A.24, where the interpolation has two "teeth" instead of the single ending in the two snapshots. To get a good-looking result, the two lines from the real region in concavity *a* would have to be matched to the two lines *c* and *d* in the other snapshot. The program does not discover this matching because of the different positions of these two lines in the convex hull tree.

However, this problem is most often caused by using objects with few lines and sharp angles, and is therefore less likely to happen with real objects. For instance, in the first example in Figure A.24, a real object would probably have a rounded corner, which would have caused a small concavity which might be matched to concavity *a* in the rightmost figure. In the few remaining cases the concavities will likely be so small and/or thin that it will be hard to

observe the error. However, for test data which use relatively few lines, this will continue to be a problem.

Figure A.24 Convex concavity becomes non-convex

A.8. Related Work

As mentioned in the introduction, algorithms for creating interpolations between two snapshots already exist. One of these, [SG92], was designed to help creators of animated movies by generating intermediate shapes between two snapshots of cartoon figures. This is a very similar problem to the shape interpolation done by the rotating plane algorithm and convex hull tree in this paper. From the examples they have presented, it seems that their approach is better at preserving shape and avoiding some strange behaviors than ours. It is also definitely better at rotation, which it seems able to detect and account for. The problem, however, is that the user of the system must specify seven constants which are used in the interpolation. They present a table with the numbers they have used in each of their examples, and for four of the constants they are all different. This probably means that there is no set of numbers that works universally. In our approach, one of the goals is that this process should go completely automatically, without any user interaction at all. Also, the preservation of shape is not that important for our application, because the goal is to store a representation for amorphous objects and not objects with a fairly fixed shape, such as the dancing person used as an example in [SG92]. Another problem is the running time. If the user specifies an initial correspondence between two points, their algorithm runs in $O(n^2)$ time. However, if the user does not specify this, it runs in $O(n^2 \log n)$ time. The average case running time of our approach, however, is close to $O(n \log n)$. The d variable is somewhat dependent on n, because more details may be shown. However, it will grow only very slowly.

A.9. Conclusions

This paper has presented an approach to building the moving region representation described in [FGNS00] from a series of snapshots of an amorphous region. The combination of rotating plane algorithm and overlap graph seems to work well for most regions of this type, although there seem to be better approaches if an interpolation between two snapshots is all that one wants. However, if one instead wants to interpolate between five hundred snapshots, our approach seems to be a good one, because it does not demand any user interaction and has a reasonable running time. The algorithms described in the paper have been implemented. The running time has not been a problem with any of the tests that have been run up until now, even though the test program has been implemented in Java. There are some interesting possibilities for future work:

1. The matching strategies described in Section A.5 should be implemented and compared systematically. So far we have only implemented one particular choice.
2. A problem with the overlap strategies is that for a large object that is translated in the plane the smaller parts (e.g. lower level concavities) move a lot relatively to their size even though the entire object moves only a little. Hence small concavities will not overlap any more. There are several ways to compensate for this, for example, by combining overlap with a distance criterion for the small components. This should be explored in more detail and evaluated experimentally.
3. Given a large collection of snapshots of an object which moves only little, techniques for data reduction need to be developed. For example, suppose an oil spill in the sea is captured every minute, constructing interpolations between all successive snapshots may lead to an unnecessary amount of data. How can one construct a minimal representation according to some required precision?
4. Precise definitions for the quality of a series of snapshots should be developed. This should allow one to decide whether a series of observations is “good enough”. Such definitions could be given in terms of the matching strategies described in the paper.

Appendix B
Class Description for the Implementation

B.1. Class hierarchy

- class uncertainspatial.**AreaUtils**
- class uncertainspatial.**CFace** (implements uncertainspatial.SpatialObject)
- class uncertainspatial.**CLine** (implements uncertainspatial.SpatialObject)
- class uncertainspatial.**CLines** (implements uncertainspatial.MSet, uncertainspatial. SpatialObject)
- class uncertainspatial.**ContainmentTree**
- class uncertainspatial.**CPoints** (implements uncertainspatial.MSet, uncertainspatial. SpatialObject)
- class uncertainspatial.**CRegion** (implements uncertainspatial.MSet, uncertainspatial. SpatialObject)
- class uncertainspatial.**CrossCurve** (implements java.lang.Cloneable)
- class uncertainspatial.**Integers** (implements uncertainspatial.MSet)
- class uncertainspatial.**Interval** (implements java.io.Serializable)
- class java.awt.geom.Point2D (implements java.lang.Cloneable)
  - class java.awt.Point (implements java.io.Serializable)

        - class uncertainspatial.**CPoint** (implements uncertainspatial. SpatialObject)
- class uncertainspatial.**ProbFunc** (implements java.io.Serializable)
    - class uncertainspatial.**EquiProbFunc**
    - class uncertainspatial.**LinearProbFunc**
- class uncertainspatial.**ProbMassFunc** (implements java.io.Serializable)
    - class uncertainspatial.**EquiProbMassFunc**
    - class uncertainspatial.**LinearProbMassFunc**
    - class uncertainspatial.**TwoDeltaProbMassFunc**
- class uncertainspatial.**Range** (implements uncertainspatial.MSet)
- class uncertainspatial.**UncertainBoolean** (implements java.io.Serializable)
- class uncertainspatial.**UncertainObject** (implements java.io.Serializable)
    - class uncertainspatial.**UncertainInteger**
    - class uncertainspatial.**UncertainIntegers** (implements uncertainspatial. MSet)
    - class uncertainspatial.**UncertainInterval**
    - class uncertainspatial.**UncertainRange** (implements uncertainspatial. MSet)
    - class uncertainspatial.**UncertainSpatialObject** (implements uncertainspatial. SpatialObject)
        - class uncertainspatial.**UncertainCurve**
            - class uncertainspatial.**UncertainCycle**
        - class uncertainspatial.**UncertainFace**
        - class uncertainspatial.**UncertainPoint**
        - class uncertainspatial.**UncertainPoints** (implements uncertainspatial. MSet)
        - class uncertainspatial.**UncertainSegment**

B.2. Interface hierarchy

- interface uncertainspatial.**MSet**
- interface uncertainspatial.**SpatialObject**

B.3. Classes and functions

B.3.1. AreaUtils

Method Summary	
static boolean	**inside**(java.awt.Shape a1, java.awt.Shape a2) Tests whether a1 is inside a2.

B.3.2. CFace

Field Summary	
protected java.awt.geom.Area	**faceArea** An Area object that represents the face
protected java.util.ArrayList	**holeCycles** A list of holes represented as polygons
protected java.awt.Polygon	**outerCycle** The polygon defining the outer boundary of the face

Constructor Summary
CFace() Constructs an empty CFace.
CFace(java.awt.Polygon oc) Constructs a CFace that is bounded by a polygon

Method Summary	
protected static void	**addChild**(java.util.ArrayList tmpres, CFace buildface, java.awt.Polygon child, ContainmentTree ct) Adds holes to a component face in a containment tree
protected static void	**addToRoots**(java.util.ArrayList tmpres, java.awt.Polygon newroot, ContainmentTree ct) Adds a new root to the containment tree
double	**area**() Computes the area covered by this face

protected static java.awt.Polygon[]	**areaToPolygons**(java.awt.geom.Area area) Returns the polygons that an area object is made up of.
protected static java.awt.Polygon	**copyPolygon**(java.awt.Polygon in) Creates a copy of a polygon.
void	**draw**(java.awt.Graphics g) Draws a representation of this face on the screen
java.awt.geom.Area	**getFaceArea**() Gets the area covered by the face.
boolean	**insertHole**(java.awt.Polygon hole) Insert a hole into the face.
protected void	**insertSureHole**(java.awt.Polygon hole) Insert a hole into the face.
CFace[]	**intersection**(CFace f) Computes the spatial intersection of two faces.
UncertainBoolean	**intersects**(CFace f) Tests whether two faces share an area
UncertainBoolean	**intersects**(CLine l) Tests whether a line lies at least patially inside this face.
UncertainBoolean	**intersects**(CPoint p) Tests whether a point lies inside this face.
UncertainBoolean	**intersects**(SpatialObject so) Tests whether the shapes of two spatial objects share points or parts.
CFace[]	**minus**(CFace f) Computes the spatial minus of two faces.
UncertainBoolean	**outside**(CFace f) Tests whether this face geometrically contains another face.
UncertainBoolean	**outside**(CRegion r) Tests whether this face geometrically contains a region.
static CFace[]	**reconstructFaces**(java.awt.geom.Area ar) Finds the faces which make up the given area and returns them as an array of faces.
CFace[]	**union**(CFace f) Computes the spatial union of two faces.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

B.3.3. CLine

Field Summary	
`protected java.awt.Rectangle`	**`bb`** The bounding box of the curve.
`protected java.util.ArrayList`	**`vertexes`** A list of the points that make up the curve

Constructor Summary
`CLine`() Constructs an empty curve

Method Summary	
`void`	**`addVertex`**(int x, int y) Adds a vertex to the curve.
`void`	**`addVertex`**`(java.awt.Point p)` Adds a vertex to the curve.
`double`	**`area`**`()` The area of a curve
`void`	**`draw`**`(java.awt.Graphics g)` Draws a representation of this curve on the screen
`boolean`	**`equals`**`(java.lang.Object o)` Checks whether two curve object are equal, that is, that they represent the same curve.
`java.awt.Rectangle`	**`getBounds`**`()` Gets the bounding box of the curve
`java.awt.Point`	**`getVertex`**`(int a)` Gets the location of a vertex in the curve

java.util.ListIterator	**getVertexIterator**() Gets an iterator that iterates over the vertexes of this curve
UncertainBoolean	**intersects**(CLine l) Tests whether the shapes of two crisp curves share points or parts.
UncertainBoolean	**intersects**(CPoint p) Tests whether a point is on this curve.
UncertainBoolean	**intersects**(SpatialObject so) Tests whether the shapes of two spatial objects share points or parts.
int	**noVertexes**() The number of vertexes in the curve

Methods inherited from class java.lang.Object
clone, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.4. CLines

Constructor Summary
CLines() Constructs an empty ALine set.

Method Summary	
boolean	**add**(java.lang.Object element) Adds an element to this set
boolean	**contains**(java.lang.Object element) Checks whether this set contains the given element
java.lang.Object[]	**decompose**() Decomposes a set into its constituent elements
void	**draw**(java.awt.Graphics g) Draws a representation of this CLines on the screen
MSet	**intersection**(MSet intersectwith) Takes the regular set intersection of two CLines.
UncertainBoolean	**intersects**(SpatialObject o) Tests whether the shapes of two spatial objects share points or parts.

MSet	**minus**(MSet subtract) Takes the regular set difference of two CLines.
int	**noComponents**() Gets the number of members in this set
boolean	**subtract**(java.lang.Object element) Removes an element from this set
MSet	**union**(MSet unionwith) Takes the regular set union of two CLines.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.5. ContainmentTree

Constructor Summary
ContainmentTree() Constructs an empty containment tree.

Method Summary	
void	**addNode**(java.awt.Polygon p) Adds a node to the containment tree
java.awt.Polygon[]	**getChildren**(java.awt.Polygon p) Gets the children of a particular node, that is, all the polygons that are contained in a particular polygon directly.
java.awt.Polygon[]	**getRoots**() Gets the roots of the containment tree (which really is a forest)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.6. CPoints

Constructor Summary
CPoints() Constructs an empty CPoints set.

Method Summary	
boolean	**add**(java.lang.Object element) Adds an element to this set
boolean	**contains**(java.lang.Object element) Checks whether this set contains the given element
java.lang.Object[]	**decompose**() Decomposes a set into its constituent elements
void	**draw**(java.awt.Graphics g) Draws a representation of this Cpoints on the screen
MSet	**intersection**(MSet intersectwith) Takes the regular set intersection of two CPoints.
UncertainBoolean	**intersects**(SpatialObject o) Tests whether the shapes of two spatial objects share points or parts.
MSet	**minus**(MSet subtract) Takes the regular set difference of two CPoints.
int	**noComponents**() Gets the number of members in this set
boolean	**subtract**(java.lang.Object element) Removes an element from this set
MSet	**union**(MSet unionwith) Takes the regular set union of two CPoints.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.7. CRegion

Constructor Summary
CRegion() constructs an empty CRegion.
CRegion(java.awt.geom.Area a) Constructs a region from an Area object containing a set of polygons
CRegion(CFace[] fs) Constructs a CRegion from a list of faces.

Method Summary	
boolean	**add**(java.lang.Object element) Adds a face to this region.
double	**area**() Computes the area covered by this region
boolean	**contains**(java.lang.Object element) Checks whether this set contains the given element
java.lang.Object[]	**decompose**() Decomposes a set into its constituent elements
void	**draw**(java.awt.Graphics g) Draws a representation of this region on the screen
protected java.awt.geom.Area	**getArea**() Gets the area covered by this region
MSet	**intersection**(MSet intersectwith) Takes the spatial set intersection of two CRegions.
UncertainBoolean	**intersects**(SpatialObject o) Tests whether the shapes of two spatial objects share points or parts.
MSet	**minus**(MSet subtract) Takes the spatial set difference of two CRegions.
int	**noComponents**() Gets the number of members in this set
UncertainBoolean	**outside**(CFace f) Tests whether this region geometrically contains a face.
UncertainBoolean	**outside**(CRegion r) Tests whether this region geometrically contains a region.
boolean	**subtract**(java.lang.Object element) Subtracts a face from this CRegion.
MSet	**union**(MSet unionwith) Takes the spatial set union of two CRegions.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.8. CrossCurve

Field Summary	
double	**angle** The angle of the crosscurve compared to the horizontal axis
int	**length** The distance from the core line of the support (half the length of the crosscurve)

Constructor Summary
CrossCurve() Constructs a crosscurve with a length and an angle of 0.

Method Summary	
java.lang.Object	**clone**() Creates a new CrossCurve that is a copy of this one.

Methods inherited from class java.lang.Object
equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.9. Integers

Constructor Summary
Integers() Constructs an empty set of crisp integers.
Integers(java.lang.Integer[] base) Constructs a set of integers containing the given integers.

Method Summary	
boolean	**add**(java.lang.Object element) Adds an element to this set
boolean	**contains**(java.lang.Object element) Checks whether this set contains the given element
java.lang.Object[]	**decompose**() Decomposes a set into its constituent elements

Integers	**intersection**(Integers intersectwith) Takes the regular set intersection of two Integers objects.
MSet	**intersection**(MSet intersectwith) Takes the regular set intersection of two sets.
Integers	**minus**(Integers subtract) Takes the regular set difference of two Integers objects.
MSet	**minus**(MSet subtract) Takes the regular set difference of two sets.
int	**noComponents**() Gets the number of members in this set
boolean	**subtract**(java.lang.Object element) Removes an element from this set
java.lang.String	**toString**() Generates a string representation of this Integers object.
Integers	**union**(Integers unionwith) Takes the regular set union of two Integers objects.
MSet	**union**(MSet unionwith) Takes the regular set union of two sets.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

B.3.10. Interval

Constructor Summary
Interval() Creates an empty interval.
Interval(int s, int e) Creates an interval that starts and ends at the specified values.

Method Summary	
boolean	**contains**(Interval i) Tests whether this interval contains the other interval.
Interval	**copy**() Creates a copy of this interval
int	**end**() Returns the end value of the interval

`Interval[]`	**`intersection`**`(Interval i)` Takes the number-line intersection of two Intervals.
`boolean`	**`intersects`**`(Interval i)` Tests whether this interval and the input interval overlap each other.
`int`	**`length`**`()` Finds the length of the interval, that is, the distance along the number line from the start value to the end value.
`Interval[]`	**`minus`**`(Interval i)` Takes the number-line difference of two Intervals.
`int`	**`start`**`()` Returns the start value of the interval
`java.lang.String`	**`toString`**`()` Generates a string representation of this object.
`Interval[]`	**`union`**`(Interval i)` Takes the number-line union of two Intervals.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

B.3.11. CPoint

Inner classes inherited from class java.awt.geom.Point2D
`java.awt.geom.Point2D.Double, java.awt.geom.Point2D.Float`

Fields inherited from class java.awt.Point
`x, y`

Constructor Summary
`CPoint``()` Constructs an empty CPoint.
`CPoint``(int xc, int yc)` Constructs a CPoint from two coordinate values.
`CPoint``(java.awt.Point p)` Constructs a CPoint from a Point

Method Summary	
double	**area**() The area of a CPoint
void	**draw**(java.awt.Graphics g) Draws a representation of this CPoint on the screen
UncertainBoolean	**intersects**(CPoint p) Tests whether the shapes of two CPoint share points or parts.
UncertainBoolean	**intersects**(SpatialObject so) Tests whether the shapes of two spatial objects share points or parts.

Methods inherited from class java.awt.Point
equals, getLocation, getX, getY, move, setLocation, setLocation, setLocation, toString, translate

Methods inherited from class java.awt.geom.Point2D
clone, distance, distance, distance, distanceSq, distanceSq, distanceSq, hashCode, setLocation

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

B.3.12. ProbFunc

Constructor Summary
ProbFunc() Constructs a new ProbFunc object

Method Summary	
ProbFunc	**copy**() Creates a copy of this object
ProbMassFunc	**derivative**() Returns the derivative of this function.

double	**evaluate**(double input) Returns the function value for a particular x-value.
double	**inversEvaluate**(double value) Returns the inverse of the function for a particular value.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.13. EquiProbFunc

Constructor Summary
EquiProbFunc() Constructs a new EquiProbFunc object

Method Summary	
ProbFunc	**copy**() Creates a copy of this object
ProbMassFunc	**derivative**() Returns the derivative of this function.
double	**evaluate**(double input) Returns the function value for a particular x-value.
double	**inversEvaluate**(double value) Returns the inverse of the function for a particular value.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.14. LinearProbFunc

Constructor Summary
LinearProbFunc() Constructs a new LinearProbFunc object

Method Summary	
ProbFunc	**copy**() Creates a copy of this object
ProbMassFunc	**derivative**() Returns the derivative of this function.
double	**evaluate**(double input) Returns the function value for a particular x-value.
double	**inversEvaluate**(double value) Returns the inverse of the function for a particular value.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.15. ProbMassFunc

Constructor Summary
ProbMassFunc() Constructs a new ProbMassFunc object

Method Summary	
ProbFunc	**copy**() Creates a copy of this object
double	**evaluate**(double input) Returns the function value for a particular x-value.
double	**inversEvaluate**(double value) Returns the inverse of the function for a particular value.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.16. EquiProbMassFunc

Constructor Summary
`EquiProbMassFunc``()` Constructs a new EquiProbMassFunc object

Method Summary	
`ProbFunc`	**`copy`**`()` Creates a copy of this object
`double`	**`evaluate`**`(double input)` Returns the function value for a particular x-value.
`double`	**`inversEvaluate`**`(double value)` Returns the inverse of the function for a particular value.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

B.3.17. LinearProbMassFunc

Constructor Summary
`LinearProbMassFunc``()` Constructs a new LinearProbMassFunc object

Method Summary	
`ProbFunc`	**`copy`**`()` Creates a copy of this object
`double`	**`evaluate`**`(double input)` Returns the function value for a particular x-value.
`double`	**`inversEvaluate`**`(double value)` Returns the inverse of the function for a particular value.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

B.3.18. TwoDeltaProbMassFunc

Constructor Summary
TwoDeltaProbMassFunc() Constructs a new TwoDeltaProbMassFunc object

Method Summary	
ProbFunc	**copy**() Creates a copy of this object
double	**evaluate**(double input) Returns the function value for a particular x-value.
double	**inversEvaluate**(double value) Returns the inverse of the function for a particular value.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.19. Range

Constructor Summary
Range() Constructs an empty range object.
Range(Interval[] base) Constructs a range object containing a set of intervals.

Method Summary	
boolean	**add**(java.lang.Object element) Adds an interval to this range.
boolean	**contains**(java.lang.Object element) Checks whether this set contains the given element.
java.lang.Object[]	**decompose**() Decomposes a set into its constituent elements
MSet	**intersection**(MSet intersectwith) Takes the number-line intersection of two Ranges.
Range	**intersection**(Range intersectwith) Takes the number-line intersection of two Range objects.

MSet	**minus**(MSet subtract) Takes the number-line difference of two Ranges.
Range	**minus**(Range subtract) Takes the number-line difference of two Range objects.
int	**noComponents**() Gets the number of members in this set
boolean	**subtract**(java.lang.Object element) Removes an interval from this range.
java.lang.String	**toString**() Generates a string representation of this range.
MSet	**union**(MSet unionwith) Takes the number-line union of two Ranges.
Range	**union**(Range unionwith) Takes the number-line union of two Range objects.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

B.3.20. UncertainBoolean

Field Summary	
static UncertainBoolean	**FALSE** The uncertain boolean value FALSE.
static UncertainBoolean	**MAYBE** The uncertain boolean value MAYBE.
static UncertainBoolean	**TRUE** The uncertain boolean value TRUE.
protected static int	**V_FALSE** The integer value of FALSE.
protected static int	**V_MAYBE** The integer value of MAYBE.
protected static int	**V_TRUE** The integer value of TRUE.

Method Summary	
UncertainBoolean	**and**(UncertainBoolean in) Uncertain AND.
boolean	**equals**(UncertainBoolean o) Tests whether two uncertain booleans are equal.
UncertainBoolean	**not**() Uncertain NOT.
UncertainBoolean	**or**(UncertainBoolean in) Uncertain OR.
java.lang.String	**toString**() Generates a string representation of this uncetrain boolean.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

B.3.21. UncertainObject

Field Summary	
protected double	**probExistence** The probability that the uncertain object exists at all

Constructor Summary
UncertainObject() This class is only used as a superclass to other classes. This constructor therefore does nothing.

Method Summary	
abstract double	**accuracy**() The *Accuracy* of the uncertain object.
abstract java.lang.Object	**alphaCut**(double threshold) The *Alpha_Cut* of the uncetrain object.

abstract java.lang.Object	**core**() The *Core* of the uncertain object.
abstract UncertainBoolean	**equ**(UncertainObject o) This is the *Equals* operation.
abstract double	**existence**() This is the *Existence* operator.
abstract java.lang.Object	**expectedValue**() The *Expected_Value* of the uncertain object.
abstract UncertainBoolean	**moreAccurateThan**(UncertainObject o) This is the *More_Accurate_Than* operator.
abstract java.lang.Object	**support**() The *Support* of the uncertain object.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.22. UncertainInteger

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainInteger() Creates an empty uncertain integer.
UncertainInteger(int c, int deviation) Creates an uncertain integer with a given center value and deviation.
UncertainInteger(int start, int c, int end) Creates an uncertain integer with a given start, peak value and end.
UncertainInteger(int start, int c, int end, ProbMassFunc pmf) Creates an uncertain integer with a given start, peak value, end and probability density function.

Constructor Summary
`UncertainInteger``(int start, int c, int end, ProbMassFunc pmf, double pe)` Creates an uncertain integer with a given start, peak value, end, probability density function and probability of existence.
`UncertainInteger``(int c, int deviation, ProbMassFunc pmf)` Creates an uncertain integer with a given center value, deviation and probability mass function.
`UncertainInteger``(int c, int deviation, ProbMassFunc pmf, double pe)` Creates an uncertain integer with a given center value, deviation, probability mass function and probability of existence.

Method Summary	
`double`	**`accuracy`**`()` Computes the accuracy of the uncertain integer.
`java.lang.Object`	**`alphaCut`**`(double threshold)` Computes the alpha-Cut of the uncertain integer.
`java.lang.Object`	**`core`**`()` Generates the core of the uncertain integer.
`UncertainBoolean`	**`equ`**`(UncertainObject o)` Uncertain equality.
`double`	**`existence`**`()` Checks the probability that this integer exists.
`java.lang.Object`	**`expectedValue`**`()` Computes the expected value of the integer.
`UncertainBoolean`	**`intersects`**`(UncertainInterval o)` Tests whether the shapes of two uncetrain integers share points or parts.
`UncertainBoolean`	**`moreAccurateThan`**`(UncertainObject o)` This is the *More_Accurate_Than* operator.
`java.lang.Object`	**`support`**`()` Generates the support of the uncertain integer.
`java.lang.String`	**`toString`**`()` Creates a string representation of this uncertain integer.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

B.3.23. UncertainIntegers

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainIntegers() Constructs an empty set of uncertain integers.
UncertainIntegers(UncertainInteger[] base) Constructs a set of uncertain integers containing the given integers.

Method Summary	
double	**accuracy**() Computes the accuracy of the set of uncertain integers.
boolean	**add**(java.lang.Object element) Adds an element to this set
java.lang.Object	**alphaCut**(double threshold) Computes the alpha-Cut of this set of uncertain integers.
boolean	**contains**(java.lang.Object element) Checks whether this set contains the given element
java.lang.Object	**core**() Generates the core of the set of uncertain integers.
java.lang.Object[]	**decompose**() Decomposes a set into its constituent elements
UncertainBoolean	**equ**(UncertainIntegers o) Uncertain equality.
UncertainBoolean	**equ**(UncertainObject o) Uncertain equality.
double	**existence**() Checks the probability that this set of uncertain integers exists.

java.lang.Object	**expectedValue**() Computes the expected value of the set of uncertain integer.
MSet	**intersection**(MSet intersectwith) Takes the regular set intersection of two sets.
UncertainIntegers	**intersection**(UncertainIntegers intersectwith) Takes the regular set intersection of two UncertainIntegers objects.
UncertainIntegers	**intersectsWith**(UncertainInterval o) Checks which of the elements in this set can possibly be inside the given uncertain interval.
MSet	**minus**(MSet subtract) Takes the regular set difference of two sets.
UncertainIntegers	**minus**(UncertainIntegers subtract) Takes the regular set difference of two UncertainIntegers objects.
UncertainBoolean	**moreAccurateThan**(UncertainObject o) This is the *More_Accurate_Than* operator.
int	**noComponents**() Gets the number of members in this set
boolean	**subtract**(java.lang.Object element) Removes an element from this set
java.lang.Object	**support**() Generates the support of the set of uncertain integers.
java.lang.String	**toString**() Creates a string representation of this set of uncertain integers.
MSet	**union**(MSet unionwith) Takes the regular set union of two sets.
UncertainIntegers	**union**(UncertainIntegers unionwith) Takes the regular set union of two UncertainIntegers objects.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

B.3.24. UncertainInterval

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainInterval() Creates an empty uncertain interval.
UncertainInterval(int[] bs) Creates an uncertain interval with the given parameters.
UncertainInterval(int[] bs, double pe) Creates an uncertain interval with the given parameters.
UncertainInterval(int[] bs, ProbFunc pf) Creates an uncertain interval with the given parameters.
UncertainInterval(int[] bs, ProbFunc pf, double pe) Creates an uncertain interval with the given parameters.
UncertainInterval(int startsup, int startcore, int endcore, int endsup) Creates an uncertain interval with the given parameters.
UncertainInterval(int startsup, int startcore, int endcore, int endsup, double pe) Creates an uncertain interval with the given parameters.
UncertainInterval(int startsup, int startcore, int endcore, int endsup, ProbFunc pf) Creates an uncertain interval with the given parameters.
UncertainInterval(int startsup, int startcore, int endcore, int endsup, ProbFunc pf, double pe) Creates an uncertain interval with the given parameters.
UncertainInterval(UncertainInterval a) Creates an uncertain interval that is an exact copy of another one.

Method Summary	
double	**accuracy**() Computes the accuracy of the uncertain interval.

java.lang.Object	**alphaCut**(double threshold) Computes the alpha-Cut of the uncertain interval.
UncertainBoolean	**contains**(UncertainInterval i) Tests whether this uncertain interval contains the other uncertain interval.
java.lang.Object	**core**() Generates the core of the uncertain interval.
UncertainBoolean	**equ**(UncertainObject o) Uncertain equality.
double	**existence**() Checks the probability that this uncertain interval exists.
java.lang.Object	**expectedValue**() Computes the expected value of the interval.
UncertainInterval	**intersection**(UncertainInterval i) Takes the number-line intersection of two uncertain intervals.
UncertainBoolean	**intersects**(UncertainInterval i) Tests whether the shapes of two uncetrain intervals share numbers or parts.
UncertainInterval[]	**minus**(UncertainInterval i) Takes the number-line difference of two uncertain intervals.
UncertainBoolean	**moreAccurateThan**(UncertainObject o) Returns TRUE if this object is a more accurate version of the given object.
java.lang.Object	**support**() Generates the support of the uncertain interval.
java.lang.String	**toString**() Generates a string representation of this object.
UncertainInterval[]	**union**(UncertainInterval i) Takes the number-line union of two uncertain intervals.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

B.3.25. UncertainRange

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainRange() Constructs an empty uncertain range object.
UncertainRange(UncertainInterval[] base) Constructs an uncertain range that contains the given uncetrain intervals.

Method Summary	
double	**accuracy**() Computes the accuracy of the uncertain range.
boolean	**add**(java.lang.Object element) Adds an uncertain interval to this uncertain range.
java.lang.Object	**alphaCut**(double threshold) Computes the alpha-Cut of the uncertain range.
boolean	**contains**(java.lang.Object element) Checks whether this set contains the given element.
java.lang.Object	**core**() Generates the core of the uncertain range.
java.lang.Object[]	**decompose**() Decomposes a set into its constituent elements
UncertainBoolean	**equ**(UncertainObject o) Uncertain equality.
UncertainBoolean	**equ**(UncertainRange o) Uncertain equality.
double	**existence**() The probability that a range exists is the probability that at least one of the intervals it contains exists.
java.lang.Object	**expectedValue**() Computes the expected value of the range.
MSet	**intersection**(MSet intersectwith) Takes the number-line intersection of two uncertain Ranges.

UncertainRange	**intersection**(UncertainRange intersectwith) Takes the number-line intersection of two UncetrainRange objects.
UncertainBoolean	**intersects**(UncertainRange intersectswith) Tests whether the shapes of two uncetrain ranges share numbers or parts.
MSet	**minus**(MSet subtract) Takes the number-line difference of two uncertain Ranges.
UncertainRange	**minus**(UncertainRange subtract) Takes the number-line difference of two UncertainRange objects.
UncertainBoolean	**moreAccurateThan**(UncertainObject o) MoreAccurateThan for a range is true iff all the intervals of the given range (O) are more accurate versions of intervals in this range, and there is a one-to-one correspondence between intervals in O and in this range.
int	**noComponents**() Gets the number of members in this set
boolean	**subtract**(java.lang.Object element) Removes an uncetrain interval from this ucnertain range.
java.lang.Object	**support**() Generates the support of the uncertain range.
java.lang.String	**toString**() Generates a string representation of this uncertain range.
MSet	**union**(MSet unionwith) Takes the number-line union of two uncertain Ranges.
UncertainRange	**union**(UncertainRange unionwith) Takes the number-line union of two UncertainRange objects.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

B.3.26. UncertainSpatialObject

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainSpatialObject() This class is only used as a superclass to other classes. This constructor therefore does nothing.

Method Summary	
abstract double	**area**() The *Area* operator.
abstract double	**resemble**(UncertainSpatialObject comparison) The *Resemble* operator is defined for all uncetrain spatial objects, and can be used to test degree of equality.

Methods inherited from class uncertainspatial.UncertainObject
accuracy, alphaCut, core, equ, existence, expectedValue, moreAccurateThan, support

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface uncertainspatial.SpatialObject
draw, intersects

B.3.27. UncertainCurve

Field Summary	
protected CrossCurve	**finalCC** The last crosscurve in this curve.
protected double	**finalExist** The probability of existence of the curve in the last point.

protected java.awt.Point	**finalPoint** The last point in this curve.
protected java.util.ArrayList	**segments** The list of segments that this curve consists of.

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainCurve() Creates an empty uncertain curve object.
UncertainCurve(java.awt.Point cp1, java.awt.Point cp2, CrossCurve c1, CrossCurve c2, ProbFunc pf, ProbMassFunc pmf, double pexist1, double pexist2, double pexistline) Creates an uncertain curve that consists of one segment.

Method Summary	
double	**accuracy**() Computes the accuracy of the uncertain curve
void	**addSegment**(java.awt.Point cp, CrossCurve c, ProbFunc pf, ProbMassFunc pmf, double pexist) Adds a segment to the uncertain curve.
java.lang.Object	**alphaCut**(double threshold) Computes the alpha-Cut of the uncertain curve.
double	**area**() Computes the area of the support of the uncertain curve.
protected CFace	**buildFace**(CLine upper, CLine lower) This function creates a face (with no holes) from two CLine objects, each representing one side of the uncertain curve.
java.lang.Object	**core**() Computes the core of the uncertain curve.
void	**draw**(java.awt.Graphics g) Draws a representation of this uncertain curve on the screen
UncertainBoolean	**equ**(UncertainObject o) Uncertain equality.

double	**existence**() This is the *Existence* operator.
java.lang.Object	**expectedValue**() Computes the expected value of the curve.
UncertainBoolean	**intersects**(SpatialObject o) Tests whether the shapes of two spatial objects share points or parts.
UncertainBoolean	**moreAccurateThan**(UncertainObject o) This is the *More_Accurate_Than* operator.
double	**resemble**(UncertainSpatialObject comparison) The *Resemble* operator is not implemented for curves yet.
java.lang.Object	**support**() Computes the support of the uncertain curve.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.28. UncertainCycle

Field Summary	
protected ProbFunc	**allProbFunc** The probability function that is used along all the segments of this cycle.
protected UncertainSegment	**closingSegment** The segment that closes the cycle.
protected java.awt.Polygon	**innerPoly** The inner polygon of the uncertain cycle.
protected double	**outerMult** The multiplier to apply to the crosscurves to get the outer polygon.
protected java.awt.Polygon	**outerPoly** The outer polygon of the uncertain cycle.

Fields inherited from class uncertainspatial.UncertainCurve
finalCC, finalExist, finalPoint, segments

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainCycle(java.awt.Point cp1, java.awt.Point cp2, CrossCurve c1, CrossCurve c2, ProbMassFunc pmf, double pexistline) Creates an uncertain cycle that consists of one segment.

Method Summary	
void	**addSegment**(java.awt.Point cp, CrossCurve c, ProbFunc pf, ProbMassFunc pmf, double pexist) Adds a segment to the uncertain cycle.
void	**addSegment**(java.awt.Point cp, CrossCurve c, ProbMassFunc pmf) Adds a segment to the uncertain cycle.
protected void	**computePolys**() Computes the two polygons that bound the support of this uncertain cycle and finds which is the inner and which is the outer.
java.awt.Polygon	**getFaceHoleAlpha**(ProbFunc pf, double threshold) Computes the part of the alpha-cut of a face that has this cycle as one of its holes.
java.awt.Polygon	**getFaceOuterAlpha**(ProbFunc pf, double threshold) Computes the part of the alpha-cut of a face that has this cycle as its outer boundary.
java.awt.Polygon	**getInnerPolygon**() Computes the inner polygon of the uncertain cycle.
java.awt.Polygon	**getOuterPolygon**() Computes the outer polygon of the uncertain cycle.

Methods inherited from class uncertainspatial.UncertainCurve
accuracy, alphaCut, area, buildFace, core, draw, equ, existence, expectedValue, intersects, moreAccurateThan, resemble, support

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.29. UncertainFace

Field Summary	
protected java.util.ArrayList	**holeCycles** The list of hole cycles.
protected UncertainCycle	**outerCycle** The outer cycle of the uncertain face.
protected ProbFunc	**probDistFunc** the probability distribution function that applies over all the uncertain cycles of this face.

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainFace() Creates an empty uncertain face.
UncertainFace(UncertainCycle oC, double pe, ProbFunc pf) Creates an uncertain face

Method Summary	
double	**accuracy**() Computes the accuracy of the uncertain face.
java.lang.Object	**alphaCut**(double threshold) Computes the alpha-Cut of the uncertain face.
double	**area**() Computes the area of this uncertain face.
java.lang.Object	**core**() Generates the core of the uncertain face.

void	**draw**(java.awt.Graphics g) Draws a representation of this uncertain face on the screen
UncertainBoolean	**equ**(UncertainObject o) Uncertain equality.
double	**existence**() Checks the probability that this face exists.
java.lang.Object	**expectedValue**() Computes the expected value of the face.
boolean	**insertHole**(UncertainCycle hC) Inserts a hole cycle if it is completely inside the support of the outer cycle.
UncertainBoolean	**intersects**(SpatialObject o) Tests whether the shapes of two uncetrain faces share points or parts.
UncertainBoolean	**moreAccurateThan**(UncertainObject o) This is the *More_Accurate_Than* operator.
double	**resemble**(UncertainSpatialObject comparison) The *Resemble* operator for uncertain faces.
java.lang.Object	**support**() Generates the support of the uncertain face.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.30. UncertainPoint

Field Summary	
protected java.awt.Point	**core** The location of the core.
static int	**NOCORNERS** This constant indicates how many corners the support of an uncertain point should have.

protected ProbMassFunc	**probmass** The probability mass function that applies along all lines from the core to the support.
protected int[]	**support** The distance from the core to each of the corners.

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainPoint() Constructs an empty uncertain point.
UncertainPoint(int[] corners, java.awt.Point c) Constructs an uncertain point with the given parameters.
UncertainPoint(int[] corners, java.awt.Point c, ProbMassFunc fs) Constructs an uncertain point with the given parameters.
UncertainPoint(int[] corners, java.awt.Point c, ProbMassFunc fs, double pe) Constructs an uncertain point with the given parameters.

Method Summary	
double	**accuracy**() Computes the accuracy of the uncertain point.
java.lang.Object	**alphaCut**(double threshold) Computes the alpha-Cut of the uncertain point.
double	**area**() Computes the area of the support of the uncertain point.
java.lang.Object	**core**() Generates the core of the uncertain point.
void	**draw**(java.awt.Graphics g) Draws this uncertain point.
UncertainBoolean	**equ**(UncertainObject o) Uncertain equality.
double	**existence**() Checks the probability that this uncertain point exists.

java.lang.Object	**expectedValue**() Computes the expected value of the uncertain point.
protected int	**getSupXCoordinate**(int c) Computes the x-coordinate of a given corner of the support
protected int	**getSupYCoordinate**(int c) Computes the y-coordinate of a given corner of the support
UncertainBoolean	**intersects**(CFace f) Tests whether this point may lie inside the given crisp face.
UncertainBoolean	**intersects**(CLine l) Tests whether a crisp curve lies at least partially inside the support of this point.
UncertainBoolean	**intersects**(CPoint p) Tests whether a crisp point lies inside the support of this point.
UncertainBoolean	**intersects**(SpatialObject so) Tests whether the shapes of two spatial objects share points or parts.
UncertainBoolean	**intersects**(UncertainPoint ip) Tests whether the shapes of two uncertain points share points or parts.
UncertainBoolean	**moreAccurateThan**(UncertainObject o) Returns TRUE if this object is a more accurate version of the given object.
double	**resemble**(UncertainSpatialObject comparison) The resemplance of two uncertain points is the area of the intersection of the supports of the two points divided by the area of the uncertain point with the largest area.
java.lang.Object	**support**() Generates the support of the uncertain point.
protected java.awt.Polygon	**supportAsPolygon**() Generates a polygon representation of the support of this uncertain point.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.31. UncertainPoints

Field Summary	
protected java.util.HashSet	**points** The set of uncertain points.

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainPoints() constructs an empty set of uncertain points.

Method Summary	
double	**accuracy**() Computes the accuracy of the set of uncertain point.
boolean	**add**(java.lang.Object element) Adds a new member to this set.
java.lang.Object	**alphaCut**(double threshold) Computes the alpha-Cut of this set of uncertain points.
double	**area**() Computes the area of the support of this set of uncertain points.
boolean	**contains**(java.lang.Object element) Checks whether this set contains the given element
java.lang.Object	**core**() Generates the core of the set of uncertain points.
java.lang.Object[]	**decompose**() Decomposes a set into its constituent elements
void	**draw**(java.awt.Graphics g) Draws all the members of this set of uncertain points.
UncertainBoolean	**equ**(UncertainObject o) Uncertain equality.

double	**existence**() An uncertain points object exists if at least one of its members exist.
java.lang.Object	**expectedValue**() Computes the expected value of this set of uncertain points.
MSet	**intersection**(MSet intersectwith) Takes the regular set intersection of two $DM_{UPoints}$.
UncertainPoints	**intersection**(UncertainPoints intersectwith) Takes the regular set intersection of two $DM_{UPoints}$.
UncertainBoolean	**intersects**(SpatialObject o) Finds whether this set of uncertain points intersects the given spatial object.
MSet	**minus**(MSet subtract) Takes the regular set difference of two $DM_{UPoints}$.
UncertainPoints	**minus**(UncertainPoints subtract) Takes the regular set difference of two $DM_{UPoints}$.
UncertainBoolean	**moreAccurateThan**(UncertainObject o) Returns TRUE if this object is a more accurate version of the given object.
int	**noComponents**() Gets the number of members in this set
double	**resemble**(UncertainSpatialObject comparison) The resemplance of two sets of uncertain points.
boolean	**subtract**(java.lang.Object element) Removes an element from this set
java.lang.Object	**support**() Generates the support of the set of uncetrain points.
MSet	**union**(MSet unionwith) Takes the regular set union of two $DM_{UPoints}$.
UncertainPoints	**union**(UncertainPoints unionwith) Takes the regular set union of two $DM_{UPoints}$.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.32. UncertainSegment

Field Summary	
protected ProbMassFunc	**cCFunction** The probability mass function along the crosscurves.
protected ProbFunc	**coreFunction** The probability distribution function along the core segment.
protected CrossCurve	**endCC** The crosscurve at the end of this segment.
protected double	**endExist** The probability of existence in the end point.
protected java.awt.Point	**endPoint** The point in which this segment ends.
protected CrossCurve	**startCC** The crosscurve at the start of this segment.
protected double	**startExist** The probability of existence in the start point.
protected java.awt.Point	**startPoint** The point in which this segment starts.

Fields inherited from class uncertainspatial.UncertainObject
probExistence

Constructor Summary
UncertainSegment() Constructs an empty uncetrain line segment.
UncertainSegment(java.awt.Point start, java.awt.Point end, CrossCurve sc, CrossCurve ec, ProbFunc pf, ProbMassFunc pmf, double se, double ee, double pe) Constructs an uncertain line segment with all parameters.

Method Summary	
double	**accuracy**() Computes the accuracy of this uncertain segment.

java.lang.Object	**alphaCut**(double threshold) Alpha-Cut of uncertain segment.
protected CLine[]	**alphaCutCurve**(double threshold) This function computes the contribution of this segment to the alpha-cut of an uncertain curve.
protected java.awt.Point[]	**alphaPoints**(java.awt.Point cp, CrossCurve cc, ProbMassFunc pmf, double threshold) Computes the relative alpha-cut along a single crosscurve.
double	**area**() Computes the area of the support of the uncertain segment.
protected double	**areaOfQuadrangle**(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4) Returns the area of a given quadrangle (shape with four corners).
protected double	**areaOfTriangle**(int p1x, int p1y, int p2x, int p2y, int p3x, int p3y) Computes the area of a triangle given by the positions of its corners.
void	**changeBeginning**(java.awt.Point start, CrossCurve sc, double se) Alters this uncertain segment by changing its beginning.
void	**changeEnd**(java.awt.Point end, CrossCurve ec, double ee) Alters this uncertain segment by changing its end.
java.lang.Object	**core**() Computes the core of this uncertain line segment.
void	**draw**(java.awt.Graphics g) Draws a representation of this uncertain segment on the screen
UncertainBoolean	**equ**(UncertainObject o) Uncertain equality of uncertain line segments has not been implemented yet.
double	**existence**() This is the *Existence* operator.
java.lang.Object	**expectedValue**() Computes the expected value of the segment.
protected CLine	**getPartFaceAlpha**(double mult) Gets the contribution of this segment to the alpha-cut of an uncertain face.

UncertainBoolean	**intersects**(SpatialObject o) Tests whether the shapes of two spatial objects share points or parts.
UncertainBoolean	**moreAccurateThan**(UncertainObject o) This is the *More_Accurate_Than* operator.
protected java.awt.Point	**pointOnLine**(java.awt.Point oldpoint, java.awt.Point cpoint, double position) Computes a point along the core line segment.
double	**resemble**(UncertainSpatialObject comparison) The *Resemble* operator is not implemented for segments yet.
java.lang.Object	**support**() Computes the support of the uncertain segment.
protected CLine[]	**supportCurve**() This function computes the contribution of this segment to the support of an uncertain curve.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

B.3.33. Interface MSet

Method Summary	
boolean	**add**(java.lang.Object element) Adds an element to this set
boolean	**contains**(java.lang.Object element) Checks whether this set contains the given element
java.lang.Object[]	**decompose**() Decomposes a set into its constituent elements
MSet	**intersection**(MSet intersectwith) Takes the regular set intersection of two sets This is the Intersection operation for all data types except those that represent sets in themselves (such as intervals, faces and regions).
MSet	**minus**(MSet subtract) Takes the regular set difference of two sets.
int	**noComponents**() Gets the number of members in this set

`boolean`	**`subtract`**`(java.lang.Object element)` Removes an element from this set
`MSet`	**`union`**`(MSet unionwith)` Takes the regular set union of two sets.

B.3.34. Interface SpatialObject

Method Summary	
`void`	**`draw`**`(java.awt.Graphics g)` Draws a representation of this spatial object on the screen
`UncertainBoolean`	**`intersects`**`(SpatialObject o)` Tests whether the shapes of two spatial objects share points or parts.

References

[Alt94] D. Altman: Fuzzy set theoretic approaches for handling imprecition in spatial analysis. In *Int. J. Geographical Information Systems*, 8(3), pages 271-289, 1994.

[AR99] T. Abraham and J. F. Roddick: Survey of Spatio-Temporal Databases. *Geoinformatica* 5(1), pp. 61-100, 1999.

[BCW95] W. C. Booth, G. C. Colomb, J. M. Williams: The Craft of Research. *The University of Chicago Press*, 1995.

[Ben75] J. L. Bentley: Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM* 18(9), 1975, pp. 509-517.

[Ben79] J. L. Bentley: Multidimensional Binary Search Trees in Database Applications. *IEEE Trans. on Software Engineering* 4(5), 1979, pp. 333-340.

[BF79] J. L. Bentley and J. H. Friedman: Data Structures for Range Searching. *ACM Computing Surveys* 11(4), 1979, pp. 397-409.

[BJSS98] R. Blijute, C. S. Jenses, S. Saltenis, G. Slivinskas: R-tree Based Indexing of Now-Relative Bitemporal Data. *Proc. 24th VLDB Conference*, 1998, pp. 345-356.

[BKOS98] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf: *Computational Geometry: Algorithms and Applications, 2nd edition*. Springer, 1998.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger: The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1990, pp. 322-331.

[BM72] R. Bayer and E. M. McCreight: Organization and maintenance of large ordered indices. *Acta Informatica*, vol. 1, 1972, pp. 173-189.

[CF96] E. Clementini and P. Di Felice: An Algebraic Model for Spatial Objects with Indeterminate Boundaries. In *Geographic Objects with Indeterminate Boundaries*, GISDATA series vol. 2, Taylor & Francis, 1996, pages 155-169.

[CG94] T. S. Cheng and S. K. Gadia: A Pattern Matching Language for Spatio-Temporal Databases. In *Proc. ACM Conf. on Information and Knowledge Management,* 1994, pp. 288-295.

[CG96] A. G. Cohn and N. M. Gotts: The 'Egg-Yolk' Representation of Regions with Indeterminate Boundaries. In *Geographic Objects with Indeterminate Boundaries*, GISDATA series vol. 2, Taylor & Francis, 1996, pages 171-187.

[CMB97] T. Cheng, M. Molenaar, T. Bouloucos: Identification of Fuzzy Objects from Field Observation Data. In S. C. Hirtle and A. U. Frank (eds.) *Spatial Information Theory: A Theoretical Foundation for GIS*, LNCS vol. 1329, Springer-Verlag, 1997, pages 241-259.

[CM99a] T. Cheng and M. Molenaar: Objects with Fuzzy Spatial Extent. In *Photogrammetric Engineering and Remote Sensing*, 65(7), 1999, pages 797-801.

[CM99b] T. Cheng and M. Molenaar: Diachronic Analysis of Fuzzy Objects. In *Geoinformatica* 3(4), 1999, pages 337-355.

[CR97] J. Chomicki and P. Revesz: Constraint-Based Interoperability of Spatio-Temporal Databases. In *Proc. 5th Int. Symp. on Large Spatial Databases,* pp. 142.161, Berlin, Germany, 1997.

[CR99] J. Chomicki and P. Revesz: A Geometric Framework for Specifying Spatiotemporal Objects. In *Proc. 6th Int. Workshop on Temporal Representation and Reasoning (TIME)*, pp. 41-46, 1999.

[DMSW01] M. Duckham, K. Mason, J. Stell, M. Worboys: A formal approach to imperfection in geographic information. In *Computers, Environment and Urban Systems*, 25, 2001, pages 89-103.

[DMY93] K. Dobrindt, K. Mehlhorn, M. Yvinec: A Complete and Efficient Algorithm for the Intersection of a General and a Convex Polyhedron. In *Proc. 3rd Workshop on Algorithms and Data Structures*, pp. 314-324, 1993.

[DP73] D. H. Douglas and T. K: Peucker: Algorithms for the reduction of the number of points required to represent a digitized line or its carricature. *The Canadian Cartographer,* 10(2), 1973, pages 112-122.

[DS98] C. E. Dyreson and R. T. Snodgrass: Supporting Valid-time Indeterminacy. In *ACM Trans. on Database Systems*, 23(1), 1998, pages 1-57.

[Dut92] G. Dutton: Handling Positional Uncertainty in Spatial Databases. *SDH'92*, vol. 2, pages 460-469

[Edw94] G. Edwards: Characterizing and Maintaining Polygons with Fuzzy Boundaries in Geographic Information Systems. In *6th Int. Symp. on Spatial Data Handling (SDH'94)* vol. 1, pages 223-239

[EGSV98] M. Erwig, R. H. Güting, M. Schneider, M. Vazirgiannis: Abstract and Discrete Modeling of Spatio-Temporal Data Types. *ACM-GIS 1998*: pages 131-136

[ES97] M. Erwig and M. Schneider: Vague Regions. In *Proc. 5th Symp. on Advances in Spatial Databases (SSD)*, LNCS 1262, 1997, pages 298-320.

[FGNS00] L. Forlizzi, R. H. Güting, E. Nardelli, M. Schneider: A Data Model and Data Structures for Moving Objects Databases. *Proc. ACM SIGMOD Int. Conf. on Management og Data* (Dallas, Texas), 2000, pp. 319-330.

[FP95] L. de Floriani and E. Puppo: Hierarchical Triangulation for Multiresolution Surface Description. *ACM Transactions on Graphics*, 14(4), 1995, Pages 363-411

[Free87] M. Freeston: The BANG file: a new kind of grid file. *ACM SIGMOD Int. conf. on Management of Data*, 1987, pp. 260-269

[GBE+00] R. H. Güting, M. F. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, M. Vazirgiannis: A Foundation for Representing and Querying Moving Objects. In *ACM Transactions on Database Systems* 25(1), 2000, pp. 1-42.

[GG98] V. Gaede and O. Günther: Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), June 1998, pp. 171-231

[Gra72] R. L. Graham: An efficient algorithm for determining the convex hull of a finite planar set. In *Information Processing Letters*, 1, 1972.

[GS95] R. H. Güting and M. Schneider: Relm-Based Spatial Data Types: The ROSE Algebra. In *VLDB Journal 4* (1995), pages 100-143.

[Gutt84] A. Guttman: R-trees: A dynamic index structure for spatial searching. *Proc. ACM SIGMOD Int. conf. on Management of Data*, 1984, pp. 47-54.

[Hell90] M. Heller: Triangulation algorithms for adaptive terrain modelling. In *Proc. 4th Int. Symp. on Spatial Data Handling* (*SDH90*), 1990, pp. 163-174.

[HG95] G. J. Hunter and M. F. Goodchild: Dealing with error in spatial databases: A simple case study. In *Photogrammetric Engineering and Remote Sensing*, 61(5), 1995, pages 529-547.

[Hinr85] K. Hinrichs: Implementation of the Grid File: design concepts and experience. *BIT* 25, 1985, pp. 569-592.

[HSW88] A. hutflesz, H.-W. Six, P. Widmayer: Twin grid files: Space Optimizing Access Schemes. *Proc. ACM SIGMOD Int. conf. on Management of Data*, 1988, pp. 183-190.

[Kel95] S. F. Keller: Potentials and limitations of artificial intelligence techniques applied to generalization. In J.-C. Müller, J.-P. Lagrange, R. Weibel: *GIS and Generalization: Methodology and Practice*. London: Taylor & Francis, 1995.

[LAB96] P. Lagacherie, P. Andrieux and R. Bouzigues: Fuzziness and Uncertainty of Soil Boundaries: From Reality to Coding in GIS: In *Geographic Objects with Indeterminate Boundaries*, GISDATA series vol. 2, Taylor & Francis, 1996, pages 275-298.

[LG00] J. A. Cotelo-Lema and R. H. Güting: Dual Grid: A New Approach for Robust Spatial Algebra Implementation. FernUniversität Hagen, Informatik-Report 268, May 2000. To appear in *GeoInformatica*

[Low94] K. Lowell: An Uncertainty-Based Spatial Representation for Natural Resources Phenomena. *SDH'94* vol. 2, pages 933-944.

[LS80] D. T: Lee and B. J. Schachter: Two algorithms for constructing a delaunay triangulation. *Int. Journal of Computer and Information Sciences*, 9(3), 1980, pp. 219-242.

[LS89] D. B. Lomet and B. Salzberg: A Robust Multi-Attribute Search Structure. *Proc. 5th IEEE Int. conf. on Data Engineering*, 1989, pp. 296-304.

[MC89] D. M. Mark and F. Csillag: The nature of boundaries of 'area-class' maps. In *Cartographica*, 26, 1989, pages 65-77.

[MS92] R. B. McMaster and K. S. Shea: Generalization in Digital Cartography. *The Association of American Geographers*, 1992.

[NHS84] J. Nievergelt, H. Hinterberger, K. C. Sevcik: The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. on Database Systems*, 9(1), 1984, pages 38-71.

[NP82] J. Nievergeld and F. P. Preparata: Plane-Sweep Algorithms for Intersecting Geometric Figures. In *Communications of the ACM*, 25(10), 1982, pp. 739-747.

[NST99] M. A. Nascimento, J. R. O. Silva, Y. Theodoridis: Evaluation of Access Structures for Discretely Moving Points. *Int. workshop on Spatio-Temporal Database Management*, 1999, pp. 171-188

[OM84] J. A. Orenstein and T. H. Merrett: A class of data structures for Associative Searching. *Proc. 3rd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1984, pp. 100-113.

[Peu99] D. J. Peuquet: Time in GIS and geographic databases. In P. A. Longley, M. F. Goodchild, D. M. Maguire, D. W. Rhind: *Geographical Information Systems, 2nd edition.* John Wiley & Sons, 1999.

[PD95] D. J. Pequet and N. Duan: An Event-Based Spatiotemporal Data Model (ESTDM) for Temporal Analysis of Geographical Data. *Int. Journal of Geographical Information Systems*, 9(1), 1995, pp. 7-24.

[PJ99] D. Pfoser and C. S. Jensen: Capturing the uncertainty of moving-object representations. In *Proc. 6th Int. Symposium on Advances in Spatial Databases (SSD'99)*, 1999, pp. 111-132.

[PS85] F. P. Preparata and M. I. Shamos: Computational Geometry: An Introduction. *Springer-Verlag*, New York, 1985.

[Ren99] A. Renolen: Concepts and Methods for Modelling Temporal and Spatiotemporal Information. *Dr. Ing Thesis*, Norwegian University of Scirnce and Technology (NTNU), 1999

[Rob81] J. T. Robinson: The K-D-B-Tree: A search structure for Large Multidimensional Dynamic Indexes. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, 1981, pp. 10-18.

[Rod00] M. Rodríguez-Luaces: A Spatio-Temporal Algebra Implementation. In *Proc. 5th World Conf. on Integrated Design and Process Technology*, June 2000.

[Sam90] H. Samet: Applications of spatial data structures. *Addison-Wesley*, 1990.

[Sch96] M. Schneider: Modelling Spatial Objects with Undeterminate Boundaries using the Realm/ROSE Approach. In *Geographic Objects with Indeterminate Boundaries*, GISDATA series vol. 2, Taylor & Francis, 1996, pages 155-169.

[Sch99] M. Schneider: Uncertainty Management for Spatial Data in Databases: Fuzzy Spatial Data Types. In *Proc. 6th Int. Symp. on Advances in Spatial Databases (SSD)*, LNCS 1651, Springer Verlag, 1999, pages 330-351.

[Sch00a] M. Schneider: Finite Resolution Crisp and Fuzzy Spatial Objects. In *Proc. 9th Int. Symp. on Spatial Data Handling*, 5a, 2000, pages 3-17.

[Sch00b] M. Schneider: Metric Operations on Fuzzy Spatial Objects in Databases. *In Proc. 8th ACM Symp. on Geographic Information Systems (ACM GIS)*, 2000, pages 21-26.

[Sch01] M. Schneider: Fuzzy Spatial Querying Based on Topological Predicates for Complex Crisp and Fuzzy Regions. In *Proc. ER2001 Conference*, 2001.

[SG92] T. W. Sederberg and E. Greenwood: A Physically Based Approach to 2-D Shape Blending. In *Computer Graphics (Proc. ACM SIGGRAPH)*, 26(2), 1992, 25-34.

[SW99] J. G. Stell and M. F. Worboys: Generalizing Graphs Using Amalgamation and Selection. In *Proc. 6th Int. Symp. on Advances in Spatial Databases (SSD99)*, pages 19-32.

[SWCD97] A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao: Modeling and Querying Moving Objects. In *Proc. Int. Conf. on Data Engineering*, 1997, pp. 422-432.

[TG01] E. Tøssebro and R. H. Güting: Creating Representations for Continuously Moving Regions from Observations. In *Proc. 7th Int. Symp. on Spatial and Temporal Databases*, July 2001, pages 321-344.

[TG02] E. Tøssebro and R. H. Güting: Creating Representations for Continuously Moving Regions from Observations. FernUniversität Hagen, Informatik-Report, in preparation, 2002.

[TN02a] E. Tøssebro and M. Nygård: Representing Uncertainty in Spatial Databases. *Submitted for conference publication*.

[TN02b] E. Tøssebro and M. Nygård: Uncertainty in Spatiotemporal Databases. To be published in *Proc. 2nd Int. Conf. on Advances in Information Systems (ADVIS)*, October 2002.

[TN02c] E. Tøssebro and M. Nygård: An Advanced Discrete Model for Uncertain Spatial Data. To be published in *Proc. 3rd Int. Conf on Web-Age Information Management (WAIM)*, August 2002.

[TN02d] E. Tøssebro and M. Nygård: Medium Complexity Discrete Model for Uncertain Spatial Data. *Submitted for conference publication.*

[TN02e] E. Tøssebro and M. Nygård: Extending Discrete Models for Uncerta Spatial Data to Spatiotemporal Data. *Submitted for conference publication.*

[TN02f] E. Tøssebro and M. Nygård: Abstract and Discrete models for Uncertain Spatiotemporal Data. Poster presentation at *14th International Conference on Scientific and Statistical Database Management (SSDBM 2002)*. An abstract is in the proceedings

[TN02g] E. Tøssebro and M. Nygård: Representing Uncertainty in Spatiotemporal Databases. *To be submitted for journal publication*.

[TN02h] E. Tøssebro and M. Nygård: Three Discrete Models for Uncertainty in Spatiotemporal Databases. *To be submitted for journal publication*.

[Tsai93] V. J. D. Tsai: Delaunay Triangulations in TIN Creation: An Overview and a Linear-Time Algorithm. *Int. Journal of Geographical Information Systems* 7(6), 1993, pages 501-524.

[Wei95] R. Weibel: Three Essential Building Blocks for Automated Generalization. In J.-C. Müller, J.-P. Lagrange, R. Weibel: *GIS and Generalization: Methodology and Practice*. London: Taylor & Francis, 1995, pages 56-69.

[WH96] F. Wang and G. B: Hall: Fuzzy representation of geographical boundaries in GIS. In *Int. Journal of Geographical Information Systems*, 10(5), 1996, pages 573-590.

[Win00] S. Winter: Topological Relations between Imprecise Regions. In *Int. Journal of Geographical Information Science*, 14(5), 2000, pages 411-430.

[Wor94] M. F. Worboys: A Unified Model for Spatial and Temporal Information. In *The Computer Journal* 37(1), 1994, pp. 26-34.

[Wor98] M. F. Worboys: Imprecision in Finite Resolution Spatial Data. In *GeoInformatica*, 2(3), 1998, pages 257-279.

[WXCJ98] O. Wolfson, B. Xu, S. Chamberlain, L. Jiang: Moving Object Databases: Issues and Solutions. In *Proc. 10th Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, 1998, pages 111-122

[Zad65] L. A. Zadeh (1965): Fuzzy sets. *Information and Control*, 8, 1965, pp. 338-353.

[Zhan98] F. B. Zhan: Approximate analysis of binary topological relations between geographic regions with indeterminate boundaries. *Soft Computing* 2, 1998, pages 28-34.

Index

T

U

V

W