Doctoral Thesis

Algorithms, complexity, and software engineering in computational geometry case studies

Author(s):
Neyer, Gabriele

Publication Date:
2000

Permanent Link:
https://doi.org/10.3929/ethz-a-003883325

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Algorithms, Complexity, and Software Engineering in Computational Geometry
Case Studies

A dissertation submitted to the
Swiss Federal Institute of Technology (ETH) Zürich
for the degree of Doctor of Technical Sciences

presented by
Gabriele Neyer
Dipl. Informatikerin and Dipl. Mathematikerin TU-Berlin
born July 23, 1968 in Ibbenbüren, Germany
citizen of Germany

accepted on the recommendation of
Prof. Dr. Peter Widmayer, Examiner
Dr. habil. Frank Wagner, Co-Examiner
Acknowledgments

Hidden behind the lines and figures of this thesis is my life during the last three years which I was fortunate to spend with my colleagues and friends at the ETH.

First of all, I wish to express my thanks to my advisor Prof. Peter Widmayer for his support and guidance. His comprehensive knowledge and quick understanding was extremely helpful. I'm also very grateful for the freedom of choice of research topics.

Special thanks goes to Frank “Car-Walker” Wagner for being my co-examiner and for the great time at the white-board, on the balcony, and at the Werdivsel, where we had interesting and inspiring research sessions and discussions.

I also want to thank Lutz “C++” Kettner for introducing me to CGAL, his competent advice in software engineering and for proof-reading the CGAL part of this thesis. Furthermore, I thank Michael “Emergency CGAL Hacker” Hoffmann and Dimitrii “Remote Debugging” Pasechnik for their patience in helping me fix countless Internal Compiler Errors caused by my software.

I am very grateful to Mike “Drug Designer” Hallett and Arne “The Genius” Storjohann for many fruitful discussions about my research — and I really liked our “Ausländer B” meetings.

I would like to express my thanks to my friends and colleagues of our research group: Roger “The Real Cool Swiss” Wattenhofer, who enriched my social life with party invitations and reports; Zsuzsa “Tango Queen” Liptak, for brushing up my English; Stefan “Smirk” Eidenbenz, with whom it was a pleasure to work; Barbara “Social Administration” Heller; Christoph “You Can't Hurt Me” Stamm, for his help with Windows NT configurations; Konrad “Tractor Driver” Schlude, for his patience in solving wired equations; Peter “Been There, Done That, Got the T-shirt”
Remmele, Mark “Blue-Eyed Adventure-Seeker” Cieliebak and Roland “I Can Explain” Uber for many interesting discussions.

I thank Nora “Artemis” Sleumer, with whom I had a great time organizing activities to support and encourage women doing computer science — and for being a good friend and companion.

Special thanks go to my flat-mates: Karin “The Voice” Meier, I like your friendly, nice, and singing way of being; and Oliver “We HAD to Meet” Bröker, thank you for being one of my most valuable friends. I really appreciate our uncounted discussions about research, didactic, psychology, and everything that is fun.

I also want to thank my sister Birgit and especially my Mom and Dad who are steadily supporting me and believe in me. Thank you!

My unbounded thanks go to Wolfram “My Name is Wolf. I Solve Problems” Schlickenrieder, my great love, who is a friend, proof reader, coach, companion, and playmate to me in all subjects of life.
# Contents

1 Introduction

## 1 Problems in Computational Geometry

2 Singularities Make Spatial Join Scheduling Hard

### 2.1 Introduction

### 2.2 Spatial Join Scheduling without Singularities is Easy

### 2.3 Spatial Join Scheduling is \( \mathcal{NP} \)-Hard

#### 2.3.1 Construction of the Logic Graph

#### 2.3.2 Transformation into a ROG

#### 2.3.3 Embedding of "exclusive-or"s between Literals and Variables

#### 2.3.4 Routing Problem

#### 2.3.5 Combination of all ROG Components

### 2.4 Conclusion

3 From Generalization towards C-Oriented Line Simplification

### 3.1 Generalization

### 3.2 Line Simplification

### 3.3 C-oriented Line Simplification

#### 3.3.1 Preliminaries and Problem Definition

#### 3.3.2 Reachable Regions

#### 3.3.3 Speed-Up Techniques
## CONTENTS

3.3.4 Runtime and Space Calculation .......................... 63  
3.3.5 Relation to the Fréchet Metric .......................... 70  
3.3.6 Conclusion ............................................. 73

4 Labeling Downtown ............................................ 75  
4.1 Introduction .................................................. 75  
4.2 Labeling Downtown ............................................ 77  
4.2.1 Problem Definition ........................................ 78  
4.2.2 General Rules ............................................... 79  
4.2.3 A General Algorithm ...................................... 82  
4.2.4 Complexity Status ......................................... 85  
4.2.5 Solvable Special Cases .................................... 91  
4.2.6 Conclusion and Open Problems ............................ 105

II Software Engineering in Computational Geometry .............. 109

5 Software Engineering in Computational Geometry ............... 111  
5.1 Object Oriented Programming Paradigm ........................ 112  
5.2 Generic Programming Paradigm ................................ 114  
5.2.1 Templates ................................................. 115  
5.2.2 Iterators ................................................... 116  
5.2.3 Function Objects .......................................... 117  
5.2.4 Traits Classes ............................................. 118  
5.2.5 Advantages and Disadvantages of Generic Programming .... 120  
5.3 The Computational Geometry Algorithms Library (CGAL) ...... 120  
5.3.1 Related Work ............................................... 122

6 Range and Segment Trees ......................................... 123  
6.1 Introduction .................................................... 123  
6.2 Definition of a Range Tree .................................... 126  
6.3 Definition of a Segment Tree .................................. 132  
6.4 Design Goals .................................................. 137
## CONTENTS

6.5 Related Work ........................................ 138
6.6 Design Overview ...................................... 139
6.7 Program Example .................................... 143
6.8 Conclusion ........................................... 144

7 R-tree and R*-tree-like Index Structures .......... 147
7.1 Introduction .......................................... 147
7.2 Definition of an R-tree .............................. 148
  7.2.1 R-tree Searching ................................. 148
  7.2.2 R-tree Insertion ................................. 150
  7.2.3 R-tree Split ..................................... 151
  7.2.4 R-tree Deletion ................................. 152
7.3 Definition of an R*-tree ............................ 154
  7.3.1 R*-tree Insertion ............................... 154
  7.3.2 R*-tree Split ................................... 155
7.4 Design Goals ......................................... 159
7.5 Related Work ......................................... 159
  7.5.1 Generalized Search Trees (GiST) for Database Systems 160
7.6 Design Overview ...................................... 162
  7.6.1 Data handling ................................... 162
  7.6.2 R-tree_index ..................................... 163
  7.6.3 R-tree_storage ................................. 163
  7.6.4 More Design Features ........................... 164
7.7 Implementation Details .............................. 165
7.8 Program Example .................................... 166
7.9 Conclusion ........................................... 169

8 Conclusion ............................................. 171
Abstract

In this thesis we analyze and solve geometric problems that arise in applications of geographic information systems. The thesis is divided into two parts. In the first part we study three problems from a theoretical point of view. The second part explores software design issues of spatial data structures that support efficient geometric queries. These implementations are part of the computational geometry algorithms library (Cgal).

Our first investigation deals with join processing in spatial databases. This is a very expensive operation when “memory pages” have to be fetched from disk. We show that finding an optimum spatial join schedule is $NP$-hard in general. Our proof makes intensive use of a particular singularity; this leaves room for efficient and optimal join schedules for special spatial data structures.

The subject of our next studies is the automatic generation of schematic maps such as those used to describe subway maps or bus maps. In these maps, the run of a line is usually simplified and $C$-oriented. That is, the line runs parallel to a restricted set $C$ of orientations in $\mathbb{R}^2$. We focus on the $C$-oriented line simplification problem which is an important subproblem of the automatic generation of a schematic map. For a given polygonal chain $P$, a set of orientations $C$, and a constant quality bound $\epsilon$, we give an algorithm that computes a $C$-oriented polygonal chain which consists of a minimum number of links and has Fréchet distance at most $\epsilon$ to $P$. Our algorithm solves the problem in $O(kn^2 \log n)$ time, where $n$ is the number of vertices in the polygonal chain and $k$ is the minimum number of links.

In the last problem of the first part of this thesis we discuss a map labeling problem. American cities, particularly their central regions, usually have a very regular street pattern. We focus on labeling a rectangular grid of streets, such that each street is labeled with its name and no two labels overlap. This problem was recently shown to be $NP$-hard. We give an algorithm that guarantees to solve every solvable instance. Surprisingly this
algorithm is empirically efficient; this makes the analysis of the algorithm even more interesting. Furthermore, we sketch the \( \mathcal{NP} \)-completeness proof, how about a variant of this problem, namely the labeling of a cylinder shaped downtown, is \( \mathcal{NP} \)-hard, and present polynomial time algorithms for three special cases that are likely to occur.

In the second part of the thesis we describe our contribution to the CGAL project. The CGAL project aims at providing a flexible, efficient, robust, and easy-to-use library of the most important solutions in computational geometry for applications in academia and industry. We designed and implemented spatial data structures that support geometric queries for CGAL. For spatial data in internal memory we implemented the general dimensional range and segment tree. For external memory data structures we implemented a framework for the definition of an R-tree or R-tree variants, such as the R*-tree. These data structures are designed using the generic and object oriented programming paradigms. The design pursued the following goals:

**Flexibility** The data structures are highly modular, that is, each data structure consists of several components which are connected by a minimal interface. Thus, the components are easily exchangeable. E.g., all data structures can be adapted to work on data of applications that already exist.

**Efficiency and Robustness** All data structures have good performance time and meet the robustness issue, since the number type and therefore the precision can be defined by the user.

**Ease of Use** CGAL provides extensive documentation and example programs for each software package. Furthermore, intuitive naming conventions and interfaces to various tools (e.g. visualization tools) make the use of CGAL comfortable.
Kurzfassung


das Problem in Zeit $O(kn^2 \log n)$, wobei $n$ die Anzahl der Liniensegmente von $P$ und $k$ die Anzahl der Liniensegmente von $Q$ beschreibt.


Im zweiten Teil dieser Arbeit beschreiben wir unseren Beitrag zum Cgal-Projekt. Ziel des Projektes ist die Bereitstellung der wichtigsten Algorithmen und Datenstrukturen im Bereich der Computational Geometry für Anwender in Industrie und Forschung. Konkrete Design-Ziele von Cgal sind:

**Flexibilität** Die Algorithmen und Datenstrukturen sind modular aufgebaut; das heisst, sie sind in einzelne Komponenten zerlegt, die mit Hilfe von minimalen Schnittstellen verbunden werden können. Dadurch sind einzelne Komponenten leicht austauschbar, was die Algorithmen und Datenstrukturen wiederverwendbar machen soll. Ausserdem können die Algorithmen und Datenstrukturen für schon bestehende Datenstrukturen adaptiert werden.

**Effizienz und Robustheit** Bei der Auswahl der Algorithmen und Datenstrukturen zu einem Problem wurde stark auf Effizienz und Robustheit geachtet. Um Robustheitsprobleme zu umgehen, wurden die Algorithmen und Datenstrukturen mit dem Koordinatensystem und/oder den Zahlenarten parametrisiert.

**Benutzerfreundlichkeit** Zu jedem Software-Paket gibt es eine ausführliche Dokumentation und Beispielprogramme. Ausserdem wurden Schnittstellen zu verschiedenen Tools implementiert, z.B. zu Visualisierungstools, Zufallsgeneratoren und Zahlentypen.
Chapter 1

Introduction

Computational geometry deals with the design and analysis of algorithms and data structures for geometric problems involving spatial data such as points, lines, polygons, and subdivisions. Over the last two decades, the research field of computational geometry has grown significantly. The increasing interest in this area of research can be partially explained by the variety of application areas in which geometric algorithms and data structures play a fundamental role. One important area of application is that of Geographic Information Systems (GIS) [vKNRW97, vKNR+97]. A GIS is a tool for storage, retrieval, display, and analysis of geographic information. The increasing popularity of the Internet goes hand in hand with an increasing demand for tools that display geographic information. Other important application domains are computer graphics, robotics, and computer aided design (CAD).

Imagine you are a computer scientist working for a big national park. You are responsible for the storage and processing of geographic data with respect to specific applications. Your first job is to produce a map of all hiking trails that are easily accessible by public transport. For this, you are given one map of the region containing all roads, railways, and bus routes, and another map containing all hiking trails. In order to compute all hiking trails that are easily accessible by public transport, you have to overlay these two maps, that is, you have to combine the information in the two maps by locating the position of objects from one map in the other one, compute the intersection of map features, and so on. The operation of combining two inputs based on their spatial relation is called spatial join. Spatial joins are among the basic operations in a GIS. If both maps are
too big to fit in internal memory entirely, which usually is the case, the spatial data of each map is distributed on disk blocks in spatial clusters. These disk blocks are stored in external memory. In order to compute the spatial join of these two maps, you have to compute a scheduling, so that eventually each two overlapping spatial clusters are in main memory at the same time. It is long known that scheduling relational joins with relations residing on disk is \( \mathcal{NP} \)-hard in general. This result motivated a number of heuristics for scheduling spatial joins. Since spatial clustering makes use of the rich structure of Euclidean space, the \( \mathcal{NP} \)-hardness of relational join scheduling does not imply \( \mathcal{NP} \)-hardness for spatial join scheduling. In Chapter 2.1 we show that, indeed, for a class of popular spatial clustering techniques used for spatial data structures, an optimum page fetch schedule can be computed in linear time. In full generality however, we prove spatial join scheduling to be \( \mathcal{NP} \)-hard. Our proof makes extensive use of a particular singularity; this leaves room for the development of further efficient scheduling algorithms for interesting spatial data structures. For the \( \mathcal{NP} \)-hard problem we refer to a polynomial time approximation algorithm with worst case ratio \( \frac{7}{6} \) [PY93].

Let us continue the example from above. Apart from the map that displays all hiking trails and their accessibility by public transport, you are asked to produce a map of the public transport lines. Bus routes, railroad maps, and maps of subway lines are often drawn \( \mathcal{C} \)-oriented. Each line segment in these maps is parallel to an orientation in a fixed set \( \mathcal{C} \) of orientations in \( \mathbb{R}^2 \). Usually, the orientations used are horizontal, vertical, and both diagonals. The map you are to produce is required to be \( \mathcal{C} \)-oriented. Furthermore, the drawing of the individual bus or railroad line is to be simplified: The amount of line detail is to be reduced, but the simplified line should still resemble its original. Line simplification is one of the most important operations in map generalization. In Chapter 3.1, we first give an introduction to map generalization and line simplification. Following that, we study the \( \mathcal{C} \)-oriented line simplification problem: Given a polygonal chain \( P \) represented by an ordered set of vertices \( p_1, \ldots, p_n \) in the plane, a set of orientations \( \mathcal{C} \), and a constant \( \epsilon \), we search for a "\( \mathcal{C} \)-oriented" polygonal chain \( Q \) consisting of the minimum number of line segments that have distance at most \( \epsilon \) to \( P \) in the Fréchet metric. The Fréchet metric is a metric that can be used to measure the similarity between shapes [AG95]. We restrict our attention to the version of the problem where two circles of radius \( \epsilon \) formed around adjacent vertices of the polygonal chain do not intersect. We solve the \( \mathcal{C} \)-oriented line simplification problem constructively by using dynamic programming together with a simple data structure. For common cases of
our algorithm solves the problem in time \( O(kn^2 \log(n)) \) and uses \( O(kn^2) \) space, where \( k \) is the minimum number of line segments of \( Q \).

Let us again continue our example from above. At the border of the national park, there is a camping site with over 10,000 camping spots. This camping site is rectangular and has a Manhattan-like street pattern, that is, there are \( n \) horizontal streets that cross \( m \) vertical streets. For each street we are given a street name. Your next task is to label a map of the camping site, where each street is labeled with its name and no two labels overlap. We call this problem \textit{downtown labeling} problem. It belongs to the class of \textit{map labeling} problems. Map labeling is an important research area which has gained in popularity in the last decade. In Chapter 4.1 we give an introduction to map labeling before concentrating on the downtown labeling problem. We show that a slight generalization, namely the labeling of a cylinder shaped downtown, is \( \mathcal{NP} \)-hard. Recently, Unger and Seibert showed the \( \mathcal{NP} \)-completeness of the downtown labeling problem [US99]. We further give an algorithm that guarantees to solve every solvable instance and tested it on over 10,000 randomly generated instances. Surprisingly, the empirical behavior of the algorithm is polynomial without a single exception. Finally, we present efficient algorithms for three special cases including the case where all labels have length at most half the length of their respective streets.

The problems we described so far were mainly solved theoretically. Indeed, remarkable success has been made in the theory of computational geometry in general [Rep96]. However, only few theoretic results have been examined with respect to their applicability in practice and thus only few geometric codes are in use up to now [Rep96]. The main reason for this deficiency is that implementing geometric algorithms appears to be more difficult than in other fields [Rep96, Sch97]. The computational geometry algorithms library (CGAL) project was founded in 1995 in order to remedy this situation by providing correct, efficient, and reusable implementations of the most important solutions developed in computational geometry [FGKSS98, FGK+98, Ove96]. One important aspect of flexibility is that CGAL algorithms can be easily adapted to work on data types of applications that already exist. This and further design goals are achieved by using the generic programming paradigm using templates in C++ [MS96, Str97, Wei98, BKS99].

The CGAL library is composed of three different parts:

- The \textit{kernel}, which consists of basic geometric objects (e.g. points and segments) and operations on these objects (e.g. intersection of segments).
• The basic library consists of important geometric algorithms and data structures (e.g., convex hull, Delaunay triangulation, planar map and search structures for geometric queries).

• The support library contains interfaces to number types of other sources (e.g., LEDA [MN99], GNU Multiple Precision Library [Gra96]), interfaces to visualization tools, random generators, and other non-geometric tools.

When I started to work as a Ph.D. student at the ETH in 1996, the CGAL project had just started officially. It was a very good opportunity to take part in an important and big project right from the start. In particular, in the starting phase of the project, many decisions concerning the design of the library had to be taken. Being a member of the CGAL project, I was able to integrate and develop my knowledge about software engineering. Since my advisor Peter Widmayer and his group had gained special knowledge about geographic data structures, it was only natural that I combined theoretical knowledge about geographic data structures with practical knowledge about software design. Therefore, I designed and implemented data structures that can be used to perform geometric queries on point data and spatial data of arbitrary dimension.

One important geometric query we were particularly interested in is the range query. In a range query you are given set of spatial objects and a query range, which is usually a rectangular region, and the task is to report all objects that lie within or intersect the query range.

Another important query type is the inverse range query: for a query point $q$, all objects containing $q$ have to be reported. Data structures that allow geometric queries to be processed efficiently are of great importance in the area of computational geometry [Sam90, NW97b].

Representations of geometric objects can be quite complex and large. E.g. the representation of a state by its boundary may require hundreds or more vertices. Since manipulating such big objects can be cumbersome, spatial objects are usually approximated by a simpler shape. The usual technique is to bound each spatial object by the smallest axis-parallel rectangle that encloses the object. This rectangle is referred to as the spatial object's bounding box. In order to manage and query complex spatial objects efficiently, usually a two step approach is employed. In the filter step, we work with the approximation of the actual spatial objects in order to reduce the number of objects to be investigated in detail. For each object that passes the filter step, we proceed with a refinement step where
we retrieve the exact spatial extensions of the objects and check the query predicate in detail. In this work, we are only interested in the filter step.

We chose to implement range and segment trees. Range trees can be used to manage and query general-dimensional point data, while segment trees manage and allow the user to query general dimensional interval data. Note that one-dimensional interval data is simply an interval $[a,b]$, two-dimensional interval data is a two-tuple of one-dimensional intervals, which is a rectangle, and so on. Thus, a segment tree can be used to store complex spatial objects that are represented by their bounding boxes. The range tree efficiently supports range queries in $O(\log^d n + k)$ time, where $n$ is the number of elements, $d$ is the dimension, and $k$ is the number of reported elements. The tree can be built in $O(n \log^{d-1} n)$ time and needs $O(n \log^{d-1} n)$ space. The segment tree needs $O(\log^d n + k)$ time to perform an inverse range query. The segment tree can be built in $O(n \log^d n)$ time and needs $O(n \log^d n)$ space. Furthermore, range and segment trees can be combined. E.g. one can define a range tree whose secondary structure is a segment tree. This feature allows many different trees to be created, which was a special challenge for the software design.

As stated above, a GIS is an important application area of computational geometry. A GIS usually handles an enormous amount of data that cannot be kept in internal memory and therefore has to reside on disk. The $R$-tree and its variant, the $R^*$-tree, are well-studied and empirically efficient data structures. They allow the user to perform geometric queries on spatial data that is stored in external memory. Here the design challenge was not only to implement the $R$-tree and the $R^*$-tree, but to implement this data structure generically as one framework that is general enough to be used in different applications and for experimental studies. The implementation will be part of Cgal.

The most important part in the implementation of these data structures was the software design. The software was designed with respect to the design goals and design conventions of Cgal and with respect to specific requirements the data structure has to fulfill. The generic programming paradigm is the most important programming paradigm of Cgal.

In order to create a basis for the concrete software design, we first survey the most common problems when implementing geometric algorithms and data structures, give an overview of different programming paradigms, and the design goals of Cgal (see Chapter 5). In Chapter 6 we then present the design of the range and segment tree, and in Chapter 7 we present the design of the $R$-tree and its variants.
Part I

Problems in Computational Geometry
Chapter 2

Singularities Make Spatial Join Scheduling Hard

2.1 Introduction

Automated handling of spatial data has gained more and more interest during the last decade. Some application domains of Geographic Information Systems (GIS) are route planning systems, automatic traffic control systems, and flight simulator systems. Other fields where spatial data play an important role are Earth Sciences and Computer Aided Design (CAD). The availability of spatial data has increased rapidly. Satellite images, data of mapping agencies, and simulation output are some sources of data. This data has to be stored, managed, and manipulated according to the respective application. Therefore various database systems have been used to meet these requirements. Usually the amount of data is too huge to be kept in main memory. Therefore the data is distributed on disk blocks in spatial clusters. These disk blocks are also called pages. Spatial database users frequently need to combine two spatial inputs based on some spatial relationship between the objects in the two inputs. For example, map overlay is an important operation in a spatial database [Bur88, MGR91], where two maps are combined to produce a third. Suppose you are given a street map and a forest map of a certain area and you want to produce a map that contains an adequate representation of the streets and the forests. That is, selected objects of both maps have to be combined according to
their positions. This operation of combining two inputs based on their spatial relationship is called spatial join. In databases in general and spatial databases in particular, join processing is one of the most expensive operations. One of the reasons is that for large databases pages may need to be fetched from disk more than once in order to compute a join. Disk access time usually is the dominant part of the join computation time, since disk access is several orders of magnitude slower than accessing main memory. D. Corner\footnote{cited in [Arg96]} gave the following analogy for CPU and disk technology:

\begin{quote}
The difference in speed between modern CPU and disk technologies is analogous to the difference in speed in sharpening a pencil by using a sharpener on one's desk or by taking an airplane to the other side of the world and using a sharpener on someone else's desk.
\end{quote}

In other words: It pays to schedule disk accesses carefully. This is not always easy: For two relations on disk, where each page contains a set of tuples, and an equijoin over some attribute, it is an \(\mathcal{NP}\)-hard problem to find an optimal disk access schedule [MKY81]. In an equijoin all tuples that meet the attribute have to be reported.

This \(\mathcal{NP}\)-hardness result has motivated a flurry of heuristics for scheduling spatial joins in spatial databases over the past decade; for a recent account, see e.g. [DP96]. In spatial databases, as opposed to relational database systems, the data is not spread over disk pages arbitrarily: Virtually all spatial data structures partition the data space geometrically into cells, and they store those geometric objects on a memory page that lie within a cell. Therefore, the \(\mathcal{NP}\)-hardness of the relational join scheduling problem does not imply \(\mathcal{NP}\)-hardness for spatial join scheduling.

In this chapter, we study the complexity of a special case of the spatial join scheduling problem. In our case the spatial join is based on two (different) partitions of a rectangular data space into isothetic rectangles. Each rectangle is a cell that represents a page, where the geometric data stored in the page lies geometrically within the cell. The join operation computes some predicate based on spatial locality. For concreteness, let two sets of points from the same rectangular universe be maintained in the two cell partitions, and let the join predicate be the equality of two points. In order to compute the join, any two pages whose cells overlap need to be in main memory at the same time. Let us now assume that memory capacity is severely limited: Only two pages can be kept in main memory simultaneously. The spatial join scheduling problem asks for the smallest
number of disk accesses and the corresponding disk access schedule (i.e.,
sequence of page pairs) such that any two pages whose cells intersect meet
in main memory at some point in time.

We show that for an important class of cell partitions, the spatial join
scheduling problem is not hard at all: It can be solved in linear time
whenever no two rectangles, one from each partition, share some part of
their boundary. This is likely to be true for data structures that compute
the cell partition according to the data that are stored, such as k-d-trees
in their disk variant (k-d-B-trees) or hB-trees [Sam90]. The scheduling
algorithm is simple enough to be useful in practice. Therefore, in these
cases a heuristic for a spatial join is a loss of both, efficiency and quality of
the solution, as compared to the exact solution.

For the general case, where cell boundaries of both partitions are allowed
to coincide, we show that the scheduling problem is indeed \( \mathcal{NP} \)-hard.
This comes as a late (but first) justification for the search for heuristics
over the past decade. But even data structures that produce partition
lines according to some regular scheme that is fairly independent of the
data, so that cell boundaries may overlap, do exhibit a high degree of
regularity. Therefore also in this case good join scheduling algorithms might
exist. These results were obtained in cooperation with Peter Widmayer. A
conference version is published at [NW97a].

We prove \( \mathcal{NP} \)-hardness by reducing the 3SAT problem to our scheduling
problem. The reduction is based on the proof in [GJ76], but has the
extra complication of requesting that the “gadgets” be embedded into two
rectangular space partitions.

More precisely, let \( R \) be a two-dimensional rectangle that represents
the data space, and let a rectangular partition of \( R \) be a set of isothetic
rectangles that partition \( R \). For two rectangular partitions \( A, B \) of a
rectangle \( R \), a page fetch schedule for \( A \) and \( B \) is a sequence \( \alpha \) of pairs
\( \alpha_i = (a_i, b_i) \), \( a_i \in A \), \( b_i \in B \), \( i = 1, \ldots, n \), where each pair \((a, b)\)
with \( a \in A \), \( b \in B \), \( a \cap b \neq \emptyset \) appears in the sequence. Here and
throughout the paper, the intersection of two rectangles is the closure of
the intersection of their topological interiors. The number of page fetches
in a schedule \( \alpha = (\alpha_i)_{i=1,\ldots,n} \) is the number of changes in consecutive pairs
\( \alpha_i, \alpha_{i+1} \) plus 2; more precisely, it is defined as \(|\{i, 1 \leq i \leq n-1\} \mid \alpha_i \neq \alpha_{i+1} \}| + |\{i, 1 \leq i \leq n-1\} \mid b_i \neq b_{i+1} \}| + 2.

Let \( G = (V, E) \) be an undirected graph. \( V \) is a set of vertices, \( E \) is a
multiset of edges. A path of length \( k \geq 0 \) from vertex \( v \) to vertex \( w \) in \( G \)
is a sequence of vertices \([v = x_0, x_1, \ldots, x_k = w]\) with \( \{x_i, x_{i+1}\} \in E \) for
0 \leq i \leq k - 1. A cycle is a path from a vertex to itself, i.e., a path \([v, w]\) with \(v = w\). A Hamiltonian path of \(G\) is a path of length \(|V| - 1\) in which every vertex of \(V\) appears exactly once. A Hamiltonian cycle is a cycle in which every vertex appears exactly once, except the first vertex which has to be equal to the last vertex.

We will study the following problem:

**Problem 2.1 (Rectangular Join Scheduling)**

**Instance:** Two rectangular partitions \(A, B\) of a rectangle \(R\).

**Problem:** Find a page fetch schedule realizing the minimum number of page fetches.

In order to attack this problem, let \(A \cap B\) denote the join of two rectangular partitions \(A, B\), defined as \(A \cap B = \{a \cap b | a \in A, b \in B, a \cap b \neq \emptyset\}\).

**Proposition 2.2** Let \(A, B\) be two rectangular partitions of \(R\). Then \(A \cap B\) is also a partition of \(R\) into rectangles.

**Proof:** \(A\) and \(B\) consist of rectangles. Thus, every element in \(A \cap B\) is the intersection of a rectangle in \(A\) with a rectangle in \(B\). Since the intersection of two rectangles is a rectangle, it follows that \(A \cap B\) is a partition of \(R\) into rectangles.

We identify \(A \cap B\) with a graph, the rectangular overlay graph (ROG) \(G_{AB}\), defined as follows:

**Definition 2.3 (ROG)** Let \(R\) be a two dimensional rectangle, \(A, B\) two partitions of \(R\). Let \(A \cap B\) be the join of the two rectangular partitions. A ROG \(G_{AB}\) is a graph where each rectangle of \(A \cap B\) is associated with a vertex; there is an edge between two different vertices \((a, b)\) and \((c, d)\) if \(a = c\) or \(b = d\), \(a, c \in A, b, d \in B\).

![Figure 2.1: Rectangular partitions.](image)

Figure 2.1 shows a rectangular partition, the join of two rectangular partitions, and the corresponding ROG. Let \(n\) denote the number of
rectangles in $A \cap B$, i.e. the number of pairs of rectangles from $A$ and $B$ that intersect. Our goal is to order the $n$ pairs in such a way that the number of changes in consecutive pairs is minimum. Since any two consecutive pairs change in at least one element, $n - 1$ is the minimum possible number of changes. Furthermore, in a sequence with $n - 1$ changes, any two consecutive vertices have a common edge. Since a change corresponds to a page fetch, we get:

**Proposition 2.4** There is a page fetch schedule with $n+1$ page fetches if and only if $G_{A \cap B}$ has a Hamiltonian path.

**Proof:** $\Rightarrow$: Let $\alpha$ be a sequence of pairs $(a_i, b_i)$ that solves Problem 2.1 and has $n-1$ consecutive changes. Each pair in the sequence corresponds to a rectangle of $A \cap B$ and each rectangle of $A \cap B$ corresponds to a vertex in $G_{A \cap B}$. Since any two consecutive pairs have one element in common they correspond to two pairs of rectangles $((a_i, b_i) \text{ and } (a_{i+1}, b_{i+1}))$ in $A \cap B$ that fulfill $a_i = a_{i+1}$ or $b_i = b_{i+1}$. With Definition 2.3, the corresponding vertices of $G_{A \cap B}$ are incident. Since every rectangle of $A \cap B$ corresponds to exactly one pair in $\alpha$, it follows that the sequence defines a Hamiltonian path in $G_{A \cap B}$.

$\Leftarrow$: Let $\gamma$ be the sequence of vertices in a Hamiltonian path of a graph $G_{A \cap B}$. Each vertex corresponds to a rectangle of $A \cap B$ and therefore to a pair $(a, b)$ of rectangles with $a \in A$ and $b \in B$. Two vertices $u$ and $v$ of $G_{A \cap B}$ are incident if the corresponding rectangles $(u = a_u \cap b_u$ and $v = a_v \cap b_v$) fulfill $a_u = a_v$ or $b_u = b_v$, $a_u, a_v \in A$, $b_u, b_v \in B$. Therefore, it follows that in the corresponding sequence of pairs of rectangles consecutive pairs differ in exactly one element. Thus, the number of consecutive changes is $n - 1$.

This leaves us with the Hamiltonian path problem for $G_{A \cap B}$:

**Problem 2.5 (ROG Hamiltonian Path)**

**Instance:** A ROG $G_{A \cap B}$.

**Problem:** Does $G_{A \cap B}$ contain a Hamiltonian path?

In the next section, we will argue that the scheduling problem is easy whenever the rectangles of $A$ and $B$ are in general position. Section 2.3 shows that for unrestricted rectangle positions, the problem is hard. An approximate solution for the scheduling problem can be obtained as follows: We assign to each edge of $G_{A \cap B}$ a distance one and make the graph complete by adding edges of distance two. Now it is easy to see that
the scheduling problem corresponds to finding a traveling salesperson tour of minimum length in the complete graph. Papadimitriou and Yannakakis have given a polynomial time approximation algorithm with worst-case ratio $\frac{5}{4}$ for the special case of the traveling salesperson problem for which all distances are either one or two [PY93]. This algorithm can be directly applied to $G_{A \cap B}$, where the approximation ratio is conserved.

2.2 Spatial Join Scheduling without Singularities is Easy

Let $R$ be a rectangle. Let $A, B$ be two rectangular partitions of $R$. We request that the rectangles lie in general position in the sense that no two rectangles $a \in A$ and $b \in B$ share a (part of a) common boundary (apart from the common boundary sides of $R$). More precisely:

**Definition 2.6 ("common boundary")** Let $[x_1, x_2]$ be an interval on the $x$-axis. Let $[y_1, y_2]$ be an interval on the $y$-axis. Then $[x_1, x_2] \times [y_1, y_2]$ defines an axis parallel rectangle containing all points $(x, y)$ with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$.

Let $a = [x^a_1, x^a_2] \times [y^a_1, y^a_2] \in A$ and $b = [x^b_1, x^b_2] \times [y^b_1, y^b_2] \in B$ be two rectangles. Two rectangles $a$ and $b$ share a common boundary if $(x^a_i, x^a_j) \cap (x^b_i, x^b_j) \neq \emptyset$ and $y^a_i = y^b_j$ or $(y^a_i, y^a_j) \cap (y^b_i, y^b_j) \neq \emptyset$ and $x^a_i = x^b_j$ for at least one pair, $i, j \in \{1, 2\}, i \neq j$.

**Definition 2.7 (Dual Graph)** Let $R$ be a partition of a rectangle into rectangles. The dual graph $G_d$ of $R$ is a graph where each rectangle of $R$ is associated with a vertex. Two different vertices $u$ and $v$ are joined by an edge if the corresponding rectangles touch (in more than a single point).

Kranakis [Kra97] pointed out to us that Czyzowicz et al. [CRCS+94] showed the following theorem:

**Theorem 2.8 ([CRCS+94])** If a rectangle $R$ is partitioned into rectangles, then the dual graph of $R$ has a Hamiltonian path.

In order to apply Theorem 2.8 to a ROG $G_{A \cap B}$ we identify the dual graph of $R$ with the ROG $G_{A \cap B}$:

**Lemma 2.9** Let $A, B$ be two rectangular partitions of a rectangle $R$ that lie in general position. Let $A \cap B$ be the join of $A$ and $B$. Let the ROG be
2.3. SPATIAL JOIN SCHEDULING IS \( NP \)-HARD

Let \( G_{A\cap B} = (V, E) \) and the dual graph be \( (A \cap B)_d = (V', E') \). Then \( V = V' \) and \( E \supseteq E' \).

Proof: Clearly, the definition of the vertices in \( G_{A\cap B} \) corresponds to the definition of the vertices in the dual graph \( (A \cap B)_d \). Let \( \{u, v\} \) be an edge in \( (A \cap B)_d \). By the definition of the dual of a rectangular partition it follows that \( u \) and \( v \) correspond to two rectangles \( u = a_u \cap b_u \) and \( v = a_v \cap b_v \) of \( A \cap B \) that share a common boundary side. Since the rectangular partitions \( A \) and \( B \) lie in general position, the corresponding rectangles \( u \) and \( v \) fulfill:

\[ a_u = a_v \text{ or } b_u = b_v, \quad a_u, a_v \in A, \quad b_u, b_v \in B. \]

Thus the corresponding vertices of \( u \) and \( v \) in \( G_{A\cap B} \) are connected by an edge.

It has been shown [CRCS94] that the rectangular dual graphs are internally four-connected, hence they are Hamiltonian, by a theorem of Tutte [Tut56]. Thus we can use the algorithm in [CN89] to find a Hamiltonian path in \( G_{A\cap B} \) in linear time.

2.3 Spatial Join Scheduling is \( NP \)-Hard

We now consider the unrestricted case, i.e., two rectangles of different partitions may have a common boundary.

Problem 2.10 (ROG Hamiltonian Circuit)

Instance: A ROG \( G_{A\cap B} \) for two rectangular partitions \( A, B \) of a universe \( R \).

Problem: Does \( G_{A\cap B} \) contain a Hamiltonian circuit?

In the remainder of this chapter we prove our main result:

Theorem 2.11 The ROG Hamiltonian circuit problem is \( NP \)-complete.

Clearly, the ROG Hamiltonian circuit problem is in \( NP \). The \( NP \)-hardness proof for the ROG Hamiltonian circuit problem is based on the \( NP \)-hardness proof of the planar Hamiltonian circuit problem [GJT76]. Thus, in both proofs the reduction starts at an 3SAT instance.

Problem 2.12 (3SAT)

Instance: A boolean formula in conjunctive normal form with exactly three literals per clause.

Problem: Is there a satisfying truth assignment for the formula?
CHAPTER 2. SINGULARITIES MAKE SPATIAL JOIN SCHEDULING HARD

Garey, Johnson, and Tarjan express the formula by combining logic elements like “exclusive-or” and “three-input-or” to a logic graph. In a second step, they design graph components that have the functionality of the logic elements, and then embed these components into a graph according to the logic graph.

2.3.1 Construction of the Logic Graph

For the sake of making the presentation self-contained, let us recall the construction of the logic graph as defined in [GJT76]. This section closely follows the presentation in [GJT76], and it is even taken literally in some places.

First, we introduce the logic “exclusive-or” graph.

Definition 2.13 (Logic “exclusive-or”): The “exclusive-or” graph is a subgraph that acts like two separate edges, one connecting vertices \( u \) and \( u' \) and the other connecting vertices \( v \) and \( v' \), with the constraint that for each graph \( G \) that contains this graph as a vertex induced subgraph, exactly one of these two edges must occur in any Hamiltonian circuit of \( G \). In this case, we say that the edges \( \{u, u'\} \) and \( \{v, v'\} \) have been “connected” by an “exclusive-or”.

Definition 2.14 (Logic “three-input-or”): The “three-input-or” graph is a subgraph that acts like three separate edges, one connecting vertices \( u \) and \( u' \), one connecting vertices \( v \) and \( v' \), and the other one connecting vertices \( w \) and \( w' \), with the constraint that for each graph \( G \) that contains this graph as a vertex induced subgraph, at least one of these three edges must occur in any Hamiltonian circuit of \( G \). In this case, we say that the edges \( \{u, u'\} \), \( \{v, v'\} \) and \( \{w, w'\} \) have been “connected” by a “three-input-or”.

The logic graph consists of edges as usual and the logic components “exclusive-or” and “three-input-or”, as follows:

Definition 2.15 (Logic Graph): Let an instance of 3SAT be given by a set \( V \) of \( n \) variables and a set \( C \) of \( m \) clauses over \( V \). For each of the variables \( x_i \), \( 1 \leq i \leq n \), we construct four vertices \( v_{i1}, v_{i2}, v_{i3}, v_{i4} \) and for each clause \( C_j \), \( 1 \leq j \leq m \) we construct six vertices \( w_{j1}, w_{j2}, w_{j3}, w_{j4}, w_{j5}, w_{j6} \). These vertices are connected by the following edges:

1. two copies each of \( \{v_{i1}, v_{i2}\} \) and \( \{v_{i3}, v_{i4}\} \), \( 1 \leq i \leq n \);
2. \{v_{i2}, v_{i3}\}, 1 \leq i \leq n;
3. \{v_{i4}, v_{i+1,4}\}, 1 \leq i \leq n - 1;
4. \{v_{i4}, w_{m4}\};
5. \{v_{11}, w_{11}\};
6. two copies of \{w_{j1}, w_{j2}\}, \{w_{j3}, w_{j4}\}, and \{w_{j5}, w_{j6}\}, 1 \leq j \leq m;
7. \{w_{j2}, w_{j3}\}, \{w_{j4}, w_{j5}\}, 1 \leq j \leq m;
8. \{w_{j6}, w_{j+1,1}\}, 1 \leq j \leq m - 1;

For each \( i \), we connect one copy of \( \{v_{i1}, v_{i2}\} \) to one copy of \( \{v_{i3}, v_{i4}\} \) with an “exclusive-or”. For each \( j \), we connect one copy each of \( \{w_{j1}, w_{j2}\}, \{w_{j3}, w_{j4}\}, \) and \( \{w_{j5}, w_{j6}\} \) with a “three-input-or”.

Now let us consider each literal \( p_{jk} \) in \( F \). If \( p_{jk} = x_i \), we use an “exclusive-or” to connect the copy of \( \{w_{j,2k-1}, w_{j,2k}\} \) not connected to a “three-input-or” with the copy of \( \{v_{i1}, v_{i2}\} \) which is not connected to \( \{v_{i3}, v_{i4}\} \) with an “exclusive-or”. If \( p_{jk} = \overline{x_i} \), we use an “exclusive or” to connect that copy of \( \{w_{j,2k-1}, w_{j,2k}\} \) with a copy of \( \{v_{i3}, v_{i4}\} \) which is not connected to \( \{v_{i1}, v_{i2}\} \) with an “exclusive-or”.

See Figure 2.2 for a schematic of this construction for \( F = (x \lor y \lor z) \land (\bar{x} \lor \bar{y} \lor \bar{z}) \land (y \lor \bar{z} \lor w) \). A \( \wedge \) symbolizes an “exclusive-or” relation and a \( \lor \) symbolizes a “three-input-or” relation. The dark shaded part is the logic graph for a variable and its negation, and the light shaded part is the logic graph for a clause. Garey, Johnson, and Tarjan prove:
Theorem 2.16 ([GJT76]) The logic graph constructed for a 3SAT formula $F$ as defined in Definition 2.15 has a Hamiltonian circuit if and only if $F$ is satisfiable.

Thus, the Hamiltonian circuit problem for these logic graphs is $\text{NP}$-complete. This completes our presentation of the logic graph from [GJT76].

The idea of the proof condensed in four sentences is the following: We use the same transformation of a 3SAT formula into a logic graph. In a second step, we transform the logic elements into ROG components which have the same functionality as the logic elements. The ROG components are then combined according to the logic graph. In order to embed crossing logic "exclusive-or" elements into a ROG, we solve a channel routing problem in knock knee mode and then transform the solution into a ROG which has a Hamiltonian circuit if and only if the corresponding 3SAT formula is satisfiable.

2.3.2 Transformation into a ROG

In order to be able to combine the ROG components, we design each component to have a standard length (SL) or a multiple of SL. Furthermore, we unify the interfaces of the components: For the left (right, top, bottom) front of a component we define interfaces LFI (RFI, TFI, BFI).

![Figure 2.3: TFI](image)

Figure 2.3: TFI

Its mirror image on the horizontal axis is a BFI. A 90° (270°) rotation and swapping the dashed with the dotted lines gives a LFI (RFI). A dotted and dashed border line is defined as LFB (RFB, TFB, BFB). All interfaces have standard length (SL) or a multiple of SL and can be "plugged together".

Each logic component consists of logic edges with a relation (e.g. "exclusive-or"). For each logic edge $e$ of a logic component there exist two vertices $u, v$ in the corresponding ROG component such that for any Hamiltonian circuit in the logic graph that contains edge $e$, there is a Hamiltonian circuit in the ROG that enters the component through $u$ and leaves it through $v$. We say $u, v$ act like an edge (build a logic edge). Any two vertices $p, q$ of the ROG component that are connected to other components and do not build a logic edge have the following property: For any ROG, there is no Hamiltonian circuit that enters the component at $p$ and leaves it at $q$. 
Lemma 2.17 ("exclusive-or" ROG) Figure 2.4 shows a ROG component that has the functionality of the logic "exclusive-or" graph.

More precisely, Figure 2.4 shows a horizontal "exclusive-or" ROG component tied to edge \( \{u, u'\} \), and \( \{v, v'\} \). The dark gray shaded part shows the connection of the component with its surroundings. A 90° rotation and swapping the dashed with the dotted lines results in a vertical "exclusive-or". The proof of this theorem as well as the proofs of the following theorems concerning ROG components is based on the following observation: Consider two edges \( (x, y), (y, z) \), where \( y \) has degree two. Thus, any Hamiltonian circuit has to visit these edges in succession. A connected vertex induced subgraph that is connected to other components by exactly two edges is called a chain. Any Hamiltonian circuit through a graph containing such a component as a vertex induced subgraph enters the component and visits all vertices before the component is left. E.g. in Figure 2.4 there are only four vertices \( u, u', v, v' \) where a path can enter or leave the component, and four horizontal chains. The theorem is proven by observing that "any path from \( u \) to \( v' \) (\( v \) to \( u' \), \( u \) to \( v \), \( u' \) to \( v' \)) can only visit one or three chains" and "there is no Hamiltonian circuit that enters and leaves the component twice" combined with showing that there is a Hamiltonian
path that enters and leaves the component through \( u, u' \) (or \( v, v' \)). For the sake of completeness we will spell out the proofs in full.

**Proof:** We show that the graph fulfills the ROG properties and has the functionality of the logic “exclusive-or” as defined in Definition 2.13.

1. The dotted lines divide the area of the component into rectangles; the dashed lines do the same. In the join of the dotted and dashed rectangular partition, each rectangle is identified with a vertex. Two vertices \( u \) and \( v \) are joined by an edge if the corresponding rectangles \( u = a_u \cap b_u \) and \( v = a_v \cap b_v \) from the original partitions satisfy: \( a_u = a_v \) or \( b_u = b_v \), \( a_u, a_v \in A, b_u, b_v \in B \). Furthermore, the left (right, bottom, top) interface corresponds to LFI (RFI, BFI, TFI).

2. First, we observe that the vertices in each of the sets \( \{3, 4, 5\}, \{11, 12, 13\}, \{19, 20, 21\}, \{27, 28, 29\} \) are chains. The component can be visited through four edges: \( \{u, 1\}, \{r, 8\}, \{u', 25\}, \{v', 34\} \). Assume there exists a Hamiltonian path that enters this component through \( u \), visits all vertices of the component, and leaves it through \( v \). Any Hamiltonian path has to visit chain \( \{3, 4, 5\} \). It follows that the path starting with \( \{u, 1\} \) has to visit chain \( \{3, 4, 5\} \) before any
other chain is visited, since the only possibilities to visit this chain are given by path \((1, 2, 3, 4, 5, \ldots)\) and path \((9, 2, 3, 4, 5, \ldots)\), where vertex 1 only is adjacent to 2, 7, and 9. One possibility to leave the chain is \((3, 4, 5, \ldots, 6, 7, 8)\). Since 8 is the only vertex adjacent to \(v\), there is no path visiting the other chains before leaving the component through \(v'\). The other possibility to leave the chain is given by \((1, 2, 3, \ldots, 7, 16)\). Since \(v, 7, \) and 16 are the only vertices that are adjacent to vertex 8, it also follows that there is no path visiting the other chains before leaving the component through \(v'\). Thus, there exists no Hamiltonian path that enters this component through \(u\) and leaves it through \(v\).

Since the number of chains is even and any path from \(u\) to \(v\) visits an odd number of chains, it follows that there is no Hamiltonian path that enters the component through \(u\) and leaves it through \(v'\).

From the symmetry of this component and the above reasoning it follows that there is no Hamiltonian path that uses all four edges \(\{u, 1\}, \{v, 8\}, \{u', 25\}\), and \(\{v', 34\}\).

Figure 2.5 shows a Hamiltonian path that enters the component through \(u\) and leaves it through \(u'\); we call this a local state of the component. A Hamiltonian path that enters the component through \(v\) and leaves it through \(v'\) is symmetric to this path. Any Hamiltonian path through a graph containing this component as a vertex induced subgraph has only these two possibilities to enter and leave the component.

Thus, it follows that the component acts like two separate edges, one connecting \(u\) and \(u'\) and the other connecting \(v\) and \(v'\), with the constraint that exactly one of these two logic edges must occur in any Hamiltonian circuit of \(G\). Thus, the edges \(\{u, u'\}\) and \(\{v, v'\}\) have been “connected” by an “exclusive-or”.

This ROG with the functionality of the logic “exclusive-or” graph can be modified geometrically for various purposes. Figure 2.6 shows a ROG component for a variable and its negation. Without the gray shaded part (the “double-edges”), the configuration corresponds to a “exclusive-or” where the pairs of vertices \(\{u, u'\}\) and \(\{v, v'\}\) are laid out along one straight boundary.

Figure 2.7 shows the layout of the pairs of vertices \(\{u, u'\}\) and \(\{v, v'\}\) around the corner, which is called a “turn”. More precisely, it is a “left-top-turn”
Figure 2.6: Variable and its negation.

Figure 2.7: "turn".
A rotation of this configuration of $180^\circ$ ($90^\circ$ swapping the dashed lines with the dotted lines, $270^\circ$ swapping) is a "right-bottom-turn", ("left-bottom-turn", "right-top-turn").

### Figure 2.8: "double-edge"

A logic "double-edge" consists of two different vertices that are combined by two edges. Any Hamiltonian path visits a vertex exactly once and from the construction of the logical graph it follows that exactly one edge of each "double-edge" occurs in a Hamiltonian path.

**Lemma 2.18 ("double-edge" ROG)** In combination with an "exclusive-or" and "three-input-or" ROG component the ROG for a "double-edge" as shown in Figure 2.8 acts like the logic "double-edge".

We say the upper (lower) logic edge of the logic "double-edge" occurs in a path, if in the "double-edge" ROG edges $\{b,c\}$ and $\{f,g\}$ ($\{i,j\}$ and $\{m,n\}$) occur in a path.

**Proof:** First, it is easy to see, that the dotted (resp. dashed) lines form a rectangular partition. Furthermore, the component has bottom (top) interface BFI (TFI). Since the graph fulfills the properties of Definition 2.3 it follows that this component is a ROG.

A "double-edge" exists in the logic graph only in combination of two "exclusive-ors" or one "exclusive-or" and one "three-input-or". In each local state of an "exclusive-or" ROG component (and "three-input-or" ROG component, as we will see later) in a Hamiltonian path either both edges $\{b,c\}$ and $\{f,g\}$ ($\{i,j\}$ and $\{m,n\}$ resp.) occur or none of them. Since $u$ has degree two, edge $\{u,a\}$ has to appear in any Hamiltonian circuit, thus exactly one of the edges $\{b,c\}$ and $\{m,n\}$ has to appear in any Hamiltonian circuit. The "double-edge" configuration is symmetric, such that this condition also holds for the other direction and other side.

In the logical graph, the configurations for the variables are simply combined with edges. This functionality is provided by placing the ROG components of Figure 2.6 next to each other.

**Lemma 2.19 ("three-input-or" ROG)** Figure 2.9 shows the "three-input-or" component tied to the (logic) "double-edge" $\{u,u'\}$, $\{v,v'\}$, and
\{w, w'\}. Without the light gray shaded part, the component corresponds to the pure "three-input-or" ROG component.

The entire figure is a ROG configuration for a clause.

---

**Proof**: First, we observe that the component fulfills the ROG properties and has top interface TFI three times in succession. We subdivide the figure into three components (dark shaded in Figure 2.9). Each component is linked to other components by four edges. E.g., the leftmost dark shaded component is combined to the edges \(a, c, g,\) and \(h\). Observe that any Hamiltonian circuit in a graph \(G\) which contains this graph as a vertex induced subgraph must visit the edges \(a\) through \(f\). Note that any dark shaded component has four edges connecting it to the outside, at least two of them have to be taken, and every Hamiltonian circuit visits a component an even number of times. Thus, the following condition holds: If edge \(g\) is visited in a Hamiltonian circuit, then edge \(h\) is also visited. Similarly, if edge \(i\) is visited, then also edge \(j\) is visited; if edge \(k\) is visited then also edge \(l\) is visited. Furthermore, any Hamiltonian circuit has to visit at least one of these pairs of edges, since these edges are the only edges that enter the component. Figure 2.10 shows a possible local state with the lower edge of "double-edge" \{\(u, u'\)\} and \{\(v, v'\)\} taken. Thus, this subgraph acts like three "double-edges", one connecting \(u\) and \(u'\), one connecting \(v\) and \(v'\) and the other connecting \(w\) and \(w'\), with the constraint that at least one of the three lower edges must occur in any Hamiltonian circuit of \(G\).

Since the ROG for the logical graph of a clause shown in Figure 2.2 consists of a "three-input-or" combined with three "double-edges", the
2.3. SPATIAL JOIN SCHEDULING IS \( \mathcal{NP} \)-HARD

entire Figure 2.9 builds a ROG for a clause. In the logical graph, the configurations for the clauses are combined with single edges. This functionality is provided by placing these components next to each other.

2.3.3 Embedding of "exclusive-or"s between Literals and Variables

We now have constructed the ROG components for the variables and the clauses. To conclude the construction we have to embed the "exclusive-or" connections between the literals of the clauses and the variables. Each literal of each clause is connected to exactly one variable. But a variable can be connected to more than one clause. We call this configuration a "multiple-exclusive-or" graph.

We combine a configuration of a variable \( x \) and its negation \( \bar{x} \) to \( k + l \) "exclusive-ors" that combine the variable with the corresponding literals, where \( k \) (l) is the number of occurrences of variable \( x \) (\( \bar{x} \)). We divide each ROG component for a variable and its negation (see Figure 2.6) at its vertical central line. Then, we stretch each part such that it has width \( k \) (l) times SL. The gray shaded part of Figure 2.11 shows the "multiple-dou-

Figure 2.11: Variable with two adjacent "exclusive-or"s.

ble-edge" ROG which acts like a "double-edge" and builds the connection between the stretched configuration of a variable and its negation and \( 2 + 1 \) "exclusive-ors".
CHAPTER 2. SINGULARITIES MAKE SPATIAL JOIN
SCHEDULING HARD

32

Figure 2.12: "multiple-double-edge", local state with upper edge taken and lower edges taken.

Lemma 2.20 ("multiple-double-edge") The ROG shown in Figure 2.12 acts like a "double-edge" where the gray shaded part can be copied \((k - 2)\) times and juxtaposed horizontally. The ROG has to be combined with \(k\) "exclusive-or"s on its lower interface and one stretched "exclusive-or" on its upper interface.

Proof: For the logic lower (upper) edge it is easy to see that the vertices \(a, b, c,\) and \(d (f, g, h,\) and \(i\) can be visited independently from the fact whether the lower (upper) edge is taken or not. Since \(v\) and \(v'\) have degree two and there is no path from \(v\) to \(v'\) visiting both edges the component acts like a "double-edge".

Observe that the horizontal length of the configuration for all variables is equal to the horizontal length of the configuration for all clauses (which is \(3m\) times \(S_{k}\), where \(m\) is the number of clauses).

In our embedding of the "exclusive-or" lines we have to handle "crossing-exclusive-or" lines. The property which permits this is that "exclusive-or" lines can be connected in series, to cross over an edge of \(G\), when that edge is required to occur in any Hamiltonian circuit.

Lemma 2.21 ("crossing-exclusive-or" ROG) Figure 2.13 shows the ROG component of two "crossing-exclusive-or" lines in connection with the edges \(\{u, u'\}, \{v, v'\}, \{p, p'\},\) and \(\{q, q'\}\). This component acts like four separate edges with the property that in any Hamiltonian circuit in a graph \(G\) which contains this graph as a vertex induced subgraph, either \(\{u, u'\}\) or \(\{v, v'\}\) are connected by an edge and either \(\{p, p'\}\) or \(\{q, q'\}\) are connected by an edge.
Figure 2.13: "crossing-exclusive-or".
CHAPTER 2. SINGULARITIES MAKE SPATIAL JOIN

SCHEDULING HARD

Proof: First, observe that the ROG in Figure 2.13 (without the gray shaded parts) fulfills the ROG properties.

Figure 2.14 shows the logic components the configuration is composed of: the bold dashed lines show a possible local state. Between \( \{p, p'\} \) and \( \{q, q'\} \) we have four chains, as in the "exclusive-or" configuration, except that these chains are discontinued in the middle by "double-edge" configurations. These "double-edges" are combined with "exclusive-or" configurations. Similarly to all other "double-edges" in this construction, exactly one edge of these "double-edges" has to be in any Hamiltonian circuit. Thus, this graph acts like an "exclusive-or" for edge \( \{p, p'\} \) and \( \{q, q'\} \). Which one of these "double-edges" is in the Hamiltonian circuit is determined by the edge \( \{u, u'\} \). Edge \( \{u, u'\} \) is connected to the rest of the component by an "exclusive-or". Thus, if \( \{u, u'\} \) is in a Hamiltonian circuit, the left edge of a "double-edge" cannot be in the Hamiltonian circuit and all right edges have to. Therefore, it follows that edge \( \{v, v'\} \) is not in the Hamiltonian circuit. Thus, it follows that edge \( \{u, u'\} \) and \( \{v, v'\} \) are joined by an "exclusive-or", independently from the edges \( \{p, p'\} \) and \( \{q, q'\} \). \qed

Another important ROG configuration for the embedding of the "exclusive-or" lines connecting literals with variables is the one shown in Figure 2.15. This component is called "knock-knee", reminiscent of a certain wiring mode in VLSI design. The ROG component is tied to the edges \( \{u, u'\} \), \( \{v, v'\} \), \( \{p, p'\} \), and \( \{q, q'\} \). Without the dark shaded part the component is a top-left-knock-knee; its mirror image with respect to the horizontal axis is a top-right-knock-knee. An inspection of the figure reveals that this configuration consists of two independent "exclusive-or" configurations \( \{u, u'\} \) "exclusive-or" \( \{v, v'\} \), \( \{p, p'\} \) "exclusive-or" \( \{q, q'\} \) providing interfaces LFI, RFI, TF1, and BF1.
2.3.4 Routing Problem

We now transform the problem of embedding the "exclusive-or" lines connecting literals with variables into a channel routing problem in knock-knee mode.

Let $p_{ij}$ be the $i$-th literal in the $j$-th clause, $1 \leq i \leq 3$, $1 \leq j \leq m$. Let $x_1, \ldots, x_i$ be the variables; let $k_r$ be the number of occurrences of literal $x_r$, and let $k_t$ be the number of occurrences of literal $\bar{x}_r$. In VLSI design terminology, we create a channel with $3m$ bottom terminals $p_{ij}$ in the order of their appearance in the clauses. Then, we create $3m$ top terminals $t_{1,1}, \ldots, t_{1,k_1}, t_{1,1}, \ldots, t_{1,k_1}, \ldots, t_{n,1}, \ldots, t_{n,k_n}, t_{n,1}, \ldots, t_{n,k_n}$, such that each literal occurs exactly as many times as needed in the clauses. Then, the following top to bottom nets are created: Terminal $p_{ij}$ builds a top to bottom net with $x_{p,q}$ ($\bar{x}_{p,q}$), if $p_{ij}$ is the $q$-th occurrence of $x_p$ ($\bar{x}_p$) in the ordered set of clauses.

Now, the problem is to find a routing that connects the terminals of each net, where knock-knees are allowed. This can be done in $O(m)$ time with algorithm CHANNELROUTING due to [MPSS86].
CHAPTER 2. SINGULARITIES MAKE SPATIAL JOIN
SCHEDULING HARD

For a net consisting of two terminals $t_i$ and $t_j$, we say $t_i$ is the starting terminal, if the column of $t_i$ lies left of the column of $t_j$. In this case $t_j$ is the terminating terminal. The width of the channel is $O(m)$ and the height is bounded by the density, which in turn is $O(m)$.

Algorithm 1 [MPS86] ChannelRouting (columns, nets)

01) forall columns from left to right do
02)   if there are only starting terminals in the current column then
03)     if both starting terminals belong to one net then
04)        combine the terminals by a straight line;
05)     else
06)        if there are two tracks occupied by one net then
07)           close the net and use its tracks for the starting nets;
08)        else
09)           use a free track for every starting net;
10)    else
11)      close a terminating net and use its track for the other net
         (if any) which has a terminal in the current column;

Figure 2.16: Solved routing problem.

Figure 2.16 shows the solved routing problem according to formula $F = (x \lor y \lor z) \land (\overline{x} \lor \overline{y} \lor w) \land (y \lor \overline{z} \lor \overline{w})$. 
2.3.5 Combination of all ROG Components

From a solution of the knock-knee routing problem we get a ROG layout as follows: We draw a dotted square around each internal grid point, such that the square is axis parallel, the corners are equidistant from the grid points, and the side length corresponds to the minimum distance of two grid-points. Figure 2.17 shows all possible dotted squares that can occur in the routing.

![Figure 2.17: Routing configurations](image)

In order to transform the solution into a ROG, the configurations for the variables are placed above the configuration of the clauses, with distance SL times the number of needed tracks in the solution of the routing problem. Thus, the space between the clause and variable ROG configurations can be subdivided into squares of size SI by SL. These squares are then filled with the corresponding ROGs.

Observe that an empty dotted square of a column only occurs in connection with a horizontal "exclusive-or", a "turn", or another empty dotted square. In this empty dotted square we fill in ROG component "empty" which is shown in Figure 2.18, stretch it to fill all adjacent empty dotted squares of the column, and combine it with the adjacent "exclusive-or" or "turn".

In algorithm Transformation, the construction of the ROG is described more formally.

Up to now we left open how to embed the leftmost logic edge \(v_{1,1}, w_{1,1}\) and the rightmost logic edge \(v_{n,1}, w_{n,1}\) (see Figure 2.2). Observe from the logic graph that both edges must be used in any Hamiltonian circuit. Thus, we can embed the leftmost edge with the configuration in Figure 2.19, with the S part stretched according to the size for the embedding of the "exclusive-or" lines. Similarly, the rightmost edge can be embedded (see Figure 2.20). This edge can be used to fill the empty space behind the variables and the clauses according to the number of additionally used channels for the routing (stretch S1 and S2). The graph thus constructed fulfills the ROG properties and has a Hamiltonian circuit if and only if the initial 3SAT formula has a truth assignment.
CHAPTER 2. SINGULARITIES MAKE SPATIAL JOIN

SCHEDULING HARD

Figure 2.18: "empty" (shaded).

Figure 2.19: "left-most edge"
Algorithm 2  Transformation \((F,G_{A\lor B})\)

**Input:** An instance \(F\) of 3SAT.

**Output:** A ROG \(G_{A\lor B}\) which has a Hamiltonian circuit if and only if \(F\) is satisfiable.

01) Solve the Routing Problem described in Section 2.3.4 for \(F\).

02) Arrange the ROG component of the logical graph for a clause in times in consecuption (see Figure 2.9).

03) For the \(i\)-th variable and its negation we take ROG component “exclusive-or” with logic edges in a horizontal line (see Figure 2.4). Let \(k\) \((l)\) be the number of occurrences of literal \(x^i\) \((\bar{x})\). We decide this component at its vertical central line. Then, we stretch the left (right) part in the horizontal direction, such that it has width \(k\) \((l)\) times \(SL\) (see Figure 2.11). We then combine the left (right) part with a “double-edge” (see Figure 2.8) in case \(k = 1\) \((l = 1)\) and a “multiple-double-edge” (see Figure 2.12), otherwise. After that, we put it next the ROG component of the \(i - 1\)-th variable and its negation.

04) Arrange the row of ROG components for the variables above the row of ROG components for the clauses, with distance \(SL\) times the number of used tracks in step 1.

05) We divide the space between the row of the ROG components for the variables and the row of the ROG components for the clauses into squares \(SL \times SL\), which correspond to the dotted squares from step (1). for each dotted square do 

07) Insert the corresponding ROG component at the corresponding place in the ROG 

08) case edge: “exclusive-or” (see Figure 2.4).

09) case cross: “crossing-exclusive-or” (see Figure 2.13).

10) case knock-knee: “knock-knee” (see Figure 2.15).

11) case turn: “turn” (see Figure 2.7).

12) case empty: “empty”, connect it to a “turn” or “horizontal-edge” ROG as described below (see Figure 2.18).

13) }}

14) Insert ROG component left edge at the left side of the ROG (see Figure 2.19). For this, stretch side \(S\) to the size of the number of used tracks times \(SL\).

15) Insert ROG component right edge at the right side of the ROG (see Figure 2.20). For this, stretch side \(S1\) to the size of the number of additionally used channels times \(SL\) and stretch side \(S2\) to the size of the number of used tracks times \(SL\).
CHAPTER 2. SINGULARITIES MAKE SPATIAL JOIN
SCHEDULING HARD

Figure 2.20: “rightmost edge”

Figure 2.21: Complete ROG.
2.3. SPATIAL JOIN SCHEDULING IS \( \mathcal{NP} \)-HARD

Figure 2.21 shows the complete ROG which has a Hamiltonian circuit if and only if the corresponding 3SAT formula \( F = (x \lor y \lor z) \land (x \lor y \lor w) \land (y \lor z \lor w) \) is satisfiable.

**Lemma 2.22 (Polynomial Reduction)** The reduction from 3SAT to ROG Hamiltonian Circuit takes polynomial time.

**Proof:** The ROG component for the clauses can be constructed in linear time, since the ROG component for each clause is equal to Figure 2.9 which has fixed size.

The ROG component for the variables can also be constructed in linear time, since each component is either equal to Figure 2.6 or equal to a stretched version of that figure, and the stretching factor is bounded by the number of clauses times SL.

The routing problem is solved in linear time, using at most \( O(n^2) \) space, where \( n \) is the number of clauses. Thus, the transformation from the routing problem to the construction of the ROG for the whole embedding of the "exclusive-or" lines, takes at most quadratic time and needs \( O(n^3) \) space, since the ROG components for the routing configurations (see Figure 2.17) have fixed size.

The ROG configurations of edge \( \{v_{13}, w_{14}\} \) and edge \( \{v_{14}, v_{11}\} \) have a fixed size and can be embedded in \( O(1) \) time and space. Thus, it follows that the reduction is polynomial.

**Theorem 2.23** The ROG Hamiltonian Path Problem 2.3 is \( \mathcal{NP} \)-complete.

**Proof:** Since the leftmost logic edge \( \{v_{13}, v_{14}\} \) has to be in any Hamiltonian circuit, by deletion of this edge and closing the left side with a LFB, the \( \mathcal{NP} \)-completeness proof of the Hamiltonian path problem carries over. \( \square \)

As a consequence, we get:

**Theorem 2.24 (Rectangular Join Scheduling)** Rectangular join scheduling is \( \mathcal{NP} \)-complete.

**Proof:** An algorithm solving the rectangular join scheduling problem also solves the ROG Hamiltonian path problem. \( \square \)
2.4 Conclusion

In case that the rectangles of the rectangular partitions lie in general position the rectangular join scheduling problem is linear time solvable. For the other case, where the rectangles of the rectangular partitions may share a (part of a) common boundary, we proved this problem to be \( \mathcal{NP} \)-hard.

The condition that only two rectangles (pages) fit in internal memory is quite restrictive. The complexity of the general problem, where \( k \) rectangles fit in internal memory is also interesting. We conjecture that this problem is \( \mathcal{NP} \)-hard for \( k \ll n \). It is easy to see that the problem is easy for \( k = n \). Thus it is an open problem to find a \( k \) such that the problem is \( \mathcal{NP} \)-hard for \( k \) and polynomial time solvable for \( k+1 \). We studied the problem for \( k = 3 \) and showed with a very similar reduction its \( \mathcal{NP} \)-hardness. This reduction implies a reduction for \( k \ll n \). Unfortunately the number of rectangles in the rectangular partitions grows exponentially with \( k \) and the construction of the rectangular partitions gets more and more complicated [Ney97]. We therefore did not follow this direction.
Chapter 3

From Generalization towards C-Oriented Line Simplification

3.1 Generalization

Techniques for automated \textit{generalization} of spatial data are of premier importance in the application area of Geographic Information Systems (GIS). Map generalization is necessary for the creation and maintenance of spatial databases of multiple scales. A good survey on generalization is given in [Wei97], other good sources are [BM91, MLW95]. The maps are commonly produced at a series of scales, such as $1:25,000$, $1:50,000$, $1:100,000$, $\ldots$, $1:1,000,000$. Usually the map scale is halved at each step of generalization. This has the consequence that the space on the map is divided by 4 and small objects might approach the limits of visual perceptibility. Therefore, essential objects have to be enlarged to remain still visible, while the total amount of data has to be reduced. Furthermore, logical and unambiguous relations between the map objects have to be maintained in order to preserve aesthetic quality. Generalization consists of the selection and simplified representation of detail appropriate to the scale and the purpose of the map. The kind of objects that are essential in a specific map depends on the purpose of the map. E.g. clarity of hiking trails is very essential in a hiking map, but unimportant in a road map.
There exist two types of generalization: context dependent generalization and context independent generalization. The context dependent generalization involves operators such as aggregation or displacement which are controlled involving the spatial context. Polygonal subdivision simplification is a context dependent generalization task. E.g. state borders and land borders usually are represented by polygons, which together build a polygonal subdivision. The generation of a simplified representation of the polygonal subdivision is called polygonal subdivision simplification. Cartographic line characterization and segmentation are other examples of context dependent generalization. Here the objective is to maintain characteristic forms of objects such that they can be recognized.

In the context independent generalization, generalization methods are applied to individual objects independently of their spatial context. Basic algorithms in this field of generalization are selection, elimination, simplification, and smoothing of spatial objects. For the selection and elimination of objects the number of objects and the kind of object that are selected have to be defined. Simplification and smoothing algorithms are applied to complex shapes in order to generate a much less complex shape that resembles the original one. In the next section we discuss line simplification which is the most studied simplification task.

3.2 Line Simplification

Line simplification is often regarded as the most important generalization operation. The majority of map features is directly represented by lines (e.g. roads or streams) or form polygons which are bounded by lines (e.g. forest stand or state borders). The objective of line simplification is to reduce the amount of line detail. Many line simplification algorithms have been developed over the last three decades [Wei97]. Usually, the simplification algorithms starts with a polygonal chain $P$ represented by an ordered set of vertices $(p_1, \ldots, p_n)$, where adjacent vertices $p_i$ and $p_{i+1}$ are connected by a (closed) line segment $p_i p_{i+1}$, also called link. $P$ is then turned into a simplified polygonal chain $Q$ which consists of a smaller set of vertices and represents $P$ well. The classical criteria for line simplification are:

- Minimization of the distance between $P$ and $Q$, and
- Minimization of the number of vertices of the simplification.
In order to mathematically quantify how much two given polygonal chains resemble each other, we first have to define a distance measure that reflects our intuitive notion of “resemblance”. One possible metric that is often used to measure the similarity of shapes is the so-called Hausdorff-metric $\delta_H$ [AG96]: For arbitrary bounded sets $A, B \subseteq \mathbb{R}^2$, the Hausdorff distance is the maximum distance of all minimum distances between an element of $A$ and each element of $B$ and vice versa:

$$\delta_H(A, B) = \max\{\sup_{a \in A} \inf_{b \in B} d(a, b), \sup_{b \in B} \inf_{a \in A} d(a, b)\}$$

where $d$ is the underlying metric in the plane. For the line simplification problem, this is usually the Euclidean metric.

The Hausdorff metric is not an appropriate metric for the line simplification problem. This follows from Figure 3.1. The two curves have small Hausdorff distance but do not resemble each other. The reason for this is that the Hausdorff metric does not reflect the course of the curves. This discrepancy can be overcome in using the Fréchet metric which was first defined by Fréchet [AG95].

Intuitively, two curves $\alpha$ and $\beta$ have Fréchet distance at most $\varepsilon$, if a person walking along $\alpha$ can walk a dog along $\beta$ with a leash of length $\varepsilon$. More precisely, two curves have distance at most $\varepsilon$ in the Fréchet metric if and only if they have monotone parameterizations $\alpha$ and $\beta (\alpha, \beta : [0, 1] \to \mathbb{R}^2)$, such that $\|\alpha(t) - \beta(t)\|_2 \leq \varepsilon$ for all $t \in [0, 1]$.

In the last few years, the Fréchet metric gained more and more attention in research areas about geometric techniques that have been used to measure similarity or distance between shapes. For a survey on these techniques see [AG95, AG96].

Probably the most popular line simplification algorithm is the Douglas Peucker Algorithm [DP73] and its improvement [HS92] which has a worst case time complexity of $O(n \log n)$. It starts by connecting the two endpoints $p_1$ and $p_n$ of the original polygonal chain $P = (p_1, \ldots, p_n)$ with a straight line. If the perpendicular distances of all intermediate vertices are within a tolerance $\varepsilon$ from the base line, these vertices are eliminated and the original polygonal chain is represented by this line.
CHAPTER 3. FROM GENERALIZATION TOWARDS C-ORIENTED LINE SIMPLIFICATION

Figure 3.2: The Douglas-Peucker Algorithm. a) Initial line with furthest vertex v4. b) The line is split, the furthest vertex is v8.

Figure 3.3: The Douglas-Peucker Algorithm continued. c) The second split yields the simplified line.

Guibas et al. have studied the line simplification problem with respect to the Fréchet metric. We are given a polygonal chain $P = (p_1, \ldots, p_n)$ and a constant quality bound $\epsilon$. The problem is to find a polygonal chain $Q$ that consists of a minimum number of links and has Fréchet distance at most $\epsilon$ to $P$. Let $C_j$ be the circle with radius $\epsilon$ and center $p_j$. Guibas et al. gave a linear time greedy algorithm for the case that two consecutive $\epsilon$-circles $C_j$ and $C_{j+1}$ are disjoint. In case that two consecutive $\epsilon$-circles may intersect they gave a $O(n^2 \log^2 n)$ time algorithm that uses dynamic programming.

Both algorithms are based on an approach called ordered stabbing, which was first studied by Egyed and Wenger [EW91]. In the ordered stabbing problem we are given a list of disjoint convex objects $C_1, \ldots, C_n$ and want to find a polygonal chain that consists of a minimum number of links and visits (stabs) the objects in order. The authors define a region $R_j$ to consist of all points in $\mathbb{R}^2$ that can be reached with one line that visits $C_1, \ldots, C_j$ in order. They showed that this region is limited by so-called limiting lines $t$ and $t'$. Figure 3.4 shows the limiting lines of $R_j$. Let the limiting lines of $R_j$ be computed. In order to determine the limiting lines of $R_{j+1}$, $t$ has to be turned clockwise until it coincides with $C_{j+1}$ and such that $t$ intersects...
3.2. LINE SIMPLIFICATION

C₁, ..., Cₐ₊₁. Analogously, $t'$ has to be turned counterclockwise until it coincides with $C_{a+1}$ and such that $t'$ intersects $C₁, ..., C_a$ (see Figure 3.5).

In order to compute a stabbing polygonal chain instead of a single stabbing line the definition of $R_j$ is adapted. We call the convex hull of the two adjacent circles $C_j$ and $C_{a+1}$ tube $T_j$. Then, $R_j$ is the set of all points in $T_j$ that can be reached with a minimum number of links, such that the polygonal chain visits $C₁, ..., C_a$ in order, and "stays within $\epsilon$ to $P'"$. The term "stays within $\epsilon$ to $P'" is not precise enough. To be precise, we have to say that a line segment of the simplification enters, resp. leaves, a tube $T_j$ only through the circles $C_j$ or $C_{a+1}$. Note, a so computed polygonal chain to a point in $R_j \cap C_j$ has Fréchet distance at most $\epsilon$ to $\langle p₁, ..., p₉ \rangle$.

Figure 3.5: Updating the limiting lines.

$R_{a+1} = \emptyset$. Depending on the position of $C_{a+1}$ with respect to $C_j$ and $C_{a-1}$ a new starting region is determined and the approach is repeated until $C_a$ is reached. The determination of the starting region depends on the condition $C_j \cap C_{a+1} = \emptyset$. The simplified line is then determined from the sets $R_j, R_{a+1}$.

In case that the neighboring $\epsilon$-circles $C_j$ and $C_{a+1}$ may intersect, Guibas et al. solve the line simplification problem with dynamic programming. For this algorithm the sets $R_j$ have to be modified again. Let $R'_j$ be the set of points in $T_j$ that can be reached with $i$ links, such that the polygonal chain visits $C₁, ..., C_a$ in this order and "stays within $\epsilon$ to $P'". Their algorithm starts with a computation of the sets $R'_1, ..., R'_{a+1}$ with the line stabbing method. These sets are the starting points for the computation of the sets $R_1, ..., R_{a+1}$, which again are computed with the line stabbing method. This procedure is repeated until $R'_{a+1} \neq \emptyset$ with minimal $k$. The simplified line is then determined from these sets.

In the next section we concentrate on line simplification with additional restrictions to the simplified line: All line segments have to be parallel to an orientation in a set of orientations $C$.
3.3 C-oriented Line Simplification

Maps such as those used to describe subway routes, bus plans, or cartographic schemas for gas, water or electricity mains are often drawn C-oriented. Each line segment in these plans is parallel to an orientation in a fixed set of orientations (vectors) $\mathcal{C}$ in $\mathbb{R}^2$. We assume that for each $c \in \mathcal{C}$ the orthogonal orientation to $c$, denoted as $\bar{c}$, is element of $\mathcal{C}$. This restriction is trivially fulfilled for an even number of uniformly distributed orientations. Requiring the maps to be $\mathcal{C}$-oriented helps to make the maps look graphically clearer, more structured, and hence easier to read. Consider, for example, the creation of a subway line map given a city map. Typically, the number of $\mathcal{C}$-orientations used would be 4: horizontal, vertical, and both diagonals. The subway line normally maintains geographical informations like “is in the north of”, “crosses”, etc. We restrict attention to one central task in the automatic generation of $\mathcal{C}$-oriented maps: the C-oriented line simplification problem.

Let $P = (p_1, \ldots, p_n)$ be a polygonal chain. In the C-oriented line simplification problem we want to compute a C-oriented polygonal chain (i.e. each of the links is C-oriented) with a minimum number of links that represents $P$ “well”. Representing $P$ well means that the approximation leads along $P$ and remains within $\epsilon$-distance of $P$, where $\epsilon$ is a constant quality bound.

We would like to make the following annotations in respect to the the applicability of the Douglas and Peucker algorithm for the C-oriented line simplification problem. The simplification in the Douglas and Peucker algorithm uses only original data points. Since the line formed from two
3.3. C-ORIENTED LINE SIMPLIFICATION

data points usually is not C-oriented, a variant of the Douglas and Peucker algorithm for the C-oriented line simplification problem would have to use non original data points. A simple variant in which each line formed by the Douglas Peucker algorithm is represented by a C-oriented polygonal chain that consists of two links, might need at least twice as many links as the line simplification algorithm that we present in the rest of this chapter.

Adegeest et al. give a \(O(c^2 n \log^3 n)\) time algorithm for computing minimum C-oriented link paths between a pair of points in the plane that avoids a set of \(n\) obstacles [AOS94]. In their definition of C-orientation \(c\) is a number greater or equal to two defining C unit vectors where each pair of clockwise adjacent vectors has the angle \(\frac{\pi}{c}\). For a given starting point \(A\), they iteratively determine the sets of all points that can be reached by paths of one link, two links, etc. This is the first approach solving the C-oriented line simplification. We use ideas from Guibas et al. [GHMS83] and Adegeest et al. [AOS94], but we note that both methods can not be applied without major adjustments. A conference version of this chapter has been published in [NeY99].

3.3.1 Preliminaries and Problem Definition

In this section we first introduce some necessary notations and definitions. Then, we formalize the C-oriented line simplification problem.

For two points \(p\) and \(q\) we denote \(\overrightarrow{pq}\) as the vector \((q - p)\). We distinguish between points and vertices of a polygonal chain: we call any \(q \in \overrightarrow{p_j p_{j+1}}\) a point of \(P\) and each \(p_j\) a vertex, \(1 \leq j \leq n\). Let \(p\) and \(q\) be two points on a simple polygonal chain \(P\). If \(p\) precedes \(q\) on \(P\) we write \(p < q\).

Let \(P = (p_1, \ldots, p_n)\) be a polygonal chain. Call a circle with radius \(c\) and center \(p_j\) the \(\epsilon\)-circle \(C_j\) of \(p_j\). The convex hull of two \(\epsilon\)-circles \(C_j\) and \(C_{j+1}\) from two consecutive vertices \(p_j\) and \(p_{j+1}\) is called the tube \(T_j\). Figure 3.8 shows two tubes \(T_j, T_{j+1}\), and a point \(q\) lying in both tubes but not in the \(\epsilon\)-circle they share. We need to distinguish between \(q \in T_j\) and \(q \in T_{j+1}\). Therefore, to a tube \(T_j\) including the link \(\overrightarrow{p_j p_{j+1}}\) we assign a level \(j\), and to a \(\epsilon\)-circle \(C_j\) including its center \(p_j\) we assign levels \(j\) and \(j - 1\), if \(j > 1\). All operations like \(\cap, \epsilon, <\) now refer to their operands and their level. As a consequence \(P\) is simple,
T_j \cap T_{j+1} = C_{j+1}, and T_j \cap T_{j+1} = \emptyset, for i > 1. Let \( T = \bigcup_{i=1}^{n-1} T_j \). (Figure 3.7 shows a set of polygonal chains with the area of \( T \) shaded gray.) Clearly, each point in \( T \) has distance at most \( e \) to the polygonal chain. We require that the approximation lies in \( T \), starts at a point in a start set \( S \subseteq C_1 \), ends at a point in an end set \( E \subseteq C_n \), uses only orientations in \( C \), and leads through consecutive tubes.

Problem 3.1 (\( C \)-oriented Line Approximation)

**Instance:** Let \( P = (p_1, \ldots, p_n) \) be a polygonal chain in \( \mathbb{R}^2 \), let \( C \) be a set of orientations, \( e \) a positive constant, such that \( C_j \cap C_{j+1} = \emptyset \), \( j = 1, \ldots, n-1 \), \( S \subseteq C_1 \) a connected set of possible start points and \( E \subseteq C_n \) a set of possible end points.

**Problem:** Find a \( C \)-oriented line approximation of \( P \). That is a \( C \)-oriented polygonal chain \( Q = (q_1, \ldots, q_m) \), such that \( q_1 \in S \), \( q_m \in E \), and \( q_i \in T_i \), \( i = 1, \ldots, m \). Furthermore, link \( q_i q_{i+1} \) intersects all \( C_j, C_{j+1} \) in this order, in case \( q_i \in T_j \), \( q_{i+1} \in T_i \), \( j \) maximal, \( l \) minimal, and \( j < l \). In case that \( q_i \in T_j \), \( q_{i+1} \in T_j \), \( j \) maximal, \( l \) minimal, and \( j < l \), link \( q_i q_{i+1} \) intersects all \( C_i, C_{j+1} \) in this order.

Problem 3.2 (\( C \)-oriented Line Simplification (\textsc{COLS}))

**Instance:** An instance of the \( C \)-oriented line approximation problem.

**Problem:** Find a \( C \)-oriented line approximation of \( P \) that consists of a minimum number of links.

### 3.3.2 Reachable Regions

In this section we introduce the concept of reachable regions, modified from [AOS94]. For any tube \( T_j \) and any number of links \( i \) we will define a set that contains all points \( q \in T_j \) for which a \( C \)-oriented link path exists that starts in \( S \) and ends through \( T_1, \ldots, T_j \). For these sets we define a recursion formula that can be calculated in a dynamic program.

Let \( p \) and \( q \) be two points. We say \( p \) reaches \( q \) w.r.t. \( e \) if there exists a \( \lambda \in \mathbb{R} \) such that...
3.3. C-ORIENTED LINE SIMPLIFICATION

$q = p + \lambda c$. Locally in a tube we define the set of points that are reachable with one single link within a single tube as follows.

**Definition 3.3 (Reachable Region $\mathcal{R}(S, c)$)** Let $T$ be a tube, $S \subseteq T$, and $c \in \mathcal{C}$. We denote by $\mathcal{R}(S, c)$ the set of all points $q \in T$ for which there exists a point $p \in S$ such that $p$ reaches $q$ with respect to $c$. For $\bigcup_{c \in \mathcal{C}} \mathcal{R}(S, c)$ we simply write $\mathcal{R}(S)$.

Figure 3.9 gives an example of a reachable region. Now, we extend the notion of a reachable region to allow multiple links within a sequence of tubes. Note that the following definition is consistent with the definition of Problem 3.1.

**Definition 3.4 (Reachable Region $\mathcal{R}(S, T_j)$)** Let $T_j$ be the $j$th tube of a polygonal chain $P$ and $S$ the start set. We denote by $\mathcal{R}(S, T_j)$ the set of all points $q \in T_j$ for which a $C$-oriented $i$-link path $Q = (q_1, \ldots, q_i)$ from $S$ to $q$ exists such that $q_l \in T_l$, $l = 1, \ldots, i$. Furthermore, link $\overline{q_l q_{l+1}}$ intersects all $C_{j+1}, \ldots, C_{j}$ in this order, in case that $q_l \in T_j$, $q_{j+1} \in T_j$, $j$ maximal, $d$ minimal, and $j < d$. In case that $q_l \in T_{d}$, $q_{j+1} \in T_{d}$, $j$ maximal, $d$ minimal, and $j < d$, link $\overline{q_l q_{j+1}}$ intersects all $C_{d}, \ldots, C_{j+1}$ in this order.

Note that $\mathcal{R}(S, T_1) = S$, $\mathcal{R}(S, T_{1}) = \emptyset$ for $j > 1$ and $\mathcal{R}(S, T_{1}) = \mathcal{R}(S)$ in respect to tube $T_1$. Throughout this chapter we measure the distance between two points $p = (x_1, y_1)$ and $q = (x_2, y_2)$ in $\mathbb{R}^2$ by the $L_2$ metric. That is, $||p - q||_2 := (|x_1 - x_2|^2 + |y_1 - y_2|^2)^{1/2}$. In the next lemma and observation we prove that COLS always admits a solution.

**Lemma 3.5 (Existence)** Let $p, q \in T_1$, there exists a nonnegative integer $k$ such that $q \in \mathcal{R}(p, T_1)$. $k$ is bounded by $\frac{||p - q||_2}{2\epsilon} + 1$.

**Proof:** If $p = q$ we may choose $k = 0$. Assume henceforth that $p \neq q$. There exists a point $p'$ along $\overline{pp_{j+1}}$ such that $||p - p'||_2 \leq \epsilon$. Analogously, there exists a point $q'$ along $\overline{qq_{j+1}}$ such that $||q - q'||_2 \leq \epsilon$. $p$ lies in an $\epsilon$-circle $C'$ around $p'$ which is contained in $T_1$. If $pp' \in \mathcal{C}$ then $p$ can be reached with one link. Otherwise, $p$ lies in the wedge of 2 orientations $c$ and $\hat{c}$ in $\mathcal{C}$ with respect to $pp'$. Since the angle $\angle(c, \hat{c}) \leq \frac{\pi}{2}$, there exists a $\lambda, \nu \in \mathbb{R}$ such that $p = p' + \lambda c + \nu \hat{c}$, where the links $p'[p' + \lambda c]$ and $(p' + \lambda c)p$ lie inside $C''$. Let $\overline{p'q}$ be in the wedge of $c$ and $\hat{c}, c, \hat{c} \in \mathcal{C}$ (see Figure 3.10). We now show that there exists a $C$-oriented link path from $p'$ to $q'$ inside $T_1$ using only
CHAPTER 3. FROM GENERALIZATION TOWARDS
\textit{C}-ORIENTED LINE SIMPLIFICATION

Figure 3.10: There exists a \textit{C}-oriented link path consisting of a bounded number of links between each pair of points in a tube.

the two orientations \(c\) and \(\hat{c}\). Let \(\gamma\) be the angle between \(c\) and \(\overrightarrow{pq}\). Let \(\delta\) be the angle between \(\hat{c}\) and \(\overrightarrow{pq}\). Starting at \(p'\) using only directions \(c\) and \(\hat{c}\) one can reach all points on the lines \(p' + \lambda c\) and \(p' + \lambda \hat{c}\) that lie inside \(T_1\), \(\lambda \in \mathbb{R}\). With one more link, all points in the light shaded region are reachable. With two more links also the points in the dark shaded region are reachable and so on. Within each additional link an additional length of length \(L = 2c(\cot \gamma + \cot \delta)\) can be reached on \(\overrightarrow{pq}\). Since \(\gamma + \delta \leq \frac{\pi}{2}\) it follows that \(L \geq 2c\). Thus the number of links that are necessary to reach \(q'\) from \(p'\) is bounded by \(\frac{||p-q'||}{2c} + 4\). Therefore, \(q \in \mathcal{RR}^k(p, T_1)\), with \(k = \frac{||p-q'||}{2c} + 4\).

\begin{observation}
Every polygonal chain \(P\) has a \textit{C}-oriented line simplification for arbitrary \(c > 0\).
\end{observation}

\textbf{Proof:} Let \(S\) be the set of start points in \(C_1\). Lemma 3.5 yields that for any point \(q \in C_2\) there exists an \(i \in \mathbb{N}\) such that \(q \in \mathcal{RR}^i(S, T_1)\). Repetitive application of this observation yields that for any point \(q \in T_j\) there exists an \(i\) such that \(q \in \mathcal{RR}^i(S, T_1)\), \(j = 1, \ldots, n\). In particular, for each \(q \in E\) there exists an \(i \in \mathbb{N}\) such that \(q \in \mathcal{RR}^i(S, T_{n-1})\). Let \(Q\) be the according \textit{C}-oriented link path. From the definition of the reachable regions \(\mathcal{RR}^i(S, T_j)\) it follows that the path is a \textit{C}-oriented approximation of \(P\). From the existence of a \textit{C}-oriented line approximation of \(P\) follows
3.3. C-ORIENTED LINE SIMPLIFICATION

the existence of a C-oriented line approximation consisting of a minimum number of links.

We now introduce the concept of extensions. This allows us to extend a reachable region \( r \subset RR'(S, T_k) \) into \( RR'(S, T_j) \) whenever possible, \( k \neq j \). That is, we compute the C-oriented reachable region \( r' \) of a C-oriented reachable region \( r \), \( c \neq \hat{c} \), and try to extend it into as many consecutive tubes as possible. Figure 3.11 shows an extension of a C-oriented reachable region from \( T_{j-1} \) to \( T_j \). Note that the extension in Figure 3.12 is bounded by a boundary segment of \( C_j \). A point \( q \) in the striped black region of Figure 3.12 does not belong to the extension of \( r' \) from \( T_{j-1} \) to \( T_j \), since a link from \( r' \) into that region does not cut \( C_j \).

We define a forward extension as an extension from a tube \( T_k \) into a tube \( T_j \) with \( k < j \). Similarly, a backward extension is an extension from a tube \( T_j \) into a tube \( T_k \) with \( j > k \).

![Figure 3.11: The C-oriented reachable region \( r' \) is extended (shaded dark).](image1)

![Figure 3.12: The extension of \( r' \) is bounded by a segment of \( C_j \) in (shaded dark).](image2)
CHAPTER 3. FROM GENERALIZATION TOWARDS
\[\ldots\]
\[\ldots\]

Definition 3.7 (Extension)

Forward Extension \( \mathcal{F}(U, T_k, T_j) \): Let \( U \subseteq T_k \) for some \( k < j \). We denote by \( \mathcal{F}(U, T_k, T_j) \) the set of all points \( q \in T_j \) for which a \( C \)-oriented link from a point \( p \in U \setminus C_{k+1} \) exists, such that \( pq \) intersects all \( C_{k+1}, \ldots, C_j \) in this order. With \( \mathcal{F}(U, T_k, T_j, c) \) we denote the subset of \( \mathcal{F}(U, T_k, T_j) \) where all links are \( c \)-oriented, \( c \in C \). Then, \( \mathcal{F}(U, T_k, T_j) = \bigcup_{c \in C} \mathcal{F}(U, T_k, T_j, c) \) follows.

Backward Extension \( \mathcal{B}(U, T_k, T_j) \): Let \( U \subseteq T_j \) for some \( k < j \). We denote by \( \mathcal{B}(U, T_k, T_j) \) the set of all points \( q \in T_k \setminus C_{k+1} \) for which a \( C \)-oriented link from a point \( p \in U \) exists, such that \( pq \) intersects all \( C_j, \ldots, C_{k+1} \) in this order.

We will show in the next lemma that we do not need backward extensions to reach all reachable points with a minimum number of links. This does not follow directly from the problem definition. We have to make use of the assumption that for each orientation \( c \in C \) also the orthogonal orientation \( \overline{c} \) is element of \( C \).

Lemma 3.8 (Backward Extensions) Let \( q \in \mathbb{R} \setminus (S, T_j) \). Then, there exists a \( C \)-oriented path \( Q = (q_1, \ldots, q_t = q) \) from \( q_1 \in S \) to \( q \) such that each link \( q_{k-1}q_k \) is either contained in a tube \( T_l \) for some \( l \leq j \), or \( q_{k-1}q_k \) intersects all \( C_{k+1}, \ldots, C_j \) in this order with \( q_k \in T_k, q_{k+1} \in T_j \), \( c \) maximal, \( f \) minimal, and \( e < f \).

Proof: Assume there exists a \( q \in \mathbb{R} \setminus (S, T_j) \) that can only be reached with \( i \) links, when backward extensions are also considered. Let \( Q = (q_1, \ldots, q_t = q) \) be a path from \( q_1 \in S \) to \( q \). Then, there exists a minimum \( k, \) a maximum \( e, \) and a minimum \( f \) with \( e < f \), \( q_k \in T_k, q_{k+1} \in T_f \) such that \( q_{k-1}q_k \) intersects \( C_{k+1}, \ldots, C_{k+1} \) in this order. We now show how this link, its predecessor link, and successor link can be replaced by 3 other \( C \)-oriented links that are not backward extensions. Observe that \( Q \) has no self intersections, when we assign to a tube \( T_i \) a level \( i \) and to a circle \( C_i \) level \( i - i \) and level \( i \), as introduced in Section 3.3.1.

Note that \( Q \) enters the circle \( C_{k+1} \) three times. Let the intersecting points of \( Q \cap \partial C_{k+1} \) be \( u < v < v' < u'' < v'' \) (see Figure 3.13, 3.14). Since \( u \) can reach \( v'' \) within \( C_{k+1} \) in at most 2 links we can assume that \( q_{k-1} \) lies in a tube \( T_x \) with \( x \leq e \) and \( q_{k+2} \) lies in a tube \( T_y \) with \( y > e \). Therefore,
3.3. C-ORIENTED LINE SIMPLIFICATION

Figure 3.13: The solid polygonal subchain can be replaced by the dashed one.

Figure 3.14: The solid polygonal subchain can be replaced by the dashed one.
CHAPTER 3. FROM GENERALIZATION TOWARDS C-ORIENTED LINE SIMPLIFICATION

$\overline{uv}$, $u''v''$, and $\overline{u''v''}$ are segments of $Q$ and therefore $\mathcal{C}$-oriented. Let $\overline{uv}$ have orientation $\delta$ and let $\overline{u''v''}$ have orientation $\delta'$.

We make a case distinction according to the location of $u$, $v$, $u''$, and $v''$:

There exists a line $a_m$ that cuts $p_{i+1}$, $\overline{uv}$ and $\overline{u''v''}$: In this case we claim that there exists a $p \in \overline{uv}$ such that $p + \lambda \delta$ has non empty intersection with $\overline{u''v''}$. Let $R = \{x \in C_{2+1} | \exists y \in \overline{uv}, \lambda \in \mathbb{R} \text{ such that } x = y + \lambda \delta \}$. Observe that $R$ is point symmetric to the midpoint $p_{i+1}$.

Since $u$ and $v$ are positioned on opposite sides of $a_m$ in $C_{2+1}$, it follows that each point on $a_m$ can be reached with an $\mathcal{C}$-oriented link starting at a point on $\overline{uv}$. Since $\overline{u''v''}$ cuts $a_m$ it follows that there exists a point $z$ on $\overline{uv}$, a point $z'$ on $\overline{u''v''}$, and a $\lambda \in \mathbb{R}$ with $z' = z + \lambda \delta$. Thus, replacing $Q$ by $Q' = \langle q_1, \ldots, q_{k-1}, z, z', q_{k+2}, \ldots, q_i \rangle$ yields that $Q'$ has as many links as $Q$ and $\langle q_1, \ldots, q_{k-1}, z, z', q_{k+2} \rangle$ has no backward extension.

The points $u, v, u''$ and $v''$ lie on one half circle $H$ of $\partial C_{2+1}$: We make a case distinction according to the possible orders of $u, v, u''$, and $v''$ on $H$:

Order $u''$, $u, v, u''$ and $u, u'', v, v$ and their reversals; See Figure 3.14 for an example. Then, either there exists a $\lambda \in \mathbb{R}$ and a $z$ on $\overline{u''v''}$ such that $z = u + \lambda \delta$ or there exists a $\lambda \in \mathbb{R}$ and a $z$ on $\overline{uv}$ such that $z = u'' + \lambda \delta$. In the first case, $Q' = \langle q_1, \ldots, q_{k-1}, u, z, q_{k+2}, \ldots, q_i \rangle$ is a link path with at most as many links as $Q$ has and $\langle q_1, \ldots, q_{k-1}, u, z, q_{k+2} \rangle$ has no backward extension. In the second case $Q' = \langle q_1, \ldots, q_{k-1}, z, u'', q_{k+2}, \ldots, q_i \rangle$ is a link path with at most as many links as $Q$ has and $Q' = \langle q_1, \ldots, q_{k-1}, z, u'', q_{k+2} \rangle$ has no backward extension.

Order $u, v, v''$, $u''$, order $v, u, u''$, $v''$, and their reversals; In this case either link $\overline{u_q q_{k+2}}$ lies inside $T_s$ or $\overline{q_{k+1} q_{k+2}}$ lies inside $T_i$. In both cases $q_{k+1}$ can be reached without backward extensions.

This procedure can be applied to all backward extensions in $Q$. Thus, all points in $\mathcal{R}(\mathcal{R}(S, T_j))$ reachable without backward extensions.

We now can express the calculation rule for the reachable regions as a recursion formula.

**Theorem 3.9 (Recursion Formula)** For $i, j > 0$ we have

$$\mathcal{R}(S, T_j) = \mathcal{R}(\mathcal{R}(\mathcal{R}(S, T_j))) \cup \bigcup_{k=1}^{j-1} \mathcal{F}(\mathcal{R}(\mathcal{R}(\mathcal{R}(S, T_j))), T_k, T_j)$$
3.3. C-ORIENTED LINE SIMPLIFICATION

With \( \mathcal{R}^i(S, T, c) \) we denote the subset of \( \mathcal{R}^0(S, T) \) where the last link is c-oriented. Then, \( \mathcal{R}^i(S, T) = \bigcup_{c \in \mathcal{C}} \mathcal{R}^i(S, T, c) \) follows.

**Proof:** In the previous lemma we showed that backward extensions are not necessary to reach a point within a minimum number of C-oriented links. Thus, the correctness of this formula follows.

**Corollary 3.10** Let \( i \) be minimal such that \( E \cap \mathcal{R}^i(S, T_{n-1}) \neq \emptyset \). The minimum C-oriented link path from \( S \) to \( E \) has \( i \) links.

**Proof:** With Theorem 3.9 it follows that the corresponding C-oriented \( i \)-link path for each \( q \in (E \cap \mathcal{R}^i(S, T_{n-1})) \) consists of a minimum number of links.

The sets \( \mathcal{R}^i(S, T) \) can be computed by dynamic programming according to the Recursion Formula (see algorithm Basic COLS). Let \( q \in \mathcal{R}^i(S, T, c) \). The C-oriented link path from \( q \) to \( S \) can be computed as follows: Starting at \( q \) we create a line \( q + \lambda c, \lambda \in \mathbb{R} \). The c-oriented reachable region of \( q \) is associated with its generating reachable region \( r' \in \mathcal{R}^{i-1}(S, T_k, c) \). We compute a point \( q' \in r' \) which lies on \( q + \lambda c \). This point \( q' \) is the next vertex of the link path. Then, we repeat this procedure starting at \( q' \) and continue until we reach a point in \( S \).

**Algorithm 3** Basic COLS \((p_1, \ldots, p_n, c, \epsilon)\)

01) \( i := 1; \)
02) for \( j := 1 \) to \( n-1 \) do \( \mathcal{R}^0(S, T, c) := \emptyset; \)
03) \( \mathcal{R}^0(S, T_1, c) := S \forall c \in \mathcal{C}; \)
04) while \( E \cap \mathcal{R}^i(S, T_{n-1}) = \emptyset \forall c \in \mathcal{C} \) do {
05) \( \mathcal{R}^i(S, T, c) := \emptyset \forall c \in \mathcal{C} \) do {
06) for \( j := 1 \) to \( n-1 \) do {
07) for each \( c \in \mathcal{C} \) do {
08) \( \mathcal{R}^i(S, T_j, c) := \mathcal{R}^i(S, T_j, c) \cup \mathcal{R}^i(S, T_{j+1}, c) \); 
09) for \( j := j \) to \( n-1 \) do 
10) \( \mathcal{R}^i(S, T_{j'}, c) := \mathcal{R}^i(S, T_j, c) \cup \bigcup_{k=j}^{j-1} \mathcal{F}(\mathcal{R}^i(S, T_k), T_k, T_{j'}, c); \)
11) } }
12) \( i := i + 1; \)
13) }


CHAPTER 3. FROM GENERALIZATION TOWARDS C-ORIENTED LINE SIMPLIFICATION

Inserting each reachable region into a set $\mathcal{R}^j(S, T_j)$ yields that each region $r \in \mathcal{R}^j(S, T_j)$ generates at most $|C|$ reachable regions in $\mathcal{R}^{j+1}(S, T_j)$. Furthermore, each region $r \in \mathcal{R}^j(S, T_j)$ can generate at most $|C|$ reachable regions in the extension step per tube $l = j + 1, \ldots, n$. Let $f(i, j)$ be the number of reachable regions in $\mathcal{R}^j(S, T_j)$, $1 \leq j \leq n - 1$. The following equations hold:

$$f(i, j) \leq \begin{cases} |C| \sum_{l=1}^{i} f(i-1, l) & \text{if } i > 1, j > 1 \\ |C| & \text{if } i = 1, j \geq 1 \end{cases}$$

These recurrence equations lead to a number of reachable regions that grows exponentially in the number of line segments $k$ of the minimum link path.

3.3.3 Speed-Up Techniques

In order to consider only a polynomial (in $n, k$, and $|C|$) number of reachable regions we study the form of reachable regions and classify them. This allows us to handle and store the reachable regions efficiently.

Lemma 3.11 Let $r$ be a reachable region in $\mathcal{R}^j(S, T_j)$. Then, $r \cap C_j \neq \emptyset$.

Proof: This follows inductively. The reachable region of $\mathcal{R}^1(S, T_1, c)$ has nonempty intersection with $C_1$ since $S \subseteq C_1$, $c \in C$. Therefore $\mathcal{R}^j(S, T_1, c) \cap C_1 \neq \emptyset$, for any $c \in C$. Let $r \in \mathcal{R}^j(S, T_k)$ be a reachable region in $T_k$, $k < j$. Since the extension $\mathcal{E}(r, T_k, T_j, c)$ has nonempty intersection with $C_j$ (by definition) the same follows for any reachable region in $\mathcal{R}^j(S, T_j)$, $1 \leq j \leq n - 1$.

Let $r$ be a c-oriented reachable region in a tube $T_j$. Then, the boundary of $r$ has the nice property that is determined by the boundary of $T_j$, the boundary of $C_j$ and c-oriented lines. In case that $C_j \cap C_{j+1} \neq \emptyset$ the boundary of a reachable region can also contain boundary segments of non local circles, which would make the handling much more difficult.

Theorem 3.12 (Type of a Reachable Region) Let $S \subseteq C_1$ be a connected set of points in $C_1$. Let $c \in C$. A c-oriented reachable region $r$ in $T_j$ can be categorized to have the following forms:

1. $r$ is determined by the intersection of two parallel c-oriented lines and the boundary of $T_j$. We say that type($r$) = 1. (See Figure 3.11.)
2. $r$ is a reachable region in $T_j$ that is defined by a connected boundary segment $s_j \subseteq \partial C_j \setminus \partial T_{j-1}$ and a direction $\tau \in \{c, -c\}$. $r = \{q \in T_j \mid \exists q' \in s_j, \lambda > 0$ such that $q = q' + \lambda \tau\}$. We say that type($r$) = 2. (See Figure 3.12.)
3.3. C-ORIENTED LINE SIMPLIFICATION

Proof: Let \( r \) be the c-oriented reachable region generated from \( r' \) in \( T_j \).
We show by induction on \( i \) that each reachable region \( r \in \mathcal{R}R^i(S, T_j, c) \) is of type 1 or type 2.

\( i = 1, j = 1 \): The c-oriented reachable region of \( S \) in \( T_1 \) clearly is of type 1 (see also Figure 3.9).

\( i = 1, j > 1 \): Let \( r = \mathcal{F}E(r', T_1, T_j, c) \). Let \( s = \partial C_j \setminus \partial T_{j-1} \cap r \). Clearly, for each \( q \in r \) there exists a \( q' \in s \) and a \( \lambda \in \mathbb{R} \) such that \( q = q' + \lambda c \) and the line \( q'q \setminus s \neq \emptyset \). For each pair \( (q, q') \) with \( q \in r \) and \( q' \in s \) such that \( q = q' + \lambda c \), the sign of \( \lambda \) is either positive or negative since \( C_j \cap C_{j+1} = \emptyset \). Let \( \tau = c \) if \( \lambda > 0 \), \( \tau = -c \) otherwise. Therefore, \( r = \mathcal{F}E(r', T_1, T_j, c) = \{ q \in T_j \mid \exists q' \in s, \lambda > 0 \text{ such that } q = q' + \lambda \tau \} \).

\( s \) is a connected boundary segment since for each two points \( q_1 \) and \( q_2 \) in \( s \) with \( q_1 + \lambda \tau \) intersects \( s \) and \( q_2 + \lambda \tau \) intersects \( s \) there is a path from \( q_1 \) to \( q_2 \) in \( s \) and all points \( q_3 \) on this path fulfill that \( q_3 + \lambda \tau \) also intersects \( \partial C_j \setminus \partial T_{j-1} \) and therefore \( s \). Since \( s = \partial C_j \setminus \partial T_{j-1} \cap r \), \( s \) can be at most a half circle. If \( s \subset \partial T_j \) then type(\( r \)) = 1. Otherwise, type(\( r \)) = 2.

Induction precondition: Let the reachable regions of \( \mathcal{R}R^1(S, T_j), \ldots, \mathcal{R}R^{i-1}(S, T_j) \) be either of type 1 or of type 2.

\( i - 1 \Rightarrow i \):

\( r' \subset T_j \): The c-oriented reachable region of \( r' \) in \( T_j \) clearly is of type 1.

\( r' \subset T_{j,k} \), \((k < j - 1) \) or \((k = j - 1 \text{ and } r' \cap C_j = \emptyset) \): Analogously to case \( i = 1, j > 1 \), \( r' \subset T_{j-1,k} \), \( r' \cap C_j \neq \emptyset \): Let \( r^+ = \{ q \in T_j \mid \exists q' \in r' \setminus C_j, \lambda > 0, q' + \lambda c = q \} \) and \( r^- = \{ q \in T_j \mid \exists q' \in r' \setminus C_j, \lambda < 0, q' + \lambda c = q \} \). Clearly, \( r = r^+ \cup r^- \). Let \( s^+ = (\partial C_j \setminus \partial T_{j-1}) \cap r^+ \) and \( s^- = (\partial C_j \setminus \partial T_{j-1}) \cap r^- \).

We clearly now can subdivide \( s^+ \) and \( s^- \) into connected boundary segments.

For complexity reasons we show that we yield altogether at most two connected boundary segments. That is, if \( s^+ (s^-) \) contains two disjoint segments then \( s^- = \emptyset (s^+ = \emptyset) \)). Figures 3.15–3.18 show the possibilities of a region of type 1 to intersect \( C_j \). Note that a region in \( T_{j-1} \) of type 2 has empty intersection with \( C_j \). In the first two cases we yield one connected boundary segment (see Figure 3.15, 3.16). In the third case depending on \( c \) we either yield two boundary segments in \( s^+ \) and \( s^- = \emptyset \) (see Figure 3.18) or two boundary segments in \( s^- \) and \( s^+ = \emptyset \) or one in \( s^+ \) and one in \( s^- \) (see Figure 3.17).
CHAPTER 3. FROM GENERALIZATION TOWARDS 
C-ORIENTED LINE SIMPLIFICATION

Figure 3.15: $r'$ (shaded light) generates one connected boundary segment $s^+$ (solid) for the forward extension in orientation $c$.

Figure 3.16: $r'$ generates one connected boundary segment $s^+$ (dashed) for the forward extension in orientation $c$.

Figure 3.17: $r'$ generates one connected boundary segment $s^+$ (dashed) and one boundary segment $s^-$ (solid) for the forward extension in orientation $c$.

Figure 3.18: $r'$ generates two connected boundary segment $s^+$ (dashed) for the forward extension in orientation $c$. 
In the next two observations we show how to shrink the number of reachable regions by deleting redundant reachable regions and unifying reachable regions.

**Observation 3.13** Let \( r \) be a c-oriented reachable region in \( \mathcal{R}[R]^i(S, T_j, c) \). Then, \( r \) can be deleted in case that there exists a \( r' \in \mathcal{R}[R]^i(S, T_j, c) \) with \( r \subseteq r' \).

**Observation 3.14** Let \( r \) and \( r' \) be two c-oriented reachable regions in \( \mathcal{R}[R]^i(S, T_j, c) \) with \( r \cap r' \neq \emptyset \), \( r \not\subseteq r' \), and \( r' \not\subseteq r \). Let \( r \) and \( r' \) be reachable regions of type 1 or of type 2 with equal \( \tau \). We can unify \( r \) and \( r' \) for the computation of the reachable region of \( r \) and \( r' \).

Note that we must store a back pointer from \( r \cup r' \) to \( r \) and \( r' \) since during the computation of the link path we have to follow the links backwards until we reach the start set \( S \). The classification of a reachable region in type 1 and type 2 leads to the notation that \( \mathcal{R}[R]^i(S, T_j, c) \) consists of a set of reachable regions and a reachable region is an element of \( \mathcal{R}[R]^i(S, T_j, c) \).

Let \( \bar{c} \) be the orthogonal orientation to \( c \). The projection of a tube \( T_j \) to \( \bar{c} \) yields an interval \( \mathcal{I}(T_j, \bar{c}) \).

**Observation 3.15** Each c-oriented reachable region in \( T_j \) is uniquely represented by an interval in \( \mathcal{I}(T_j, \bar{c}) \) (namely the projection of the reachable region to \( \bar{c} \)) and a tag specifying whether the reachable region is of type 1 or type 2 accompanied by the orientation \( \tau \).

In a further tag field a reference to the generating reachable region is stored. This property we use to store the reachable regions efficiently. We associate three sets of intervals with each \( \mathcal{R}[R]^i(S, T_j, c) \): \( \mathcal{I}(S, T_j, c, \pm \) contains all intervals corresponding to reachable regions of type 1. \( \mathcal{I}(S, T_j, c, +) \) contains all intervals corresponding to reachable regions of type 2 with \( \tau = +c \) and \( \mathcal{I}(S, T_j, c, -) \) with \( \tau = -c \).

We insert a c-oriented reachable region \( r \) into \( \mathcal{R}[R]^i(S, T_j, c) \) only if it is not yet contained in \( \mathcal{R}[R]^i(S, T_j, c) \). In this case we delete all c-oriented reachable regions \( r' \) that are contained in \( r \) see Observation 3.13. and procedure **insertion**. Before we compute \( \mathcal{R}[R]^i+1(S, T_j) \) we unify the reachable regions in \( \mathcal{R}[R]^i(S, T_j, c) \) for each \( c \in C \) in order to shrink their number (see Observation 3.14. and procedure **update**). Let \( x \in \{ \pm, +, - \} \).

The insertion operations in \( \mathcal{I}(S, T_j, c, x) \) have the property that each two intervals \( i_1 \) and \( i_2 \) in \( \mathcal{I}(S, T_j, c, x) \) are either disjoint or \( i_1 \not\subseteq i_2 \) and \( i_1 \not\supseteq i_2 \).

We suggest handling each \( \mathcal{I}(S, T_j, c, x) \) as an ordered list. Thus, the
CHAPTER 3. FROM GENERALIZATION TOWARDS C-ORIENTED LINE SIMPLIFICATION

Procedure 1: Insertion (e-oriented reachable region r, $\mathcal{RR}(S,T_j,c)$)

1. $r_p :=$ the projection of r onto c;
2. if \( \text{type}(r) = 1 \) then {
   3. \( \exists r'_p \in I_i(S,T_j,c,\pm) \) such that \( r_p \subseteq r'_p \) then
      4. \( \text{return; \} \}
   5. else {
      6. \( \forall r'_p \in I_i(S,T_j,c,\pm) \cup I_i(S,T_j,c,+ ) \cup I_i(S,T_j,c,-) \)
         \( \text{ such that } r'_p \subseteq r_p \text{ do} \)
      7. delete \( r'_p \) from the structure;
      8. insert \( r_p \) into \( I_i(S,T_j,c,\pm) \);
   9. \} \}
10. else {
11. \( \exists r'_p \in I_i(S,T_j,c,\pm) \cup I_i(S,T_j,c,+ ) \cup I_i(S,T_j,c,-) \)
      \( \text{ such that } r'_p \subseteq r_p \text{ then} \)
12. \( \text{return; \} \}
13. else {
14. \( \text{if } \text{type}(r) = 2 \text{ with } \tau = c \text{ then} \) 
15. \( \forall r'_p \in I_i(S,T_j,c,+) \) such that \( r'_p \subseteq r_p \text{ do} \)
16. delete \( r'_p \) from the structure;
17. insert \( r_p \) into \( I_i(S,T_j,c,+) \);
18. \} \}
19. else {
20. \( \forall r'_p \in I_i(S,T_j,c,-) \) such that \( r'_p \subseteq r_p \text{ do} \)
21. delete \( r'_p \) from the structure;
22. insert \( r_p \) into \( I_i(S,T_j,c,-) \);
23. \} \}

insertion, deletion, and search for an interval can be done in $O(\log l)$ time, when l is the length of the list. Let $u$ be an interval that contains $k$ intervals of the list. Since these $k$ intervals are consecutive in the list, the deletion or search of these intervals can be done in $O(\log (l + k)$ time.

If we once have reached a point $p \in T_j$ we can estimate the maximum number of iterations that we have to compute extensions into that tube.

Observation 3.16 Let $p \in \mathcal{RR}(S,T_j)$. Then, $C_j \subset \mathcal{RR}(S,T_j,c)$, for each $c \in C$. 

3.3. C-ORIENTED LINE SIMPLIFICATION

Procedure 2  \textit{Update} (\textit{RR}'(S, T_j, c), c)

1) \textbf{for each} \( x \in \{\pm, +, -\} \) \textbf{do}
   2) \hspace{0.5cm} let \( r_1, \ldots, r_l \) be the intervals of \( I_i(S, T_j, c, x); \)
   3) \hspace{0.5cm} compute the minimum set of intervals \( r_1, \ldots, r_k^* \) such that
      \[ \bigcup_{i=1}^{l} r_i = \bigcup_{j=1}^{k} r_j^*; \]
   4) \hspace{0.5cm} with each \( r_j^* \) we associate all \( r_i \) with \( r_j^* \cap r_i \neq \emptyset; \)
   5) \hspace{0.5cm} insert \( r_1, \ldots, r_k^* \) into \( I_i(S, T_j, c, x); \)
   6) \textbf{for all} \( r_j \in \mathbb{I}_j(S, T_j, c, \pm) \cup \mathbb{I}_j(S, T_j, c, -) \) \textbf{do}
      \hspace{0.5cm} delete \( r_j \) from the data structure;
   7) \textbf{return} \( I_i(S, T_j, c, \pm), I_j(S, T_j, c, +), I_j(S, T_j, c, -); \)

\textbf{Proof:} Since \( p \in \textit{RR}'(S, T_j) \), it follows that there exists a \( p' \in C_j \) with \( p' \in \textit{RR}'(S, T_j) \). Each point \( q \in C_j \) is reachable from \( p' \) with at most two links. Since for each two clockwise next orientations \( c, \tilde{c} \in C \) the angle \( \angle c(q, \tilde{c}) \leq \frac{\pi}{2} \), \( C_j \) is covered with reachable regions in the second iteration and thus with an arbitrary c-oriented reachable region in the third iteration.

This ends our speed-up investigations. Recapitulating, we compute the reachable regions \( \textit{RR}'(S, T_j) \) by using dynamic programming, together with the interval lists as data structure, the insertion and update technique, and the \textit{shape} computation of reachable regions. That is, we compute a c-oriented reachable region in a tube only in case when the tube is not yet covered with c-oriented reachable regions. Furthermore, we compute a forward extension into a tube \( T_j \) only in case when the circle \( C_j \) is not yet covered with reachable regions. (See algorithm \textit{COLS}.)

3.3.4 Runtime and Space Calculation

We now calculate the maximum number of reachable regions in each tube. According to our Recursion Formula 3.9 we will estimate the number of c-oriented reachable regions in a tube \( T \) that is generated by

1. non c-oriented reachable regions in \( T \) (see Lemma 3.17),
2. forward extensions of non c-oriented reachable regions into \( T \) (see Corollary 3.18).
Algorithm 4  \( \text{COLS}(p_1, \ldots, p_n, c, \epsilon) \)

01) \( i := 1; \)
02) \( \text{for } j := 1 \text{ to } n \text{ do } \{ \)
03) \( \text{for each } c \in C \text{ do } \mathcal{R} \mathcal{R}^0(S, T_j, c) := \emptyset; \}
04) \( \mathcal{R} \mathcal{R}^0(S, T_1, c) := S, \text{ for each } c \in C; \)
05) \( \text{while } E \cap \mathcal{R} \mathcal{R}^i(S, T_{n-1}, c) = \emptyset, \text{ for each } c \in C \text{ do } \{ \)
06) \( \mathcal{R} \mathcal{R}^i(S, T_j, c) := \mathcal{R} \mathcal{R}^{i-1}(S, T_j, c) \text{ for } j := 1, \ldots, n, \text{ for each } c \in C; \)
07) \( \text{for } j := 1 \text{ to } n - 1 \text{ do } \{ \)
08) \( \text{if } nr(j + 1) = -4 \text{ or } nr(j + 1) + 4 > i \text{ then } \{ \)
09) \( \text{for each } c \in C \text{ do } \{ \)
10) \( \text{for each } e \in E \text{ do } \{ \)
11) \( \text{INSERT}(\mathcal{R} \mathcal{R}(r, e), \mathcal{R} \mathcal{R}^i(S, T_j, c)); \}
12) \( \text{for each } r \in \mathcal{R} \mathcal{R}^{i-1}(S, T_j, c) \text{ do } \{ \)
13) \( \text{for each } \hat{c} \in C \text{ do } \{ \)
14) \( \text{if } j < n - 1 \text{ and } nr(j + 1) < i + 4 \text{ then } \{ \)
15) \( \text{for each } e \in E \text{ do } \{ \)
16) \( \text{for each } r \in \mathcal{R} \mathcal{R}^{i-1}(S, T_j, c) \text{ do } \{ \)
17) \( j' := j + 1; \)
18) \( \text{while } (E \mathcal{E}(r, T_j, T_j, c) \neq \emptyset) \text{ do } \{ \)
19) \( \text{if } nr(j) = -4 \text{ or } nr(j) + 4 > i \text{ then } nr(j') := i; \)
20) \( \text{INSERT}(E \mathcal{E}(r, T_j, T_j, c), \mathcal{R} \mathcal{R}^i(S, T_j, c), c); \}
21) \( j' := j' + 1; \}
22) \( \text{for each } c \in C \text{ do } \{ \)
23) \( \mathcal{R} \mathcal{R}^i(S, T_j, c) := \mathcal{R} \mathcal{R}^{i-1}(S, T_j, c) \text{ for each } c \in C; \)
24) \( \text{UPDATE}(\mathcal{R} \mathcal{R}^i(S, T_j, c), c) \text{ for each } c \in C; \)
25) \( \text{for each } r \in \mathcal{R} \mathcal{R}^{i-1}(S, T_j, c) \text{ do } \{ \)
26) \( \text{for each } c \in C \text{ do } \{ \)
27) \( i := i + 1; \}
28) \} \}
29) \} \}

CHAPTER 3. FROM GENERALIZATION TOWARDS
C-ORIENTED LINE SIMPLIFICATION

64
3.3. C-ORIENTED LINE SIMPLIFICATION

3. c-oriented reachable regions in $T$ (see Observation 3.19).

4. forward extensions of c-oriented reachable regions into $T$ (see Lemma 3.20).

With these calculations we are able to estimate the maximum number of reachable regions in $R^R(S, T_j, c)$ (see Theorem 3.21) which leads to the runtime and space of the algorithm.

We call two reachable regions $r_1$, $r_2$ of one orientation in a tube disjoint, if $r_1 \cap r_2 = \emptyset$, type($r_1$) ≠ type($r_2$), or type($r_1$) = type($r_2$) = 2 and $\tau(r_1) \neq \tau(r_2)$. We call procedure Update with $R^R(S, T_j, c)$ before computing the sets $R^R(S, T_j, c)$, for $j = 1, \ldots, n - 1$. Therefore, we only have to take into account the number of disjoint reachable regions in $R^R(S, T_j, c)$.

**Lemma 3.17** Let $r_1, \ldots, r_k$ be the c-oriented reachable regions in $R^R(S, T_j, c)$. Let the angle between $c$ and $\hat{c}$ be $\alpha_r, r_1, \ldots, r_k$ create at most $\left\lceil \frac{\alpha_r}{2\pi} \right\rceil$ disjoint $\hat{c}$-oriented reachable regions in $R^R(S, T_j, \hat{c})$.

![Diagram](image)

**Figure 3.19:** By knowing $C_j, c$, and $\alpha$ we can successively compute $\beta, d_1$, $d_2, \ldots, g, h_1, h_2, \ldots, e, \ldots$.

**Proof:** From Theorem 3.12 we know that the reachable regions can be categorized into 2 types. Each reachable region $r_3 \cap C_j$ is defined by
the inner of two parallel c-oriented lines cut with \( C_j \), \( 1 \leq a \leq k \). The c-oriented reachable region of \( r_a \) and \( r_b \) can only be disjoint if the c-oriented reachable region of \( r_a \cap C_j \) and \( r_b \cap C_j \) are disjoint, \( 1 \leq a, b \leq k \).

We therefore get an upper bound on the number of disjoint c-oriented reachable regions of \( r_1, \ldots, r_k \) when we compute the number of disjoint c-oriented reachable regions of \( r'_1 = r_1 \cap C_j \), \ldots, \( r'_k = r_k \cap C_j \). Now, let \( r_a \subseteq r_b \). Since \( \mathcal{R}\mathcal{R}(r_a, c) \subseteq \mathcal{R}\mathcal{R}(r_b, c) \), we can assume that \( r'_1, \ldots, r'_k \) have minimum diameter and are disjoint. We further assume that \( r'_1, \ldots, r'_k \) have minimum distance to each other and are ordered according to orientation \( c \). Therefore, \( r'_1 \) has distance \( \epsilon - \delta \) to the center of \( C_j \) where \( \delta \) is a number approaching 0. For convenience we say that \( r'_1 \) has distance \( \epsilon \) to the center of \( C_j \). Please confer Figure 3.19 for the following considerations.

Trigonometric calculations yield that \( r'_2 \) must have distance greater than \( d_1 = 2\epsilon \sin^2(\alpha) \) to \( r'_1 \), otherwise the c-oriented reachable regions of \( r'_1 \) and \( r'_2 \) would intersect. In case that \( d_1 > \epsilon \) we know that at most 2 disjoint c-oriented reachable regions can be created. Similar computation yields that \( r'_2 \) must have distance greater than \( d_2 + d_1 = 2\epsilon \sin(\alpha) \sin(3\alpha) \) to \( r'_2 \) or distance greater than \( d_2 + 2d_1 \) to \( r'_1 \). In general \( r'_j \) has to have distance greater than \( d_{i-1} + d_{i-2} \) to \( r'_{i-1} \) or distance greater than \( \sum_{j=1}^{i-2}2d_j + d_{i-1} \) to \( r'_i \), with

\[
    d_i = \begin{cases} 
    2\epsilon \sin(\alpha) \sum_{j=1}^{i}(-1)^j \sin((2j-1)\alpha) & \text{if } i \text{ is even}, \\
    2\epsilon \sin(\alpha) \sum_{j=1}^{i-1}(-1)^{j+1} \sin((2j-1)\alpha) & \text{otherwise}. 
    \end{cases}
\]

These equations are only valid while \( r'_i \) has distance smaller than \( \epsilon \) to \( r'_1 \) which is the case for \( i < \frac{\pi}{4\alpha} - \frac{1}{2} \). It follows that at most \( \lceil \frac{\pi}{4\alpha} \rceil \) disjoint c-oriented reachable regions can be created from the c-oriented reachable regions with distance at most \( \epsilon \) to \( r_1 \) (including \( r_1 \)). Thus at most \( \lceil \frac{\pi}{2\alpha} \rceil \) disjoint c-oriented reachable regions can be created from \( r_1, \ldots, r_k \).

Corollary 3.18 A maximum of \( 2\lceil \frac{\pi}{2\alpha} - \frac{1}{2} \rceil (|C| - 1) \) disjoint c-oriented reachable regions can exist in \( \mathbb{R}\mathbb{R}(\mathbb{R}\mathbb{R}^i(S, T_j), c) \setminus \mathbb{R}\mathbb{R}^i(S, T_j, c) \cup \bigcup_{k=1}^{l-1} \mathbb{F}\mathbb{E}((\mathbb{R}\mathbb{R}^i(S, T_k) \setminus \mathbb{R}\mathbb{R}^i(S, T_k, c)), T_k, T_j, c) \).

Observation 3.19 1. Each \( \mathbb{R}\mathbb{R}^i(S, T_j, c) \) contains one unique reachable region, \( c \in C \).

2. Let \( r \in \mathbb{R}\mathbb{R}^i(S, T_j, c) \) with type\((r) = 1 \). Then, \( \mathbb{R}\mathbb{R}(r, c) = r \).

3. Let \( r \in \mathbb{R}\mathbb{R}^i(S, T_j, c) \) with type\((r) = 2 \) and \( \tau \in \{+c, -c\} \). Then, \( \mathbb{R}\mathbb{R}(r, c) \supset r \) and type\( (\mathbb{R}\mathbb{R}(r, c)) = 1 \).
In the next lemma we show that in case that a c-oriented reachable region has a c-oriented forward extension from tube $T_k$ to tube $T_j$, then at most one c-oriented reachable region in each tube $T_{k+1}, \ldots, T_{j-1}$ has a forward extension into $T_j$.

**Lemma 3.20** Let $\mathcal{F}(\mathbb{R}^d(S, T_k, c), T_k, T_j, c) \neq \emptyset$, $k < j < n$. Then, $\mathcal{F}(\mathbb{R}^d(S, T_l, c), T_l, T_j, c)$ contains at most one reachable region for any $l$, $k < l < j$.

**Proof:** We start with the study of some special properties of a c-oriented reachable region that is extended with orientation $c$. Let $r' \in \mathbb{R}^d(S, T_k, c)$ such that $r = \mathcal{F}(r', T_k, T_j, c) \neq \emptyset$. Then, type$(r') = 2$ with $\tau(r') \in \{c, -c\}$. From Theorem 3.12, it follows that there exists a $s' \subset \partial C_k$ which is the $r'$ defining boundary segment. The definition of a forward extension yields that for all $u \in r'$ there exists a $v \in s'$ and a $\lambda > 0$ such that $u = \lambda \tau(r')$. Since $r'$ can only be extended in direction $-\tau(r')$ we get $\tau(u) = -\tau(r')$. Furthermore, for all $u \in r$ there exists a $v \in s$ and a $\lambda > 0$ such that $u = v - \lambda \tau(r')$.

Secondly, under the assumption that $\mathcal{F}(\mathbb{R}^d(S, T_k, c), T_k, T_j, c) \neq \emptyset$ and $\mathcal{F}(\mathbb{R}^d(S, T_l, c), T_l, T_j, c) \neq \emptyset$, $k < l < j$. $r'_l \subset \mathbb{R}^d(S, T_l, c)$ with $\mathcal{F}(r'_l, T_l, T_j, c) \neq \emptyset$ and any $r''_r \subset \mathbb{R}^{d-1}(S, T_m, c)$ with $\mathcal{F}(r''_r, T_m, T_l, c) \neq \emptyset$ it follows that $r'_l \cap \mathcal{F}(r''_r, T_m, T_l, c) = \emptyset$, for any $m < l$. Let $r'_f \in \mathbb{R}^d(S, T_k, c)$ such that $r_f = \mathcal{F}(r'_f, T_k, T_j, c) \neq \emptyset$ and let $r'_j \in \mathbb{R}^d(S, T_i, c)$ such that $r_j = \mathcal{F}(r'_j, T_i, T_j, c) \neq \emptyset$, with $k < l < j$. It follows that $\tau(r'_f) = -\tau(r_k)$ and $\tau(r'_j) = -\tau(r_l)$. Let $s'_k$ (resp. $s'_j$) be the $r'_k$ (resp. $r'_j$) defining boundary segment. Then, there exists a $u \in s'_k$ such that $u + \lambda \tau(r_k)$ intersects $C_1, \ldots, C_j$ in this order. Analogously, there exists a $v \in s'_j$ such that $v + \lambda \tau(r_l)$ intersects $C_1, \ldots, C_j$ in this order. Since $k < l < j$ and $C_j \cap C_{j+1} = \emptyset$, for $1 \leq j \leq n - 1$, it follows that $\tau(r_k) = \tau(r_l)$.

We now assume that there exists a reachable region $r''_r \in \mathbb{R}^{d-1}(S, T_m, c)$ such that $r''_r \supset \mathcal{F}(r''_r, T_m, T_l, c)$. It follows that type$(r''_r) = 2$ with $\tau(r''_r) = -\tau(r')$. Let $s''_m$ be the $r''_r$ defining boundary segment. Then, $s''_m \subset \partial C_m$ and there exists a $v \in s''_m$ such that $v + \lambda \tau(r''_r)$ intersects $C_m, \ldots, C_1$ in this order. Since there exists a $u \in s'_k$ such that $u + \lambda \tau(r_k)$ intersects $C_k, \ldots, C_j$ in this order, $\tau(r_k) = -\tau(r_l)$, $m, k < l$, and $C_{l-1} \cap C_l = \emptyset$ we come to a contradiction.

Thirdly, we assume that $\mathcal{F}(\mathbb{R}^d(S, T_l, c), T_l, T_j, c)$ contains 2 disjoint reachable regions and lead this to a contradiction: Let $r'_1 \in \mathbb{R}^d(S, T_l, c)$ such that $r_1 = \mathcal{F}(r'_1, T_l, T_j, c) \neq \emptyset$ and let $r'_2 \in \mathbb{R}^d(S, T_i, c)$ such that $r_2 = \mathcal{F}(r'_2, T_i, T_j, c) \neq \emptyset$. We show that $r_1 \cap r_2 \neq \emptyset$: From the
first case it follows that \( \tau(r'_{i_1}) = \tau(r''_{i_1}) = \tau(r'_{i_2}) = -\tau(r_{i_1}) = -\tau(r_{i_2}) = -\tau(r_k) \). Hence, there exists a reachable region \( r''_{i_1} \in \mathcal{R}\mathcal{R}^{i-1}(S,T_{m_1},c) \) such that \( r'_{i_1} \supseteq \mathcal{FE}(r''_{i_1},T_{m_1},T_k,c) \) and \( r''_{i_2} \in \mathcal{R}\mathcal{R}^{i-1}(S,T_{m_2},c) \) such that \( r'_{i_2} \supseteq \mathcal{FE}(r''_{i_2},T_{m_2},T_k,c), m_1, m_2 < l \).

Claim: \( m_1 = m_2 = l - 1 \). Suppose \( m_1 < l - 1 \). Then, there exists a \( v \in r''_{i_2} \) such that \( v + \lambda \tau(r_k) \) intersects \( C_{m_1+1}, \ldots, C_l \) in this order. Since there exists a \( u \in s'_l \) such that \( u + \lambda \tau(r_k) \) intersects \( C_0, \ldots, C_j \) in this order, \( \tau(r_k) = -\tau(r'_{i_1}), m_1, k < l - 1, \) and \( C_{l-1} \cap C_l = \emptyset \) we come to a contradiction.

Let \( X \) be an arbitrary region in the plane. Let \( I(X) \) be the interval when projecting \( X \) onto \( c \). By Lemma 3.11 follows that \( r''_{i_1} \cap C_{l-1} \neq \emptyset \) and \( r''_{i_2} \cap C_{l-1} \neq \emptyset \). Therefore, \( I(r''_{i_1}) \cap I(C_{l-1}) \neq \emptyset \) and \( I(r''_{i_2}) \cap I(C_{l-1}) \neq \emptyset \). Furthermore, since \( r'_{i_1} \supseteq \mathcal{FE}(r''_{i_1},T_{l-1},T_k,c) \neq \emptyset \) and \( r''_{i_2} \supseteq \mathcal{FE}(r''_{i_2},T_{l-1},T_k,c) \neq \emptyset \) we get that \( I(r''_{i_1}) \cap I(C_l) \neq \emptyset \) and \( I(r''_{i_2}) \cap I(C_l) \neq \emptyset \). A necessary condition for \( r'_{i_1} \cap r''_{i_2} = \emptyset \) is that \( I(r''_{i_1}) \cap I(r''_{i_2}) = \emptyset \). This is only possible if \( I(r''_{i_1}) \subseteq I(C_{l-1}) \cap I(C_l) \) and \( I(r''_{i_2}) \subseteq I(C_{l-1}) \cap I(C_l) \). But then, there exists a \( v \in r''_{i_1} \cap C_{l-1} \) such that \( v + \lambda \tau(r_k) \) intersects \( C_{l-1}, C_l \) in this order. Since there exists a \( u \in s'_l \) such that \( u + \lambda \tau(r_k) \) intersects \( C_0, \ldots, C_j \) in this order, \( \tau(r_k) = -\tau(r'_{i_1}), k < l - 1, \) and \( C_{l-1} \cap C_l = \emptyset \) we come to a contradiction here.

\[ \text{Theorem 3.21} \quad \mathcal{R}\mathcal{R}^i(S, T_j, c) \text{ contains at most } 3j\lceil \frac{|c| - 1}{2|c|} \rceil \text{ disjoint reachable regions.} \]

\[ \text{Proof:} \quad \text{The number of reachable regions in } \mathcal{R}\mathcal{R}^{i+1}(S, T_j, c) \text{ is the sum of the reachable regions in:} \]

\[ (1) \quad \mathcal{R}\mathcal{R}(\mathcal{R}\mathcal{R}^i(S, T_j) \setminus \mathcal{R}\mathcal{R}^i(S, T_j, c), c), \]

\[ (2) \quad \bigcup_{k=1}^{j-1} \mathcal{FE}((\mathcal{R}\mathcal{R}^i(S, T_k) \setminus \mathcal{R}\mathcal{R}^i(S, T_k, c)), T_k, T_j, c), \]

\[ (3) \quad \mathcal{R}\mathcal{R}(\mathcal{R}\mathcal{R}^i(S, T_j, c), c), \]

\[ (4) \quad \bigcup_{k=1}^{j-1} \mathcal{FE}(\mathcal{R}\mathcal{R}^i(S, T_k, c), T_k, T_j, c). \]

\[ \text{(1) and (2): Lemma 3.17 and Corollary 3.18 yield that a maximum of } 3j\lceil \frac{|c| - 1}{2|c|} \rceil \text{ disjoint } \mathcal{R}\mathcal{R}^i \text{ reachable regions that were generated by the reachable regions in } \mathcal{R}\mathcal{R}^i(S, T_j) \setminus \mathcal{R}\mathcal{R}^i(S, T_j, c) \text{ and } \bigcup_{k=1}^{j-1} \mathcal{FE}((\mathcal{R}\mathcal{R}^i(S, T_k) \setminus \mathcal{R}\mathcal{R}^i(S, T_k, c)), T_k, T_j, c) \text{ can exist in } \mathcal{R}\mathcal{R}^{i+1}(S, T_j, c). \]

\[ \text{(3): Observation 3.19 yields that the number of reachable regions in } \mathcal{R}\mathcal{R}(\mathcal{R}\mathcal{R}^i(S, T_j, c), c) \text{ is at most as big as the number of reachable regions in } \mathcal{R}\mathcal{R}^i(S, T_j, c). \]

\[ \text{(4): Lemma 3.20 yields that the number of reachable regions in } \bigcup_{k=1}^{j-1} \mathcal{FE}(\mathcal{R}\mathcal{R}^i(S, T_k, c), T_k, T_j, c) \text{ is smaller or equal to the maximum number of } \mathcal{R}\mathcal{R}^i \text{ reachable regions in } \mathcal{R}\mathcal{R}^i(S, T_k, c) \text{ plus } j - k - 1. \]
3.3. C-ORIENTED LINE SIMPLIFICATION

By induction on \( j \) we show that the number of reachable regions in \( RR'(S,T_j,c) \) is bounded by \( 3j \left\lceil \frac{x}{2^\alpha_{\min}} \right\rceil (|C| - 1) \):

\[ j = 1: \text{Observation 3.19 yields that the number of reachable regions in } \]
\[ \text{RR}'(S,T_1,c) \text{ is equal to } 1 \leq 3 \left\lceil \frac{x}{2^\alpha_{\min}} \right\rceil (|C| - 1). \]

Induction precondition: Let the condition be true for \( 1, \ldots, j-1 \).

\[ j - 1 \rightarrow j: \text{The number of reachable regions in } \text{RR}'(S,T_j,c) \text{ is bounded by the number of reachable regions in } \]
\[ \text{RR}(RR'(S,T_j) \setminus \text{RR}'(S,T_j,c),c), \text{ in } \]
\[ \bigcup_{k=1}^{j-1} fE((RR'(S,T_k)) \setminus \text{RR}'(S,T_k,c)), T_k, T_j, c \text{ and the maximum number of } c-\text{oriented reachable regions in } \text{RR}'(S,T_k,c) \text{ plus } j - k - 1. \]

Applying the induction precondition we yield in total at most \( 3 \left[ \frac{x}{2^\alpha_{\min}} \right] (|C| - 1) + 3(j - 1) \left[ \frac{x}{2^\alpha_{\min}} \right] (|C| - 1) = 3 \left[ \frac{x}{2^\alpha_{\min}} \right] (|C| - 1) \) disjoint reachable regions.

\[ \square \]

The estimation of the number of reachable regions leads to the running time and space needed by the algorithm.

**Theorem 3.22** Let \( L_{\max} = \max j \geq 2, \ldots, n \left\lceil \frac{\log_{2^\alpha_{\min}} |C|}{2} \right\rceil + 4 \). The running time of the algorithm is bounded by \( O(n^2|C|^2) \left[ \frac{x}{2^\alpha_{\min}} \right] \log(n|C|) \left[ \frac{x}{2^\alpha_{\min}} \right] \).

Expressed in the minimum number of links \( k \) of \textsc{cols} the algorithm needs \( O(kn^2|C|^2 \left[ \frac{x}{2^\alpha_{\min}} \right] \log(n|C|) \left[ \frac{x}{2^\alpha_{\min}} \right]) \) time.

**Proof:** Let \( L_j = \left\lceil \frac{\log_{2^\alpha_{\min}} |C|}{2} \right\rceil + 4 \). Lemma 3.5 yields that at most \( L_j \) links are needed to reach each point in \( T_j \) from a starting set of points in \( C_j \). Therefore, \( L_{\max} \) corresponds to the maximum number of iterations needed to cover any \( T_j \) with reachable regions, \( j = 1, \ldots, n \). Theorem 3.21 yields that the number of reachable regions in \( RR'(S,T_j) \) is bounded by \( |C| \left[ \frac{x}{2^\alpha_{\min}} \right] (|C| - 1) \). For each reachable region in \( RR'(S,T_j) \), \( |C| \) new reachable regions are computed. We assume that the computation of a reachable region in a tube from a reachable region can be done in constant time. Each reachable region is inserted into a list structure. The insertion of all reachable regions into \( RR'(S,T_j) \) requires at most \( O(|C|^2 \left[ \frac{x}{2^\alpha_{\min}} \right] \log(n|C|) \left[ \frac{x}{2^\alpha_{\min}} \right]) \) time. The forward extension of reachable regions into tube \( T_j \) is processed in four sequencing iterations. Thus, at most \( 4|C| \left[ \frac{x}{2^\alpha_{\min}} \right] (|C| - 1) \) reachable regions are extended into \( T_j \). This requires time \( O(|C|^2 \left[ \frac{x}{2^\alpha_{\min}} \right] \log(n|C|) \left[ \frac{x}{2^\alpha_{\min}} \right]) \).

Altogether, we require at most \( O(nL_{\max} \log(n|C|) \left[ \frac{x}{2^\alpha_{\min}} \right] \log(n|C|) \left[ \frac{x}{2^\alpha_{\min}} \right]) + O(n^2|C|^2 \left[ \frac{x}{2^\alpha_{\min}} \right] \log(n|C|) \left[ \frac{x}{2^\alpha_{\min}} \right]) = O(n^2|C|^2 L_{\max} \left[ \frac{x}{2^\alpha_{\min}} \right] \log(n|C|) \left[ \frac{x}{2^\alpha_{\min}} \right]) \) time in the algorithm. Expressed in the minimum number of links \( k \) of \textsc{cols} the algorithm requires at most \( O(kn^2|C|^2 \left[ \frac{x}{2^\alpha_{\min}} \right] \log(n|C|) \left[ \frac{x}{2^\alpha_{\min}} \right]) \) time.
Corollary 3.23 The algorithm needs at most $O(n^2 |C|^2 L_{\text{max}} \left( \frac{\pi}{2\alpha_{\text{min}}} \right) )$ space. Expressed in the minimum number of links $k$ of COLS the space is bounded by $O(kn^2 |C|^2 \left( \frac{\pi}{2\alpha_{\text{min}}} \right) )$.

Proof: Each $RR'(S, T_j, e)$ needs $O(|C|^2 \left( \frac{\pi}{2\alpha_{\text{min}}} \right) )$ space. Thus, $RR'(S, T_j)$ needs $O(|C|^2 jL_{\text{max}} \left( \frac{\pi}{2\alpha_{\text{min}}} \right) )$ space and $\bigcup_{j=1}^{j_{\text{max}}} RR'(S, T_j)$ needs $O(|C|^2 jL_{\text{max}} \left( \frac{\pi}{2\alpha_{\text{min}}} \right) )$ space. Thus $\bigcup_{j=1}^{j_{\text{max}}} \bigcup_{j=1}^{j_{\text{max}}} RR'(S, T_j)$ needs $O(n^2 |C|^2 L_{\text{max}} \left( \frac{\pi}{2\alpha_{\text{min}}} \right) )$ space. Expressed in the minimum number of links $k$ of COLS we have: $\bigcup_{j=1}^{j_{\text{max}}} RR'(S, T_j)$ needs $O(n^2 |C|^2 \left( \frac{\pi}{2\alpha_{\text{min}}} \right) )$ space and therefore all reachable regions together need $O(kn^2 |C|^2 \left( \frac{\pi}{2\alpha_{\text{min}}} \right) )$ space.

3.3.5 Relation to the Fréchet Metric

In this section we note relationships to the Fréchet metric for curve similarity and line simplification (see Section 3.2). We show that a minimum $C$-oriented line simplification with Fréchet distance at most $\epsilon$ to $P$ has as many links as a minimum $C$-oriented line simplification has. Furthermore, we show how to obtain a minimum $C$-oriented line simplification with Fréchet distance at most $\epsilon$ to $P$ from the reachable regions computed in the algorithm. Figure 3.7 shows for each polygonal chain a minimum $C$-oriented line simplification that has Fréchet distance at most $\epsilon$. In the next definition we define the neighborhood of a point $q$ as the connected component of the intersection of $P$ with the $\epsilon$-circle around $q$.

Definition 3.24 (Neighborhood $N(q)$) Let $P = \{p_1, \ldots, p_n\}$ be a polygonal chain and $Q = \{q_1, \ldots, q_m\}$ be a $C$-oriented line approximation of $P$. Let $q$ be a point on $Q$, $q \in T_j$, $j$ minimal, and $1 \leq j \leq n - 1$.

$q \in T_j \setminus C_{j+1}$: Let $N(q) = \{x \in p_j p_{j+1} \text{ such that } ||q - x||_2 \leq \epsilon\}$.

$q \in C_{j+1}$: Let $N(q) = \{x \in p_j p_{j+1} \cup p_{j+1} p_{j+2} \text{ such that } ||q - x||_2 \leq \epsilon\}$.

Let $Q$ be a curve with Fréchet distance at most $\epsilon$ to $P$. From this definition follows that $P$ and $Q$ have monotone parameterizations $\alpha$ and $\beta$ such that $\alpha(t)$ is in the neighborhood of $\beta(t)$.
3.3. C-ORIENTED LINE SIMPLIFICATION

Observation 3.25

1. \( N(q) \) is a continuous subset of \( \overline{p_jp_{j+1}} \) or \( \langle p_j,p_{j+1},p_{j+2} \rangle \), respectively.

2. Let \( Q \) have Fréchet distance at most \( \epsilon \) to \( P \). Let \( \alpha \) and \( \beta \) be monotone parameterizations of \( P \) and \( Q \), such that \( ||\alpha(t) - \beta(t)||_2 \leq \epsilon \) for all \( t \in [0,1] \). If \( \beta(t) = q \) then \( \alpha(t) \in N(q) \), \( t \in [0,1] \).

We show that a \( C \)-oriented line approximation \( Q \) of \( P \) has Fréchet distance at most \( \epsilon \) to \( P \) if and only if the neighborhood of \( Q \) contains \( P \) and for any two points \( q' \) and \( q'' \) on \( Q \) where \( q' \) precedes \( q'' \) on \( Q \) the neighborhood of \( q'' \) does not precede the neighborhood of \( q' \) on \( P \).

**Lemma 3.26** Let \( P = \langle p_1, \ldots, p_n \rangle \) be a polygonal chain and \( Q = \langle q_1, \ldots, q_m \rangle \) be a \( C \)-oriented line approximation of \( P \). \( Q \) has Fréchet distance at most \( \epsilon \) to \( P \) if and only if

1. for all points \( p \) on \( P \) there exists a \( q \) on \( Q \) such that \( p \in N(q) \), and
2. for all points \( q' \prec q'' \) on \( Q \) we have that \( N(q'') \not\subset N(q') \).

**Proof:** 

"\( \Rightarrow \):" (1) is trivially fulfilled. (2) Assume there exists \( t < t', t,t' \in [0,1] \) with \( N(\beta(t')) \not\subset N(\beta(t)) \). Let \( \alpha(t) \) be the a monotone parameterization of \( P \). Since \( \alpha(t) \in N(\beta(t)) \) and \( \alpha(t') \in N(\beta(t')) \) it follows that \( \alpha(t'') \not\subset \alpha(t') \) - contradiction.

"\( \Leftarrow \):" Let \( \alpha' \) and \( \beta' \) be two arbitrary monotone parameterizations of \( P \) and \( Q \), respectively. Let \( F_e = \{(s,t) \in [0,1] \times [0,1] \mid ||\alpha'(s) - \beta'(t)||_2 \leq \epsilon \} \). From [AG95] Lemma 4 follows that \( P \) and \( Q \) have Fréchet distance at most \( \epsilon \) if a curve within \( F_e \) from \((0,0)\) to \((1,1)\) exists that is monotone in both coordinates. Since \( P \) and \( Q \) are piecewise linear we know that \( N(q) \cap N(q + \delta) = N(q) \). Additionally, since \( N(q) \neq \emptyset \) and for all points \( p \) on \( P \) there exists a \( q \) on \( Q \) such that \( p \in N(q) \), it follows that \( F_e \) is connected. Since for \( t' < t'', t',t'' \in [0,1] \) follows that \( N(\beta'(t'')) \not\subset N(\beta'(t')) \) and the existence of such a curve within \( F_e \).

Now, we show that any \( C \)-oriented polygonal chain with Fréchet distance at most \( \epsilon \) to \( P \) is a \( C \)-oriented approximation of \( P \).

**Theorem 3.27** Let \( P = \langle p_1, \ldots, p_n \rangle \) be a polygonal chain. Let \( Q = \langle q_1, \ldots, q_m \rangle \) be a polygonal chain with \( q_1 \in S, q_m \in E \) and \( \overline{q_iq_{i+1}} \in C \). Furthermore, let \( Q \) have Fréchet distance at most \( \epsilon \) to \( P \). Then, \( Q \) is a \( C \)-oriented approximation of \( P \).

**Proof:** Clearly \( q_i \in T, i = 1, \ldots, m \). We have to show that the links fulfill the requirements of Problem 3.1. Let \( \alpha \) and \( \beta \) be two monotone parameterizations of \( P \) and \( Q \) such that \( ||\alpha(t) - \beta(t)||_2 \leq \epsilon \) for all \( t \in [0,1] \). Let
CHAPTER 3. FROM GENERALIZATION TOWARDS
C-ORIENTED LINE SIMPLIFICATION

$q_i q_{i+1}$ be a link of $Q$.

Case $q_i \in T_j$ and $q_{i+1} \in T_j$: In this case $q_i q_{i+1}$ lies inside $T_j$.

Case $q_i \in T_j$ and $q_{i+1} \in T_j$, such that $l$ is maximal, $j$ is minimal, and $l < j$:

Then, there exists $t, t' \in [0, 1]$ with $\beta(t) = q_i$ and $\beta(t') = q_{i+1}$. Furthermore, $\alpha(t) \in N(\beta(t))$ and $\alpha(t') \in N(\beta(t'))$. Since $\alpha$ is a monotone parameterization of $P$ there exists $t_{i+1}, \ldots, t_j$ with $t \leq t_{i+1} < \cdots < t_j \leq t'$ and $\alpha(t_k) = p_k$, $k = l + 1, \ldots, j$. It follows that $\beta(t_k) \in C_k$, $k = l + 1, \ldots, j$. Thus, $q_i q_{i+1}$ intersects all $C_{i+1}, \ldots, C_j$ in this order.

Case $q_i \in T_j$ and $q_{i+1} \in T_j$, such that $l$ is maximal, $j$ is minimal, and $l < j$:

This can not happen since two consecutive $c$-circles have empty intersection and $Q$ is a Fréchet curve.

After these preparations we are ready to prove our main theorem of this section.

**Theorem 3.28** Let $P = (p_1, \ldots, p_n)$ be a polygonal chain and $Q = (q_1, \ldots, q_m)$ be a C-oriented line simplification of $P$. Then, there exists a C-oriented line simplification $Q'$ with Fréchet distance at most $\epsilon$ to $P$ that has as many links as $Q$ has.

**Proof:** From the Backward Extension Lemma 3.8 follows that there exists a C-oriented line simplification $Q$ of $P$ that contains no backward extensions. Let $Q$ be a C-oriented line simplification of $P$ without backward extensions with Fréchet distance greater than $\epsilon$ to $P$. We give a constructive proof that successively transforms $Q$ into a C-oriented line simplification $Q'$ with Fréchet distance at most $\epsilon$ to $P$ that has at most as many links as $Q$ has. From Lemma 3.26 follows that there exists a first $q' < q''$ on $Q$ with $N(q'') < N(q')$. Since $Q$ contains no backward extensions it follows that $q'$ and $q''$ are located in one tube $T_j$, $1 \leq j \leq n - 1$. Let $m$ be a point on $P$ with $N(q'') < m < N(q')$. Then $|q' - m| \geq \epsilon$ and $|q'' - m| \geq \epsilon$. Let $C''_m$ be the $c$-circle around $m$. Particularly, $m$ is a point on link $p_j p_{j+1}$. There exists a $u \in \partial C''_m \cap Q$ with $u < v$, for all $w \in \partial C''_m \cap Q$. Analogously, there exists a $v'' \in \partial C''_m \cap Q$ with $w \leq v''$, for all $w \in \partial C''_m \cap Q$. Then, $u$ is a point on $q_k q_{k+1}$ and $v''$ is a point on $q_k q_{k+1}$.

Case $l = 1$: Since $v''$ can be reached from $u$ inside $C''_m$ within two C-oriented links, there exists a $w \in C''_m$ with $\overrightarrow{w} \in C$ and $\overrightarrow{w} \in C$. Therefore, $Q' = \{q_1, \ldots, q_k, u, v, v'', q_{k+1}, \ldots, q_m\}$ is a C-oriented link path with at most as many links as $Q$ has. For all $q' < q''$ on $\{q_1, \ldots, u', v', v'', q_{k+1}\}$ we have $N(q'') \neq N(q')$.

Case $l = 2$: Then, $\|q_k - m\| \geq \epsilon$ and $\|q_{k+1} - m\| \geq \epsilon$. Let $\overrightarrow{q_k q_{k+1}} \cap C''_m = \overrightarrow{w}$ and $\overrightarrow{q_{k+1} q_{k+2}} \cap C''_m = \overrightarrow{w'}$ have orientation $c$ and $\epsilon$.
3.3. C-ORIENTED LINE SIMPLIFICATION

respectively. This configuration is very similar to that in Figure 3.13. We simply have to embed the circle in a tube and rename $p_i$ by $m_i$. Furthermore, analogously to the proof of Lemma 3.8 if follows that there exists a $z \in \mathbb{R}^2$ such that $z + \lambda \bar{c}$ has non empty intersection $z'$ with $w^2 \cup l'$. Thus, replacing $Q$ by $Q' = \langle q_1, \ldots, q_{k-1}, z', q_{k+2}, \ldots, q_m \rangle$ yields that $Q'$ has as many links as $Q$ and for all $q' \prec q''$ on $\langle q_1, \ldots, q_{k-1}, z', q_{k+2} \rangle$ we have $N(q'') \neq N(q')$.

Remember that we construct the $C$-oriented line simplification by following the links backwards until we reach the start set $S$. We can construct a $C$-oriented line simplification with Fréchet distance more that $\epsilon$ to $P$ by the following rule.

**Lemma 3.29** Following a link backwards until it hits a point in its generating reachable region for the first time yields an approximation with Fréchet distance at most $\epsilon$ to $P$.

**Proof:** Assume that the resulting path $Q$ has Fréchet distance more that $\epsilon$ to $P$. Then, we can apply the transformations of the proof of Theorem 3.28 to $Q$. In each transformation step a sequence of links $\langle q_{k-1}, \ldots, q_{k+i} \rangle$ is replaced by a sequence $\langle q_{k-1}, q'_1, \ldots, q'_{k+i-1}, q_{k+i} \rangle$, where $q'_{k+i-1}$ is a point on $q_{k+i-1}$. Let $q_{k+i-1}$ be reachable with $i$ links. From the transformation follows that $q'_{k+i-1}$ must also be reachable with $i$ links. Since $q'_{k+i-1} \prec q_{k+i-1}$ on $q_{k+i-1}$ and the construction of the approximation follows a link backwards until it hits a point in its generating reachable region for the first time, we yield a contradiction.

3.3.6 Conclusion

We presented an algorithm that solves the $C$-oriented line simplification problem with the additional restriction that the $C$-oriented polygonal chain has Fréchet distance at most $\epsilon$ to the original chain in polynomial time.

**Theorem 3.30** The $C$-oriented line simplification problem (COLS) with the additional restriction that the $C$-oriented polygonal chain has Fréchet distance at most $\epsilon$ to the original chain can be solved in $O(kn^2 |C|^2 \log n)$ time. The algorithm needs $O(kn^2 |C|^2 [\frac{n}{2\epsilon \min}])$ space ($k$ is the minimum number of links, $k \leq 4n + \sum_{i=1}^{n-1} \frac{\|p_i - p_{i+1}\|}{2\epsilon}$).

For sets of orientations $C$ that do not contain the orthogonal orientation to each orientation we have a negative result:
Corollary 3.31 In case that $\mathcal{C}$ does not contain the orthogonal orientation to each orientation, the minimum $\mathcal{C}$-oriented line simplification can have fewer links than the minimum $\mathcal{C}$-oriented approximation that has Fréchet distance at most $\epsilon$ to $P$.

Proof: Figure 3.20 shows a polygonal chain and its minimum $\mathcal{C}$-oriented line simplification that has Fréchet distance more than $\epsilon$ to $P$ since $N(q'') < N(q')$ on $P$, with $q' < q''$. Each $\mathcal{C}$-oriented approximation that has Fréchet distance at most $\epsilon$ to $P$ consists of at least 4 links. 

Figure 3.20: Example for Corollary 3.31.

In case that $\mathcal{C}$ does not contain the orthogonal orientation to each orientation our algorithm still computes a $\mathcal{C}$-oriented approximation but the resulting approximation might neither be minimal, nor have Fréchet distance $\epsilon$. In order to gain minimality it is necessary to allow backward extensions from a tube $T_j$ to $T_k$ with $k < j$. An open question is how to compute the minimum $\mathcal{C}$-oriented approximation that has Fréchet distance at most $\epsilon$ to $P$ in this case.

The restriction that the $\epsilon$-circles of two consecutive vertices of the original polygonal chain do not intersect is necessary when we want to solve this problem by computing the sets $RR'(S, T_j)$ in polynomial time. There exist examples where the number of disjoint reachable regions in $RR'(S, T_j)$ gets exponential in $j$ when allowing $C_i \cap C_{i+1} \neq \emptyset$. Since the profit of deciphering such an example is rather small we decided not to expose them.

The complexity of the $\mathcal{C}$-oriented line simplification problem when allowing $C_j \cap C_{j+1} \neq \emptyset$ remains open.
Chapter 4

Labeling Downtown

4.1 Introduction

When visualizing information it is often essential to display text by graphical objects. This means text labels have to be associated with graphical features. Until now, the placement of labels is primarily performed manually, especially in map production. In the area of Cartography, Geographic Information Systems, and Graph Drawing map labeling usually has to be performed efficiently. Therefore, it is highly desirable to use automatic map labeling algorithms. The ACM Computational Geometry Task Force [For96] has identified label placement as an important area of research in Computational Geometry.

We distinguish between three kinds of graphical features according to their dimension.

**Point Features**  Cities, summits, area features on small scale maps and vertices of graphs or diagrams.

**Line Features**  Rivers, boarders, streets, straight edges, polygonal lines and edges or arcs of graphs or diagrams.

**Area Features**  Mountains, islands, countries, and lakes.

Point and line feature labels are arranged next to the object and area feature labels are usually placed within the boundary of the feature to be labeled.
Extensive effort have been spent by cartographers like Imhof [Imh62, Imh75] and Yoeli [Yoe72] to devise rules that measure the semantic clarity of a labeling assignment. We state three concepts that are widely accepted as the basic rules for accurate map labeling.

**Readability**  The labels are of legible size.

**Unambiguity**  Each label can be easily identified with exactly one graphical feature of the layout.

**Avoidance of Overlaps**  Labels should not overlap with other labels or other graphical features of the layout.

We denote the possible label positions of a feature as its *label candidates*. Sometimes, a *cost* is assigned to an individual label candidate which reflects the quality of this label in terms of unambiguity, overlap with graphical features, and preferences between the label candidates.

How features are labeled depends on the specific labeling model. The most important models are:

**Fixed Position Model**  Each feature has a finite set of label candidates. For point labeling, typical examples are the 2- or 4-position model as shown in Figure 4.1.

**Fixed Position Model with Scalable Labels**  Each feature has a finite set of label candidates, where the size of all labels can be scaled.

**Slider Model**  Each feature has a fixed label that can be placed at any position that touches the feature. Figure 4.2 shows the 1-, 2-, and 4-slider model for point features, where the labels can be shifted continuously as indicated by the arrows.

![Figure 4.1: Fixed position models](http://lilinwww.ira.uka.de/bibliography/Theory/map.labeling.html)  
![Figure 4.2: Slider models.](http://lilinwww.ira.uka.de/bibliography/Theory/map.labeling.html)

In the last ten years, the amount of research in automatic map making has increased significantly, as the number of published articles illustrates\(^1\). Although most variants of map labeling are \(NP\)-complete.

\(^1\)http://lilinwww.ira.uka.de/bibliography/Theory/map.labeling.html
4.2 Labeling Downtown

The downtown labeling problem is a special case of a map labeling problem. The downtown labeling problem was recently shown to be \( \text{AP-complete} \) by Unger and Seibert [US00].

One clear way to model it, is to abstract a grid-shaped downtown street pattern into a chessboard of adequate size. We abstract the names to be placed along their streets as tiles that w.l.o.g. span an integer number of fields.

Now a feasible labeling is a conflict free tiling of the board placing all the labels along their streets.

Our main algorithm is a kind of an adaptive backtracking algorithm that is guaranteed to find a solution if there is one. We tested our algorithm on over 10,000 randomly generated instances. Surprisingly it has an empirically strictly bounded depth of backtracking, namely one. Under the assumption that the algorithm has backtracking depth one, the algorithm has polynomial runtime, which makes it an empirically efficient algorithm.
Given this experience this makes the theoretical analysis of our algorithm even more interesting. For the sake of completeness, we sketch the \(\mathcal{NP}\)-completeness proof of Unger and Seibert.

There is a well studied family of related problems from Discrete Tomography [Woc96, GGS95] that yields a \(\mathcal{NP}\)-hardness result for a slightly different labeling problem, taking place on a cylinder instead of a rectangle.

We round up this chapter by giving efficient solutions to special cases: There is a polynomial algorithm, if

- no label is longer than half of its street length
- all vertical labels are of equal length
- the map is quadratic and each label has one of two label lengths

One general remark that helps to suppress a lot of formal overhead: We often only discuss the case of horizontal labels or row labels and avoid the symmetric cases of vertical labels and columns or vice versa.

### 4.2.1 Problem Definition

Let \(G\) be a grid consisting of \(n\) rows and \(m\) columns. Let \(R = \{R_1, \ldots, R_n\}\) and \(C = \{C_1, \ldots, C_m\}\) be two sets of labels. The problem is to label the \(i^{th}\) row of \(G\) with \(R_i\) and the \(j^{th}\) column of \(G\) with \(C_j\) such that no two labels overlap. We will represent the grid \(G\) by a matrix.

**Definition 4.1 (Label Problem \((G, R, C, n, m)\))**

**Instance:** Let \(G_{n,m}\) be a two dimensional array of size \(n \times m\). \(G_{i,j} \in \{\emptyset, r, c\}\). Let \(R_i\) be the label of the \(i^{th}\) row and let \(r_i\) be the length of label \(R_i\), \(1 \leq i \leq n\). Let \(C_i\) be the label of the \(i^{th}\) column and let \(c_i\) be the length of label \(C_i\).

**Problem:** For each row \(i\) set \(r \) consecutive fields of \(G_{i,\cdot}\) to \(r\) and for each column \(j\) set \(c\) consecutive fields of \(G_{\cdot,j}\) to \(c\).

Of course no label can be longer than the length of the row or column, respectively.

Initially, we set \(G_{i,j} = \emptyset\) which denotes that the field is not yet set. Let \([a, b]\) be a half open interval such that \(G_{i,x} \in \{\emptyset, r\}\), for \(x \in [a, b]\). We say that \(G_{i,[a,b]}\) is free for row labeling. Furthermore, this interval has length \(b-a\). We also say that \(G_{i,\cdot}\) contains two disjoint intervals of length at least \([\frac{b-a}{2}]\) that are free for row labeling, namely \([a, a+\lfloor\frac{b-a}{2}\rfloor]\) and \([a+\lfloor\frac{b-a}{2}\rfloor, b]\).
4.2. LABELING DOWNTOWN

4.2.2 General Rules

Assume we have a label with length longer than half of its street length. No matter how we position the label on its street, there are some central fields in the street that are always occupied by this label. We therefore can simply mark these fields occupied. It is easy to see that these occupied fields can produce more occupied fields. The following rules check whether there is sufficiently large space for each label and determines occupied fields.

**Rule 4.2 (Conflict)** Let $l = [a, b]$ be the longest interval of row $i$ that is free for row labeling. If $r_i > b - a$, then row $i$ can not be labeled, since it does not contain enough free space for row labeling. In this case we say that a conflict occurred and it follows that the instance is not solvable.

**Rule 4.3 (Large Labels)** Let $l = [a, b]$ be the only interval in $G_i$, that is free for feasible row labeling. Observe that the fields that are occupied simultaneously when $R_i$ is positioned leftmost and rightmost in $I$ have to be occupied by $R_i$ no matter where it is placed. These fields we set to $r$ and call them preoccupied.

---

**Procedure 3** Preprocessing $(G, R, C, n, m)$

1) repeat {
2) $G' := G$;
3) run Rule 4.3 on $(G, R, C, n, m)$ and on $(G^T, C, R, m, n)$;
4) if Rule 4.2 yields a conflict on $(G, R, C, n, m)$ or on $(G^T, C, R, m, n)$ then
5) return "conflict";
6) } until $(G = G')$;
7) return true;

---

Our Preprocessing procedure iteratively executes the Rules 4.2 and 4.3 until none of them yields a further change to the label problem or a conflict occurs. In the latter case we have that the instance is not solvable. We will spell out special cases where the successful preprocessing implies solvability. Furthermore, the preprocessing underlies the following considerations.

For each unfixed label that is limited to just one interval of at most twice its length or to two intervals of exactly its length we can check whether these labels can be simultaneously positioned without conflicts. This can be done since all possible label positions of these rows and columns can be encoded.
in a set of 2SAT clauses, the satisfaction of which enforces the existence of
a conflict free label positioning of these labels. On the other hand a conflict
free label positioning of these labels implies a satisfying truth assignment
to the set of clauses. Even, Itai, and Shamir [EIS76] proposed a polynomial
time algorithm that solves the 2SAT problem in time linear in the number
of clauses and variables.

We therefore represent each matrix field \( G_{i,j} \) by two boolean variables. We
have the boolean variable \( G_{i,j} = r \) and its negation \( \overline{G_{i,j}} = \overline{r} \) which means
\( G_{i,j} \neq r \) or \( G_{i,j} \in \{ \emptyset, c \} \). As the second variable we have \( G_{i,j} = c \) and its
negation \( \overline{G_{i,j}} = \overline{c} \) which means \( G_{i,j} \neq c \) or \( G_{i,j} \in \{ \emptyset, r \} \). Of course these
two variables are coupled by the relation \( (G_{ij} = r) \rightarrow (\overline{G_{ij}} = \overline{c}) \).

Those rows and columns, where the possible label positions are limited to
just one interval of at most twice its length or to two intervals of exactly
its length, we call dense. We now encode all possible label positions of the
dense rows and columns in a set of 2SAT clauses the satisfaction of which
yields a valid labeling of these rows and columns and vice versa.

**Property 4.4 (Density Property I)** Let \( G_{i,} \) be a row that contains
exactly two maximal intervals each of length \( r_i \) that are disjoint and free
for feasible row labeling. Let these intervals be \([a, b]\) and \([c, d]\), \(1 \leq a < b < c < d \leq n + 1\). Then, a valid labeling exists if and only if the conditions

1. \( (G_{i,a} = r) \leftrightarrow (G_{i,a+1} = r) \leftrightarrow (G_{i,a+2} = r) \leftrightarrow \cdots \leftrightarrow (G_{i,b-1} = r) \),
2. \( (G_{i,c} = r) \leftrightarrow (G_{i,c+1} = r) \leftrightarrow (G_{i,c+2} = r) \leftrightarrow \cdots \leftrightarrow (G_{i,d-1} = r) \),
3. \( (G_{i,a} = r) \rightarrow (G_{i,c} = r) \)

are fulfilled.

\( (G_{i,a} = r) \leftrightarrow (G_{i,a+1} = r) \) can be written as the 2SAT clauses
\( (G_{i,a} = r \lor G_{i,a+1} = r) \), \( (G_{i,a} = r \lor G_{i,a+1} = \overline{r}) \) and since the condition
\( (G_{i,a} = r) \leftrightarrow (G_{i,c} = r) \) can be written as \( (G_{i,a} = r \lor G_{i,c} = r) \), \( (G_{i,a} = r \lor G_{i,c} = \overline{r}) \), it is easy to see that the complete Density Property 4.4
can be written as a set of 2SAT clauses. The feasible label placements are
\( (G_{i,a} = r, \ldots, G_{i,b-1} = r) \) and \( (G_{i,c} = r, \ldots, G_{i,d-1} = r) \).

**Property 4.5 (Density Property II)** Let \( G_{i,} \) be a row that contains
only one maximal interval \([a, b]\) that is free for feasible row labeling,
\( r_i < b - a \leq 2r_i \). Then, a valid labeling for the row exists if and only
if the conditions
4.2. LABELING DOWNTOWN 

1. \((G_{i,a} = r) \rightarrow (G_{i,a+1} = r) \rightarrow (G_{i,a+2} = r) \rightarrow \cdots \rightarrow (G_{i,b-r_i-1} = r)\),

2. \(G_{i,b-r_i} = r, \ldots, G_{i,a+r_i-1} = r\), and

3. \((G_{i,a} = r) \rightarrow (G_{i,a+r_i} = r), (G_{i,a+1} = r) \rightarrow (G_{i,a+1+r_i} = r), \ldots,
   \quad (G_{i,b-r_i-1} = r) \rightarrow (G_{i,b-1} = r)\)

are fulfilled.

Analogously to the first Density Property, the conditions of the second Density Property can be formulated as a set of 2SAT clauses. All feasible label placements are \((G_{i,a} = r, G_{i,a+1} = r, \ldots, G_{a+r_i-1} = r)\), \((G_{i,a+1} = r, G_{i,a+2} = r, \ldots, G_{a+2+r_i} = r)\), \((G_{i,a+2} = r, G_{i,a+3} = r, \ldots, G_{a+3+r_i} = r)\), \ldots, \((G_{i,b-r_i} = r, G_{i,b-r_i+1} = r, \ldots, G_{i,b-1} = r)\). Note that the properties work analogously for columns.

**Theorem 4.6** The 2SAT formula of all dense rows and columns can be created in \(O(nm)\) time. The 2SAT instance can be solved in \(O(nm)\) time.

**Proof:** The number of variables is limited by \(2nm\). For each dense row we have at most \(\frac{3}{2}n\) clauses. Analogously, for each dense column we have at most \(\frac{3}{2}m\) clauses. Altogether we have \(O(nm)\) clauses. Thus, the satisfiability of the 2SAT instance can be checked in \(O(nm)\) time [EIS76].

Procedure **DrawConclusions** calls procedure **Preprocessing**. In case of success, all dense rows and columns are encoded as a set of 2SAT clauses with the aid of Density Property 4.4 and 4.5. Then, their solvability is checked e.g. by invoking the 2SAT algorithm of Even, Itai, and Shamir [EIS76].

**Procedure 4** **DrawConclusions** \([G, R, C, n, m]\)

1) if **Preprocessing** \((G, R, C, n, m)\) then

2) \(F := \) the set of 2SAT clauses of the dense rows and column;

3) if \(F\) is satisfiable then return true;

4) return false;

**Lemma 4.7** Procedure **Preprocessing** and procedure **DrawConclusions** can be implemented to use at most \(O(m(n+m))\) time.
Proof: The rules only need to be applied to those rows and columns in which an entry was previously set to $r$ or $c$. A setting of a field $G_{i,j}$ can only cause new settings in row $i$ or column $j$, which by themselves can again cause new settings. The application of the rules on a row and a column takes $O(n + m)$ time. Since at most $2nm$ fields can be set we have that the preprocessing can be implemented such that its running time is $O(nm(n + m))$ time. In Theorem 4.6 we proved that the 2SAT clauses can be generated and checked for solvability in $O(nm)$ time. Thus, in total we need at most $O(nm(n + m))$ time. 

Thus, we can solve dense problems:

**Theorem 4.8** In case that each row and each column of a preprocessed labeling instance $(G, R, C, n, m)$ either fulfills the Density Property 4.4 or 4.5, procedure DrawConclusions decides if the instance is solvable. In case of solvability we can generate a valid labeling from a truth assignment. The overall running time is bounded by $O(nm(n + m))$ time.

### 4.2.3 A General Algorithm

In this section we describe an algorithmic approach with a backtracking component that solves any label problem. Empirically it uses its backtracking ability in a strictly limited way such that its practical runtime stays in the polynomial range. After performing the Preprocessing and satisfiability test for dense rows and columns (see procedure DrawConclusions), we adaptively generate a tree that encodes all possible label settings of the label problem. Each node in the first level of the search tree corresponds to a possible label setting for the first row label. In the $i^{th}$ level the nodes correspond to the possible label settings for the $i^{th}$ row, depending on the label settings of all predecessor rows. Thus, we have at most $m$ possible positions for a row label and at most $n$ levels. Our algorithm searches for a valid label setting in this tree by traversing the tree, depth-first, generating the children of a node when necessary.

In the algorithm, we preprocess the matrix $G$ and check the solvability of the dense rows and columns by invoking procedure DrawConclusions. We further mark all settings of this procedure permanently. When we branch on a possible label setting for a row, we increase the global timestamp, draw all conclusions this setting has for the other labels by invoking procedure DrawConclusions, and timestamp each new setting. These consequences can be a limitation on the possible positions of a label or even the impossibility of positioning a label without conflicts. After
that, we select one of the newly generated children, increase the timestamp and again timestamp all implications. When a conflict occurs, the process resumes from the deepest of all nodes left behind, namely, from the nearest decision point with unexplored alternatives. We mark all timestamps invalid that correspond to nodes that lie on a deeper level than the decision point. This brings the matrix $G$ into its previous state without storing each state separately. Let the algorithm return a valid label setting for all rows. Since procedure \texttt{Preprocessing} ensures that each column $i$ contains an interval of length at least $c_i$ that is free for column labeling we can simply label each column and yield a valid label setting. The algorithm is given in the Label Algorithm 5, in procedure \texttt{Preprocessing}, in procedure \texttt{DrawConclusions}, and in procedure \texttt{PositionAndBacktrack}.

\begin{algorithm}
\begin{algorithmic}
\State \textit{timestamp} := $1$;
\State \textbf{if} \texttt{DrawConclusions}(G,R,C,n,m) \textit{yields a conflict}
\State \hspace{1em} \textbf{return} \textit{"not solvable"};
\State \textbf{if} \textit{timestamp} each setting:
\State \hspace{1em} \textbf{let} $w$ \textbf{be the first row that is not yet labeled};
\State \textbf{if} \texttt{PositionAndBacktrack}(w,G,R,C,n,m,timestamp) \{ \\
\hspace{1em} \textbf{label} all columns that are not yet completely labeled;
\State \hspace{1em} \textbf{return} $G$; \\
\} \\
\textbf{else}
\State \hspace{1em} \textbf{return} \textit{"not solvable"};
\end{algorithmic}
\end{algorithm}

We implemented the backtracking algorithm and tested it on over 10000 randomly generated labeling instances with $n$ and $m$ at most 100. After at most one backtracking step per branch the solvability of any instance was decided. The algorithm is constructive: for each solvable instance a labeling was produced. This makes it reasonable to study the worst case run time behavior of the algorithm under the assumption that the backtracking depth is always at most one. The algorithm behaves in the worst way when each label is positioned first in all places that cause a conflict, before it is positioned in a conflict free place. A row label can be positioned in at most $m$ different places. Each time when a label is positioned procedure \texttt{DrawConclusions} is called, which needs at most $O(nm(n+m))$ time. Thus, the time for positioning a row label is bounded by $O(nm^2(n+m))$ time. Since $n$ rows have to be labeled the backtracking approach with backtracking depth one needs at most $O(n^2m^2(n+m))$
time. If the assumption of limited backtracking behavior does not hold the runtime is exponential.
4.2.4 Complexity Status

In this section we first show the $\mathcal{NP}$-completeness of the labeling of a cylinder-shaped downtown. This problem is a slight generalization of the labeling problem: A cylinder labeling instance in which one column label has full length is a downtown labeling instance. Recently, Unger and Seibert showed the $\mathcal{NP}$-completeness of the labeling problem. For the sake of making the presentation self-contained, we sketch the $\mathcal{NP}$-completeness proof for the labeling problem from Unger and Seibert. The reason why we give the proof for the cylinder labeling problem is that our reduction could be helpful in understanding the complexity of the original problem.

4.2.4.1 Complexity of the Cylinder Label Problem

Instead of labeling an array we now label a cylinder consisting of $n$ cyclic rows and $m$ columns. Figure 4.4 shows an example of a cylinder instance. We show that this problem is $\mathcal{NP}$-complete by reducing a version of the Three Partition problem to it. Our proof is similar to an $\mathcal{NP}$-completeness proof of Woeginger [Woe96] about the reconstruction of polyominoes from their orthogonal projections. Woeginger showed that the reconstruction of a two-dimensional pattern from its two orthogonal projections $H$ and $V$ is $\mathcal{NP}$-complete when the pattern has to be horizontally and vertically convex. This and further related problems, also discussed in [Woe96] show up in the area of discrete tomography.

**Definition 4.9 (Cylinder Label Problem ($Z$, $R$, $C$, $n$, $m$))**

**Instance:** Let $Z_{n,m}$ be a cylinder consisting of $n$ cyclic rows and $m$ columns. Let $R_i$ be the label of the $i^{th}$ row and let $r_i$ be the length of label $R_i$, $1 \leq i \leq n$. Let $C_j$ be the label of the $j^{th}$ column and let $c_j$ be the length of label $C_j$, $1 \leq j \leq m$. 
Problem: For each row $i$ set $r_i$ consecutive fields of $Z_{i,:}$ to $r$, for each column $j$ set $c_j$ consecutive fields of $Z_{:,j}$ to $r$.

Our reduction is done from the following version of the $\mathcal{NP}$-complete Three Partition problem [GJ79].

Problem 4.10 (Three Partition)

Instance: Positive integers $a_1, \ldots, a_{3k}$ that are encoded in unary and that fulfill the two conditions

(i) $\sum_{i=1}^{3k} a_i = k(2B + 1)$ for some integer $B$, and
(ii) $(2B + 1)/4 < a_i < (2B + 1)/2$ for $1 \leq i \leq 3k$.

Problem: Does there exist a partition of $a_1, \ldots, a_{3k}$ into $k$ triples such that the elements of every triple add up to exactly $2B + 1$?

Theorem 4.11 The Cylinder Label problem is $\mathcal{NP}$-complete.

Proof: Cylinder Label problem $\in \mathcal{NP}$: The Cylinder Label problem is in $\mathcal{NP}$ since it is easy to check whether a given solution solves the problem or not.

Transformation: Now let an instance of Tree Partition be given. From this instance we construct a Cylinder Label problem consisting of $n = k(2B + 2)$ rows and $m = 3k$ columns. The vector $r$ defining the row label length is of the form:

\[(m, m - 1, \ldots, m - 1, m - 1, \ldots, m - 1)\]

\[(2B + 1)-times \quad \quad \quad (2B + 1)-times\]

Since a row label of length $m$ occupies the whole row, those rows with label length $m$ have no free space for column labeling. Therefore the rows with label length $m$ subdivide the rows in $k$ blocks, each containing $2B + 1$ rows each of which has one entry that is free for column labeling when the row is labeled. The vector defining the column label length is of the form:

\[(a_1, a_2, \ldots, a_{3k})\]

The transformation clearly is polynomial.

The Tree Partition instance has a solution $\Leftrightarrow$ the Cylinder Label instance has a solution:
Let \((x_1, y_1, z_1), \ldots, (x_k, y_k, z_k)\) be a partition of \(a_1, \ldots, a_{3k}\) into \(k\) triples such that \(x_i + y_i + z_i = 2B + 1\), \(1 \leq i \leq k\). For each \(i\), \((x_i, y_i, z_i) = (a_{ij}, a_{ig}, a_{ih})\), for some indices \(j, g, \) and \(h\), \(1 \leq i, j, g, h \leq 3k\). We now label the columns \(j\), \(g\), and \(h\) among themselves in the \(i\)-th block of rows. More precisely, in column \(f\) we label the fields \(Z_{f, (i-1)(2B+2) + j} = c\), \(Z_{f, (i-1)(2B+2) + g} = c\), \(Z_{f, (i-1)(2B+2) + h} = c\). In column \(g\) we label the fields \(Z_{g, (i-1)(2B+2) + j} = c\), \(Z_{g, (i-1)(2B+2) + g} = c\), \(Z_{g, (i-1)(2B+2) + h} = c\). In column \(h\) we label the fields \(Z_{h, (i-1)(2B+2) + j} = c\), \(Z_{h, (i-1)(2B+2) + g} = c\), \(Z_{h, (i-1)(2B+2) + h} = c\). It then follows that the rows \(j(2B+2) + 1\) are free for row labeling, for \(0 \leq j \leq k\). Thus, we can label them with their labels of length \(3k + m\). All other rows have exactly one entry occupied by a column label. Since the rows are cyclic we can label each of these rows with a label of length \(3k - 1\).

Let \(Z\) be a solution of the Cylinder Label instance. Each row contains at most one entry that is occupied by a column label. Each column label \(a_i\) has length \((2B+1)/4 < a_i < (2B+1)/2\), \(1 \leq i \leq 3k\). Therefore, exactly three columns are label in the rows \(j(2B+2) + 2, \ldots, (j+1)(2B+2)\), for \(0 \leq j \leq k - 1\). Furthermore the label length of each triple sums to \(3k\) and thus partitions \(a_1, \ldots, a_{3k}\) into \(k\) triples. Thus solves the Three Partition instance.

### 4.2.4.2 Complexity of the Downtown Labeling Problem

Unger and Seibert give a reduction from 3SAT to the downtown labeling problem. The proof needs a lot of notation, is technically involved, and 10 pages long. In order to make this presentation self-contained we sketch the idea of the proof; for details we refer to [US00].

For a given 3SAT formula we construct a map \(M_\phi\) so that \(M_\phi\) can be labeled if and only if the corresponding 3SAT formula is satisfiable. A 3SAT formula \(\phi\) consists of \(t\) clauses \(c_1, \ldots, c_t\) over \(m\) variables \(x_1, \ldots, x_m\). Each clause \(c_i\) is of the form \(z_{i,1} \lor z_{i,2} \lor z_{i,3}\), the \(z_{i,j}\) being from \(\{x_1, \ldots, x_m\} \cup \{\overline{x_1}, \ldots, \overline{x_m}\}\). For such a given instance of 3SAT, the authors construct a map \(M_\phi\) consisting of \(N_c = 12n\) rows and less than \(N_\phi = 36n^2\) columns, for \(n = 3l + m\). The map is split into several vertical stripes. Each stripe is limited to the left and to the right by a vertical label of full length \(N_c\). The map consists of three groups of vertical stripes (see Figure 4.5). There is a pair of vertical stripes for each variable in the left part of the map. Furthermore, for each negative occurrence of a variable
in a clause, there is a stripe in the middle part of the map. Furthermore, for each clause there is a triple of stripes.

<table>
<thead>
<tr>
<th>Variable part</th>
<th>Middle (Mirror) part</th>
<th>Clauses part</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 ) ( x_2 ) ( \ldots ) ( x_k ) ( \ldots ) ( \bar{x}<em>1 ) ( \bar{x}<em>2 ) ( \bar{x}<em>k ) ( \ldots ) ( z</em>{i,j} ) ( z</em>{i,j} ) ( \ldots ) ( z</em>{1,1} ) ( z_{1,2} ) ( z_{1,3} ) ( \ldots )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.5: The outline of the map.

The set of rows also consists of different groups. The first \( k \) rows and the last \( k \) rows are the variable setting rows. These rows reflect the setting of a variable. The parameter \( k \) will be specified below. Right below the upper variable setting rows, also in the upper half of the map, are the mirror rows. These rows mirror negative occurrences of variables as we will describe below. In the lower half of the map, above the lower variable setting rows, are the clause rows located.

The Variable Part

Let \( s_i \) be the number of occurrences of literal \( x_i \) or \( \bar{x}_i \). The stripes of the variable part have the following form: The stripes for variable \( x_1 \) are at least \( s_1 + 1 \) columns wider than the stripes for variable \( x_2 \); the stripes for variable \( x_1 \) are at least \( s_2 + 2 \) columns wider than the stripes for variable \( x_3 \); and so on. Thus, for each variable, the pair of stripes has a different size.

The rows in the variable setting part are organized as follows. The first \( s_1 + 1 \) rows and the last \( s_1 + 1 \) rows together with the variable stripes of \( x_1 \) encode the variable setting of \( x_1 \). The neighboring \( s_2 + 1 \) rows together with the important role for the stripes of \( x_2 \), and so on. Indeed the label positions in these rows reflect the variable setting.
The left stripe of the pair of variable stripes for \( x_i \) is \( s_i + 1 \) columns wider than the right stripe of \( x_j \). In the middle of each stripe for a variable \( x_i \) there is a column label defined which has size \( N_e \) minus the row number of the first important row for \( x_i \). Thus, for variable \( x_1 \) the column length is \( N_e - s_1 \); for \( x_2 \) the column length is \( N_e - s_1 - s_2 \); and so on.

Let \( r_{11}, \ldots, r_{8i+1} \) and \( r_{8i+1}, \ldots, r_{8j+1} \) be the by row number sorted set of rows that encode the variable setting of \( x_i \). Then, the rows \( r_{8i+1} \) and \( r_{8i+2} \) have labels of length equal to the width of the right stripe of this variable. This length we call \( l_e \). Let \( z_{j,k} \) be the \( l \)-th occurrence of variable \( x_j \) in \( \phi \). In case that \( z_{j,k} \) is a positive occurrence, then row \( r_{8i+1} \) has a horizontal label of length \( l_e + l \). Otherwise, row \( r_{8i+2} + l \) has a horizontal label of length \( l_e + l \). As a consequence, in a setting as shown in Figure 4.6 a), all labels in the rows that encode the variable setting of variable \( x_i \) and that are in the upper half of the map can be positioned. Furthermore, the uppermost label in the lower half of the map out of the rows that encode the variable setting of variable \( x_i \) can be positioned. This setting corresponds to a truth setting of the variable. The setting in Figure 4.6 b) corresponds to a false setting. Here, all labels in the rows that encode the variable setting of variable \( x_i \) and that are in the lower half of the map can be positioned and the lowermost label in the upper half of the map out of the rows that encode the variable setting of variable \( x_i \) can be positioned.

**The Middle Part**

The aim of the middle part is to mirror the horizontal labels representing the negative occurrences of variables into a new part of the map, which is unused by the variables so far. This new part is located in the upper half of the rows, right below the important horizontal labels (see Figure 4.5). For each occurrence of a variable \( z_{i,j} = \bar{x}_u \) there is a stripe which has exactly the width of the horizontal label that is to be mirrored. For each stripe there is a horizontal label in the mirror part. In this stripe there are
two vertical labels of unequal length defined. If \( x_u = 0 \), then the lower important horizontal label can be positioned in the left variable stripe and the mirror horizontal label can be positioned in the mirror stripe, together with the vertical labels. Otherwise, if \( x_u = 1 \), the lower important horizontal label must be positioned in the mirror stripe. Then, the vertical labels are positioned uppermost, which makes it impossible to position the horizontal mirror label, too. This is illustrated in Figure 4.7.

Figure 4.7: Mirror negative occurrences of variables in the upper half of the map.

The Clause Part
For a clause \( c_i = z_{i,1} \lor z_{i,2} \lor z_{i,3} \) three vertical stripes are constructed. If the literal \( z_{i,j} \) corresponds to a positive occurrence of a variable, then its stripe has a width equal to its horizontal label length in variable setting part. Otherwise, its stripe has the width of its horizontal mirror label. In each stripe there are two vertical column labels of non equal length. Furthermore, there is a horizontal label in the clause part of the rows for each triple of stripes. The vertical label lengths are determined such that it is possible to place either the corresponding horizontal label of the upper variable or mirror part, or the horizontal clause label next to the two vertical labels. This is illustrated in Figure 4.8. Therefore, the clause label can only be set (that is, the clause can only be fulfilled) if and only if at least one stripe of the three stripes is not occupied with a horizontal variable label or horizontal mirror label.

From the construction then follows the existence of a conflict free labeling if and only if the 3SAT formula is satisfiable.
4.2. LABELING DOWNTOWN

Fig. 4.8: Clause stripes: The clause label can only be positioned, if at least one literal is fulfilled, that is, the literal label is positioned in a variable stripe.

4.2.5 Solvable Special Cases

In the following section we derive an $O(nm)$ time algorithm for the special case where no label is longer than half of its street length. We think that this case in particular applies to large downtown maps, where the label length is short in respect to the street length. In Section 4.2.5.2 we solve the label problem for the case where each vertical label is of equal length. In many American cities the streets in one orientation (e.g., north-south) are simply called 1-st Avenue, 2-nd Avenue, .... These names have all the same label length and thus the label problem can be solved with the algorithm in Section 4.2.5.2. In Section 4.2.5.3 we give an algorithm for quadratic maps, where each label has one of two label lengths. The algorithm for the latter two cases have runtime $O(nm(n + m))$ time.

4.2.5.1 Half Size

In this section we study the case where each row label has length at most $\left\lfloor \frac{n}{2} \right\rfloor$ and each column label has length at most $\left\lfloor \frac{m}{2} \right\rfloor$, where $n$ is again the number of rows and $m$ is the number of columns. We show that the label problem is solvable in this case.

**Theorem 4.12** Let $(G, R, C, n, m)$ be a label problem. Let $r_i \leq \left\lfloor \frac{n}{2} \right\rfloor$ and let $c_i \leq \left\lfloor \frac{m}{2} \right\rfloor$. Then, algorithm HALF SOLUTION computes a solution to Problem 4.1 in $O(nm)$ time.

**Proof:** Take a look at Figure 4.9.
Algorithm 6 \( \text{HALF SOLUTION} \{G, R, C, n, m\} \)

1) label the rows \( 1, \ldots, \left\lceil \frac{n}{2} \right\rceil \) leftmost;
2) label the rows \( \left\lceil \frac{n}{2} \right\rceil + 1, \ldots, n \) rightmost;
3) label the columns \( 1, \ldots, \left\lceil \frac{m}{2} \right\rceil \) bottommost;
4) label the columns \( \left\lceil \frac{m}{2} \right\rceil + 1, \ldots, m \) topmost;

---

Figure 4.9: Solution of a typical half size label problem according to algorithm \( \text{HALF SOLUTION} \).

---

Figure 4.10: Typical downtown map where the vertical street names have constant length.
4.2. LABELING DOWNTOWN

4.2.5.2 Constant Vertical Street Length

In this section we consider the special case of the label problem \((G, R, C, n, m)\) where all column labels have length \(l\). This problem we denote with \((G, R, C, n, m, l)\). We show that we can decide whether the label problem \((G, R, C, n, m, l)\) is solvable or not. We further give a simple algorithm that labels a solvable instance correctly. All results of this section are assignable for the constant row length case.

**Theorem 4.13 (Constant Column Length)** Let \((G, R, C, n, m, l)\) be a label problem with \(c_i = l\), \(1 \leq i \leq n\). The instance is solvable if and only if no conflict occurred in procedure **PREPROCESSING**.

We assume that \(l \leq \left\lfloor \frac{n}{2} \right\rfloor\). Otherwise, row \(\left\lfloor \frac{n}{2} \right\rfloor \) contains no fields that are free for row labeling. The next lemma states that the preoccupied fields are symmetric to the vertical central axis of \(G\).

**Lemma 4.14** Let \((G, R, C, n, m, l)\) be a successfully preprocessed label problem. After the preprocessing each row has the form \([aba]\), where:

\[
G_{i,1} = \emptyset, \ldots, G_{i,a} = \emptyset, G_{i,a+1} = x, \ldots, G_{i,a+b} = x, G_{i,a+b+1} = \emptyset, \ldots, G_{i,m} = \emptyset
\]

for \(x = r\) or \(x = c\), \(m \geq b \geq 0\) and \(2a + b = m\).

**Proof:** Initially we have \(G_{i,j} = \emptyset\) for \(1 \leq i \leq n, 1 \leq j \leq m\). Executing Rule 4.3 on each row \(i\) with length \(r_i > \frac{n}{2}\) yields \(G_{i,m-r_i+1} = r, \ldots, G_{i,r_i} = r\). Thus, all \(r\)-entries of \(G\) are symmetric to the vertical mid axis of \(G\). Remember that each column has label length \(l\). Therefore, executing Rule 4.3 on each column \(i\) that is set to \(c\) the fields \(G_{i,j+1} = c, \ldots, G_{i,m-j+1} = c, \) if \(1 \leq i \leq \frac{n}{2}\); and \(G_{i,m-j+1} = c, \ldots, G_{i,j-1} = c, \) if \(\frac{n}{2} < i \leq m\). Therefore, all \(c\)-entries of \(G\) are symmetric to the central vertical axis of \(G\). Thus, until now each row \(i\) has the form \([aba]\), where \(G_{i,1} = \emptyset, \ldots, G_{i,a} = \emptyset, G_{i,a+1} = x, \ldots, G_{i,a+b} = x, G_{i,a+b+1} = \emptyset, \ldots, G_{i,m} = \emptyset\) for \(x \in \{r, c\}, b \geq 0, 2a + b = m\) and \(1 \leq i \leq n\). Assume that the repeated execution of Rule 4.3 on row \(i\) of form \([aba]\) and \(x = c\) does change an entry of \(G_{i,..}\). In this case \(a < r_i\) and it follows that the instance is not solvable. Therefore, the repeated execution of Rule 4.3 on a column can not change the instance and the lemma follows. \(\square\)

**Lemma 4.15** Let \((G, R, C, n, m, l)\) be a successfully preprocessed label problem. Then algorithm **CONSTANT COLUMN LENGTH** computes a feasible solution to \((G, R, C, n, m, l)\) in \(O(nm(n+m))\) time.
Algorithm 7  \textsc{ConstantColumnLength} \( (G, R, C, n, m, l) \)

\begin{enumerate}
\item if \textsc{Preprocess}ing\( (G, R, C, n, m) \) \{ \\
\item label the columns \( 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \) bottommost; \\
\item label the columns \( \left\lceil \frac{n}{2} \right\rceil + 1, \ldots, m \) topmost; \\
\item label the rows \( 1, \ldots, n \) in the free space; \\
\}
\end{enumerate}

\textbf{Proof:} Since \( (G, R, C, n, m, l) \) is preprocessed successively it follows that each column contains an interval of length at least \( l \) that is free for column labeling. Assume that after processing steps 2-3 there exists a row \( i \) not containing an interval of length \( r_i \) that is free for row labeling. We make a case distinction according to the length of \( R_i \):

\textbf{Case } \( r_i > \left\lceil \frac{m}{2} \right\rceil \): We know that the fields \( G_{i,m-r_i+1} = r_i \ldots, G_{i,r_i} = r_i \) were set in the preprocessing. Furthermore, Lemma 4.14 yields that no other entry of \( G_{i,\cdot} \) was set to \( c \) in the preprocessing. Therefore, each column \( G_{i,j} \) with \( 1 \leq j < m - r_i + 1 \) or \( r_i + 1 \leq j \leq m \) contains either one interval of length at least \( 2l \) that is free for column labeling or two intervals each of length at least \( l \) that is free for column labeling. From the symmetry of the label problem and since the column labels of the columns \( j \) with \( 1 \leq j < m - r_i + 1 \) are labeled bottom most and the labels \( j \) with \( r_i + 1 \leq j \leq m \) are labeled top most it follows that either the fields \( G_{i,1} \ldots, G_{i,m-r_i+1} \) are free for row labeling or the fields \( G_{i,r_i+1} \ldots, G_{i,m} \). Thus, \( G_{i,\cdot} \) contains an interval of length \( r_i \) that is free for row labeling. Contradiction.

\textbf{Case } \( r_i \leq \left\lceil \frac{m}{2} \right\rceil \): Lemma 4.14 yields that this row has the form \([aba]\) with \( G_{i,1} = \emptyset, \ldots, G_{i,a} = \emptyset, G_{i,a+1} = c, \ldots, G_{i,a+b} = c, G_{i,a+b+1} = \emptyset, \ldots, G_{i,m} = \emptyset \) and \( 2a + b = m \). Since the instance is solvable it follows that \( a \geq r_i \). With the same arguments as above it follows that either the fields \( G_{i,1} \ldots, G_{i,r_i} \) are free for row labeling or the fields \( G_{i,m-r_i+1} \ldots, G_{i,m} \) are free for row labeling. Contradiction.

The running time is dominated by the preprocessing and thus \( O(n^3) \) time.

See Figure 4.11-4.14 for an example. Figure 4.10 shows a typical downtown city map in which all vertical streets have the same length.
4.2. LABELING DOWNTOWN

Figure 4.11: Matrix of a label problem with constant column length after the successful preprocessing. Entries that are set in the preprocessing are colored black and gray.

Figure 4.12: Solution of the label problem of the left figure.
Figure 4.13: Matrix of a constant column length problem after the successful preprocessing. Entries that are set in the preprocessing are colored black and gray.

Figure 4.14: Solution of the label problem of the left figure.
Two Different Label Lengths

Let \((G, R, C, n, n)\) be a label problem. In this section we study the case when any row label and any column label has length \(l_1\) or length \(l_2\). This special case we denote with \((G, R, C, n, n, l_1, l_2)\). We give an algorithm which solves this problem.

**Theorem 4.16 (Two Different Label Lengths)**

Let \((G, R, C, n, n, l_1, l_2)\) be a quadratic label problem such that \(r_i \in \{l_1, l_2\}\) and \(c_j \in \{l_1, l_2\}\). The instance is solvable if and only if no conflict occurred in procedure Preprocessing.

W.l.o.g let \(l_1 \geq l_2\). In case that \(l_1 \leq \left\lfloor \frac{n}{2} \right\rfloor\) Theorem 4.12 yields that the instance is solvable. In case that \(l_1 \geq \left\lceil \frac{n}{2} \right\rceil\) and \(l_2 \geq \left\lfloor \frac{n}{2} \right\rfloor\) it is easy to see that the instance is not solvable. Therefore, throughout this section and the next two subsections we assume that \(l_1 > \left\lfloor \frac{n}{2} \right\rfloor\) and \(l_2 < \left\lceil \frac{n}{2} \right\rceil\).

**The Case \(l_1 + l_2 > n\)**

Similarly to the half size solution we are going to label these instances in a windmill like manner. The critical rows and columns are the central rows and columns \(n - l_1 + 1, \ldots, l_1\). From the regularity of the label length we can make some useful observations about the existence of long labels (of length \(l_1\)) in the central rows and columns.

**Observation 4.17** Let \((G, R, C, n, n, l_1, l_2)\) be a solvable label problem. At most one of the label sets \(\{R_{n-l_1+1}, \ldots, R_{l_1}\}\) and \(\{C_{n-l_1+1}, \ldots, C_{l_1}\}\) has elements of length \(l_1\).

The label problem has the nice property that intervals that are free for labeling in the central rows and columns are also free for labeling in the outer rows and columns.

**Observation 4.18** Let \([a, b]\) be an interval of row \(G_{i,j}\), that is free for row labeling, \(n - l_1 < i \leq l_1\). Then, \([a, b]\) is free for row labeling in all rows.

**Proof:** Assume there exists a row \(i'\) and a column \(j \in [a, b]\) such that \(G_{i',j} = c\). In case that \(c_j = l_1\) it follows that \(G_{i',j} = c\) for \(n - l_1 < i \leq l_1\). Contradiction. In case that \(c_j = l_2\) it follows that \(G_{i',j} = c\) for \(n - l_1 < j \leq l_1\). Therefore, row \(i'\) contains not enough free space for row labeling and we get a contradiction since the preprocessing would not be successful in this case. \(\square\)
Observation 4.19 Let \((G, R, C, n, n, l_1, l_2)\) be a label problem that was successfully preprocessed. There exist two disjoint intervals \([a_1, a_2], [a_3, a_4]\) each of length at least \(l_2\) such that \(G_{i_1, a_1}, \ldots, G_{i_1, a_2-1}\) and \(G_{i_2, a_3}, \ldots, G_{i_2, a_4-1}\) are free for row labeling for all rows \(1 \leq i \leq n\).

Proof: Assume there exists a row \(i\) with \(r_i = l_2\) that does not contain two intervals of length at least \(l_2\) that are free for row labeling (with Observation 4.18 we can assume that \(n - l_1 < i \leq l_1\)). From the preprocessing Rule 4.3 follows that \(R_i\) causes an occupied entry \(G_{i,j} = r\) for some \(1 < j \leq n\). Then, each row label \(R_j\) of length \(l_2\) \((n - l_1 < k \leq l_1)\) causes an occupied row entry \(G_{k,j} = r\) for \(j\). Each row label \(R_k\) of length \(l_1\) and \(n - l_1 \leq k \leq l_1\) causes an occupied row entry \(G_{k,j} = r\) for \(j\) as well since \(l_1 > l_2\). From \(c_j \geq l_2\) and since \(n - l_1 < l_2\) and \(n - l_1 \geq l_2\) it follows that there is not enough free space for column label \(C_j\). Therefore, the preprocessing would have found a conflict which is a contradiction to our assumption. \(\Box\)

Algorithm 8 TWOLEABLENTHS1, \(l_1 + l_2 \geq n\) \((G, R, C, n, n, l_1, l_2)\)

1) if Preprocessing \((G, R, C, n, n)\) { 
2) \(\text{let } [a_1, a_2], [a_3, a_4] \text{ be two disjoint maximal intervals of } G \text{ each of size at least } l_2 \text{ that are free for row labeling in each row;}\)
3) \(\text{let } [b_1, b_2], [b_3, b_4] \text{ be two disjoint maximal intervals of } G \text{ each of size at least } l_2 \text{ that are free for column labeling in each column;}\)
4) \(\text{label the rows } 1, \ldots, b_3 - 1 \text{ in the leftmost possible sufficiently large interval that is free for row labeling;}\)
5) \(\text{label the rows } b_3, \ldots, n \text{ in the rightmost possible sufficiently large interval that is free for row labeling;}\)
6) \(\text{label the columns } 1, \ldots, a_3 - 1 \text{ in the bottommost possible sufficiently large interval that is free for column labeling;}\)
7) \(\text{label the columns } a_3, \ldots, n \text{ in the topmost possible sufficiently large interval that is free for column labeling;}\)
8)}

These observations lead to algorithm TWOLABELNTHS1 which solves these kind of instances.

Lemma 4.20 Let \((G, R, C, n, n, l_1, l_2)\) be a label problem with two different label lengths. If no conflict occurred in the procedure Preprocessing, then algorithm TWOLABELNTHS1 computes a feasible solution for \((G, R, C, n, n, l_1, l_2)\) in \(O(n^3)\) time.
4.2. LABELING DOWNTOWN

**Proof:** The rows \( i \in \{1, \ldots, b_3 - 1\} \) of label length \( l_2 \) are labeled inside \( G_{i,1}, \ldots, G_{i,2^s-1} \), not conflicting with any preoccupied column entry from the preprocessing. The rows \( i \in \{b_3, \ldots, n\} \) of label length \( l_2 \) are labeled inside \( G_{i,3}, \ldots, G_{i,n} \), not conflicting with any preoccupied column entry from the preprocessing. The columns \( j \in \{1, \ldots, a_3 - 1\} \) of label length \( l_2 \) are labeled inside \( G_{b_3,j}, \ldots, G_{n,j} \), not conflicting with any preoccupied row entry from the preprocessing. The columns \( j \in \{1, \ldots, a_3 - 1\} \) of label length \( l_2 \) are labeled inside \( G_{1,j}, \ldots, G_{b_3-1,j} \), not conflicting with any preoccupied row entry from the preprocessing. Since all four submatrices are disjoint it follows that no conflict occurs between any two labels of length \( l_2 \). Assume that there exist two indices \( i, j, 1 \leq i, j \leq n \) such that \( G_{i,j} \) is set to \( r \) and \( c \).

**Case 1 \( i \leq b_3, 1 \leq j \leq a_3 \):** From the observations made above follows that either a label of length \( l_2 \) overlaps with a label of length \( l_1 \) or two labels of length \( l_1 \) overlap.

Let two labels of length \( l_1 \) overlap. Since the preprocessing was successful and the \( j^{th} \) column is labeled bottommost it follows that all entries \( G_{b_3,j} = c \) with \( 1 \leq k < b_3 \) are preoccupied fields and therefore set in the preprocessing. These preoccupied fields have been considered in the positioning of label \( r \). Contradiction. In case that \( r_1 = l_2 \) and \( c_j = l_1 \) we come to a contradiction with analog considerations. In case that \( r_1 = l_1 \) and \( c_j = l_2 \) we know that column \( j \) is labeled below row \( b_3 - 1 \) and yield a contradiction.

**Other Cases:** All other cases work analogously, since the whole labeling is symmetric.

The preprocessing clearly dominates the running time. Thus the running time is \( O(n^3) \) time.

**The Case \( l_1 + l_2 \leq n \)**

We subdivide each column and each row into intervals. As interval points we define \( x_1 = l_2, x_2 = n - l_1, x_3 = l_1, \) and \( x_4 = n - l_2 \). Note that \( x_1 \leq x_2 \leq x_3 \leq x_4 \). These intervals subdivide the matrix \( G \) into several areas. We denote these areas with the indices \( q_{11}, \ldots, q_{55} \) as defined in Figure 4.16.

Similarly to the case \( l_1 + l_2 > n \) we can make some observations about the label length of solvable instances in the central rows and columns \( x_2 + 1, \ldots, x_3 \).

**Observation 4.21** Let \( (G, R, C, n, n, l_1, l_2) \) be a solvable label problem. At most one of the label sets \( \{R_{x_2+1}, \ldots, R_{x_3}\} \) and \( \{C_{x_2+1}, \ldots, C_{x_3}\} \) has elements of length \( l_1 \).
Figure 4.15: Solution of a problem with two different label length and \( l_1 + l_2 > n \) after the execution of algorithm Two\labelengths. Entries that were occupied in the preprocessing are colored dark, all others are shaded.
4.2. LABELING DOWNTOWN

**Observation 4.22** A preoccupied row entry $G_{i,j} = r$ for rows with label length $l_1$ can only occur if $i \in \{x_2 + 1, \ldots, x_3\}$ or $j \in \{x_2 + 1, \ldots, x_3\}$. For rows with label length $l_2$ it can only occur if $i \in \{x_2 + 1, \ldots, x_3\}$.

Analogously, a preoccupied column entry $G_{i,j} = c$ for columns with label length $l_1$ can only occur if $i \in \{x_2 + 1, \ldots, x_3\}$ or $j \in \{x_2 + 1, \ldots, x_3\}$. For columns with label length $l_2$ it can only occur if $j \in \{x_2 + 1, \ldots, x_3\}$.

**Proof:** Assume $G_{i,j} = r$ and $0 \leq i \leq x_2$ and $0 \leq j \leq x_2$. In case that $r_i = l_2$ we yield that this preoccupied entry must have been induced by at least one column entry $G_{i',j'} = c$ with $x_3 < i' \leq n$. In case that $r_i = l_1$ we also yield that this preoccupied entry must have been induced by at least one column entry $G_{i',j'} = c$ with $x_3 < i' \leq n$ since otherwise all preoccupied fields are contained in the interval $[x_2 + 1, x_3]$ of row $i$. Assume $G_{i,j} = r$ and $0 \leq i \leq x_2$, $x_2 < j \leq x_3$ and $r_i = l_2$. We again yield that this preoccupied entry must have been induced by at least one column entry $G_{i',j'} = c$ with $x_3 < i' \leq n$ and one with $1 \leq i' \leq x_2$. Since both possibilities are again caused by preoccupied fields where neither of
the two indices lies inside \([x_2 + 1, x_3]\) we come to a contradiction. All other cases work exactly analogously. □

**Lemma 4.23** We differentiate between two types of instances.

**Case 1:** No label of length \(l_2\) causes a preoccupied entry. Let \([a, b]\) be an interval of row \(G_i\), that is free for row labeling, \(x_2 < i \leq x_3\). Then, \([a, b]\) is free for row labeling in all rows. Analogously, let \([c, d]\) be an interval of column \(G_j\) that is free for column labeling, \(x_2 < j \leq x_3\). Then, \([c, d]\) is free for column labeling in all columns.

**Case 2:** There exist labels of length \(l_2\) that cause preoccupied fields. Then, there exists an interval \([a, b]\) of length at least \(l_2\) and \([a, b]\cap [x_2 + 1, x_3] = \emptyset\) such that either the fields \(G_{i,x}, \ldots, G_{i,x-1}\) are free for row labeling for each \(i \in \{x_2 + 1, x_3\}\) or the fields \(G_{x,j}, \ldots, G_{x,j-1}\) are free for column labeling for each \(i \in \{x_2 + 1, x_3\}\).

**Proof:**

**Case 1:** Let \([a, b]\) as defined in the lemma, case 1. Assume there exists a row \(i'\) and a column \(j \in [a, b]\) such that \(G_{i',j} = c\). Since no column with a label of length \(l_2\) causes a preoccupied entry we get that \(c_j = l_1\). It follows that \(G_{i,j} = c\) for \(x_2 < i \leq x_3\). Contradiction.

**Case 2:** There exists labels of length \(l_2\) that cause preoccupied fields. Consider the matrix \(G\) with all preoccupied fields of labels of length \(l_2\). Observe that in case that a preoccupied column entry \(G_{i,j} = c\), with \(i \in [x_2 + 1, x_3]\) exists, then also \(G_{i',j} = c\) for all \(i \in [x_2 + 1, x_3]\). Furthermore, in case that a preoccupied row entry \(G_{i,j} = r\), with \(j \in [x_2 + 1, x_3]\) exists, then also \(G_{i,j'} = r\) for all \(j' \in [x_2 + 1, x_3]\). These preoccupied fields induce (a part of the) preoccupied fields of labels of length \(l_2\). Concrete, let a column \(j \in [x_2 + 1, x_3]\) with \(c_j = l_2\) cause a preoccupied entry \(G_{i,j} = c\), then also \(G_{i,j'} = c\) for all columns \(j' \in [x_2 + 1, x_3]\), which follows from the observation made above.

**Case 1:** Since the preprocessing was successful there exists a row interval \([a, b]\) in row \(i\) of length at least \(l_2\) that is free for row labeling and \([a, b]\subseteq [1, x_2]\) or \([a, b]\subseteq [x_3 + 1, n]\). Assume there exists a row \(i' \in [x_2 + 1, \ldots, x_3]\) with \(i' \neq i\) and a \(j' \in [a, \ldots, b - 1]\) with \(G_{i',j'} = c\). Then, \(c_{j'} = l_2\) since otherwise also \(G_{i,j'} = c\). Since \(j' \notin [x_2 + 1, \ldots, x_3]\) we yield with Observation 4.22 a contradiction.

**Case 2:** In case that no \(i \in \{x_2 + 1, \ldots, x_3\}\) exists such that \(G_{i,j} = c\), \(j \in \{x_2 + 1, \ldots, x_3\}\) and \(c_j = l_2\), it follows there
exists an interval \([a, b]\) of length at least \(l_2\) and \([a, b] \cap [x_2 + 1, x_3] = \emptyset\)
such that the fields \(G_{a, j_1}, \ldots, G_{b, j}\) are free for row labeling for each
\(j \in \{x_2 + 1, \ldots, x_3\}\)

All non-listed cases are symmetric and follow analogously.

\[\Box\]

**Lemma 4.24** Let \((G, R, C, n, n, l_1, l_2)\) be a label problem with two different
label lengths and with \(l_1 + l_2 < n\). If no conflict occurred in procedure Pre-
processing, then algorithm **TwoLabelLengthsII** computes a feasible
solution for \((G, R, C, n, n, l_1, l_2)\) in \(O(n^3)\) time.

**Proof:** Let \((G, R, C, n, n, l_1, l_2)\) be a solvable and preprocessed label problem.
We first we show that no conflict occurs in part 1 of algorithm **TwoD-
ifferentLabelLengthsII**. Note that in step 2 only fields in the fields
\(q_{11}, q_{12}, q_{13}, q_{21}, q_{22}\), and \(q_{32}\) of matrix \(G\) are set to \(r\). In step 3 only fields
in fields \(q_{13}, q_{44}, q_{45}, q_{33}, q_{34}\), and \(q_{35}\) of matrix \(G\) are set to \(r\). In step 4 only
fields in the fields \(q_{31}, q_{32}, q_{44}, q_{42}, q_{41}\), and \(q_{32}\) of matrix \(G\) are set to \(c\). In
step 5 only fields in the fields \(q_{44}, q_{15}, q_{24}, q_{25}, q_{44}\), and \(q_{35}\) of matrix \(G\) are
set to \(c\). Since no field is named twice and since no conflict occurred in the
preprocessing no conflict occurred until now. Note, that the labels of length
\(l_1\) only occupy fields in the fields \(q_{11}, q_{23}, q_{35}, q_{32}, q_{14}, q_{15}, q_{14}, q_{15}\), and \(q_{35}\).
Thus, in the first part of the algorithm we found a conflict free position for the
labels \(\{R_1, \ldots, R_{x_2}, R_{x_2 + 1}, \ldots, R_n, C_1, \ldots, C_{x_2}, C_{x_2 + 1}, \ldots, C_n\}\). Figure 4.17
shows an example.

Let \(i \in \{x_2 + 1, x_3\}\) such that \(r_i = l_2\). Let \(I_1 = [a_1, b_1], \ldots, I_p = [a_p, b_p]\) be all
maximal intervals of \(G_{i, j}\) of length at least \(l_2\) that are free for row labeling,
for all \(i \in \{x_2 + 1, x_3\}\) with \(r_i = l_2\). Let \(J_1 = [e_1, f_1], \ldots, J_q = [e_q, f_q]\) be
all maximal intervals of \(G_{j, i}\) of length at least \(l_2\) that are free for column
labeling, for all \(j \in \{x_2 + 1, x_3\}\) with \(c_j = l_2\). We assumed w.l.o.g. that
\(r_{x_2 + 1} = l_2, \ldots, r_{x_3} = l_2\). Since the preprocessing was successful, for each
\(i \in \{x_2 + 1, \ldots, x_3\}\) with \(c_i = l_1\) there is an interval \([e, f]\) of \(G_{i, j}\) that
is free for column labeling. These columns are labeled in steps 6 and 7.
All fields of \(G_{i, j}\) with \(i, j \in \{x_2 + 1, \ldots, x_3\}\) that are set to \(c\) in steps 6-7
were occupied in the preprocessing. Thus, none of the label placements
further influences the amount of free space for row labeling in the rows
\(x_2 + 1, \ldots, x_3\).

Case \(\{I_1 \setminus [x_2 + 1, x_3]\}, \ldots, I_p \setminus [x_2 + 1, x_3]\) contains an interval \([a, b]\) of length
at least \(l_1\). This case is treated in steps 10-13. The labeling is trivially
correct.
Algorithm 9  \textsc{TwoLabelLengthsII}, \( l_1 + l_2 \leq n \) (G, R, C, n, n, \( l_1, l_2 \))

00) if not \textsc{Preprocessing}(G, R, C, n, n) then return

Part 1
01) \( x_1 := l_2; x_2 := n - l_1; x_3 := l_1; x_4 := n - l_2 \);
02) label the rows 1, \ldots, \( x_2 \) leftmost;
03) label the rows \( x_3 + 1, \ldots, n \) rightmost;
04) label the columns 1, \ldots, \( x_2 \) bottommost;
05) label the columns \( x_3 + 1, \ldots, n \) topmost;

Part 2
06) forall \( i \in \{ x_2 + 1, x_3 \} \) such that \( c_i = l_1 \)
07) label an interval of \( B_{i,j} \) of size \( l_1 \) that is free for column labeling with \( C_i \);
08) let \( I := \{ I_1 := [a_1, b_1], \ldots, I_p := [a_p, b_p] \} \) be all maximal intervals of length at least \( l_2 \) that are free for row labeling, for all rows and \( x_2 < i \leq x_3 \);
09) let \( J := \{ J_1 := [e_1, f_1], \ldots, J_q := [e_q, f_q] \} \) be all maximal intervals of length at least \( l_2 \) that are free for column labeling, for all columns and \( x_2 < i \leq x_3 \);
10) if \( \{ I_1 \setminus [x_2 + 1, x_3], \ldots, I_p \setminus [x_2 + 1, x_3] \} \) contains an interval \( [a, b[ \) of length at least \( l_2 \)\)
11) label the rows \( x_2 + 1, \ldots, x_3 \) leftmost, right of column \( a - 1 \);
12) for each \( i \in \{ x_2 + 1, \ldots, x_3 \} \) label column \( i \) in an arbitrarily chosen interval that is free for column labeling;
13) } else {
14) if \( \{ J_1 \setminus [x_2 + 1, x_3], \ldots, J_q \setminus [x_2 + 1, x_3] \} \) contains an interval \( [e, f[ \) of length at least \( l_2 \)\)
15) for each \( i \in \{ x_2 + 1, \ldots, x_3 \} \) with \( c_i = l_2 \) label columns \( i \) topmost, below row \( e - 1 \);
16) } else {
17) let \( [a_1, a_2], [a_3, a_4] \) be two disjoint intervals from \( \{ I_1, \ldots, I_p \} \) each of length at least \( l_2 \) such that \( A_{i,j} \) is free for row labeling, \( x_2 < i \leq x_3 \);
18) let \( [b_1, b_2], [b_3, b_4] \) be two disjoint intervals from \( \{ J_1, \ldots, J_q \} \) each of length at least \( l_2 \) such that \( B_{i,j} \) is free for column labeling, \( x_2 < i \leq x_3 \);
19) label the rows \( x_2 + 1, \ldots, b_3 - 1 \) leftmost, right of column \( a_1 - 1 \);
20) label the rows \( b_3, \ldots, x_3 \) rightmost, left of column \( a_1 \);
21) for each \( i \in \{ x_2 + 1, \ldots, b_3 - 1 \} \) with \( c_i = l_2 \) label column \( i \) bottommost, above row \( b_1 \);
22) for each \( i \in \{ b_3, \ldots, x_3 \} \) with \( c_i = l_2 \) label column \( i \) topmost, below row \( b_1 - 1 \);
23) } }
4.2. LABELING DOWNTOWN

Case \{J_i\backslash[x_2 + 1, x_3], \ldots, J_n\backslash[x_2 + 1, x_3]\} contains an interval \([c, f]\) of length at least \(l_2\): Analogously to the previous case steps 14-17 yield a valid labeling.

Case \(\{-\} \) all intervals of \(\{I_1\backslash[x_2 + 1, x_3], \ldots, I_n\backslash[x_2 + 1, x_3]\}\) have size smaller than \(l_2\): With Lemma 4.23 we yield that labels of length \(l_2\) do not produce preoccupied fields. Thus each row \(i \in \{x_2 + 1, \ldots, x_3\}\) with \(r_i = l_2\) contains at least two disjoint intervals of length at least \(l_2\) that are free for row labeling. Analogously, each column \(j \in \{x_2 + 1, \ldots, x_3\}\) with \(c_j = l_2\) contains at least two disjoint intervals of length at least \(l_2\) that are free for column labeling. With Lemma 4.23 follows that \(\{I_1, \ldots, I_n\}\) contains two disjoint intervals \([a_1, a_2], [a_3, a_4]\) each of length at least \(l_2\) such that \(G_{i,a_1}, \ldots, G_{i,a_2-1}\) and \(G_{i,a_3}, \ldots, G_{i,a_4-1}\) are free for row labeling, \(x_2 < i \leq x_3\). Analogously, \(\{J_1, \ldots, J_n\}\) contains two disjoint intervals \([b_1, b_2], [b_3, b_4]\) each of length at least \(l_2\) such that \(G_{b_1,i}, \ldots, G_{b_2-1,i}\) and \(G_{b_3,j}, \ldots, G_{b_4-1,j}\) are free for column labeling, \(x_2 < i \leq x_3\). Assume that steps 18-24 do not yield a conflict free labeling. Assume that there exists two indices \(i, j, 1 \leq i \leq n, 1 \leq j \leq m\) such that \(G_{i,j}\) was set to \(r\) and to \(c\).

We make a case distinction according to \(i\) and \(j\).

Case \(x_2 < i < b_1\) and \(x_2 < j < a_3\): No conflict can occur since the columns \(x_2 + 1, \ldots, a_3 - 1\) with length \(l_2\) are labeled below \(b_1 - 1\).

The columns with label length \(l_1\) are labeled right of \(a_2 - 1\).

Case \(x_2 < i < b_1\) and \(a_3 \leq j \leq x_3\): No conflict can occur since the rows \(x_2 + 1, \ldots, b_1 - 1\) are labeled left of the column \(a_3\).

Case \(b_3 \leq i \leq x_3\) and \(x_2 < j < a_3\): No conflict can occur since the rows \(b_3, \ldots, x_3\) are labeled right of the column \(a_3 - 1\).

Case \(b_3 \leq i \leq x_3\) and \(a_3 \leq j \leq x_3\): No conflict can occur since the columns \(x_2 + 1, \ldots, x_3\) are labeled above row \(b_3\).

Thus, in all cases we found a conflict free labeling and the running time, which is again dominated by the preprocessing, is \(O(n^3)\) time. 

\(\square\)

See Figure 4.17 and 4.18 for an illustration.

4.2.6 Conclusion and Open Problems

Studying the downtown labeling problem led to surprising results. On the one hand, the downtown labeling problem was shown to be \(\text{NP}\)-complete. Indeed Unger and Seibert showed the problem to be \(\text{APX}\)-hard. On the other hand the backtracking algorithm is empirically efficient.
Figure 4.17: Matrix of a problem with two different label lengths and $l_1 + l_2 \leq n$ after the execution of algorithm TwoLABELLENGTHS II part I. Entries that were occupied in the preprocessing are colored black, all others are shaded.
4.2. LABELING DOWNTOWN

Furthermore, the analysis of special cases led to nice algorithms with polynomial running time for non-trivial special cases.

There are several further variations of the downtown labeling problem which are worth studying. Usually very long streets are labeled more than once. It would be interesting to study the downtown labeling problem, where e.g., every street label is positioned twice (three times,...).

In our definition all label positions are of equal "goodness". A cartographer would probably invoke aesthetic criteria on the label positions. These aesthetic criteria could be reflected by a weighted version of the label problem in which "good" label positions have small penalty and "bad" label positions have high penalty.
Part II

Software Engineering in Computational Geometry
Chapter 5

Software Engineering in Computational Geometry

Although remarkable progress has been made in the theory of computational geometry [GO97], implementing geometric algorithms presents difficulties of its own. The most common problems are of numerical nature, problems with degeneracies, sophisticated data structures, and huge hidden constants in asymptotic worst case analysis [Rep96, Sch97].

- Theoretical papers assume exact arithmetic with real numbers. The use of floating point arithmetic can lead to rounding errors that can destroy geometric consistency. If an algorithm relies on this geometric consistency this can result in errors.

- Often, theoretical papers exclude degenerate configurations in the input ("Without loss of generality, we assume that all points are in general position"). Nevertheless, an algorithm also has to work on degenerate input.

- Algorithms in geometry are among the most advanced algorithms with respect to algorithm design and frequent use of complicated data structures.

- The hidden constant in asymptotic worst case complexity analysis can easily outweigh asymptotic factors in practice. Furthermore, the worst case usually depends on a general input model that may be unrealistic. E.g. in many applications the input data lies on a grid
CHAPTER 5. SOFTWARE ENGINEERING IN COMPUTATIONAL GEOMETRY

of specified granularity. For these reasons, the asymptotic worst case complexity analysis often does not match the need in practice.

The Computational Geometry Algorithms Library CGAL\(^1\) [Ove96, FGK, BKSV, FGKSS98] aims at providing a library that consists of correct, robust, efficient, flexible, and reusable algorithms and data structures in computational geometry. The library is being continuously developed by several universities and research institutes in Europe and Israel. One important aspect of flexibility is that algorithms can be easily adapted to work on data types in applications that already exist. This flexibility is reached through applying the generic programming paradigm using C++ templates [Aus98, Str97]. The object oriented programming (OOP) paradigm in C++ is not encouraged to be used in the library. However, in several appropriate places CGAL makes use of object oriented solutions and design patterns [GHVJ94]. Generic programs with templates are strongly type checked at compile time. The C++ abstractions used in CGAL cause no runtime overhead [FGKSS98]. Therefore, generic programming goes hand in hand with flexible, correct, and efficient programming.

In most discussions about OOP and generic programming, the programming paradigms are contrasted with each other [Ket99, Wei98]. This makes sense when comparing their advantages and drawbacks. Nevertheless, OOP and generic programming are not incompatible as we will see in the design of the range and segment tree data structures (see Chapter 6).

We now give an introduction into both paradigms and a short comparison of both paradigms. We assume that the reader is familiar with the basic concepts of the programming language C++. We recommend reading *The C++ Programming Language* book from Stroustrup [Str97] and the *Generic Programming and the STL* written by Austern [Aus98]. The introduction into the programming paradigms is followed by an overview of CGAL (see Section 5.3).

5.1 Object Oriented Programming Paradigm

Usually, OOP means programming with inheritance and dynamic binding. Inheritance is a kind of abstraction that allows the programmer to ignore differences between similar objects at some time and to exploit these differences at other times. Dynamic binding takes care of calling the right

\(^1\)See [http://www.cs.uu.nl/CGAL](http://www.cs.uu.nl/CGAL)
5.1. OBJECT ORIENTED PROGRAMMING PARADIGM

function at run time, based on the actual type of an object. An introduction into OOP can be found in [Str97, McL94, KM96].

The flexibility by OOP is achieved by defining a base class which defines an interface, and derived classes that implement this interface. Typically, this base class is pure virtual, that is, its functions are declared in use of the keyword virtual and not implemented. The generic functionality is that whenever an instance of the base class is required, any derived class can be used. This is achieved through the concept of virtual member functions. The effect of virtual member functions becomes clear through the class hierarchy illustrated in Figure 5.1 and the following piece of code:

```cpp
void draw_something(Graphic &g) {
    g.draw();
}

int main() {
    Rectangle r = ...;
    draw_something(r); /* draws the rectangle r */
}
```

Here, the class `Graphic` is a pure virtual base class which defines an interface consisting of a pure function `virtual void draw()`. The classes `Line`, `Rectangle`, and `Text` are derived from class `Graphic`. Each class provides an implementation of the function `draw()`. The function `draw_something(Graphic & g)` expects an instance of type `Graphic` as argument and calls `g.draw()`. Since `draw_something` is called with an instance of class `Rectangle`, and since `draw()` is a virtual member function of class `Graphic`,

![Figure 5.1: Class hierarchy: Class Graphic is a pure virtual base class. The classes Line, Rectangle, and Text are derived from class Graphic (indicated by the triangle, for an explanation of the notation see [GHV94]).](image-url)
the member function `draw()` of class `Rectangle` is called and not the one of class `Graphic`.

Assume an instance of a derived class is used in a function where an instance of the base class is required. Then the "real" type of the derived class is unknown in this function. Recovering the "lost" type of an object at runtime can be achieved through `dynamic_cast`. The use of type information at runtime is called runtime type information (RTTI). The following extension of the example above is an example for the use of dynamic casts. Here, an object of type `Graphic` is casted dynamically to an object of type `Rectangle`. Then, a function only existing in class `Rectangle` is called. Note that `dynamic_cast<Rectangle*>(g)` returns 0 if `g`'s class is not a subclass of class `Rectangle`.

```cpp
void manipulate(Graphic *g) {
    Rectangle *r;
    if (r = dynamic_cast<Rectangle*>(g)) {
        r->rotate();
    }
}
```

Unfortunately, there are several disadvantages for the object oriented programming paradigm: Whenever dynamic casts are used, no strong type checking can be done at compile time. When using virtual functions, additional memory is required for each object (the so called `virtual function table pointer`). Furthermore, for each call to a virtual member function an indirection through the virtual function table is added [Lip96]. This indirection affects the runtime performance since it cannot be made `inline`. This means that the compiler cannot optimize out the indirection. These effects are negligible for larger functions, but small functions will suffer from an increased runtime of one or two orders of magnitude [FGK+98]. Since most optimization techniques of compilers fail for virtual member functions, the space and runtime overhead introduced through virtual member functions is not negligible.

### 5.2 Generic Programming Paradigm

Generic programming focuses on subdividing a program package into small generic components that can be combined flexibly. The flexibility of a component is achieved by `class templates`, `function templates`, the use of programming concepts such as the `iterator` concept, the `traits` concept,
and the use of function objects. We now introduce these different generic concepts, beginning with templates.

### 5.2.1 Templates

A template argument of an algorithmic component is a formal placeholder for a type that is left open. Whenever a component with a template argument is used, the computer generates a separate translation of the component where the template argument is replaced by the actual type. For example, the `swap` function exchanges the values of two variables of arbitrary type. This can be realized by the following template function:

```cpp
template<class T> void swap(T& a, T& b){
    T tmp = a;
    a = b;
    b = tmp;
}
```

Let \( x, y \) be two integers, then `swap(a,b)` is a possible call. Note that the actual types for a function template are implicitly given by the types of the function arguments at instantiation time. As opposed to function templates, the actual types for class template arguments have to be explicitly provided by the programmer. The template class below is a generic list class for arbitrary item types. A list with elements of type `int` is then defined by `list<int>`.

```cpp
template<class T> class list{
    void push_back(T& t); // appends t to the list */
    ...
}
```

The example of the `swap` function shows that a template argument has to fulfill certain requirements. In this case the variables of type \( T \) have to be assignable. The existence of an assignment operator for \( T \) is a syntactic requirement. Syntactic requirements are checked at compile time. Apart from syntactic requirements there are semantic requirements which cannot be checked by a compiler. The semantic requirements to the assignment operator \( T \) would be that the assignment operator actually copies the value. Since the syntactic requirements of a template argument can be spread out in large functions or classes, the syntactic (and semantic) requirements are hard to extract and therefore have to be documented carefully.
5.2.2 Iterators

The Standard Template Library (STL) [MS96] is a good and well known example of generic programming that greatly influenced the definition of the C++ Standard Library. In the Standard Template Library, algorithms and data structures are independent of each other and are of equal significance. Consequently, the algorithms and data structures can be combined flexibly. One key concept in STL is the iterator concept which works on sequences of items of one type. The data structure of an item is called container in STL. The iterator concept is an abstraction of the interface between an algorithm and a sequence container by means of a small set of functions that allow to refer to an item and to traverse a sequence of items. The iterators are grouped into five categories, according to the capabilities of the iterator. The categories are the input, output, forward, bidirectional, and random access iterator. E.g. the usual C-pointer to a C-array fulfills the requirements of a random access iterator. Usually, an algorithm is given two iterators first and last, one referring to the first item and the other referring behind the last item of the sequence. \( ++\text{first} \) advances the iterator to the next element and \( *\text{first} \) returns the item the iterator refers to. STL sequence containers such as list and vector provide two functions begin() and end() each returning an iterator which refers to the first, resp. behind the last item of the sequence. E.g. with these iterators it is possible to use the good old for loop for iteration:

```cpp
list<int> li; // initialize li ...
list<int>::iterator i;

for(i=li.begin(); i!=li.end(); i++)
    cout << *i; // do something else ...
```

A typical STL algorithm working on a sequence of items is parameterized with an iterator referring to the first and an iterator referring behind the last item.

A call to the max_element algorithm from STL, which returns a maximum element in a sequence, can be written as

```cpp
int main()
{
    list<int> li;
    //initialize li ...

    int max_list_item = max_element(li.begin(),li.end());
}
```
5.2. GENERIC PROGRAMMING PARADIGM

Everyone who has implemented a max_element algorithm knows that a comparison operator has to be defined for the container type the function is instantiated with. This operator is predefined for the type int. One can think of more complicated container classes than int, consisting of more than one comparable members. In this case, the comparison operator may be different in different applications, and it should be possible to use different comparison operators in different contexts. This leads to the function object concept — a very important concept in generic programming which is also widely used in STL.

5.2.3 Function Objects

Assume we want to run function max_element on a sequence of cars in order to find the fastest car in the sequence.

```cpp
struct car{
    char *car.name;
    int max.km.per.h;
    int cost;
};
```

Then, we have to inform function max_element about the comparison function to use. Therefore, the more generic version of the STL function max_element has an additional template argument Compare and the function call expects a function object of a comparison function.

```cpp
template <class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first,
    ForwardIterator last, Compare comp) {
    if (first == last) return first;
    ForwardIterator result = first;
    while (++first != last)
        if (comp(*result, *first)) result = first;
    return result;
}
```

A function object basically is an instance of a class that contains an implementation of the operator() function. E.g. `comp_speed()` is a function object that can be used as comparison function:

```cpp
struct comp_speed{
```
bool operator()(car x, car y) {
    return (x.max_km_per_h < y.max_km_per_h);
}

main() {
    car max_speed_car;
    list<car> lc;
    /* initialize the list of cars: lc */
    max_speed_car=max_element(lc.begin(),lc.end(),comp_speed());
}

In comparison to function pointers, function objects have the following advantages:

- function objects can be declared *inline*. Therefore, the function object call can be optimized out by the compiler.
- the function object's class can be used to store a local state.

The function object concept is of great value in the design of algorithmic components that have to be independent of the data structures they should be applied to. Since the use of function objects allows to use an algorithmic component and one data structure with different function objects in different applications, the algorithmic component gains flexibility.

### 5.2.4 Traits Classes

In order to satisfy flexibility requirements in CGAL, the geometric algorithms and data structures are separated into the algorithm itself, the data structure, the number type and the coordinate representation. This has the effect that the algorithms and data structures are easily exchangeable and can therefore be reused in new and unforeseen areas. The use of traits classes is an effective method to plug different algorithms, data structures, and number types, etc, together. A traits class builds the interface between two algorithmic components, e.g. between an algorithm and a data structure. Assume we have an algorithm which has a template argument `data_traits` which builds the interface to the data. All data accesses and operations on the data are `data_traits` data accesses and operations. Since the `data_traits` class is a template argument of the algorithm, it follows that this traits class has to fulfill specific syntactic and semantic requirements. E.g. it has to provide a set of types, members, and functions which may
5.2. GENERIC PROGRAMMING PARADIGM

have names and meanings different from those of the actual data structures the algorithm has to work on.

The *data_traits* class is the only algorithmic component in which both the specific data structure and its requirements in the algorithm are known and have to be linked together. The traits class is an implementation of the syntactic and semantic requirements of the algorithm with respect to the specific data structure. In other words, the traits class concept is a structured abstraction of the interface between two algorithmic components which is realized with the help of templates and function objects.

![Diagram](image.png)

**Figure 5.2: Traits class example.**

For a better understanding of the traits class concept, we give an example. Figure 5.2 shows a class *Drawer* which has a template argument a class called *Traits*. This traits class specifies the output medium on which class *Drawer* operates. The requirements on class *Traits* are to provide the functions `void drawPolygon(..)` and `void drawText(..)`. There are two output media classes: class *DrawPostscript* which produces postscript output and class *XWinDraw* which outputs in an XWindow. For each class a traits class is defined which builds the interface between the specific output medium and class *Drawer*. As a consequence, if class *Drawer* is instantiated with the traits class *PSTraits* it will produce postscript output. Otherwise, if class *Drawer* is instantiated with the traits class *XWinTraits*, the output will be given in a window.

Depending on the functionality of the specific output class, the implementation of a traits class function can either be a simple call of the corresponding
function of the output class or of more complex nature. E.g. class XWinDraw does not provide a function that draws a polygon. Therefore, in function void drawPolygon(...) of class XWinTraits function void line(...) of class XWinDraw is called for each segment of the polygon.

5.2.5 Advantages and Disadvantages of Generic Programming

Generic programming has the following advantages: Generic programs are flexible and can be reused in new and unforeseen applications. Generic programming is type secure since strong type checking at compile time is performed during the template instantiation. No extra storage or indirection is needed during function calls and inline function calls are optimized out at compile time [Str97]. As a disadvantage we would like to mention the lack of notation in C++ to declare syntactical requirements of a template argument. The syntactical requirements of a template argument can only be found out if the whole function or class is examined. Thus, these requirements have to be well documented.

5.3 The Computational Geometry Algorithms Library (CGAL)

Since the implementation of geometric algorithms has its own difficulties, the library intends to make important computational geometry results available to users in academia and industry. Furthermore, the library is intended to serve as a framework for algorithm engineering that can easily be used for rapid prototyping and for experimental studies about geometric algorithms. In order to reach these goals, the algorithms and data structures in CGAL have to be correct, easy to use, robust, efficient, flexible, and reusable. A survey about CGAL can be found in [Ove96, FGK+98, BKSV99, FGKSS98].

In order to reach ease of use, CGAL introduced intuitive naming conventions, input/output support for debugging, several interfaces to various visualization tools, and above all, a detailed documentation of the data structures, types, and the design concepts and conventions.

The algorithms and data structures in CGAL are often proven to be asymptotically optimal. The exchangeability of the used number type is another efficiency issue of CGAL.
In order to meet the correctness issue, the algorithms and data structures are implemented by experts in the concerning areas. Furthermore, a sophisticated system of checks and warnings has been defined for the methods in the kernel and the basic library. The possibility to use exact arithmetic makes the implementations robust.

Flexibility and reuseability are gained in use of the generic programming paradigm which we describe in Section 5.2 and by the highly modular structure of the library.

The Computational Geometry Algorithms Library consists of several modular units. These units are grouped into the following bigger units: The Cgal kernel, the basic library and the support library. For the user, the modular structure has the advantage that the library can be used in parts; thus one only has to know about a part of the library instead of the whole library. For the developers the modular structure made it easy to organize the development of the library, since the parts could be implemented and tested separately.

The Cgal kernel consists of primitives, constant-size geometric objects (such as points, lines, and spheres) and predicates on them (such as orientation test for points and intersection tests). It is split in 3 parts that deal with 2-dimensional, 3-dimensional, and general-dimensional objects. For each dimension there are Cartesian and homogeneous representations for the coordinates available. Geometry in 2 and 3 dimensions is well studied and has many applications, which justifies the special status.

The basic library contains more complex geometric algorithms and data structures: Polygons, planar maps, range, and segment trees are some of the data structures; an algorithm for the computation of a convex hull or the Delaunay triangulation of a set of points are examples for algorithms in the basic library.

Functionality that is not purely geometric is provided in the support library. Here, number types provided by Cgal [BBP98, BEPP97] and interfaces to number types of other libraries such as LEDA\(^2\) and the Gnu Multiple Precision library [Gra96] can be found.

The use of exact arithmetic instead of floating point arithmetic can be necessary when robustness problems occur. Since exact arithmetic usually is much more time consuming than floating point arithmetic, a comfortable exchange of the number type is desirable. Cgal meets this goal by templatizing data structures and algorithms with a number type.

\(^2\)See http://www.mpi-sb.mpg.de/LEDA/leda.html
In addition to number types the support library contains random generators, circulators [BKS09], and interfaces to visualization software like GeomView and LEDA windows.

### 5.3.1 Related Work

An overview on computational geometry software until 1997 is given in [Ame97]. The Directory of Computational Geometry Software provides a collection of isolated implementations of algorithms and data structures in computational geometry. Since these algorithms and data structures have not been implemented to work together, it usually requires some effort to adapt them.

In software libraries the components are designed to work seamlessly together, to be extensible, and flexible. The most important software libraries in computational geometry are the GeomLib [BTV97] which is a computational geometry library (implemented in Java) and the precursors of CGAL which are PLAGEO [Gie94], C++GAL [Avn94], and the geometry part of LEDA.

---

3 See [http://www.geom.umn.edu/software/download/geoview.html](http://www.geom.umn.edu/software/download/geoview.html)

4 See [http://www.geom.umn.edu/software/cglist/](http://www.geom.umn.edu/software/cglist/)
Chapter 6

Range and Segment Trees

6.1 Introduction

Consider a database of a company for personnel administration. This database contains the name, address, salary, date of entrance, etc. of each employee. A typical query on this database is to report all employees that started to work in the company between 1994 and 1996 and earn between USD 4000 and USD 6000 per month. In order to formulate this as a geometric problem, we represent each employee by a point in the plane. The first coordinate of the point is the date of entrance and the second coordinate is the monthly salary. With the point we store the other information we have about the employee. The query can now be transformed in a geometric query: All points whose first coordinate lies between 1994 and 1996 and whose second coordinate lies between 4000 and 6000 have to be reported. Thus, we want to report all points in the axis parallel rectangle of Figure 6.1. This type of query is called range query or window query.

Another example for geometric queries is the following. We are given a database of the private estates and their owners in a country. This database contains the position and the geometry of each estate and the address of its owner. Due to a chemical accident the owners of the estates in a certain region have to be informed. Therefore, all estates in this region have to be found. The geometry of an estate can be very complex. Therefore, a testing if an estate has non empty intersection with the query -region might be very time consuming. There is a trick which can be used to reduce this problem to a range searching problem on axis parallel rectangles, which can
be solved efficiently. Each estate is represented by its axis parallel bounding box in the plane. A bounding box of a geometric object is the smallest axis parallel rectangle that encloses the object. The range searching problem then is to report all rectangles that have non empty intersection with the query rectangle (see Figure 6.2). In a second step the more time consuming test that checks if an estate intersects the query region can be invoked on the estates of the reported rectangles. A further important query type is the inverse range query. For a query point \( q \) all objects containing \( q \) have to be reported.

In the first example we saw range queries on two-dimensional point data and in the second example we saw range queries on rectangles that we call two-dimensional interval data. We call one-dimensional data types
with a strict ordering (like integer or double) point data. d-dimensional point data are \(d\)-tuples of one-dimensional point data. One-dimensional interval data is a pair \((a, b)\), where \(a\) and \(b\) are one-dimensional point data and \(a < b\). The pair \((a, b)\) represents the half open interval \([a, b)\). Analogously, a \(d\)-dimensional interval is represented by a \(d\)-tuple of one-dimensional intervals.

Because of the wide range of applications for geometric queries, efficient data structures that support geometric queries are of great interest in computational geometry. A lot of research on these data structures has been done, especially in the end of the seventies. A good overview over many data structures that support geometric queries can be found in [Sam90, dByKOS97]. Some important data structures that support geometric queries on \(d\)-dimensional point data are the range tree, the \(k\)-\(d\)-tree, and the quad tree. Furthermore, the segment tree supports inverse range queries and range queries on \(d\)-dimensional interval data, and the interval tree supports range queries on one-dimensional interval data. In order to provide one data structure that supports range queries on \(d\)-dimensional point data and one on \(d\)-dimensional interval data, we implemented the range tree and the segment tree data structure, for CGAL.

We preferred the implementation of the range tree over the implementation of the \(k\)-\(d\)-tree or the quad tree because of the better query performance time. In the range tree, the \(d\)-dimensional points that lie in the query rectangle can be reported in \(O(\log^d n + k)\) time, where \(n\) is the total number of points and \(k\) is the number of reported points. The tree can be built in \(O(n \log^{d-1} n)\) time and needs \(O(n \log^{d-1} n)\) space. In comparison, the \(k\)-\(d\)-tree has \(O(n^{1 - \frac{1}{d}})\) worst case running time and the quad tree has \(O(n)\) worst case running time for range queries. The decision to implement the segment tree was relatively easy, since it has a good worst case time bound for inverse range queries of \(O(\log^d n + k)\) time, where \(k\) is the number of reported \(d\)-dimensional intervals. The segment tree can also be used to perform range queries. The worst case range query performance time is \(O(k \log n + \log n)\) time for one-dimensional segment trees and \(O(n^{d-1} \log n + k) + k \log n)\) time for \(d\)-dimensional segment trees (which is extremely bad). The tree can be built in \(O(n \log^d n)\) time and needs \(O(n \log^d n)\) space. All named data structures are static, that is, insertions and deletions are disallowed.

One very useful and interesting aspect of the range and segment tree data structure is that both data structures can be combined.

That is, one can e.g. define a range tree on one-dimensional point data in the first dimension which has a segment tree as secondary structure on one-dimensional interval data in the second dimension. E.g. assume that in
the first example about the database for personal administration for each employee we are given a time period the employee was working on project X. A search for all employees who worked on project X between 1996 and 1997 and earn between USD 4000 and USD 5000 can be solved by a range query on a range tree which has as secondary structure a segment tree. The data type the tree handles is two-dimensional, in the first dimension we have Y's salary, and in the second dimension we have an one-dimensional interval denoting the time period person Y was working on project X. Then, the query range is two-dimensional interval which is the money range 4000-5000 USD in the first dimension and the time period in the second dimension. A tree consisting of different types of tree layers we call mixed multilayer tree. Mixed multilayer trees play an important role in computational geometry. For further applications we refer to [Ove88]. The realization of this aspect in the design of the data structures was very challenging.

The rest of this chapter is organized as follows: We first give a definition of the range and segment tree (see Section 6.2 and Section 6.3). Following that, we present our design goals for an implementation of the two data structures (see Section 6.4). In the following Section 6.5 we evaluate related work in respect to our design goals. Our design is then presented in Section 6.6. The program examples in Section 6.7 should give the reader a better understanding of the usage of the data structures. In Section 6.8 we conclude with an evaluation of the data structures in respect to the design goals.

6.2 Definition of a Range Tree

The range tree by Bentley and Mauer [Ben79, BM80] is best understood by first examining the one-dimensional range tree. A one dimensional range tree is a balanced binary search tree on one-dimensional point data. The data points are stored in the leaf nodes of the tree which are sorted such that the leftmost leaf contains the smallest point and the rightmost the biggest point. Each inner node stores the highest entry of its left subtree. The tree is static which means that after construction of the tree, no elements can be inserted or deleted. See Figure 6.3 for an example.

To report the points in a query range \([a, b]\), two leaf nodes \(u\) and \(v\) are determined such that \(u\) is the smallest leaf node with \(a \leq v\) and \(v\) is the smallest leaf node with \(b \leq v\). Then, all leaf nodes that lie between the two search paths to \(u\) and \(v\) have to be reported including \(u\). E.g., in order to search for the interval \([11, 25]\) in the tree of Figure 6.3, the two search
6.2. DEFINITION OF A RANGE TREE

Figure 6.3: Range tree on one-dimensional point data.

paths which are shaded light grey lead to the leaf with point data 12 and 26, respectively. All leaf nodes between these two search paths (shaded dark) and their entries have to be reported, including the entry of the left leaf node.

A $d$-dimensional range tree is a 1-dimensional range tree with respect to the first dimension of the $d$-dimensional point data. Each inner node of the tree is associated with a $(d-1)$-dimensional range tree on the points of its subtree. More precisely, let $p_v$ be the points in the subtree of a node $v$. Then, $v$ is associated with a $(d-1)$-dimensional range tree on the points $p_v$ with respect to the second, ..., $d$-th dimension. We call this tree the sublayer tree of $v$. Figure 6.4 shows a two-dimensional range tree with an exemplary illustration of the sublayer tree of $v$. In the first layer, the two-dimensional points are sorted with respect to the first dimension. In the sublayer of $v$, the child points of $v$ are sorted with respect to the second dimension. Figure 6.5 shows a $d$-dimensional range tree, for each dimension one sublayer tree is illustrated exemplarily.

The $d$-dimensional range tree can be constructed with an recursive algorithm. The algorithm gets as input a set of $d$-dimensional points $P = \{p_1, \ldots, p_n\}$. Let $p_{i,1}, \ldots, p_{i,d}$ be the coordinates of point $p_i$. The root of the tree is associated with the median $p_m$ of $P$ which is built with respect to the first dimension. For $d > 1$ each inner node of the tree has a $(d-1)$-dimensional sublayer range tree. The sublayer tree of the root is a $(d-1)$-dimensional range tree on the points $P$, which is built with respect to the second dimension. The left child of the root is a $d$-dimensional range tree on all points of $P$ that are smaller or equal to the median with respect to the first dimension. The right child is a $d$-dimensional range tree on all points of $P$ that are greater than the median in respect to the first dimension. These subtrees are build recursively by applying this algorithm to the appropriate point sets. In the case that a point set consists of
Figure 6.4: Range tree on two-dimensional point data.

Figure 6.5: Range tree on $d$-dimensional point data.
6.2. DEFINITION OF A RANGE TREE

one single point the point becomes a leaf and the recursion stops. The
procedure is illustrated in procedure Build-d-dimRangeTree.

**Procedure 6** Build-d-dimRangeTree (point set $P = \{p_1, \ldots, p_n\}$,
dimension $k=1$)

01) $p_m = \text{median of } P \text{ with respect to the } k^{\text{th}} \text{ coordinate};$
02) let the point associated with the root $v$ be $p_m$;
03) if $k < d$ then let the sublayer tree of $v$ be
         Build-d-dimRangeTree($P, k+1$);
04) $P_L := \{p_i \in P|p_{i,k} \leq p_{m,k}\};$
05) $P_R := \{p_i \in P|p_{i,k} > p_{m,k}\};$
06) if $P_L \text{ contains only one point then}$
07)     left_child($v$) = a leaf associated to $P_L$;
08) else left_child($v$) = Build-d-dimRangeTree($P_L, k$);
09) if $P_R \text{ contains only one point then}$
10)     right_child($v$) = a leaf associated to $P_R$;
11) else right_child($v$) = Build-d-dimRangeTree($P_R, k$);

The $d$-dimensional range tree can be constructed in $O(n \log^{d-1} n)$
time and needs $O(n \log^{d-1} n)$ storage. The construction consists of building
a balanced binary search tree, which takes time $O(n \log n)$, plus the
construction of the secondary structures. The needed time can be expressed
in a recurrence equation: $T_d(n) = O(n \log n) + \sum_{e = 0}^{\log(n-1)} 2^e T_{d-1}(\frac{n}{2^e})$, where
$T_d(n)$ is the construction time of a $d$-dimensional range tree with $n$ points.
This equation resolves to $O(n \log^{d-1} n)$ with $T_1(n) = n \log n$.

The query algorithm for the $d$-dimensional range tree which reports all
points in a query range $[a = (a_1, \ldots, a_d), b = (b_1, \ldots, b_d)]$ works recursively.
For a vertex $v$, let $u_1, \ldots, u_d$ be the $d$-dimensional point associated with $u$.
First, two search paths $p$ and $p'$ are determined. $p$ leads to the smallest leaf
node $v$ with $a_1 \leq u_1$, and $p'$ leads to the smallest leaf node $v$ with $b_1 \leq u_1$.
Let $v_1, \ldots, v_m$ be the set of vertices that are either a right child of $p$ and
left of $p'$, or a left child of $p'$ and right of $p$. Then, the query algorithm is
called for the $(d-1)$-dimensional sublayer trees of these vertices, where the
search paths are determined with respect to the second dimension of the
query range. Even in the one-dimensional sublayer tree two search paths $p$
and $p'$ are determined. $p$ leads to the smallest leaf node $u$ for which $a_d \leq u_d$
and $p'$ leads to smallest leaf node $v$ for which $b_d \leq v_d$. But now, all points
that lie between these two paths together with $u$ are reported. The pseudo
code of this procedure is illustrated in procedure D-dimRangeQuery.
**Procedure 7**  
D-DIMRANGEQUERY (d-dim range tree \( \tau \), query range \([a = (a_1, \ldots, a_d), b = (b_1, \ldots, b_d)]\), dimension \(k = 1\))

1. \( v_{split} = \text{FindSplitNode}(\tau, [a, b], k) \);
2. if \( v_{split} \) is a leaf then
   - check if the point stored at \( v_{split} \) must be reported;
3. else
   - \( v = \text{left}_\text{child}(v_{split}) \);
   - while \( v \) is not a leaf do
     - if \( a_k \leq v_k \) then
       - if \( k < d \) then
         - D-DIMRANGEQUERY (sublayer tree(\( v \)), \([a, b], k + 1 \));
     - else
       - Report all points in the subtree of \( v \);
       - \( v = \text{left}_\text{child}(v) \);
   - else
     - \( v = \text{right}_\text{child}(v) \);
   - check if a must be reported;
   - similarly, follow the path to \( b_k \), if \( k < d \) call D-DIMRANGEQUERY for sublayer trees left of the path, otherwise report the points of the subtrees; check if the point stored at the leaf must be reported;
4. \}

A range query on a \( d \)-dimensional range tree storing \( n \) \( d \)-dimensional points takes \( \mathcal{O}(\log^d n + k) \) time, where \( k \) is the number of reported points. In the first level of the tree \( \mathcal{O}(\log n) \) nodes are located whose subtrees contain all points whose first coordinate lies in the correct range. These nodes are queried further by performing a range query on the second-level structure. In each second-level structure \( \mathcal{O}(\log n) \) nodes are located whose subtrees contain all the points whose first and second coordinate lies in the query range. This means there are \( \mathcal{O}(\log^2 n) \) subtrees in the second-level in total. In the \( d \)-th level there are \( \mathcal{O}(\log^d n) \) subtrees located that contain all points in the correct range. Therefore, the query time is \( \mathcal{O}(\log^d n + k) \).
6.2. DEFINITION OF A RANGE TREE

Procedure 8  \text{FINDSPLITNODE}(\text{range tree layer } \tau, \text{query range } [a_k, b_k], \text{dimension } k)

1) \( v = \text{root} (\tau) ; \)
2) \textbf{while } \( v \) \textbf{ is not a leaf and } \((b_k \leq v_k \text{ or } a_k > v_k) \textbf{ do} \)
3) \textbf{if } \( b_k \leq v_k \textbf{ then } v = \text{left}\_\text{child}(v) ; \)
4) \textbf{else } \( v = \text{right}\_\text{child}(v) ; \)
5) \text{return } v ;

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{range_tree_example.png}
\caption{Range tree on 2-dimensional point data. Two sublayer trees are illustrated exemplarily.}
\end{figure}
6.3 Definition of a Segment Tree

The segment tree was also invented by Bentley in 1977 [Ben77]. Before we describe the general-dimensional segment tree, we describe the one-dimensional segment tree, since the definition of the general-dimensional segment tree is based on the one-dimensional one. For the one-dimensional segment tree we are given a set of one-dimensional interval data \( S = \{ s_1, \ldots, s_n \} \) where each \( s_i \in S \) is an interval of the form \([a, b]\). Let \( p_1, \ldots, p_n \) be the list of distinct interval endpoints of \( S \), sorted from left to right. These points partition the real line in elementary intervals: \((-\infty, p_1), [p_1, p_2), [p_2, p_3), \ldots, [p_n, \infty)\). The segment tree is a binary search tree \( \tau \) whose leaves correspond to the elementary intervals, such that the leftmost leaf corresponds to interval \((-\infty, p_1)\) and the rightmost leaf to interval \([p_n, \infty)\). An internal node of \( \tau \) corresponds to the interval that is the union of the elementary intervals of the leaves in its subtree. The intervals are stored in lists that are associated to the nodes of the tree. An interval \([a, b]\) is stored in the list of node \( v \), which corresponds to the interval \([a_v, b_v]\), if \([a_v, b_v] \subseteq [a, b]\) and if the interval \([a_v, b_v]\) which corresponds to the parent node of \( v \) is not contained in \([a, b] \). See Figure 6.7 for an illustration.

![Diagram of a segment tree](image)

Figure 6.7: A one-dimensional segment tree. The segments and the corresponding elementary intervals are shown below the tree. The query range is dashed, the query paths are shaded light, and the node between the query paths is shaded dark.
In an inverse range query, for a given point \( q \), all segments containing \( q \) have to be reported. Let \( p \) be the search path from the root to the leaf corresponding to the elementary interval containing \( q \). The query segments are exactly all those segments that are associated with nodes on \( p \). Since a segment is stored at at most one node on a path from the root to a leaf, the worst case processing time for inverse range queries in an one-dimensional segment tree is \( O(\log n + k) \) time.

To report the intervals that have nonempty intersection with a query range \([a, b]\), two paths \( p \) and \( p' \) are determined. Path \( p \) leads to the rightmost leaf node \( x \) such that \( a \leq b_x \). Path \( p' \) leads to the leftmost leaf node \( x' \) such that \( b \leq b_{x'} \). All intervals associated with vertices on and between these two paths match the query and have to be reported. The worst case range query processing time is \( O(k \log n + \log n) \) time, where \( k \) is the number of reported elements. The paths \( p \) and \( p' \) are determined in \( O(\log n) \) time. Since the number of elementary intervals between \( p \) and \( p' \) is bounded by \( O(k) \), the number of nodes between \( p \) and \( p' \) is also bounded by \( O(k) \). Thus, \( O(\log n + k) \) nodes are examined. Since each segment is stored at at most two nodes per level, it follows that at most \( 2k \log n \) segments are examined. Therefore, the worst case range query processing time is \( O(k \log n + \log n) \) time. In Figure 6.7 the left and right search paths are shaded light, the nodes between the search paths are shaded dark.

The \( d \)-dimensional segment tree is a one-dimensional segment tree with respect to the first dimension of the \( d \)-dimensional interval data. Each inner node of the tree is associated with a sublayer tree which is a \((d-1)\)-dimensional segment tree on the interval data that is associated with that node. The \((d-1)\)-dimensional segment tree is built with respect to the second, \ldots, \( d \)-th dimension. The algorithm for the construction of the tree is given in procedure \textsc{Build-d-dimSegmentTree}. Figure 6.8 illustrates a two-dimensional segment tree where one sublayer tree is illustrated exemplarily. The construction of the \( d \)-dimensional segment tree takes \( O(n \log^d n) \) time.

In an inverse range query of the \( d \)-dimensional segment tree, all \( d \)-dimensional intervals that contain the query point have to be reported. First, an inverse range query on the first layer of the \( d \)-dimensional segment tree with respect to the first dimension is performed. For all nodes that match the query, the \((d-1)\)-dimensional sublayer trees of these nodes are queried with respect to the second dimension, and so on. Finally, in the \( d \)-th sublayer tree, the intervals associated with the nodes that match the query are reported. Procedure \textsc{InverseRangeQuery} shows the pseudo-code for the query algorithm. The inverse range query processing time is
$O(\log^d n + k)$ since in each sublayer tree $O(\log n)$ nodes are located whose sublayer trees contain all intervals containing $q$.

The $d$-dimensional range query is performed analogously. Firstly, a range query is performed in the first layer of the segment tree with respect to the first dimension of the query range. Then, all sublayer trees of nodes that match the query are queried with respect to the second dimension, and so on. The worst case range query performance time is extremely high. The number of nodes in a segment tree layer whose sublayer trees have to be queried is not longer bounded by $\log n$ or $k$. Thus, the worst case query time is $O(n^{d-1}(\log n + k) + k \log n)$ time.
6.3. DEFINITION OF A SEGMENT TREE

Procedure 9  BUILD-d-dimSEGMENTTree (intervals $S = \{s_1, \ldots, s_n\}$, dimension $k = 1$)

1) determine a list of the elementary intervals $P = \{p_1, \ldots, p_m\}$ of $S$ sorted with respect to the $k$-th dimension;
2) build a binary tree on the elementary intervals with root $\tau$;
3) for each interval $s_i \in S$ do
   4)  $InsertInterval(\tau, s_i, k)$;
   5)  if $k < d$ then {
      6)  $BuildSublayerTrees(\tau, k)$;
      7)  delete all intervals stored at nodes of $\tau$;
      8)  } return $\tau$;

Procedure 10  BUILDSublayerTrees (node $v$, dimension $k$)

1) if $k < d$ then {
   2)  let $S$ be the intervals that are associated with $v$;
   3)  associate with $v$ the sublayer tree $BUILD-d-dimSEGMENTTree(S, k + 1)$;
   4)  }
5) if $v$ is not a leaf then {
   6)  $BuildSublayerTrees(left\_child(v), k)$;
   7)  $BuildSublayerTrees(right\_child(v), k)$;
   8)  }

Procedure 11  INSERTInterval (node $v$, interval $[a, b]$, dimension $k$)

1) if $[a_v, b_v] \subseteq [a, b]$ in the $k$-th dimension then
2)  store $[a, b]$ at $v$;
3) else {
4)  $l = left\_child(v)$;  $r = right\_child(v)$;
5)  if $[a_l, b_l] \cap [a, b] \neq \emptyset$ in the $k$-th dimension then
6)  $InsertInterval(l, [a, b], k)$;
7)  if $[a_r, b_r] \cap [a, b] \neq \emptyset$ in the $k$-th dimension then
8)  $InsertInterval(r, [a, b], k)$;
9)  }
Procedure 12   INVERSE RANGE QUERY \((d\text{-dim segment tree } \tau, q = (q_1, \ldots, q_d)\), \text{ dimension } k\)

1) if \(k < d\) then INVERSE RANGE QUERY (sublayer tree \((\tau), q, k + 1\));
2) else report all intervals associated with \(\tau\);
3) let \([a_1, b_1]\) be the interval associated with left_child \((\tau)\);
4) let \([a_2, b_2]\) be the interval associated with right_child \((\tau)\);
5) if \(q_k \in [a_1, b_1] \neq \emptyset\) then
6) INVERSE RANGE QUERY (left_child \((\tau), q, k\));
7) else if \(q_k \in [a_2, b_2] \neq \emptyset\) then
8) INVERSE RANGE QUERY (right_child \((\tau), q, k\));

Figure 6.9: 2-dimensional segment tree with one sublayer tree illustrated exemplarily.
6.4 Design Goals

In this section we concentrate on the design goals for the design of the range and segment tree data structure. Before we do so, we would like to mention that the range and segment tree are “compatible” to be combined in a mixed multilayer tree. That is, one can e.g. define a range tree where each inner node is associated with a segment tree as sublayer tree. Since mixed multilayer trees play an important role in computational geometry, it was one major design goal to realize this flexibility elegantly. The other design goals follow from the general design goals of CGAL and the definition of the trees.

Independence of the data type The tree should be easily adaptable to data types in applications that already exist. Therefore, it should be possible to store arbitrary data types in the tree. The requirements on these data types should be minimal.

Independence of the input/output sequence container The tree should be independent of the type of the sequence container in which the data is stored. That is, the tree should be flexible enough to allow different types of sequence containers like list, vector, and C-array for input and output of the data.

Arbitrary dimension It should be possible to define a range or a segment tree of arbitrary dimension. Furthermore, the definition of a one-dimensional tree should be very similar to the definition of a $d$-dimensional tree, $d > 1$.

Query types The tree should support different kinds of queries. Both trees should support range queries and the segment tree should further support inverse range queries and enclosing queries. In an enclosing query, all intervals that are contained in the query interval have to be reported.

Ease of use The design should be well structured, easy to use, and have the look and feel of CGAL.

Efficiency, correctness, and robustness The implementation should be efficient, type secure, and robust.
6.5 Related Work

The only implementations of the range and segment tree data structure that have been made public is their implementation in LEDA. These trees are used in LEDA for the implementation of the dictionary_2d, point_set, and segment_set data structures. The range and segment tree data structures are not meant to be used separately, since no documentation is made public. Therefore, we do not evaluate these data structures here.

Another data structure we would like to mention is the implementation of the k-d-tree in CGAL. The k-d-tree can be used to store and query d-dimensional point data. In respect to our design goals we have the following annotations to make:

**Independence of the data type** The k-d-tree can be adapted to work with arbitrary data types in arbitrary dimension. The data type and the necessary functionality on the data type is accessed through a traits class, the so-called Interface_traits. The functionality a data type has to provide is small: A comparison function for each dimension of the data type, and a copy constructor for each coordinate has to be provided.

**Independence of the input/output sequence container** The input/output sequence container is the STL-list. Thus, in this point the tree is not very flexible.

**General dimensionality** The tree is designed to handle data of arbitrary dimension. The definition of a one-dimensional tree is very similar to the definition of a d-dimensional tree.

**Query types** The tree supports range queries. The query time is $O(n^{1-\frac{1}{d}})$, where $n$ is the total number of points and $d$ the dimension of the tree.

**Ease of use** The data structure is easy to use. Furthermore, default interface traits classes are provided.

**Efficiency, correctness, and robustness** The implementation is type secure and robust. The preprocessing time is $O(n \log^d n)$, it uses $O(n \log^{d-1} n)$ space, and has a query time of $O(n^{1-\frac{1}{d}})$. 
6.6 Design Overview

In order to be able to define a multilayer tree we first designed the range and segment tree to have a template argument defining the type of the sublayer tree. With this sublayer tree type information the sublayers could be created. This approach lead to nested template arguments, since the sublayer tree can again have a template argument defining the sublayer. Therefore, the internal class and function identifiers got longer than a compiler-dependent limit. This happened already for \( d = 2 \).

Therefore, we chose another, object oriented, design. We defined a pure virtual base class called `Tree_base` from which we derived the classes `Range_tree_d` and `Segment_tree_d`. The constructor of these classes expects an argument called `sublayer_prototype` of type `Tree_base`. Since class `Range_tree_d` and class `Segment_tree_d` are derived from class `Tree_base`, one can use an instantiation of class `Range_tree_d` or class `Segment_tree_d` as constructor argument. This argument defines the sublayer tree of the tree. E.g., you can construct a `Range_tree_d` with an instantiation of class `Segment_tree_d` as constructor argument. You then have defined a range tree with a segment tree as sublayer tree. Since both classes `Range_tree_d` and `Segment_tree_d` expect a sublayer tree in their constructor we had to derive a third class called `Tree_anchor` from class `Tree_base` which does not expect a constructor argument. An instantiation of this class is used as constructor argument of class `Range_tree_d` or `Segment_tree_d` in order to stop the recursion.

All classes provide a `clone()` function which returns an instance (a copy) of the same tree type. The `clone()` function of the `sublayer_prototype` is called in the construction of the tree. In case that the sublayer tree again has a sublayer, it also has a `sublayer_prototype` which is also cloned and so on. Thus, a call to the `clone()` function generates a sublayer tree which has the complete knowledge about its sublayer tree.

The design partly follows the `prototype design pattern` in [GHV94]. In comparison to our first approach using templates we want to note the following: In this approach the sublayer type is defined in use of object oriented programming at run time, while in the approach using templates, the sublayer type is defined at compile time.

The runtime overhead caused in use of virtual member functions in this object oriented design is negligible since all virtual functions are non-trivial. The design concept is illustrated in Figure 6.10.

E.g. in order to define a two dimensional multilayer tree, which consists of a range tree in the first dimension and a segment tree in the second dimension...
we proceed as follows: We construct an object of type Tree_anchor which stops the recursion. Then we construct an object of type Segment_tree_d, which gets as prototype argument our object of type Tree_anchor. After that, we define an object of type Range_tree_d which is constructed with the object of type Segment_tree_d as prototype argument. The following piece of code illustrates the construction of the two-dimensional multilayer tree.

```cpp
int main(){
    Tree_anchor *anchor=new TreeAnchor;
    Segment_Tree_d *segment_tree = new Segment_Tree_d(*anchor);
    Range_Tree_d *range_segment_tree = new Range_Tree_d(*segment_tree);
    // let data.items be a list of Data items */
    range_segment_tree->make_tree(data.items.begin(), data.items.end());
}
```

Here, class Tree_Archor, Segment_Tree_d, and Range_Tree_d are defined by typedefs:

```cpp
typedef Tree_anchor<Data,Window> TreeAnchor;
typedef Segment_tree_d<Data,Window,Interval_traits> Segment_Tree_d;
typedef Range_tree_d<Data,Window,Point_traits> Range_Tree_d;
```

Class Tree_base and class Tree_anchor get two template arguments: a class Data which defines the type of data that is stored in the tree, and a class Window which defines the type of a query range. The derived classes Range_tree_d and Segment_tree_d additionally get an argument called Tree_traits which defines the interface between the Data and the tree. Let the Data type be a $d$-dimensional tuple, which is either a point data or an interval data in each dimension. Then, the class Tree_traits provides accessors to the point (resp. interval) data of that tree layer and a compare function. Remind our example of the two-dimensional tree which is a range tree in the first dimension and a segment tree in the second dimension. Then, the Tree_traits class template argument of class Segment_tree_d defines an accessor to the interval data of the Data, and the Tree_traits class template argument of class Range_tree_d defines an accessor to the point data of Data. An example implementation for these classes is listed below.

```cpp
struct Data{
    int min,max; // interval data */
```
Figure 6.10: Design of the range and segment tree data structure. The symbol triangle means that the lower class is derived from the upper class.
142 CHAPTER 6. RANGE AND SEGMENT TREES

double point; // point data */
};

struct Window{
    int min, max;
    double min_point, max_point;
};

class Point_traits{
public:
    typedef double Key,
    Key get_key(Data& d){return d.point;} //key accessor */
    Key get_left(Window& w){return w.min_point;}
    Key get_right(Window& w){return w.max_point;}
    bool comp(Key& key1, Key& key2){return (key1 < key2);
}
}

class Interval_traits{
public:
    typedef int Key,
    Key get_left(Data& d){return d.min;} 
    Key get_right(Data& d){return d.max;}
    Key get_left_win(Window& w){return w.min;}
    Key get_right_win(Window& w){return w.max;}
    bool comp(Key& key1, Key& key2){return (key1 < key2);
}
}

CGAL provides predefined range and segment tree classes for the dimensions one to four. They are called Range_tree_1, ..., Range_tree_4 and Segment_tree_1, ..., Segment_tree_4. These classes get one template argument called Traits. By implementing this Traits class, the predefined trees can be adapted to work with arbitrary Data. The implementation of this Traits class is straightforward. CGAL further provides predefined traits classes for Data of the CGAL type Point_3<Rep> and Point_3<Rep>. Rep is a template argument specifying the number type and the representation type of the coordinates. These predefined classes are easy to use and probably meet the need of the majority of users. For the user manual of these implementations we refer to http://www.cs.uu.nl/CGAL/Information/doc_html/index.html. The example in the next section will illustrate the use of these predefined classes.
6.7 Program Example

The following example illustrates the use of the predefined `Segment_tree_2` class on two-dimensional interval data pair `<Point_2<Cartesian<double>>, Cartesian<double>>`. `Point_2` is the CGAL data structure for two-dimensional points in Cartesian or homogeneous representation. After including the necessary include files, the representation system and number type of the coordinates is defined. Then, the type of the traits class and the type of the `Segment_tree_2` class is defined. In the main function the tree data is inserted into a list. After that, the tree is built according to that data in the list and a `window_query` is performed on the tree.

```cpp
#include <CGAL/Segment_tree.h>
#include <CGAL/Range_segment_tree_traits.h>

/* the coordinates are represented in the Cartesian system */
typedef CGAL::Cartesian<double> Representation;
typedef CGAL::Range_segment_tree_set_traits_2<Representation> Traits;
/* definition of the two-dimensional segment tree */
typedef CGAL::Segment_tree_2<Traits> Segment_tree_2_type;

int main(){
  typedef Traits::Interval Interval; /* is a pair<Key,Key> */
  typedef Traits::Key Key; /* is a Point_2<Rep> */
  std::list<Interval> InList, OutList;
  /* insertion of the tree elements into the sequence container */
  InList.push_back(Interval(Key(1,5), Key(2,7))); /* ... further insertions */
  /* construction of the tree */
  Segment_tree_2_type Segment_tree_2(InList.begin(),InList.end());
  /* perform a window query */
  Interval a=Interval(Key(3,6), Key(7,12));
  Segment_tree_2.window_query(a,std::back_inserter(OutList));
}
```

This example shows that the predefined tree classes are very easy to use. By providing a user-defined traits class, the user can adapt the predefined `Range_tree_1`, ..., `Range_tree_4` and `Segment_tree_1`, ..., `Segment_tree_4` classes to work with arbitrary data types.
6.8 Conclusion

We presented the design of the range and segment tree data structures in Cgal. We showed that the design is flexible enough to easily define a general-dimensional range or segment tree. Even the construction of a mixed multilayer tree is possible with this design. The predefined Range_tree_1, ..., Range_tree_4 and Segment_tree_1, ..., Segment_tree_4 classes allow the user to adapt the one- to four-dimensional range and segment tree to arbitrary Data. The user only has to provide a Traits class which builds the interface between the tree and the data.

Note that the design is extendible. E.g., one can derive an interval tree from class Tree_base which can be used to define a mixed multilayer tree which is an interval tree in the d-th sublayer.

We now evaluate the design with respect to our design goals from Section 6.4.

Independence of the data type Data and Keys are only accessed through their Traits class. For this, the tree can be adapted to work with Data types from applications that already exist. The functionality which has to be defined in the Traits class is minimal. Only an access function and a compare function has to be defined for the data in each dimension.

Independence of the input/output sequence container In use of member template functions the tree works with arbitrary input/output sequence containers.

Arbitrary dimension The design allows to construct d-dimensional range and segment trees as well as mixed multilayer trees of arbitrary dimension. The definition of a d-dimensional tree is very similar to the definition of a (d − 1)-dimensional tree.

Query types The range tree supports range queries and the segment tree supports inverse range queries, range queries, and enclosing queries.

Ease of use The design is well-structured and has the look and feel of Cgal. The predefined tree and traits classes make the tree easy to use in standard applications and serve as examples for the definition of more sophisticated tree structures.

Efficiency, correctness, and robustness The implementation is type secure and well tested. Robustness problems are prevented by the use
6.8. CONCLUSION

of the traits technique: The user can use exact arithmetic in the case of numerical instabilities.
Chapter 7

R-tree and R*-tree-like Index Structures

7.1 Introduction

In the section about range and segment trees we pointed out that there are numerous applications for data structures for spatial objects that efficiently support range queries. The R-tree and its variant, the R*-tree, is a data structure that can be used to manage and query general-dimensional spatial data in external memory. The tree structure is dynamic in the sense that insertions, deletions, and range searches can be intermixed. Although it is not possible to guarantee good worst case performance, empirical tests show that the structure performs well. The R-tree was introduced by Antonin Guttman in 1984 [Gut84]. Beckmann et al. [BKSS90] developed the R*-tree, which empirically outperforms the R-tree.

In this section we explore the specification of the R-tree and R*-tree-like index structure that we implemented for CGAL. We start with a description of the R-tree (see Section 7.2) and of the R*-tree (see Section 7.3). Under consideration of the general design goals in CGAL (see Section 5.3) and the definition of the R-tree and R*-tree, we formulate the design goals for an implementation of the index structure (see Section 7.4). After that, we discuss related work which we evaluate according to our design goals (see Section 7.5). Our design is then presented in Section 7.6. Important implementation details are discussed in Section 7.7. Finally, we give an example program in order to give a better understanding of the design (see...
Chapter 7. R-Tree and R*-Tree-Like Index Structures

An evaluation of our design according to the design goals is given in the conclusion (see Section 7.9).

7.2 Definition of an R-tree

The R-tree [Gut84] is a hierarchical data structure that can be viewed as a multidimensional generalization of a B*-tree [Com79]. It can be used to organize arbitrary $d$-dimensional geometric objects by representing them by their bounding boxes. A bounding box of a $d$-dimensional geometric object is the smallest $d$-dimensional axis-parallel rectangle that encloses the geometric object. Each node in the tree corresponds to the smallest $d$-dimensional rectangle that encloses its child nodes. The leaf nodes contain the geometric objects. The tree is balanced in the sense that all leaf nodes appear on the same level. See Figure 7.1 for an example. The R-tree index structure is completely dynamic, it supports insertions, deletions, and searches, which can be intermixed. Note that rectangles corresponding to different nodes may overlap, and an object can be spatially contained in several nodes. Since each geometric object is associated with exactly one node, a spatial query often requires several nodes to be visited before answering the query.

If the data structure is stored in external memory, each node is stored as a disk block on disk. Often the size of a node is a fraction of the size of the internal memory. Therefore, the R-tree has variable parameters that define the minimum and maximum size of a node. Let $M$ be the maximum number of entries that fit in one node and let $m \leq \frac{M}{2}$ be the minimum number of entries in a node. Then, each node except for the root node has between $m$ and $M$ entries and the root node has between 2 and $M$ entries unless the root node is a leaf.

We now describe the search, insertion, and deletion algorithms for the R-tree.

7.2.1 R-tree Searching

For a given $d$-dimensional query rectangle, the search algorithm descends the tree from the root to all child nodes that match the query rectangle. Since more than one subtree of a visited node may need to be searched, it is not possible to guarantee good worst-case performance. Therefore, it is important that the rectangles corresponding to different nodes have small area. The procedure SEARCH describes the search algorithm, which returns
Figure 7.1: The upper left figure shows the bounding boxes of the data that is to be stored in the R-tree. The upper right figure shows how the data is partitioned into two sets R1 and R2. The lower left figure shows how the data of the partitions R1 and R2 is further partitioned. The lower right figure shows the corresponding R-tree.
all geometric objects for which the corresponding $d$-dimensional rectangle overlaps the query rectangle $S$.

**Procedure 13** \texttt{Search(root node } T\text{, query rectangle } S)\texttt{)}

1) if $T$ is not a leaf then
2) for each entry $E$ of $T$ for which the corresponding rectangle overlaps $S$ call \texttt{Search}($E.S$);
3) else
4) return all entries of $T$ for which the corresponding rectangle overlaps with $S$;

### 7.2.2 R-tree Insertion

A geometric object is inserted into a leaf node of the tree. Therefore, the tree is traversed, starting at the root node, and choosing a subtree of the node where the new object fits in "best". Here, "best" means to choose the node whose area enlargement to include the geometric object increases least. Overfull nodes are split and splits are propagated up the tree. The procedure \texttt{Insert} describes the insertion procedure. The leaf node in which the new object is to be inserted is chosen in procedure \texttt{ChooseSubtree}.

**Procedure 14** \texttt{Insert(geometric object } g\texttt{)}

1) call \texttt{ChooseSubtree}(g) to select a leaf node $L$ in which to place $g$;
2) if $L$ has room for another entry then
3) insert $g$;
4) else
5) call \texttt{SplitNode} to obtain $L$ and $R$ containing $g$ and all the old entries of $L$;
6) call \texttt{AdjustTree} on $L$, also passing $R$ if a split was performed;
7) if the root was split then
8) create a new root whose children are the two resulting nodes:

In procedure \texttt{AdjustTree}, the tree is ascended from a leaf node $L$ to the root, adjusting covering rectangles and propagating node splits as necessary.
7.2. DEFINITION OF AN R-TREE

Procedure 15  \textit{ChooseSubtree} (geometric object $q$)

1) Set $N$ to the root node;
2) while $N$ is not a leaf do
3) Let $F$ be the child node of the entry in $N$ which needs least area enlargement to include the rectangle of $q$; resolve ties by choosing the entry with the rectangle of smallest area; set $N = F$;
4) return $N$;

Procedure 16  \textit{AdjustTree} (node $L$, node $R$)

01) let $P$ be the parent node of $L$. Adjust the covering rectangle of $L$'s entry in $P$;
02) if $L$ was split previously then {
03) create a new entry $E_R$ pointing to $R$ with the smallest rectangle that encloses the entries of $R$;
04) if $P$ has room for another entry then
05) add $E_R$ to $P$;
06) else
07) call \textit{SplitNode} to produce $P$ and $PR$ containing $E_R$ and all of $P$'s old entries;
08) }
09) if $P$ is not the root then
10) call \textit{AdjustTree} with $L = P$ and $R = PR$ if a split occurred.

7.2.3 R-tree Split

When adding a new entry to a full node, the node has to be divided into two. The division should make it unlikely that both nodes have to be examined in a search. Since a node has to be examined in a search when the bounding rectangle overlaps the query rectangle, Guttman aims at minimizing the total area of the two covering rectangles of the two nodes after the split. This criterion is used in the procedure \textit{ChooseSubtree}. The best known algorithm that solves this problem has a runtime of $O(n^3)$ [BGO+92]. Since this algorithm is too slow, Guttman suggested a quadratic cost split algorithm and a linear cost split algorithm. The quadratic cost split algorithm is given in procedure \textit{SplitNode}, \textit{PickSeeds}, and \textit{PickNext}. We will not describe the linear cost split algorithm since it is outperformed
CHAPTER 7. R-TREE AND R*-TREE-LIKE INDEX

STRUCTURES

by the quadratic cost split algorithm and since both algorithms are very similar.

**Procedure 17**  
\textbf{SPLIT\_NODE} \((E_1, \ldots, E_{M+1})\)

1. call \textit{PickSeeds}, passing the entries, \(L\), and \(R\), to choose two entries to be the first entries of the groups \(L\) and \(R\);
2. do{
3.   if all entries have been assigned then return;
4.   if one group has so few entries that the remaining entries have to be assigned to it then assign them and return;
5.   call \textit{PickNext}, passing the entries, \(L\), and \(R\), to choose the next entry to assign. Assign it to the group whose covering rectangle will have to be enlarged least to accommodate it. Resolve ties by adding the entry to the group with smaller area, then to one with fewer entries;
6. }while (not all entries have been assigned):

**Procedure 18**  
\textbf{PickSeeds} \((E_1, \ldots, E_{M+1})\)

1. for each pair of entries \(a, b \in \{E_1, \ldots, E_{M+1}\}\) do {
2.   let \(c\) be the enclosing rectangle of the rectangles of \(a\) and \(b\);
3.   calculate \(d = \text{area}(c) - \text{area}(\text{rectangle}(a)) - \text{area}(\text{rectangle}(b))\);
4. } return the pair with the largest \(d\) (most wasteful pair):

7.2.4 R-tree Deletion

In order to delete an object in the tree, the object has to be found and deleted from its leaf. The enclosing rectangles have to be adjusted and under filled nodes are inserted into a list and then deleted from the tree. After that, all entries of under filled nodes are reinserted into the tree. Note that an entry from a higher level is reinserted into a node which is on the same level in the tree as the entry was.
7.2. DEFINITION OF AN R-TREE

Procedure 19  **PickNext**  \( E_1, \ldots, E_{M+1}, \text{groups } L, R \)

1) for each entry not yet in a group do
2)   let \( d_1 \) be the area increase required in the covering rectangle of \( L \)
     and \( a \);
3)   let \( d_2 \) be the area increase required in the covering rectangle of \( R \)
     and \( a \);
4) }
5) return the entry with the maximum difference between \( d_1 \) and \( d_2 \);

Procedure 20  **Deletion**  \( E \)

1) call **FindLeaf** to locate the leaf node \( L \) containing \( E \);
2) if **FindLeaf** did not find the leaf then return;
3) remove \( E \) from \( L \);
4) call **CondenseTree** \( L \);
5) if the root node has only one child, make the child the new root;

Procedure 21  **FindLeaf**  \( \text{root node } T, \text{element } E \)

1) if \( T \) is not a leaf then
2)   for each entry \( P \) of \( T \) for which the corresponding rectangle
     overlaps \( C \) call **FindLeaf** \( T, E \);
3) else
4)   return the entry of \( T \) for which the corresponding rectangle
     matches \( E \);

Procedure 22  **CondenseTree**  \( \text{node } L, \text{list } Q \)

1) while \( L \) is not the root do {
2)   let \( P \) be the parent from \( L \) and \( E_E \) E's entry in \( P \);
3)   if \( E \) has fewer than \( m \) entries then
4)     delete \( E_E \) from \( P \) and add \( E \) to \( Q \);
5) else
6)     adjust the rectangle of \( E_E \) to tightly contain all entries in \( E \);
7)     call **CondenseTree** \( P, Q \);
8) }
9) reinsert all entries of nodes in \( Q \). Entries from level \( k \) are
    reinserted in nodes of level \( k \).
7.3 Definition of an R*-tree

Beckmann et al. [BKSS90] proposed the R*-tree — a variant of the R-tree — that empirically outperforms the R-tree. The \texttt{ChooseSubtree} strategy and the \texttt{SplitNode} strategy in the R-tree aim at minimizing the area of the enclosing rectangle of each node. In the R*-tree a combined optimization of area, margin, and overlap of the enclosing rectangle of each node is the key goal. Furthermore, parts of overfilled nodes are sometimes reinserted. Reinsertions have the effect that the tree structure becomes reorganized. The index of the R-tree is sensitive to the input order, i.e. different sequences of insertions will build different trees. Therefore, reinsertions reorganize old entries in places where they fit best with respect to the current index structure.

Beckmann et al. present the following optimization criteria for the enclosing rectangles of a node:

\begin{description}
\item[dead space] The area covered by the bounding rectangle of a node but not covered by the enclosed rectangles is called \textit{dead space}. The dead space of a rectangle of a node is supposed to be minimized.
\item[overlap] The overlap between the rectangles of different subtrees is supposed be minimized. Let $R_1, \ldots, R_p$ be the rectangles of the entries $E_1, \ldots, E_p$ in a node. Then, the overlap of an entry $E_k$ is defined as follows:

\[
\text{overlap}(R_k) = \sum_{i=1, i \neq k}^{p} \text{area}(R_k \cap R_i) \quad 1 \leq k \leq p
\]
\item[margin] The margin is the sum of the lengths of the edges of a rectangle. Minimizing the margin instead of the area yields that the rectangles of nodes tend to be square shaped.
\item[storage utilization] Higher storage utilization generally reduces the query cost since the height of the tree is kept low.
\end{description}

The search and deletion algorithm for R-trees and R*-trees are identical. Now, we describe the insertion and split algorithm for the R*-tree.

7.3.1 R*-tree Insertion

Beckmann et al. found out that only a very local reorganization of the rectangles of a subtree is performed during a split. Furthermore, tests
showed that deletions and reinsertions of old data rectangles improved the query performance enormously. To achieve dynamic reorganization, the R*-tree forces entries to be reinserted during the insertion procedure. The procedure $\text{Insert}^*$ describes the insertion procedure which calls procedure $\text{OverflowTreatment}^*$ whenever a node is overfilled. Either procedure $\text{Reinsert}^*$ or $\text{SplitNode}^*$ are then called to treat the overfilled node. Analogously to the $\text{Insert}$ procedure for the R-tree, procedure $\text{AdjustTree}^*$ is called at the end of procedure $\text{Insert}^*$.

**Procedure 23** $\text{Insert}^*(\text{geometric object } g)$

1. call $\text{ChooseSubtree}^*(g)$ to select a leaf node $L$ in which to place $g$;
2. if $L$ has room for another entry then
   1. insert $g$;
   2. else
   3. call $\text{OverflowTreatment}^*(g)$;
   4. if $\text{OverflowTreatment}^*$ was called and a split was performed then
      1. propagate $\text{OverflowTreatment}^*$ upwards;
   5. call $\text{AdjustTree}^*$ on $L$, also passing $R$ if a split was performed;
6. if the root was split then
   1. create a new root whose children are the two resulting nodes;

The difference between the $\text{ChooseSubtree}$ procedure of the R-tree and the $\text{ChooseSubtree}^*$ procedure of the R*-tree is the following: In the procedure $\text{ChooseSubtree}^*$ the entry of a node that needs the least overlap enlargement to include the new data rectangle is chosen in case that the children of the node are leaves. For nodes that are no immediate parents of leaves, the entry with the least area enlargement is chosen, as in the procedure $\text{ChooseSubtree}$ of the R-tree.

### 7.3.2 R*-tree Split

The split method in the R*-tree works as follows. For each dimension the corresponding interval data of the entries are sorted by the lower value, then by the upper value. For each sorted sequence of entries, the $M - 2m + 2$ distributions of the $M + 1$ entries into two groups are determined. In the $k$-th distribution, the first group contains the first $(m - 1) + k$ entries and the
CHAPTER 7. R-TREE AND R*-TREE-LIKE INDEX
STRUCTURES

Procedure 24  ChooseSubtree\textsuperscript{*} (geometric object g)

1) Set N to the root node;
2) while N is not a leaf do {
3)   if N's children are leaves then
4)     Let F be the entry in N whose rectangle needs least
      overlap enlargement to include the new data rectangle.
      Resolve ties by choosing the entry whose rectangle needs
      least area enlargement; set N = F;
5)   } else
6)     Let F be the child node of the entry in N which needs least
      area enlargement to include the rectangle of g; resolve
      ties by choosing the entry with the rectangle of smallest
      area; set N = F;
7) }
8) return N;

Procedure 25  OverflowTreatment\textsuperscript{*} (node L)

1) if the level of L is not the root level and this is the first call of
   OverflowTreatment\textsuperscript{*} in the given level then
2)   call ReInsert\textsuperscript{*}(L);
3) else
4)   call Split\textsuperscript{*}(L);

second group contains the remaining entries. Each distribution is scored in
respect to its area, margin, and overlap values which are defined as follows:

area-value  \[ \text{area}(	ext{bb(first group)}) + \text{area}(	ext{bb(second group)}) \]
margin-value \[ \text{margin}(	ext{bb(first group)}) + \text{margin}(	ext{bb(second group)}) \]
overlap-value \[ \text{overlap}(	ext{bb(first group)}) + \text{overlap}(	ext{bb(second group)}) \]

Here bb denotes the bounding box of a set of rectangles. The split procedure
is given in procedure Split\textsuperscript{*}, ChooseSplitAxis\textsuperscript{*}, and ChooseSplitIndex\textsuperscript{*}. 
7.3. DEFINITION OF AN R*-TREE

Procedure 26  ReInsert* (node N, parameter \( p = 0.5 \))

1) for all \( M + 1 \) entries of \( N \), compute the distance between the centers of their rectangles and the center of the bounding rectangle of \( N \);
2) sort the entries in decreasing order of their distances;
3) move the first \( p \) entries from \( N \) to a list \( L \) and adjust the bounding rectangle of \( N \);
4) for each entry \( e \) in \( L \), starting with the entry with minimum distance do
5)  call Insert*\((e)\);

Procedure 27  AdjustTree* (node \( L \), node \( R \))

01) while \( L \) is not the root do{ 02)  let \( P \) be the parent node of \( L \). Adjust the covering rectangle of \( L \)'s entry in \( P \);
03)  if \( L \) was split previously then { 04)    create a new entry \( E_R \) pointing to \( R \) with the smallest rectangle that encloses the entries of \( R \);
05)    if \( P \) has room for another entry then 06)      add \( E_R \) to \( P \);
07)      else 08)        call OverflowTreatment*\((P)\);
09)  } 10)  set \( R \) to the second split entry, if a split occurred and set \( L = P \);
11})

Procedure 28  Split* (node \( L \))

1) call ChooseSplitAxis*() to determine the axis perpendicular to which the split is performed;
2) call ChooseSplitIndex*() to determine the best distribution into two groups along that axis;
3) Distribute the entries in two groups:
**Procedure 29** \texttt{ChooseSplitAxis}(\texttt{x})

1) for each \texttt{axis} do {

2) sort the entries by the lower then by the upper value of their rectangles and determine all distributions as described above;

3) determine \( S \), the sum of all margin-values of the different distributions;

4) }

5) choose the axis with the minimum \( S \) as split axis;

---

**Procedure 30** \texttt{ChooseSplitIndex}(\texttt{x})

1) Along the chosen axis, choose the distribution with the minimum overlap-value. Resolve ties by choosing the distribution with minimum area-value.
7.4 Design Goals

We took the following design issues into account for the design of the spatial index structure.

**Independence of the data structure**

- The tree is supposed to be easily adaptable to data types in applications that already exist. In particular, the tree should be able to manage data types that contain pointers or have variable size.
- The key type should be flexible. That is, user defined key types like the bounding box, the smallest enclosing $d$-dimensional ball or ellipsoid should be possible key types.
- The set of properties a key has to fulfill should be minimal.

**Independence of the database** The database in which the nodes and leaves of the tree are stored is supposed to be adaptable to existing databases. The user might want to connect her R-tree with an existing database. Therefore, the database should be easily exchangeable and the functionality a database has to provide should be minimal.

**Flexibility of the index structure** The strategies that are responsible for the index structure should be easily exchangeable. Since many different strategies for the *ChooseSubtree* procedure and *Split* procedure have been proposed, the user might want to test different strategies or run experiments with own defined indexing strategies.

**Query types and extensibility** The query types should be extensible. The user should be able to define her own query predicates. The use of user defined keys can make new query predicates necessary.

**Ease of use** The design should be well structured, easy to use, and have the look and feel of CGAL.

**Correctness and robustness** The implementation should be type secure and robust with respect to degeneracies in the input.

7.5 Related Work

We classify three different kinds of R-tree implementations. First of all there are many implementations of the R-tree and variants that were
CHAPTER 7. R-TREE AND R'-TREE-LIKE INDEX STRUCTURES

implemented by their developers in order to evaluate their run time performance. These implementations do not meet our design goals, since they usually are very specific and not designed to be reused. Therefore, we do not discuss this kind of implementations. Then, some databases contain an R-tree implementation. Examples are PostGres\(^1\), Illustra\(^2\), GOODS\(^3\), and Post++\(^4\). These implementations are specified to work with the individual database and thus they were also not designed to be used in other contexts. Furthermore, there often is no documentation available for these implementations. Because of this, we do not discuss this kind of R-tree implementations. The last kind of R-tree implementation is designed to be adaptable and extendible in order to be reused in different applications. There is only one implementation made public that belongs into this category. It is the Generalized Search Trees\(^5\) (GiST) [HNP95]. We evaluate the GiST according to the design goals presented in Section 7.4.

7.5.1 Generalized Search Trees (GiST) for Database Systems

The GiST [HNP95] is an index structure that unifies disparate index structures, like the B+tree and the R-tree. This index structure can be adapted by a couple of methods that are general enough to make the tree behave like e.g. a B+-tree, an R-tree, or an RD-tree. The library is programmed in C++, using the object oriented programming paradigm.

In order to handle arbitrary data types for the data and the key, the data types are cast to \(\text{void}^*\). GiST provides predefined implementations for the B+-tree and a version of the R-tree. If these extensions do not meet the requirements, one can implement the Simple Extension Interface which implements so-called key functions, e.g. consistent, penalty, and union_key. An implementation of the Node-Layout Extension Interface gives the user control over the so-called tree functions, e.g. insert, updateKey, search, and pickSplit.

There exists a graphical development tool built in the GiST framework that is called A visual Access Method Development Tool (Amdb) [KSH98]. Amdb provides debugger-like functionality at the level of basic tree actions like node traversal, node split, node update, etc. The tree is visualized by

---

\(^1\)http://www.postgresql.org/
\(^2\)http://www.illnestra.com/
\(^3\)http://www.ispras.ru/~knizhnik/goods.html
\(^4\)http://www.ispras.ru/~knizhnik/post/readme.htmX
\(^5\)http://epoch.cs.berkeley.edu:8000/gist/
7.5. RELATED WORK

a hierarchy of visualization tools showing the tree, a subtree and the data of chooseable nodes.

In respect to our design goals we have the following annotations to make:

**Independence of the data structure**  The tree can handle data types and key types that already exist, as long as the data and key is stored in one memory location. Thus, data types that contain pointers referring to data types can not be handled. The data types for data and key is simply cast to `void *`. However, in order to generate an R-tree with keys different to general-dimensional points or rectangles, an R-tree extension has to be implemented. The inner nodes and the leaf nodes of the tree contain the same number of elements. This has disadvantages, when the data types are of much bigger size than the key types, since the fanout of the tree is limited by the number of data items that fit in a leaf.

**Independence of the database**  By an implementation of the Node-Layout Extension Interface, the user can control page-oriented actions. Among these action are insertions and deletions of pages.

**Flexibility of the index structure**  An index structure different from the predefined R-tree index structure can be defined by an implementation of the Simple Extension Interface and or the Node-Layout Extension Interface.

**Query types and extensibility**  The query predicates in the predefined R-tree implementation are `contained`, `contains`, `equal`, and `overlaps`. By providing a Simple Extension Interface, an arbitrary set of additional query types can be implemented.

**Ease of use**  The abstraction from specific search trees to the general search tree lead to a nice and well structured set of so called `key methods` and `tree methods`. This abstraction helps to understand the package and to design an own index structure. Furthermore, the visualization tool Amdb is probably of great help. One disadvantage is that when the functionality of the tree is to be changed slightly, a whole implementation of the Simple Extension Package or the Node-Layout Extension package has to be provided. A more modular structure of the extension packages would avoid this.

**Correctness and robustness**  Since the data types for the data and the key are simply cast to `void *`) neither type checking at compile time nor at run time is possible. This certainly is insecure.
7.6 Design Overview

Figure 7.2 illustrates the different components of the R-tree.

The R-tree class gets three traits classes as template arguments. The R-tree_traits class defines the interface to the Data objects, Keys and required methods of the Data and Keys. The class R-tree_index defines all index strategies. The R-tree is attached to two databases, one for the inner nodes of the tree and one for the leaf nodes. The class R-tree_storage builds the interface between the database for the inner tree nodes and the leaf nodes. We now describe each component in more detail. Note, that we provide example classes or implementations of the most important or most commonly used strategies for each component.

7.6.1 Data handling

In this section we describe how data is handled in the tree.

Tree Data

The tree is designed to handle arbitrary spatial objects (called Data) in arbitrary dimension. Each node of the tree stores a Key which usually is the bounding box of the Keys of its children. The tree is designed to be independent of the form of a Key. That is, a Key can have arbitrary dimension and arbitrary form (e.g. a $d$-dimensional bounding box, the smallest enclosing $d$-dimensional ball, etc). Nevertheless, the Key class
has to provide certain properties: E.g., for two Keys it must be decidable whether one Key is contained within the other one, whether they intersect and what the smallest Key is that encloses both Keys. In addition to access functions a cost-function has to be provided that measures the inefficiency when clustering two Keys together. This allows the user to build the R-tree with respect to different optimization functions such as minimum area, overlap, or margin enlargement when clustering two Keys together. The Data and Key functionality is accessed through the R_{tree_traits} class, only. Therefore, the functions in this traits class can be defined flexible. CGAL provides predefined Key classes for k-dimensional axis parallel rectangles, $1 \leq k \leq 4$.

**R_{tree_traits}**

The Data and Key functionality is accessed through a traits class named R_{tree_traits}. This traits class decouples the R-tree from the Data and Key class such that a modification of one of these classes only affects this traits class but not the tree class itself. Among the member functions of this traits class are a compare function that defines a search predicate, a cost function that returns a penalty for clustering two Keys together, and an intersection function that returns true if and only if two Keys intersect.

### 7.6.2 R_{tree_index}

The class R_{tree_index} defines how the data is spatially clustered in the tree. We decoupled the index strategies from the implementation of the tree since many different index strategies have been proposed for the R-tree and the R*-tree. CGAL provides two R_{tree_index} classes that can be "plugged into" the tree: The R_{tree_index} class which is the index strategy as proposed by Guttman, and the R_{star_tree_index} class which is the R*-tree index strategy as proposed by Beckmann et al. The R_{tree_index} class consists of an implementation of the procedure CHOOSE_SUBTREE and the procedure SPLIT. The R_{star_tree_index} class consists of an implementation of the procedure CHOOSE_SUBTREE*, the procedure SPLIT*, and the procedure REINSERT*. Note that for a user defined traits class, all index strategies can be intermixed arbitrarily among each other and with user-defined strategies.

### 7.6.3 R_{tree_storage}

The R_{tree_storage} class is a traits class in which all storage dependent components of the R-tree are defined. These are a database for the leaf
nodes, a database for the inner nodes of the tree, the capacities of the nodes in form of the number of elements and the size of a page, etc. 

CGAL provides two predefined \( R \)-tree storage classes called \( R \)-tree internal storage and \( R \)-tree external storage that can be plugged into the tree. We now give a short description of the components of the \( R \)-tree storage class.

Databases The \( R \)-tree has two databases. One for the inner nodes of the tree and one for the leaf nodes.

Each database is decoupled from the \( R \)-tree storage class by an \( IO \)-traits class. CGAL provides two databases for the \( R \)-tree. One stores the data and the tree in internal memory. The other database is an external memory database which allocates a memory cache for \( k \) pages of size \( IO \)-page size, where \( k \) is an integer template argument. The cache uses the LRU strategy ("least recently used") to load and unload pages. Note that further databases can be plugged into the tree by providing an appropriate \( IO \)-traits class.

Node and Leaf Capacities The \( R \)-tree storage class gets four integer template arguments. The minimum capacity (\( IO \)-min_cap_nodes) and maximum capacity (\( IO \)-max_cap_nodes) has to be defined for an inner node of the tree. Each inner node is guaranteed to have between \( IO \)-min_cap_nodes and \( IO \)-max_cap_nodes Key entries; the root node has between two and \( IO \)-max_cap_nodes entries. For the leaf nodes \( IO \)-min_cap_leaves defines the minimum number of elements in a leaf and \( IO \)-page size defines the maximum size of all Data entries of a leaf.

headerextern headerextern is a boolean template argument. If \( header \_extern = \text{true} \), a header file containing necessary informations for the reconstruction of a tree from an external database is stored on disk. Otherwise, the header file is not stored on disk.

Thus, the class \( R \)-tree internal storage defines all storage dependent components of an \( R \)-tree variant that is kept in internal memory and \( R \)-tree external storage defines all storage dependent components of an \( R \)-tree variant that is kept in external memory.

7.6.4 More Design Features

The flexibility described in the previous section is gained in use of template classes and the traits technique. Here we describe some further features of the design.
reinsertion control  For each level of the tree, a reinsertion flag can be set to \textit{true} or \textit{false}. If that the reinsertion flag of a level is true, the \texttt{reinsertion\_node (reinsertion\_leaf)} routine will be called for an overfull node of this level (instead of the split routine). When the \texttt{reinsertion\_node (reinsertion\_leaf)} routine is finished, the reinsertion flag is again set to \textit{false} for that level.

output sensitive query function  The query functions are output sensitive. That is, we provide iterators that allow to intermix user code with querying. Since the amount of query output can exceed the main memory capacity there is no alternative to providing output sensitive query functions.

Please refer the user manual for more information [Ney00a].

\section*{7.7 Implementation Details}

In this section we describe some selected implementation details which are not straightforward.

One of our design goals was to make it possible that the tree handles \textit{Data} objects of variable size. This has the impact that the number of objects that fit in one leaf is variable. Therefore, the size of a leaf is determined by the parameter \texttt{IO\_page\_size} which is defined in the class \texttt{R\_tree\_storage}. However, the size of a \textit{key} is fixed and the number of \textit{keys} that fit in one node is defined by the \texttt{R\_tree\_storage} parameter \texttt{IO\_max\_cap\_nodes}. Thus, an inner node of the tree and a leaf node may contain a different number of different data types. Nevertheless, for the tree both kinds of nodes have to be treated equally in some situations and different in some other situations. In order to realize this, we defined an abstract virtual base class \texttt{abstract\_node\_entry} from which we derived the classes \texttt{node\_entry}, which is the type of an inner node of the tree, and \texttt{data\_entry}, which is the type of a leaf node. Since all functions in the class \texttt{abstract\_node\_entry} are virtual, a tree, formally consisting of nodes of type \texttt{abstract\_node\_entry}, but actually having nodes of type \texttt{data\_entry} and \texttt{node\_entry}, can be defined. Note that some tree methods depend on the type of a node. E.g., since inner nodes and leaf nodes are stored in different databases, a \texttt{write\_to\_database} method which calls database methods to store a node, needs to know the actual type of the node. Since the \texttt{write\_to\_database} function is not a member function of the node, the “lost” type of the node has to be
recovered. In this case the type of the node can be recovered by \textit{dynamic_casts}. The use of \textit{dynamic_casts} inhibits type checking at compile time. We used \textit{dynamic_casts} only in some necessary places, therefore, type checking at compile time is performed at the major part of the implementation.

Another implementation detail we would like to discuss is how data is written to pages that are managed by the database. Usually, a memory copy \texttt{memcpy} of the Data object is performed. This yields a binary presentation of the object. Unfortunately, this only works when the Data object is located at one position in memory. Data objects that contain pointers to secondary structures can not be saved this way, since the secondary structure will not be copied. Therefore, each time a Data object is written to its page, an \texttt{R-tree_traits} method named \texttt{write} is called that has to be provided by the user. Analogously, a \texttt{read} function that reads a Data object as well as a \texttt{write_key} and a \texttt{read_key} function that writes (resp. reads), a Key object have to be provided. This strategy also has the advantage that the user can define I/O methods as e.g. a for human readable representation or a compressed version of the Data or Key.

### 7.8 Program Example

The following piece of code defines an R*-Tree. The \texttt{Data} type simply defines an \texttt{R} \_\texttt{tree}_\_\texttt{key} \_\texttt{2} type as Key data type. The \texttt{R} \_\texttt{tree}_\_\texttt{traits} class implementation is instantiated with this \texttt{Data} type. As \texttt{R} \_\texttt{tree}_\_\texttt{index} class we chose the predefined \texttt{R} \_\texttt{star}_\_\texttt{tree}_\_\texttt{index} class, which consists of an implementation of the \texttt{CHOOSE}\_\texttt{SUBTREE}\_\texttt{2} procedure, the \texttt{SPLIT}\_\texttt{2} procedure, and the \texttt{REINSERT}\_\texttt{2} procedure. As \texttt{R} \_\texttt{tree}_\_\texttt{storage} class we chose the \texttt{R} \_\texttt{tree}_\_\texttt{external}_\_\texttt{storage} class. In this class, class \texttt{R} \_\texttt{tree}_\_\texttt{external}_\_\texttt{db} defines the database for the inner nodes and the leaf nodes, which is kept in external memory. The minimum and maximum capacities of a node are 2 and 4. The \texttt{IO}_\texttt{page}\texttt{size} is set to 70.

An R*-Tree is created and 16 two-dimensional squares are inserted. Then, we perform various queries and delete some elements.

This example is a bit lengthy since we first define the \texttt{Data} type and inserted many comments.

Example

```c++
#include <CGAL/Rtree.h>
#include <CGAL/Rtree_key.h>
```
7.8. PROGRAM EXAMPLE

```cpp
#include <CGAL/FLstar_tree_mdex.h>
#include <CGAL/R_tree_traits_example.h>
#include <CGAL/Rtree_external_storage.h>
#define NUMBER 16

/* Definition of the data type */
struct Data{
  public:
    typedef CGAL::Rtree::key_2 Key;
    Key key;
    size_t size(void) const {return sizeof(*this);}
    void read(char * s) {key.read(s);}
    void write(char * s) {key.write(s);}
    void dump(int level =0) {key.dump();}
};

/* definition of the R_tree_traits - depending on the Data type */
typedef CGAL::R_tree_traits<Data> TTraits;
typedef Data::Key Key;

/* definition of the R_tree that contains Data elements, uses R-star
  tree index structure and stores the elements in external memory */
typedef CGAL::R_tree<TTraits, CGAL::R_star_tree_index<TTraits>>,
  CGAL::R_tree_external_storage> RTreeInst;

int main() {
  TTraits traits;
  Data elem;
  int k;
  Key key= Key(0,2,0,2);
  std::vector<Data> source;
  /* creation of R_tree associated to the files: */
  RTreeInst r_star_tree("_star_tree.head","_star_tree.dat",
    "_star_leaf_data.dat");

  /* create 2-dimensional squares as input data */
  for (k=0;k<NUMBER;++k) {
    elem.key=key;
    source.push_back(elem);
    key.xmin++; key.ymin++; key.xmax++; key.ymax++;
  }
```
CHAPTER 7. R-TREE AND R-TREE-LIKE INDEX

STRUCTURES

```
/* Insertion of the data elements */
for (k=0; k<NUMBER; ++k) { r_star_tree.insert(source[k]); }

/* Iteration through all elements of the tree that have 
non empty intersection with source[0].key=(0.2,0.2) */
it_begin=r_star_tree.begin(source[0].key);
it_end=r_star_tree.end(source[0].key);
while(it_begin != it_end){
  (it_begin).dump();
  ++it_begin;
}
/* End of iteration through the query elements of the tree */

/* Iteration through all elements of the tree 
that enclose source[2].key=(2.0,2.4) */
it_begin=r_star_tree.begin.enclose(source[2].key);
it_end=r_star_tree.end.enclose(source[2].key);
while(it_begin != it_end){
  (it_begin).dump();
  ++it_begin;
}
/* End of iteration through the query elements of the tree */

Data data_del;
/* Deletion of all data with key source[0].key=(0.2,0.2) */
while(r_star_tree.delete.key(source[0].key, data_del))
  traits.dump(data_del.key);

/* Check for elements that intersect source[1].key=(1.2,1.2) */
if(!r_star_tree.find.key.intersect(source[1].key))
{
  std::cerr « "no key intersects";
  traits.dump(traits.build(source[1]));
}
```
7.9 Conclusion

We presented a framework for the construction of an R-tree, R*-tree, or a variant of the R-tree. For each component of the R-tree (\texttt{R.tree_traits}, \texttt{R.tree_index}, and \texttt{R.tree_storage}) there exist example classes or implementations of the most important or most commonly used strategies. These components can be composed to form many different R-tree variants.

The generic programming paradigm, together with the careful use of the object oriented programming paradigm has led to a well structured, modular, easy to use, and flexible design. We now evaluate the design according to the design goals from Section 7.4.

**Independence of the data structure** The \texttt{Data} and \texttt{Key} objects and the \texttt{Key} functionalities are accessed through the \texttt{R.tree_traits} class. Therefore, the tree can be adapted to work with \texttt{Data} types from applications that already exist.

- The \texttt{tree} can be adapted to work with arbitrary \texttt{Data} types. The data objects may contain pointers to secondary structures and can have variable size.
- The type of the \texttt{Keys} can be user defined.
- The set of properties a \texttt{Key} has to fulfill is as small as possible. The \texttt{Key} functionality is accessed through the \texttt{R.tree_traits} class which leaves maximal flexibility for the particular definition of these properties.

**Independence of the data base** Database functionality is only accessed through the \texttt{R.tree_storage} traits class. Furthermore, the functionality of the \texttt{R.tree_storage} class is kept as small as possible. Therefore, the tree should be easily adaptable to existing databases. Since it is possible to define different databases for the internal nodes and the leaf nodes, one can e.g. define an R-tree where the internal nodes are stored in internal memory and the leaf nodes are stored in external memory.

**Flexibility of the index structure** By the definition of the \texttt{R.tree_index} class, the user can define the index strategies by either using predefined strategies or her own implementations.

**Query types and extensibility** The query predicates are defined in the user defined traits class \texttt{R.tree_traits}. This traits class allows the user to implement the \texttt{intersection} predicate, which returns true if
two *Keys* intersect, and the *include* predicate which returns true if one *Key* is contained within the other. A third predicate *compare* can be freely defined. Thus the query types leave some space for extensibility. Nevertheless, one could think of an unbounded set of further user defined query predicates.

**Ease of use**  The design is well structured and has the look and feel of CGAL. The fact that many example implementations are provided makes it easy to use, if one e.g. applies the well-known copy-and-paste method.

**Correctness and robustness**  A sophisticated system of checks and debugging output that can be turned on and off has been defined. *dynamic_casts* are only used at very few — and, as we think appropriate — places. Therefore, type checking at compile time is performed in the major part of the implementation. Since the data type and the primitives on the data type (e.g. compare function) are user defined, the user can prevent robustness problems in use of exact number types.
Chapter 8

Conclusion

The case studies of this thesis give insight into various fields of Computational Geometry. In the first part of this thesis, we studied problems that arise in Computational Geometry from a theoretical view. The second part explores specific software design issues of data structures that support geometric queries.

At first, we investigated a spatial join scheduling problem. For two given rectangular partitions $A$ and $B$ of a rectangle $R$, we want to compute the join of $A$ and $B$. We assume that each rectangle of $A$ and $B$ corresponds to a disk page and only two pages fit in internal memory. In this case, a join scheduling corresponds to a sequence of disk accesses such that any two pages whose cells intersect meet in main memory at least once. We show that this problem corresponds to finding a Hamiltonian path in a graph that is constructed according to the two rectangular partitions $A$ and $B$. In the case where no two rectangles of $A$ and $B$ have (a part of) a boundary in common (except the boundary of $R$), the problem is solvable in polynomial time. On the other hand, we show the general problem to be $AP$-complete by reducing 3SAT to it. The proof is a late (but first) justification for heuristics for the spatial join scheduling problem. Since our proof makes heavy use of a special singularity, efficient algorithms might exist for various special cases of spatial join scheduling problems.

Line simplification with restricted orientation is the subject of our next investigation. Often maps such as those used to describe subway lines or bus plans are drawn $C$-oriented, that is, each line is drawn parallel to a set of orientations $C$ in $\mathbb{R}^2$. For a given polygonal chain $P$ we want to compute a $C$-oriented polygonal chain that consists of as few line segments
as possible and represents $P$ well. The Fréchet metric is a very good metric to measure the similarity of shapes. We give a dynamic algorithm that computes a $\mathcal{C}$-oriented polygonal chain that consists of a minimum number of line segments and has Fréchet distance at most $c$ to $P$ in polynomial time, for adequate $c$.

We complete the first part of this thesis with the study of a map labeling problem. For a rectangular grid of streets, the problem is to label all horizontal and vertical streets with their street names such that no two labels overlap. We give a backtracking algorithm that guarantees to solve every solvable instance. Empirically, the algorithm uses at most one backtracking step per branch and is therefore efficient. On the other hand, Unger and Seibert recently showed that this problem is NP-complete. We round up the presentation with polynomial time algorithms for three special cases where

- all labels have length at most half the length of their respective street,
- all vertical labels have the same length, and
- the grid is a square and each label has one of two label lengths.

The second part of this thesis documents my work for the CGAL-Project (Computational Geometry Algorithms Library). The CGAL project aims at providing a flexible, efficient, robust, and easy-to-use library of the most important algorithms in Computational Geometry. After describing the generic and object-oriented programming paradigms used in CGAL, we describe the design of our contributions to CGAL. In order to provide internal- and external-memory data structures that efficiently support geometric queries, we implemented the general-dimensional range, segment, and R-tree data structures. All data structures are designed with respect to the design goals of CGAL. Their evaluation shows that the design fulfills the flexibility, efficiency, robustness, and ease-of-use requirements of CGAL.
List of Algorithms

1  [MPS86] ChannelRouting () ..................................  36
2  Transformation () .............................................  39
3  BASIC COLS () ................................................  57
4  COLS () ..........................................................  64
5  Label () ..........................................................  83
6  HalfSolution () ..................................................  92
7  ConstantColumnLength () ....................................  94
8  TwoLabelLengthsI, \( l_1 + l_2 > n \) () .........................  98
9  TwoLabelLengthsII, \( l_1 + l_2 \leq n \) () ..................... 104
## List of Procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion ()</td>
<td>62</td>
</tr>
<tr>
<td>Update ()</td>
<td>63</td>
</tr>
<tr>
<td>Preprocessing ()</td>
<td>79</td>
</tr>
<tr>
<td>DrawConclusions ()</td>
<td>81</td>
</tr>
<tr>
<td>PositionAndBacktrack ()</td>
<td>84</td>
</tr>
<tr>
<td>Build-d-dimRangeTree ()</td>
<td>129</td>
</tr>
<tr>
<td>d-dimRangeQuery ()</td>
<td>130</td>
</tr>
<tr>
<td>FindSplitNode ()</td>
<td>131</td>
</tr>
<tr>
<td>Build-d-dimSegmentTree ()</td>
<td>135</td>
</tr>
<tr>
<td>BuildSublayerTrees ()</td>
<td>135</td>
</tr>
<tr>
<td>InsertInternal ()</td>
<td>135</td>
</tr>
<tr>
<td>InverseRangeQuery ()</td>
<td>136</td>
</tr>
<tr>
<td>Search ()</td>
<td>150</td>
</tr>
<tr>
<td>Insert ()</td>
<td>150</td>
</tr>
<tr>
<td>ChooseSubtree ()</td>
<td>151</td>
</tr>
<tr>
<td>AdjustTree ()</td>
<td>151</td>
</tr>
<tr>
<td>SplitNode ()</td>
<td>152</td>
</tr>
<tr>
<td>PickSeeds ()</td>
<td>152</td>
</tr>
<tr>
<td>PickNext ()</td>
<td>153</td>
</tr>
<tr>
<td>Deletion ()</td>
<td>153</td>
</tr>
<tr>
<td>FindLeaf ()</td>
<td>153</td>
</tr>
<tr>
<td>CondenseTree ()</td>
<td>153</td>
</tr>
<tr>
<td>Insert*( )</td>
<td>155</td>
</tr>
<tr>
<td>ChooseSubtree* ()</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>List of Procedures</td>
</tr>
<tr>
<td>---</td>
<td>------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>25</td>
<td>OverflowTreatment* ()</td>
</tr>
<tr>
<td>26</td>
<td>ReInsert* ()</td>
</tr>
<tr>
<td>27</td>
<td>AdjustTree* ()</td>
</tr>
<tr>
<td>28</td>
<td>Split* ()</td>
</tr>
<tr>
<td>29</td>
<td>ChooseSplitAxis* ()</td>
</tr>
<tr>
<td>30</td>
<td>ChooseSplitIndex* ()</td>
</tr>
</tbody>
</table>
Bibliography


Curriculum Vitae
Gabriele Neyer
born July 23, 1968 in Ibbenbürener, Germany

Education
1996-present  Doctorate in Computer Science, ETH Zürich
Examiner: Prof. Dr. Peter Widmayer, ETH Zürich
Co-Examiner: Dr. habil. Frank Wagner, FU Berlin
Research fields: Computational Geometry, Complexity, GIS
1991-1996  Diploma in Computer Science at the TU Berlin
1988-1991  Student of Computer Science at the Uni-GH Paderborn

Employment
1996-present  Research Assistant and Teaching Assistant at the Computer Science Department, ETH Zürich. Main activities:
- Co-Developer of CGAL (Computational Geometry Algorithms Library).
- Teaching in Theoretical Computer Science.
- Supervision of a diploma thesis in Computer Science.
- Co-Organizer of the 25th International Workshop on “Graph-Theoretic Concepts in Computer Science”, WG'99.
- Co-Organizer of the Schulpf_CNtium Informatik für Naturwissenschaften and the Mittelschülerintagung at the ETH Zürich for encouraging women to study Computer Science.
1993-1996  Software Engineer at the Institute of Applied Mathematics, TU Berlin, and at the Institute of Computer Science, Uni Konstanz: Co-Developer of PlaNet, a software package for the visualization of algorithms on planar graphs.
1992  Software Engineer at ABK-Systeme (Frankfurt and Berlin).