

Automated Label Placement in Theory and Practice

Dissertation of Alexander Wolff
Fachbereich Mathematik und Informatik
Freie Universität Berlin

Supervisor: Dr. habil. Frank Wagner

External referees: Dr. Marc van Kreveld, Universiteit Utrecht,
and Prof. Christopher Jones, University of Glamorgan.

Date of doctoral defense: May 28, 1999

Contents

Abstract	v
1 An Introduction to Label Placement	1
1.1 Historic Development	2
1.2 Theory	3
1.3 . . . and Practice	4
1.4 Quality	4
1.5 Future Development	5
1.6 Overview	6
1.6.1 General Labeling, Compatible Representatives, and CSP .	6
1.6.2 Point Labeling: Label-Number Maximization	7
1.6.3 Point Labeling: Label-Size Maximization	8
1.6.4 Line Labeling	8
1.6.5 Designing Geometric Algorithms	9
2 General Labeling: Label-Number Maximization	11
2.1 Label Placement and CSP	13
2.2 Maximum Variable-Subset CSP	14
2.3 Irreducibility	16
2.4 An Edge-Irreducibility Algorithm	18
2.5 A General Label-Placement Algorithm	28
3 Point Labeling: Label-Number Maximization	31
3.1 Comparing Various Models	33
3.2 Fixed-Position Models	42
3.2.1 Algorithm	43
3.2.2 Experiments	47
3.2.3 Results	49
3.3 Slider Models	61
3.3.1 NP-Hardness	62

3.3.2	A Greedy Approximation Algorithm	65
3.3.3	A Polynomial Time Approximation Scheme	71
3.3.4	Implementation and Experimental Results	75
4	Point Labeling: Label-Size Maximization	81
4.1	Rectangular Labels	81
4.2	Circular Labels	82
4.2.1	Previous Work	84
4.2.2	Preliminaries	84
4.2.3	Algorithm	86
4.2.4	Analysis	88
4.2.5	NP-Hardness	91
5	Line Labeling	97
5.1	Previous Work	99
5.2	A Buffer Around the Input Polyline	100
5.3	A Candidate Strip	102
5.4	Finding Good Label Positions	105
5.5	Experimental Results	106
5.6	Discussion and Extensions	110
6	Designing Geometric Algorithms	113
6.1	Algorithm	115
6.2	Step by Step Towards Good Design	115
6.2.1	The Naive Approach	116
6.2.2	Decoupling Algorithm and Data Organization	116
6.2.3	Tightening Control	118
6.2.4	Influencing Critical Decisions	119
6.2.5	The Complete Interface	121
6.3	Experiments	124
6.3.1	Example Classes	124
6.3.2	Results	126
6.3.3	Evaluation	127
	Conclusion	131
	Zusammenfassung (Summary in German)	133
	Curriculum Vitae	135
	Bibliography	139

Abstract

Placing labels that contain text or images is a common task in information visualization. Labels convey information about objects in graphical displays like graphs, networks, diagrams, or cartographic maps. Typically, each object (or *feature*) that has to be labeled allows a number of positions where the corresponding label can be placed. However, each of these label candidates could intersect with label candidates of other features. This gives rise to the following combinatorial problem, the *general label-placement decision problem*. Given a set of features, each with a set of label candidates, can we assign each feature a label from its candidate set such that no two labels intersect? In other words, is there a *complete labeling* for the set of features?

Unfortunately, one cannot expect to find an efficient method for answering this question. Therefore most previous work has focused on developing heuristics and approximation algorithms for various optimization problems related to label placement, such as finding a placement for a subset of the features of maximum cardinality that has a complete labeling. This problem is referred to as the *label-number maximization problem*.

In this thesis we approach label placement from two sides. On the one hand we develop a general framework for label-placement problems. This framework extends the classical constraint-satisfaction framework such that the label-number maximization problem can be formulated and algorithms can be developed that reduce the size of the search space for an optimal solution considerably. We give such an algorithm and a simple, but very effective heuristic based on this algorithm.

On the other hand we investigate special cases of the label-placement problem, which is generally divided into point, line, and area labeling. First we consider the point-labeling problem. We apply our framework to labeling points with axis-parallel rectangles, which can, for example, represent the bounding boxes of textual labels. We allow the classical four label positions per point, namely those where a corner of the rectangle coincides with the point. We compare our method to five other point-labeling algorithms experimentally. Then we investigate point labeling with an infinite number of label candidates per point. We show that it is NP-hard to decide whether a set of points can be labeled with unit squares or with unit disks if we require that each label touches its point. Nevertheless we give efficient approximation algorithms for optimization versions of both problems. More specifically we give a polynomial-time

approximation scheme for maximizing the number of axis-parallel rectangular labels of common height, while we show that one cannot expect to find such a scheme for maximizing the size of uniform circular labels.

For labeling polygonal chains like rivers or railway lines on maps, we study the cartographers' requirements and put them into two categories; hard and soft constraints. Then we give an efficient algorithm that guarantees to satisfy all hard constraints. In addition we show how to optimize the soft constraints. The method we suggest is the first that simultaneously fulfills both of the following two requirements: it allows curved labels and its runtime is at most quadratic in the number of points on the given polyline.

Apart from analyzing asymptotical runtime behavior and storage requirements of our algorithms, we implemented most of them and studied their performance on synthetic as well as real-world data. The last chapter of this thesis is devoted to generic programming, a method for abstracting from concrete data representation, that we found very helpful for keeping our geometric algorithms flexible.

Chapter 1

An Introduction to Label Placement

In everyday life, we permanently categorize and label things or people according to the categories into which they in our opinion fall. If we do not know somebody's name, we may refer to him by his physical appearance, his hair style, profession or possessions, the way he dresses, behaves, or talks. That is, we label him as “tall”, “brunette”, “poor”, “extroverted”, “southern”, or with several of these predicates. If we accumulate enough labels, we get a unique description. We use labels to identify, describe or simply store something in our memory. Labels never catch all information available on an object but rather focus on features that distinguish it from others. Spoken in terms of computer science, a label can be seen as a hash key that allows us to access additional information about the labeled object in the dictionary, which is represented by our brain.

We use labels to describe objects and to communicate our ideas to others, hoping that the hash key works in their dictionaries as well as in ours. In order to illustrate our ideas, we often resort to images, i.e. two-dimensional mappings of reality. These tend to consist of rather simple, and thus abstract graphical elements, which can have an abundance of meanings. Thus we must annotate these objects with some kind of labels to clarify our intentions. Such a labeling must fulfill two important requirements, namely (a) legibility, i.e. a label must be of sufficient size and must neither overlap objects of the image nor other labels, and (b) unambiguity, i.e. it must be clear which object a label annotates. The second requirement is also valid for the use of labels in speech, while the first results from the geometric limitations of the plane. From now on we will refer to objects that are to be labeled as *features*.

Labeling is one of the key tasks in the process of information visualization. In diagrams, maps, technical or graph drawings, features like points, lines, and polygons must be labeled to convey information. The interest in algorithms that automate this task has increased with the advance in type-setting technology and the amount of information to be visualized. Cartographers, graph

drawers, and computational geometers have suggested solutions to the labeling problem such as expert systems, 0-1 integer programming, approximation algorithms, and simulated annealing to name only a few. The ACM Computational Geometry Impact Task Force report [C⁺96] denotes label placement as an important research area. To fully comprehend the interest in automated labeling systems, one has to realize that manually labeling a map, for instance, is estimated to take fifty percent of total map production time [Mor80].

1.1 Historic Development

Going back in history, cartography is probably the oldest field that combined graphical with textual elements and was thus forced to deal with the labeling problem. The first record of a scaled map was found in China and is estimated to be about 2,300 years old [SZ97].

In cartography, the basis for automation was laid in the early sixties when the prominent Swiss cartographer Eduard Imhof published a catalogue of rules for label placement including good and bad examples [Imh62, Imh75]. In the same year, but independently of Imhof, Georges Alinhac, “Artiste Cartographe Principale” at the French Institut Géographique National, published the book “Cartographie Théorique et Technique”, which includes a similar set of labeling guidelines [Ali62]. These first formalizations of label placement have certainly facilitated the step from craftsmanship to technology. Before, based on his taste and long work experience, a cartographer could judge a map as being “well” or “poorly” labeled, but since the publication of Imhof’s and Alinhac’s rules, the deficiencies of a map labeling can be named in detail. Only then, after a model was found and the objectives were made clear, could technicians and researchers without expertise in cartography start to automate the process of label placement. This was crucial for the field since the apparently most basic problem of label placement, i.e. labeling points, turned out to be hard in terms of computational complexity.

Ten years after Imhof and Alinhac declared the principles of label placement, the Israeli cartographer Pinhas Yoeli wrote the first article dealing with the automation of map labeling [Yoe72]. He suggested an interactive system consisting of a human map editor, a geographical database, an output and a “principle-of-placement” module as well as a placement and an operational program. He devoted a lot of attention to the arrangement and the interplay between the parts of his system, which must be due to the limitations in storage, computation speed and the type-setting abilities of output devices at that time. He did not give any details of his placement program, but suggested that after each run of the program, the map editor would evaluate the results and decide whether his “preliminary estimate as to the name carrying abilities of his map was too optimistic”. In that case “there will be names for which the computer could not find any place”. If the map editor cannot add these manually, he has to revise decisions (concerning font size, place selection) and rerun the program until a satisfactory solution is found.

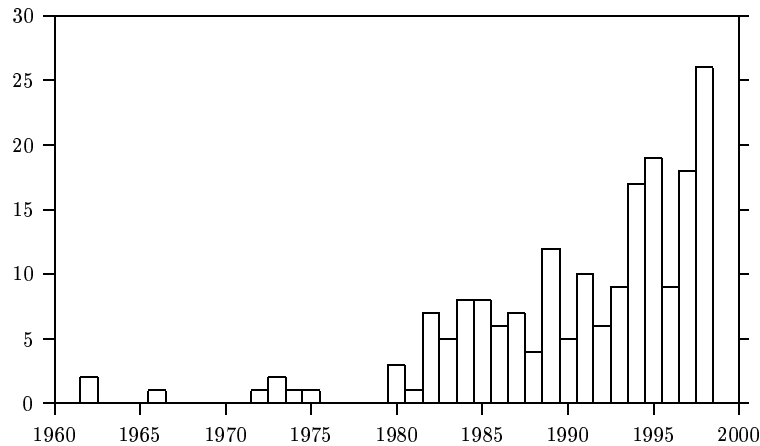


Figure 1.1: Number of publications over time in the label-placement literature.

Two articles by Boyle and a master’s thesis by Wilkie at the Department of Electrical Engineering, University of Saskatchewan followed at the beginning of the seventies [Boy73, Wil73, Boy74], but it was not until the eighties that a larger number of researchers became interested in labeling problems, see Figure 1.1. The figure shows for each year the number of references contained by the *Map-Labeling Bibliography* [WS96], an exhaustive list of literature about (mostly automated) label placement. The Map-Labeling Bibliography was integrated into the *Collection of Computer Science Bibliographies* [Ach95] in 1998.

Since the beginning of the eighties, there has been a steady flow of publications about the labeling problem in fields as diverse as artificial intelligence, cartography, geography, geology, spatial data handling, database systems, data structures, image processing, graph drawing, and computational geometry. Over the years, there were two diverging lines of research. Given the complexity of the labeling problem, most publications were directed towards developing and improving solutions for special cases. Another approach was targeting at finding a general labeling framework.

1.2 Theory...

Theoreticians, mostly computational geometers, entered the field around 1990. Nearly all work in terms of approximation algorithms and NP-hardness results has been focussed on the point labeling problem. This may be due to the fact that there are obvious models and objective functions for this problem. Nevertheless, there has been a rather controversial debate among cartographers over the “right” point-labeling model [WB91, Mil94]. Usually labels are restricted to axis-parallel rectangles, for example the bounding boxes of place names, but squares in arbitrary positions and disks have also been considered [DMM⁺97]. These labels must (a) not intersect any other label, and (b) touch the point they

label. While condition (a) guarantees legibility (given labels of sufficient size), condition (b) is responsible for the unambiguity of a labeling. Condition (b) has often been further restricted to the case where one of a label's corners must coincide with the point to be labeled. Some cartographers additionally allow the four positions where the midpoint of a label edge coincides with the point. This is how the basic labeling requirements mentioned above are modeled for points.

However, even apparently simple special cases of the point labeling problem, like deciding whether a set of points can be labeled with unit squares in one of four positions, have turned out to be NP-hard [MS91, FW91, KR92]. Therefore most theoreticians have studied approximation algorithms that maximize either the label size [FW91, DMM⁺97], the number of points with labels [AvKS98, vKSW99], or both criteria simultaneously [DMM⁺97]. Nevertheless, algorithms that can solve problems of a few dozen to a few hundred points optimally have also been studied in the past [Pre93, KMPS93, Sch95, VA99].

1.3 ... and Practice

Practitioners on the other hand have proposed an abundance of models and heuristics for labeling point, linear or area features on maps, and nodes or edges in graph drawings. While most of these algorithms may work well in practice, they lack guarantees on the quality of their results and often even asymptotic bounds on their runtime.

If quality guarantees or time bounds cannot be given, it is important to compare algorithms experimentally, both on real-world and synthetic data. There are extensive experimental comparisons of point labeling algorithms [CMS95, CFMS97, WW98]. However, we are not aware of similar work on algorithms for labeling more complex features. The reason for this may be that models for labeling one- or two-dimensional features tend to include aesthetic criteria that are highly application dependent.

1.4 Quality

Only recently another, more fundamental, question has been treated; namely how the quality of a label placement can be defined [vDvKSW99]. The basic idea is to collect a set of rules similar to those proposed by Imhof, and to quantify these rules subsequently. Such a quantification would have the task to produce formulae with a small number of parameters that can be set according to the application. The formulae should in turn be designed so that they are easily computable. Given an automatic label-quality checker, it will be possible to evaluate existing label-placement algorithms, locate their deficiencies, and ultimately come up with better algorithms. Determining a set of evaluation functions, upon which the label-placement community would generally agree, would mean a significant progress of the field, a step from a technical to a

scientific level.

So far the most common criteria taken into account for judging the quality of a label placement were either the number of features labeled or the label size. Again, only for labeling points more elaborate quality criteria have already been proposed and implemented, usually with the help of genetic algorithms [Djo94, VWS97, Pre98, Rum98, Rai98, Rai99, vDTdB99]. An exception to this is a rule-based system that Anthony Cook proposed and implemented in Prolog in collaboration with the British Ordnance Survey [Coo88]. He explicitly lists Imhof's rules and takes many of them into consideration.

1.5 Future Development

Since Yoeli opened the field of literature on automated label placement, the speed of hardware and the quality of printers has increased dramatically while the price for storage has dropped by the same order of magnitude. Furthermore, data structures and algorithms even for complex geometric problems have become widely available through libraries like LEDA [NM90], CGAL [Ove96], or the STL [MS96].

Concerning the future of label placement, I think that we can expect development into two somewhat contrary directions, namely high-quality and on-line labeling.

Assuming that the performance of computers and the price of human labour will further increase, so will the interest in high-quality labeling systems. So far, the only commercially available product is MAPLEX. This system was initially developed by a team of researchers under Christopher Jones at the University of Glamorgan, Wales [JC89]. Later the development of MAPLEX was continued by a company that was recently bought by one of the large producers of geographic information systems.

The core functionality of geographic information systems, or GIS for short, is to allow geographic data to be easily linked with other layers of information. Since GIS became available on PCs, they have gained enormous popularity for all kinds of administrative and planning tasks. One would think that labeling is a prominent feature of such systems, but so far most GIS offer only very basic placement routines. In practice, a GIS user is still forced to invest several hours in order to eliminate manually all label-label and label-feature intersections on a map—in spite of the large number of publications on automated label placement.

Given the success and the increasing availability of the Internet, the other important string of development can be expected to focus on rather simple, but very fast on-line applications. On-line mapping and label placement will certainly benefit from future extensions of the *hypertext mark-up language* (HTML). Soon it will be possible to transmit and depict vector images including text instead of bitmaps that tend to consume a lot of transmission time and computer storage. Already the current browser generation is able to place

text on top of graphics. This is one of the features of a much broader concept, namely *cascading style sheets* [JT98]. However, so far there is no generally accepted standard for font metrics. Thus the length of a textual label can vary from browser to browser, which makes it impossible to avoid intersections.

1.6 Overview

With this thesis, I would like to help narrowing the gap between theory and practice in automated label placement by presenting research in both directions. The thesis deals with the general label-placement problem, then investigates how to label points with rectangles or circles, how to label polygonal lines like rivers and finally how to design flexible geometric algorithms.

1.6.1 General Labeling, Compatible Representatives, and CSP

The general label-placement problem consists of labeling a set of *features* (points, lines, regions) given a set of *label candidates* (rectangles, circles, ellipses, irregularly shaped labels) for each feature. Each feature and each of its label candidates has a specified position in the plane. In general, a *label placement* or *labeling* simply specifies a subset of the features and chooses for each of these features a *label* from its set of label candidates such that no two labels intersect. In a *complete labeling* all features receive labels. Deciding whether a complete labeling exists is NP-hard in general [MS91, FW91]. Therefore, researchers have turned their attention mainly to developing heuristics and approximation algorithms for two obvious optimization versions of the problem, namely *label-number maximization* and *label-size maximization*.

The decision problem is a special case of the *problem of compatible representatives* introduced by Knuth and Raghunathan [KR92]. Label candidates are compatible representatives of their features if they do not intersect. In other words, we restrict compatibility to the geometric meaning implied by our context. Knuth and Raghunathan point out that “cartographers face an interesting case of the problem of compatible representatives”. The authors suggest that “it seems worthwhile to add the problem of compatible representatives to the class of ‘combinatorial problems that deserve a name’, and to investigate heuristics and additional special cases that prove to have efficient solutions.” Knuth and Raghunathan prove that the Metafont-labeling problem, a special case of the point-labeling problem, is NP-complete.

In the artificial intelligence (AI) community, the problem of compatible representatives has been addressed as the *constraint satisfaction problem* (CSP). A CSP consists of a finite set V of variables (corresponding to our features), of finite variable domains, i.e. sets D_v of at most d values (our label candidates) for each variable v in V , and of relations R on subsets V_R of V that exclude certain combinations of values for V_R . If we use symmetric binary relations that exclude intersecting candidates for each pair of features, the label-placement decision problem fits into this framework. The usual objective in the AI community

is either to list *all* assignment tuples without conflicts [MF85], to minimize the number of conflicts [FW92], or to find the maximum weighted subset of constraints that still allows an assignment (Max-CSP) [SFV95]. Since graph coloring and the decision version of the label-placement problem are NP-hard special cases of CSPs, one cannot expect to solve general CSPs in polynomial time. For this reason, the class of *network-consistency algorithms* has been invented. These algorithms use local arguments to exclude values from the domain of a variable that cannot be part of a global solution. Network-consistency algorithms can be seen as a preprocessing step to backtracking since they often reduce the search space very effectively.

In Chapter 2, we introduce a new framework for the general label-placement problem. We first extend classical CSP in order to be able to express the label-number maximization problem within this new framework. Then we develop a new form of local consistency, namely *r-irreducibility*. We present an algorithm, EI-1, that achieves 2-irreducibility in $O(d^3e)$ time using $O(de)$ space, where d is the size of the variable domains and e the number of binary relations. We also give a simple algorithm that finds near-optimal solutions for problems within our framework by combining EI-1 with a heuristic. This algorithm, EI-1* has proven to perform very well in practice, see Section 3.2, where we apply it to the point-labeling problem.

The following chapters are devoted to special cases of the general label-placement problem. In Chapters 3 to 5, we investigate the problems of labeling point and line features. When labeling a set of points, two fundamental questions can be asked. First, how many points can be labeled and second, how large can the labels be if all points must be labeled.

1.6.2 Point Labeling: Label-Number Maximization

In Chapter 3, we focus on label-number maximization given axis-parallel rectangular label candidates. First, we present two classes of models for labeling points with axis-parallel rectangles, namely so-called *fixed-position* and *slider* models. While the former restrict the number of candidates to a constant, the latter allow an infinite number. We compare some of these models theoretically by showing how many more points can be labeled in one model than in another. This is joint work with Marc van Kreveld and Tycho Strijk, both at Universiteit Utrecht [vKSW98, vKSW99].

Next, we exemplify our general framework at one of the fixed-position models. We do this such that it becomes clear how our concept can be applied to other cases. The resulting algorithm is fast, simple and performs well even on large real-world data sets. We study competing algorithms and do a thorough empirical comparison. It turns out that our algorithm produces results comparable to simulated annealing but obtains them much faster. Our algorithm outperforms a heuristic of Kakoulis and Tollis [KT98], not only in terms of time, but also in terms of quality. Like our framework, both simulated annealing [ECMS97] and the heuristic of Kakoulis and Tollis can be applied to the

general label-placement problem.

Since our framework is limited to fixed-position models, we also propose a fast greedy algorithm that works for slider and fixed-position models. Then we show that the slider models have polynomial-time approximation schemes. Finally we compare the greedy algorithms for slider models experimentally to those for fixed-position models. This part is also joint work with Marc van Kreveld and Tycho Strijk [vKSW98, vKSW99].

1.6.3 Point Labeling: Label-Size Maximization

In Chapter 4, we look at the second aspect of point labeling, namely label-size maximization. Instead of asking how many features can be labeled given candidates of a fixed size, we now assume that all points must be labeled and that their labels all have the same size. Under these circumstances it is natural to search for algorithms that simultaneously maximize the size of all labels. In the case of square label candidates, four per point, a theoretically optimal algorithm is known [FW91, Wag94] and has been extended to perform very well in practice [WW97]. We propose an algorithm for labeling points with uniform circles. The algorithm guarantees to find a placement with circles of about $1/20$ of the diameter of the labels in an optimum solution. This improves the only known algorithm [DMM⁺97] by more than 50%. We also show that it is NP-hard to approximate the problem beyond a certain constant factor. This is joint work with Tycho Strijk.

1.6.4 Line Labeling

While an abundance of solutions for point labeling and some acceptable approaches to area labeling have been suggested, mostly using the medial axis [AF84] or methods for computing the largest enclosed rectangle of given aspect ratio [vR89, AIK89, CK89, DMR97], there seems to be a gap in the literature concerning efficient geometric algorithms for labeling linear features such as rivers or streets. In Chapter 5 we turn our attention to line labeling. There the emphasis does not lie on maximizing label number or size, but on the question where to place the label in the vicinity of the object to be labeled. In other words, we are confronted with a modeling rather than an optimization problem. We first list the requirements of high-quality line labeling and divide them into two categories, *hard* and *soft* constraints.

In Section 5.3, we propose an efficient algorithm that produces a candidate strip along the input polyline. The strip has the same height as the given label, consists of rectangular and annular segments, and guarantees the hard constraints, such as a lower bound on a label's distance to the polyline and on the label's curvature.

In Section 5.4, we present algorithms for several evaluation functions whose task is to produce one or several good label placements within the candidate strip. These functions optimize soft constraints, such as the number of inflec-

tions. Again, we perform a thorough experimental analysis by applying our algorithm to synthetic as well as real-world data, see Section 5.5.

Although several line-labeling algorithms have been proposed in the literature [Coo88, DF92, BL95, AH95, ECMS97, Kra97, Bar97, PZC98, SvK99], our algorithm is the first where at the same time curved labels are allowed and bounds on the runtime given. Chapter 5 is joint work with Lars Knipping, Freie Universität Berlin, Marc van Kreveld, Tycho Strijk, both at Utrecht Universiteit, and Pankaj K. Agarwal, Duke University [WKvK⁺99].

1.6.5 Designing Geometric Algorithms

In order to support the claim of the practical relevance of our concepts, we implemented most of the algorithms we propose. The experience we have gained from implementing led to a generic design concept for geometric algorithms, which we present in Chapter 6 in the form of a tutorial. Our concept greatly increases the flexibility of an implementation without sacrificing its ease-of-use. The gain in flexibility can reduce implementation effort by facilitating code reuse. Reusability in turn helps to achieve correctness since more users mean more testing. The loss in terms of efficiency is small.

Our concept is based on the *generic programming* paradigm that has evolved over the last few years. Generic programming is about making programs more flexible by making them more general [BS98]. Abstracting from concrete in- or output data representation is an example of generic programming. This paradigm has been so successful that a model—the *Standard Template Library* (STL) [MS96]—was created and added to C++, currently one of the most popular programming languages. The STL is a library of generic components, i.e. of algorithms, data containers, and *iterators* mediating between the former two. Iterators help to decouple algorithms from the type of data container they operate on. While iterators have been known before, the real novelty of the STL was the introduction of a requirements-based taxonomy of iterators, which gives a guideline for full decoupling, and an implementation of this taxonomy using the C++ template mechanism. By becoming part of the C++ standard, the STL has attracted considerable attention and has itself set a standard for good design.

After the introduction of the STL further concepts such as *data accessors* have been suggested in the C++ literature to help programmers make their implementations even more generic [Küh96, Wei97]. Data accessors are a means to further decouple an implementation from the representation of in- and output data [KW97]. So far, these extensions have been applied predominantly to graph problems [NW96]. Exceptions such as [Wei98, Ket98] deal with the representation of geometric objects, not with the design of geometric *algorithms*, our main interest here. In order to show the relevance of STL-style generic programming including later extensions as data accessors for geometric algorithms, we investigate a simple rectangle-intersection algorithm that follows the well-known sweep-line paradigm. Using this example we give a step-by-step guide from an inflexible, naive interface to a truly flexible interface that sup-

ports code reuse. These steps reflect our own change of perspective during the implementation of our label-placement algorithms. We base our presentation on C++. While the ingredients of our concept have already been known, to our knowledge this is the first time that they are applied so rigorously to a geometric problem, that they are made accessible in the form of a tutorial and that they are accompanied by a thorough experimental analysis on random and real world data. Chapter 6 is joint work with Vikas Kapoor, Freie Universität Berlin, and Dietmar Kühl, Claas Solutions GmbH.

During the implementation phase of our algorithms, we developed a tool for the automatic generation and maintenance of makefiles that we found very helpful for administering inter-file dependencies in software projects [SW98]. This was joint work with Sven Schönherr, Freie Universität Berlin.

Chapter 2

General Labeling: Label-Number Maximization

The problem of label placement is usually divided into point, line, and area labeling, depending on the kind of features to be labeled. However, the problem can be formulated independently of the shape of features. Two interesting subproblems have been studied. In both cases, an instance consists of a set of features and a set of label candidates for each feature.

1. *The Label-Size Maximization Problem:* Find the maximum factor σ such that each feature gets a label stretched by this factor and no two labels overlap. Compute the corresponding *complete* label placement.
2. *The Label-Number Maximization Problem:* Find a maximum subset of the features, and for each of these features a label from its set of candidates, such that no two labels overlap.

The decision versions of both problems are NP-hard in general [FW91, FPT81]. The label-size maximization problem can be solved in polynomial time if all features have at most two label candidates. Then the problem can be encoded as a 2-SAT formula and tested for satisfiability in time linear in the number of pairs of intersecting candidates [EIS76]. This was already observed in [FW91]. If the label candidates of a feature overlap in a certain manner, polynomial time algorithms are known for any constant number of label candidates per feature [PZC98, SvK99], and even for an infinite number of label candidates per feature [KSY99].

In recent years, especially the point-labeling problem has achieved some attention in the algorithms community. For maximizing the number of points that are labeled with axis-parallel rectangles, the current status of the problem is described in Chapter 3. For problems related to maximizing the size of rectangular or circular labels for point features, refer to Chapter 4. In this chapter we investigate the general label-number maximization problem.

Methods that have been used for label-number maximization so far are heuristical; they include simulated annealing [CMS95, ECMS97, Zor97] and an algorithm that uses maximum-cardinality bipartite matching between features and cliques of intersecting label candidates [KT98]. Both approaches will be discussed in more detail in Section 3.2.

We propose a new framework for the general label-number maximization problem. It leads to a heuristical algorithm that is easy to implement, and, for point labeling, yields better results than the matching heuristic of [KT98] and similarly good results as simulated annealing, but obtains them much faster, see Section 3.2. Our framework is related to a concept suggested in the artificial intelligence community under the name *constraint satisfaction*, which was independently introduced into the discrete mathematics community by Knuth and Raghunathan under the name *problem of compatible representatives* [KR92]. The difference of our approach to that of the artificial intelligence community is that we try to maximize the number of variables (features) with a conflict-free assignment, while their objective is either to list *all* assignment tuples without conflicts [MF85], to minimize the number of conflicts [FW92], or to find the maximum weighted subset of constraints that still allows an assignment.

Since constraint satisfaction is NP-hard in general, the artificial intelligence community invented so-called *network-consistency algorithms*. These algorithms establish a form of consistency; i.e. they use local arguments to exclude values from the domain of a variable that cannot be part of a global solution. Network-consistency algorithms can be seen as a preprocessing step to backtracking since they often reduce the search space very effectively.

We develop the notion of *r-irreducibility*, a new form of local consistency that is comparable to consistency in classical constraint satisfaction. We give an algorithm, EI-1, that achieves 2-irreducibility in $O(d^3e)$ time using $O(de)$ space, where d is the maximum domain size and e the number of pairs of variables whose values are in conflict with each other. The domain of a variable corresponds to the set of label candidates of a feature in label placement. The value of a variable is nothing but a label candidate, and for us, two values are in conflict with each other if the corresponding candidates intersect.

While d is considered to be a small constant in point labeling (usually four or eight), there are many applications in artificial intelligence where d can be very large. Thus we take the size of d into account in this chapter. Note that k , the number of pairs of intersecting label candidates, is of $O(d^2e)$.

In addition, we present an algorithm, EI-1*, for general label-number maximization that is based on EI-1. This algorithm first establishes 2-irreducibility. Then it repeatedly makes a heuristical decision and restores 2-irreducibility until each feature is either labeled or known to constrain too many other features and therefore not labeled at all. Given the value conflict graph, EI-1* requires $O(d^3e)$ time and $O(de)$ space like EI-1.

Our new framework is called *maximum variable-subset constraint satisfaction*. Our hope is that EI-1* or other efficient algorithms based on higher

degrees of irreducibility will substitute simulated annealing for the wide variety of problems that fit into our framework. Experiments in the context of point labeling indicate that EI-1* is not only fast but also very effective in practice, see Section 3.2.

This chapter is structured as follows. In Section 2.1 we give a quick introduction into the issues relevant for label placement that have been investigated by the artificial intelligence community. We consider classical constraint satisfaction problems (CSP) and a generalization, namely Max-CSP. In Section 2.2 we extend classical CSP to maximum variable-subset CSP where the label-number maximization problem can easily be formulated. In Section 2.3 and 2.4 we define irreducibility and describe our 2-irreducibility algorithm EI-1. Finally, in Section 2.5 we present our algorithm EI-1* for the general label-number maximization problem.

2.1 Label Placement and CSP

A constraint satisfaction problem (CSP) is defined as follows. Given a set of n variables v_1, \dots, v_n , each associated with a domain D_i and a set of relations constraining the assignment of subsets of the variables, find all possible n -tuples of variable assignments that satisfy the relations [MF85]. Variable domains are restricted to discrete finite sets, and often only binary relations are considered.

Graph coloring is a special case of a CSP where the variables are nodes, the domains a given set of colors, and binary relations express the fact that a node cannot have the same color as any of its neighbors. Since graph coloring, i.e. deciding whether the nodes of a graph can be colored with the given set of colors, is NP-complete, one cannot expect to solve general CSPs in polynomial time [MF85]. For this reason, the class of *network-consistency algorithms* has been invented. These algorithms use local arguments to exclude values from the domain of a variable that cannot be part of a global solution. Network-consistency algorithms can be seen as a preprocessing step to backtracking since they often reduce the search space very effectively.

An m -consistency algorithm removes all inconsistencies among all subsets of m of the given n variables. For the special cases of $m = 1, 2,$ and 3 , polynomial-time algorithms have been suggested. They are called node-, arc-, and path-consistency algorithms, respectively.

This framework can be used nearly one-to-one for attacking the label *size* maximization problem. When maximizing simultaneously the sizes of all labels, one can do a binary search on *conflict sizes*, i.e. label sizes for which label candidates start to touch. For each conflict size, one then tries to find a complete labeling. Obviously, a feature can be seen as a variable, the set of label candidates of a feature then corresponds to the variable domain and intersections between label candidates are the constraining binary relations. Instead of computing *all* satisfying variable assignments, finding *one* is usually sufficient in the map-labeling context. This allows to reduce the search space dramati-

cally since a variable can immediately be assigned an unconstrained value from its domain if there is such a value. The algorithm for label size maximization suggested in [WW97] exploits this simplification.

When maximizing the number of labeled features, label sizes are fixed and one cannot give up and try a smaller label size as soon as it turns out that there is no complete labeling for the current label size. Systems where one cannot expect to find a *complete solution*, i.e. a non-conflicting variable assignment, are called *over-constrained systems*. In such systems one has to be content with imperfect solutions. Most effort in the CSP community has been directed to finding solutions that violate as few constraints as possible [FW92, Jam96, JFM96]. When labeling maps, such violations would result in label over-plots and thus poor legibility. It would be possible to take the output of an algorithm that minimizes the number of violated constraints and then do some post-processing. In order to get rid of the violations, one could drop a subset of the variables and resign from labeling the corresponding features. Unfortunately the problem of finding the largest violation-free subset of variables corresponds to the maximum independent set problem and is thus NP-hard.

A related problem, Max-CSP, has also been investigated. There, one is interested in finding a maximum (weighted) subset of the constraints such that there is an assignment that satisfies them all. In order to reduce label-number maximization to Max-CSP, one adds a new value Δ to the domain of each variable. Δ has a unary constraint of low weight; i.e. it only constrains itself. A variable that is assigned Δ then corresponds to an unlabeled feature in our setting. For general Max-CSP, however, even arc consistency is NP-hard [SFV95].

Therefore we take a different approach. We first extend classical CSP in order to be able to express the label-number maximization problem within this new framework, see Section 2.2. Then, in Section 2.3 we develop a new form of local consistency, namely *r-irreducibility*. In Section 2.4, we present an algorithm, EI-1, that achieves 2-irreducibility in $O(d^2e)$ time using $O(de)$ space, where d is the size of the variable domains and e the number of binary relations. Finally, in Section 2.5 we give a simple algorithm that finds near-optimal solutions for problems within our framework by combining EI-1 with a heuristic. This algorithm has proven to be very effective in practice, see Section 3.2, where we apply it to the point-labeling problem.

2.2 Maximum Variable-Subset CSP

Let us start by giving a formal definition of classical CSP [SFV95].

Definition 2.1 (CSP) *An instance of a constraint-satisfaction problem (CSP) is a triple $(V, \mathcal{D}, \mathcal{C})$ where V is a set of n variables, \mathcal{D} a collection of domains, one for each variable, and \mathcal{C} a set of constraints. A domain D_v of a variable v is a (finite) set of values of v . A constraint $C \in \mathcal{C}$ is given by a pair*

(V_C, R_C) where $V_C \subseteq V$ is a subset of the variables and $R_C \subseteq \prod_{v \in V_C} D_v$ is a relation on the variables in V_C .

A solution of a CSP is a function π that maps each variable to a value of its domain such that all constraints are satisfied, i.e. $\prod_{v \in V_C} \pi(v) \in R_C$ for all $C \in \mathcal{C}$.

In classical CSP one is either interested in finding one or in listing all solutions. We extend classical CSP in order to be able to better formulate label-number maximization. In the following definition we assume that no variable domain contains an element 0.

Definition 2.2 (MVS-CSP) A solution of a maximum variable-subset CSP (MVS-CSP) $(V, \mathcal{D}, \mathcal{C})$ is a function π that assigns every variable v in V to a value of its domain D_v or to 0 such that all relevant constraints are satisfied, i.e. for all $C \in \mathcal{C}$ if $0 \notin \pi(V_C)$ then $\prod_{w \in V_C} \pi(w) \in R_C$.

The size $|\pi|$ of a solution π is the number of variables v in V that π assigns a value $\pi(v) \in D_v$. In MVS-CSP an optimal solution is a solution of maximum size. A solution of size $|V|$ is called a complete solution,

In our definition we drop a constraint $C = (V_C, R_C)$ completely if any of the variables v in V_C is mapped to 0. The reason for this part of our definition is that the restriction of C imposed on the variables in $V_C \setminus \{v\}$ depends on the value of v , thus we cannot make any assumption about which combination of values of $V_C \setminus \{v\}$ is excluded by C . It makes sense to require that V_C is in a sense minimal, in other words that there is no $v \in V_C$ such that the projection of R_C to $\{x\} \times \prod_{w \in V_C \setminus \{v\}} D_w$ is identical for all values $x \in D_v$.

Definition 2.2 transfers the decision or enumeration problem of classical CSP into an optimization problem.

For label placement only binary constraints are relevant, i.e. $|V_C| = 2$ for all $C \in \mathcal{C}$. Given two features f and g of a label-placement instance, these binary constraints encode which pairs of label candidates b and c of f and g intersect, respectively. Thus we can use a simpler definition.

Definition 2.3 (binary MVS-CSP) An instance of a binary MVS-CSP is a triple $(V, \mathcal{D}, \mathcal{R})$ where \mathcal{R} is a set of predicates R_{vw} on $(D_v \cup \{0\}) \times (D_w \cup \{0\})$, one for each pair (v, w) of variables. For $x = 0$ or $y = 0$ $R_{vw}(x, y)$ is always true. A solution π must fulfill $R_{vw}(\pi(v), \pi(w))$ for all $v \neq w \in V$.

Given a binary CSP, constraint information can be encoded conveniently by any of the graphs that we define in the following.

Definition 2.4 (variable/value constraint/conflict graph) We say that a value x of a variable v constrains a value y of a variable w if $R_{vw}(x, y)$ is false. Let $R_{vw}^*(x, y)$ be the symmetric predicate that is true if $R_{vw}(x, y)$ and

$R_{vw}(y, x)$ are true. Then x and y are in conflict if $R_{vw}^*(x, y)$ is false. We say that w excludes a value x of v if all values of w constrain x .

We say that a variable v constrains (is in conflict with) a variable w if there is a value in the domain of v that constrains (is in conflict with) a value in the domain of w . The variable constraint graph $\vec{G}(V, \vec{E})$ has an arc for each pair (v, w) where v constrains w ; the variable conflict graph $G(V, E)$ has an edge for each pair $\{v, w\}$ where v is in conflict with w . In the value constraint (conflict) graph $\vec{G}_{\mathcal{D}}(G_{\mathcal{D}})$ there is a vertex for each variable-value pair $[v, x]$ with $v \in V$ and $x \in D_v$, and an arc (edge) between two such pairs $[v, x]$ and $[w, y]$ iff $R_{vw}(x, y)$ ($R_{vw}^*(x, y)$) false.

The question whether an instance of MVS-CSP has a complete solution corresponds to classical CSP. Thus the decision version of MVS-CSP, namely *Is there a solution of size s ?*, is NP-hard as well. Note that MVS-CSP corresponds to maximum independent set on $G_{\mathcal{D}}$ if we make the values of each variable into cliques, i.e. if we add edges between $[v, x]$ and $[v, y]$ for all $v \in V$ and $x, y \in D_v$ with $x \neq y$.

In order to approach classical CSP in spite of its NP-hardness, the notion of consistency has been developed. An instance is *m-consistent* if all m -element subsets $W \subseteq V$ are consistent, i.e. if for each value x in the domain of any variable w in W there is a complete solution for W that maps w to x . m -consistency introduces a scale between totally inconsistent and perfectly consistent. Node-, arc-, and path-consistency algorithms achieve 1-, 2-, and 3-consistency in polynomial time [Mac77]. For backtracking algorithms, achieving arc- or path-consistency is an important preprocessing step that reduces checking the same inconsistent variable assignment repeatedly. In the extreme, for each $v \in V$ and each $x \in D_v$ a $|V|$ -consistent instance yields a complete solution that maps v to x .

The input to network-consistency algorithms comprises usually the variable constraint graph, the domain of each variable, and for each arc (v, w) of the graph a method that returns the value of $R_{vw}(x, y)$ for all pairs $(x, y) \in D_v \times D_w$. The variable constraint graph can be transformed into a value constraint graph, but the latter might need up to a factor of $O(d^2)$ more storage, where d is the (maximum) size of the variable domains.

2.3 Irreducibility

The potential of network-consistency algorithms is our motivation for transferring the concept of consistency to MVS-CSP. In our setting, we refer to it as *irreducibility*, which we define as follows.

Definition 2.5 (reducible, redundant) *Given a binary MVS-CSP $(V, \mathcal{D}, \mathcal{R})$ and a subset $W \subseteq V$, a variable $v \in W$ is W -reducible iff there is a value $x \in D_v$ such that for all solutions π of W with $\pi(v) = x$ there*

is a solution π' of W with $\pi'(v) \neq x$ and $|\pi'| \geq |\pi|$. For all $w \in W$, $\pi'(w)$ must be either equal to $\pi(w)$ or not in conflict with any values of variables in $V \setminus W$, i.e. $R_{vw}^*(y, \pi'(w))$ for all $v \in V \setminus W$ and all $y \in D_v$. If such solutions π' exist, x is called W -redundant.

$W \subseteq V$ is irreducible iff there is no $v \in W$ that is W -redundant. V is r -irreducible iff all r -element subsets $W \subseteq V$ are irreducible.

Note that r -irreducibility implies i -irreducibility for all $i < r$. Node-, arc-, and path-irreducible will be used as synonyms for 1-, 2-, and 3-irreducible. If all constraints are symmetric, we will use edge-irreducible instead of arc-irreducible. The notion of reducibility helps us to remove redundant values from variable domains and thus reduce the search space for an optimal solution.

Lemma 2.6 *Let π be an optimal solution of a binary MVS-CSP $(V, \mathcal{D}, \mathcal{R})$. If there is a subset $W \subseteq V$ and a variable $v \in W$ that is W -redundant, then there is an x in the domain D_v of v such that $(V, \mathcal{D}', \mathcal{R})$ has a solution of size $|\pi|$, where $\mathcal{D}' = \{D_v \setminus \{x\} \mid v \in V\}$.*

Proof. We assume that $(V, \mathcal{D}', \mathcal{R})$ has only solutions strictly smaller than π . If $\pi(v) \neq x$ then π would also be a solution to the reduced instance, contradicting our assumption. Thus $\pi(v) = x$. Then, by definition of reducibility, there must be a solution π' of W with $\pi'(v) \neq x$ and $|\pi'| \geq |\pi^W|$ where π^W is the restriction of π to W . For each $w \in W$, π' must either fulfill $\pi'(w) = \pi(w)$ or $R_{vw}^*(y, \pi'(w))$ for all $v \in V \setminus W$ and all $y \in D_v$. Let ρ be the following function on V .

$$\rho(u) = \begin{cases} \pi(u) & \text{for all } u \in V \setminus W; \\ \pi'(u) & \text{otherwise.} \end{cases}$$

We show that ρ is a solution of the reduced instance. Let $v, w \in V$ with $\rho(v) = y \neq 0$ and $\rho(w) = z \neq 0$. We must show that y and z are not in conflict. This is clear if $\{v, w\} \subseteq V \setminus W$ and if $\{v, w\} \subseteq W$ since ρ equals π and π' on the respective subsets of V , and π and π' are solutions on $V \setminus W$ and W , respectively. Thus it is enough to consider the case $v \in V \setminus W$ and $w \in W$. On the one hand this implies $\pi(v) = \rho(v) = y$. On the other hand, we have either $\pi(w) = \pi'(w) = \rho(w) = z$ or $R_{vw}^*(y, \pi'(w))$ for all $v \in V \setminus W$ and all $a \in D_v$. In the first case y and z are both part of solution π and therefore cannot be in conflict. In the second case, too, y and z are not in conflict since $v \notin W$ and thus $R_{vw}^*(y, \pi'(w))$.

Since $|\rho^W| = |\pi'| \geq |\pi^W|$ and $\rho^{V \setminus W} = \pi^{V \setminus W}$ we have $|\rho| \geq |\pi|$, which contradicts our assumption. \square

$|V|$ -irreducibility gives us direct access to an optimal solution.

Lemma 2.7 *In a $|V|$ -irreducible binary MVS-CSP $(V, \mathcal{D}, \mathcal{R})$ all variable domains contain at most one value, i.e. $|D_v| \leq 1$ for all variables $v \in V$.*

Proof. Suppose there is a variable $v \in V$ with $|D_v| > 1$ and v is not V -reducible. There are two possibilities. Either there is an optimal solution π_{opt} of V that maps v to a $y \in D_v$ or all optimal solutions map v to 0. In the second case let π_{opt} be one of these solutions, and set y to 0.

In either case there is a value $x \in D_v \setminus \{y\}$ and for all solutions π of V with $\pi(v) = x$ there is a solution π' of V (namely π_{opt}) with $\pi'(v) \neq x$ and $|\pi'| \geq |\pi|$. Thus v is V -reducible since the additional condition, namely $R_{vw}^*(y, \pi'(w))$ for all $v \in V \setminus V$ and all $y \in D_v$, is trivial for V -reducibility. Hence our assumption is contradicted. \square

2.4 An Edge-Irreducibility Algorithm

Mackworth [Mac77] proposed an algorithm, AC-3, that achieves arc-consistency in polynomial time, see Figure 2.2. With Freuder [MF85] he showed later that AC-3 requires at least $\Omega(d^2e)$ and at most $O(d^3e)$ time, where e is the size of the variable conflict graph and d the size of the variable domains, which are assumed to be of equal size for all variables. AC-3 does not assume that constraints are symmetric. It uses $O(de)$ storage.

The heart of AC-3 is a procedure REVISE that, given a pair (v, w) of variables, eliminates all values from the domain of v that are excluded by w , see Figure 2.1. AC-3 uses a stack to keep track of all pairs of variables that potentially need revision. Initially the stack is filled with all arcs of the variable constraint graph. Until the stack (or a variable domain) is empty, AC-3 repeatedly draws a pair (v, w) from the stack, calls REVISE(v, w), and, if REVISE removed a value from the domain of v , adds all arcs (u, v) to the stack.

The task of REVISE is simple. It makes the arc (v, w) -consistent by removing all values of v that are excluded by w and therefore cannot be part of any complete solution. Without any additional data structures REVISE requires $O(d^2)$ time. The time complexity of AC-3 follows from the fact that REVISE is called at most d times for each of the e edges of the variable constraint graph.

Later, Mohr and Henderson introduced the notion of *support* [MH86]. A variable-value pair $[v, x]$ supports the value y of a variable w if $R_{vw}(x, y)$. (We will switch between value and variable-value pair depending on which is more convenient.) As soon as $[v, x]$ loses its last support from a variable w that constrains v , x must be removed from the domain of v . Mohr and Henderson gave an algorithm, AC-4, that is based on this idea. For each variable-value pair $[v, x]$, AC-4 keeps track of the number $k_{v,x}$ of values that support $[v, x]$ and maintains a list $S_{v,x}$ with all values that $[v, x]$ supports. Using these data structures yields AC-4's optimal time complexity of $O(d^2e)$. However, they are also responsible for the fact that AC-4 requires $O(d^2e)$ storage. In addition, average and worst case runtime behavior of AC-4 do not differ much. These disadvantages made AC-3 in spite of its inferior time complexity favorable in many applications [Bes94].


```

REVISE( $v, w$ )
 $deleted \leftarrow false$ 
for each  $x \in D_v$  do
  for each  $y \in D_w$  do
    if  $R_{vw}(x, y)$  then exit inner loop end
  end
  if  $\neg R_{vw}(x, y)$  then
     $D_v \leftarrow D_v \setminus \{x\}$ 
     $deleted \leftarrow true$ 
  end
end
return  $deleted$ 

```

Figure 2.1: The procedure REVISE makes the arc (v, w) consistent.

```

AC-3( $V, \mathcal{D}, \mathcal{R}$ )
 $E \leftarrow \{(v, w) \mid v, w \in V, \exists x \in D_v, y \in D_w : \neg R_{vw}(x, y)\}$ 
 $Q \leftarrow E$ 
while  $Q \neq \emptyset$  do
   $(v, w) \leftarrow Q.pop()$ 
  if  $REVISE(v, w) = true$  then
    for each  $u \in V$  such that  $(u, v) \in E$  do
       $Q.push((u, v))$ 
    end
  end
end

```

Figure 2.2: The third arc-consistency algorithm AC-3.

Bessi ere found out that it is not necessary to maintain counters and that it is enough to keep *one* support for each of the $O(de)$ arc-value pairs [Bes94]. His algorithm AC-6 exploits these observations and takes advantage of a total order on the values in each domain. AC-6 needs less storage than AC-4, namely $O(de)$. Although it shares the time complexity of $O(d^2e)$ with AC-4, it needs less predicate evaluations than both its predecessors AC-3 and AC-4. Shortly after, Bessi ere, Freuder, and R egin suggested improvements of AC-6 that led to AC-7. This algorithm requires even less predicate evaluations than AC-6 while keeping the asymptotic space and time complexity of its predecessor [BFR95].

Unfortunately the concept of support does not work in the context of MVS-CSP. If a variable-value pair $[b, 1]$ loses support from a variable c , this only means that not both $[b, 1]$ and a value $y \in D_c$ can be part of a solution. However, it does not imply that an optimal solution will not map b to 1. It does not even imply that there is an optimal solution that does not map b to 1 as the example

in Figure 2.3 demonstrates. There, variables are represented by boxes and their values by circles. Conflicting values are connected by edges; all values have degree 3 in the *value conflict graph*, except the values of c that have degree 4. While the optimal solution (indicated by bold circles) has size 4, all solutions that map b to 2 (or 0) have size at most 3.

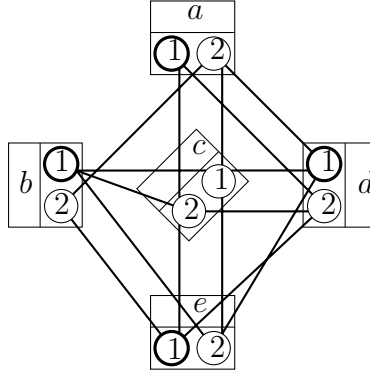


Figure 2.3: Example where the only value $(b, 1)$ that is lacking support on an edge (namely $\{b, c\}$) is in the only optimal solution (indicated by bold circles).

From now on we will only consider CSPs with symmetric constraints, i.e. for all $v, w \in V$ and $x \in D_v, y \in D_w$ we have $R_{vw}(x, y) = R_{wv}(y, x) = R_{vw}^*(y, x)$. For this reason we will avoid saying $[v, x]$ *constrains* $[w, y]$ since this induces a direction, but rather say $[v, x]$ *and* $[w, y]$ *are in conflict*. Since constraints are assumed to be symmetric, arc-irreducibility becomes edge-irreducibility according to our notation.

Since AC-3 is not based on the concept of support, we can rewrite REVISE and use AC-3 to achieve edge-irreducibility. For classical CSP, REVISE takes $O(d^2)$ time. For our purpose, however, its task becomes more involved. Given two variables v, w , REVISE must check whether v is $\{v, w\}$ -reducible. To decide whether we can remove a value x from D_v , for each solution π of $\{v, w\}$ with $\pi(v) = x$ we must find a solution π' of $\{v, w\}$ with $\pi'(v) \neq x$ and $|\pi'| \geq |\pi|$. $\pi'(w)$ must either equal $\pi(w)$ or not be in conflict with any values of variables in $V \setminus \{v, w\}$. For $\pi'(v)$ the latter condition must hold.

Using brute force, we could do the following. For each of the $O(d)$ values x of v , we enumerate each of the $O(d)$ possible solutions π_y that map v to x and w to some $y \in D_w \cup \{0\}$. For each π_y we search for a solution π'_y that maps v to a value $x' \neq x$ and fulfills the conditions stated above. To find such a solution π'_y we go through all $O(d^2)$ pairs of values (x', y') with $x' \in (D_v \cup \{0\}) \setminus \{x\}$ and $y' \in D_w$. For each pair we check $R_{vw}^*(x', y')$ and whether x' and y' (if $y' \neq y$) are not in conflict with any values of variables in $V \setminus \{v, w\}$. If this test can be done in constant time REVISE requires at most $O(d^4)$ steps. Then AC-3 achieves edge-irreducibility in $O(d^5)$ time.

Clearly this rough estimate can only serve as an upper bound. We can definitively do better. Our approach is as follows. We give a list of three rules,

each of which consists of the description of a certain conflict situation and a recipe of how to resolve it. We show that (a) only redundant values, i.e. values that prove the reducibility of a variable, are removed, (b) if all rules are applied exhaustively, the remaining instance is edge-irreducible, and (c) the application of each rule takes $O(d^2)$ time. Given Lemma 2.6, (a) implies that the size of the optimal solution remains the same until arc-consistency is achieved.

Let v and w be two variables in V , $v \neq w$. For each of the three rules below there is a figure depicting a typical situation in which the rule applies. In Figures 2.4 to 2.6 variables are represented by boxes and their values by circles. Conflicting values are connected by edges. Short line segments not ending in a circle indicate that the value from which they emanate might constrain further values possibly of other variables. The values that are removed after applying a rule are indicated by dotted circles.

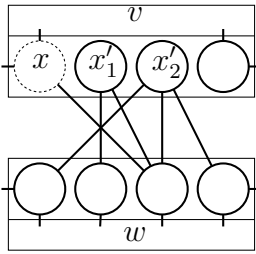


Figure 2.4: rule A1

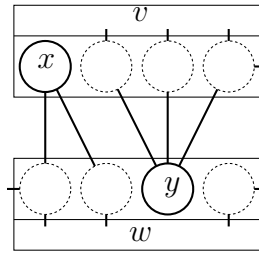


Figure 2.5: rule A2

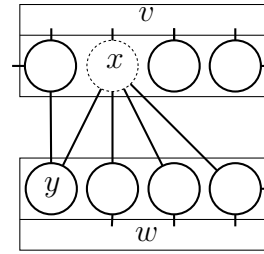


Figure 2.6: rule A3

- (A1)** If there is a value $x \in D_v$ and a subset $X \neq \emptyset$ of $D_v \setminus \{x\}$ such that all $x' \in X$ are at most in conflict with values of w , and for each value y of w that x does not constrain, there is a value $x' \in X$ that does not constrain y either, then remove x from D_v .

Special case ($X = \{x'\}$): If x' is only in conflict with values of w and those form a subset of the values that are in conflict with x , then remove x from D_v .

Special sub-case ($X = \{x'\}$ and x' has no conflicts): Then remove all values $x \neq x'$ from D_v . (This rule yields node-irreducibility.)

- (A2)** If there are values x and y of v and w , respectively, that are not in conflict with each other and with values of variables other than v and w , then set $D_v = \{x\}$ and $D_w = \{y\}$.
- (A3)** If there is a value $x \in D_v$ that is excluded by w , and there is a value $y \in D_w$ that is only in conflict with values of v , then remove x from D_v .
- Special case ($D_w = \{y\}$): If y is only in conflict with values of v , then remove all these values from D_v .

Lemma 2.8 *If any of the rules A1 to A3 are applied to two variables v and w , only $\{v, w\}$ -redundant values are removed from the domains of D_v and D_w .*

Proof. Given the situation described in rule A1, we have to show that x is $\{v, w\}$ -redundant. Let π be any solution for v and w that maps v to x . If A1 is applicable there is a subset $X \neq \emptyset$ of $D_v \setminus \{x\}$ that contains a value x' that is only in conflict with values of w , but not with $\pi(w)$. (For $\pi(w) = 0$ this is true for any $x' \in X$.) Then $\pi'(v, w) = (x', \pi(w))$ is a solution of the same size as π , and x is redundant.

For A2 we can argue as follows. Since $\{x, y\}$ is a complete solution for $\{v, w\}$, it is obvious that all other values of v and w are redundant before applying A2.

Considering A3, a solution π for v and w that maps v to x must map w to 0, hence it cannot be larger than a solution π' that maps w to y . This shows that x is redundant. \square

Lemma 2.9 *After rules A1 to A3 have been applied exhaustively to an instance $(V, \mathcal{D}, \mathcal{R})$, the resulting instance $(V, \mathcal{D}', \mathcal{R})$ is edge-irreducible.*

Proof. If $|V| < 2$ then there is nothing to show; arc-consistency is defined for pairs of variables. (Still, the special sub-case of A1 would have removed all but one value of the only variable, and the resulting instance would then be (node-)consistent.) Thus let $|V| \geq 2$. We assume that there is a subset $W = \{v, w\} \subseteq V$ and that v is W -reducible in $(V, \mathcal{D}', \mathcal{R})$. Then, due to the definition of reducibility, there is a value $x \in D_v$ such that for all solutions π of W with $\pi(v) = x$ there is a solution π' of W with $\pi'(v) \neq x$ and $|\pi'| \geq |\pi|$. $\pi'(w)$ must be either equal to $\pi(w)$ or not in conflict with any values of variables in $V \setminus W$. For $\pi'(v)$ the latter condition must hold. We have to consider the following three cases.

Case 1: $D_v = \{x\}$

All $y \in D_w$ must be in conflict with x , otherwise there would be a solution π of W with $\pi(v) = x$ and $|\pi| = 2$, and all solutions π' of W with $\pi'(v) \neq x$ would have size at most one, contradicting our assumption.

Thus x is excluded by w , and all $y \in D_w$ must be in conflict with values of variables $u \in V \setminus W$, otherwise we could have applied A3. Then, however, a solution π with $\pi(v) = x$ has size one, while all solutions π' with $\pi'(v) \neq x$ have size zero, since $\pi(w) \neq \emptyset$ would be in conflict with values of variables in $V \setminus W$. Thus x is not W -redundant, which yields the contradiction.

Case 2: $|D_v| \geq 2$ and there is a value $y \in D_w$ that is not in conflict with x .

Then for each such y there is a solution π_y of W with $|\pi_y| = 2$, namely $\pi_y(v, w) = (x, y)$. Due to our assumption for each π_y there must be a solution π'_y of W with $\pi'_y(v) \neq x$ and $|\pi'_y| = 2$. $\pi'_y(w)$ must be y , and y must constrain values of variables in $V \setminus W$, otherwise A2 would have applied and all values of v and w (among them x) would have been removed—except $\pi'_y(v)$ and $\pi'_y(w)$. Let $X = \{\pi'_y(v) \mid y \in$

D_w and y is not in conflict with x }. The condition of case 2 guarantees that $X \neq \emptyset$. All $x' \in X$ are at most in conflict with values of w ; this is due to the restrictions imposed on each solution π'_y . In this situation, however, A1 would have been applicable: for each value y of w that is not in conflict with x we have an $x' \in X$ that does not constrain y since x' and y are both part of solution π'_y . Thus x would have been removed from D_v , which contradicts our assumption.

Case 3: $|D_v| \geq 2$ and x is excluded by w .

Then a solution π of W with $\pi(v) = x$ has size one. Due to our assumption there must be a solution π' of W with $\pi(v) \neq x$ and size at least one. Suppose $x' := \pi'(v) \neq 0$. Then x' is at most in conflict with values of w , and those form a subset of the values that are in conflict with x . Thus the special case of A1 would have applied and x would have been removed.

Hence $\pi'(v) = 0$ and $y := \pi'(w) \neq 0$. In this case, however, A3 would have applied and x would have been removed from D_v , contradicting our assumption.

□

Lemma 2.10 *Suppose there is an oracle that answers question of the type “Given two variables v and w and a value $x \in D_v$, does x constrain at most values of w ?” in $O(d)$ time, and suppose that the predicate $R_{vw}(x, y)$ can be evaluated in constant time for any $x \in D_v$ and $y \in D_w$, then applying any of the rules A1 to A3 to a pair of variables $\{v, w\}$ requires at most $O(d^2)$ time.*

Proof. Let v and w be the two variables under consideration, and let $\alpha_{vw}(x)$ be the answer of the oracle applied to the variables v and w , and to a value x in D_v . For rules A1 and A3, we show that their application to (v, w) is in $O(d^2)$, then obviously the same holds for (w, v) .

Our algorithm for A1 is sketched in Figure 2.7. We assume that D_v and D_w are given as lists and that we can store an integer entry $b(y)$ with each $y \in D_w$. We initialize these entries with zero. Let X be a subset of D_v . Initially X is empty.

Our algorithm consists of two phases. In the first phase we collect in X all values $x \in D_v$ for which $\alpha_{vw}(x)$ is true, and set the entries $b(y)$ for each $y \in D_w$ to the number of $x \in X$ with $R_{vw}(x, y)$ true. The fact that each $b(y)$ equals this number is an invariant of our algorithm. In the second phase we actually remove the values from D_v for which A1 applies.

In phase 1 we go once through all values x of v . If $\alpha_{vw}(x)$ is true, we append x to X and go through D_w incrementing all $b(y)$ for which $R_{vw}(x, y)$ holds. If after this procedure $X = \emptyset$, we cannot remove any value and stop.

In phase 2 we go through D_v once more and test which values $x \in D_v$ we can remove given the entries $b(y)$ of each $y \in D_w$. In order to do so, for each

```

ALGO_A1(  $v, w; D_v, D_w, \alpha_{vw}, R_{vw}$  )
// phase 1: initialize the data structures  $X, a(D_v)$ , and  $b(D_w)$ 
 $X \leftarrow \emptyset$ 
for each  $y \in D_w$  do  $b(y) = 0$ 
for each  $x \in D_v$  do
   $a(x) \leftarrow \alpha_{vw}(x)$ 
  if  $a(x)$  then
    for each  $y \in D_w$  do if  $R_{vw}(x, y)$  then  $b(y) \leftarrow b(y) + 1$ 
     $X \leftarrow X \cup \{x\}$ 
  end
end
// phase 2: remove values from the domain of  $v$ 
for each  $x \in D_v$  do
  if  $a(x)$  then  $threshold \leftarrow 0$  else  $threshold \leftarrow 1$  end
   $can\_remove \leftarrow true$ 
  for each  $y \in D_w$  do
    if  $R_{vw}(x, y)$  and  $b(y) \leq threshold$  then  $can\_remove \leftarrow false$ 
  end
  if  $X \neq \{x\}$  and  $can\_remove$  then
     $D_v \leftarrow D_v \setminus \{x\}$ 
    if  $x \in X$  then
      for each  $y \in D_w$  do if  $R_{vw}(x, y)$  then  $b(y) \leftarrow b(y) - 1$ 
       $X \leftarrow X \setminus \{x\}$ 
    end
  end
end

```

Figure 2.7: The algorithm that implements rule A1.

$x \in D_v$, we go through D_w and check whether X covers each $y \in D_w$ that is not in conflict with x , i.e. if there is an $x' \in X$ that is not in conflict with y either. If we want to remove a value $x \in X$, we additionally have to make sure that $X \setminus \{x\}$ covers the same subset of D_w as X . The conditions for $x \in X$ ($x \notin X$) are fulfilled if the entries $b(y)$ for all $y \in D_w$ with $R_{vw}(x, y)$ are greater than 1 (0). If this is the case and $X \neq \{x\}$ holds, then we remove x from D_v . If additionally $x \in X$, we decrement the appropriate entries $b(y)$ and remove x from X . The condition $X \neq \{x\}$ ensures that we do not remove the last value x' of X . This can only happen if x' is excluded by w . Keeping x' in X is necessary to remove—in accordance with the special case of A1—all other values in $D_v \setminus X$ that are excluded by w .

Note that we do not attempt to find the set X with minimal cardinality such that X covers all $y \in D_w$ that are not in conflict with x . This would enable us to remove the maximum number of values x from D_v . However, such an attempt would correspond to solving the set-cover problem, which is

NP-complete in general [Kar72]. One cannot even expect that set cover can be approximated within a factor of $\ln N$, where N is the size of the set to be covered [Fei96]. Our objective is only to remove enough values of v such that no $\{v, w\}$ -redundant value of v remains.

For a time bound of this algorithm, observe that we need to ask the oracle $O(d)$ times, and for each of the $O(d)$ values of v we have to go through the $O(d)$ values of w at most three times. This yields a time complexity of $O(d^2)$ as desired. (The necessary operations on the set X can be done in constant time each if X is implemented by Boolean entries associated with each $x \in D_v$ and by a counter that keeps track of the current size of X .)

The algorithm for A1 is correct for the following two reasons. First, whenever we remove a candidate x , the current set X and the current Boolean entries of the values of w constitute a proof guaranteeing that A1 applies: the entry of each $y \in D_w$ that x does not constrain is marked true, thus there is a value $x' \in X$ that does not constrain y either. (If $|X| = 1$, and both x and the only element of X constrain all values of W then we can still remove x according to A1.) Note that we append to X only values x for which $\alpha_{vw}(x)$ is true.

Second, if no value is removed, there are two possibilities. If the algorithm terminates with $X = \emptyset$ then all values of v are in conflict with values of variables in $V \setminus \{w\}$ and A1 does not apply.

If $X \neq \emptyset$, suppose there is a value $x \in D_v$ and a subset $X' \neq \emptyset$ of $D_v \setminus \{x\}$ with the properties required for applying A1. We claim that then our algorithm for A1 then would have removed x from D_v . There are two cases.

Case 1: x is in conflict with values of variables in $V \setminus \{w\}$.

Then $\alpha_{vw}(x)$ is false, and x is only considered during phase 2. Since we assume that x is not removed, there must have been a value $y \in D_w$ with $R_{vw}(x, y)$ true and $b(y) = 0$. Due to our assumption, there must be a value $x' \in X'$ with $R_{vw}(x', y)$ and $\alpha_{vw}(x')$ true. If this is the case, however, $b(y)$ would have been greater than 0 when our algorithm processed x' during phase 2. The entries $b(\cdot)$ are never decreased from 1 to 0. Thus our assumption is contradicted.

Case 2: x is at most in conflict with values of w .

Then there must be a value $y \in D_w$ with $R_{vw}(x, y)$ and $b(y) = 1$ that prevented x from being removed in the second pass through D_v . Since $b(y)$ corresponds to the number of values x' in X with $R_{vw}(x', y)$ true, X did not contain such a value except x itself. During the second pass, no new values are added to X , and when the algorithm terminates, X contains all values of D_v with $\alpha_{vw}(x)$ true that have not been removed. Thus X' must be a subset of X . Since $x \notin X'$ there is no value in X' with $R_{vw}(x, y)$, which contradicts the assumption.

The algorithm for A2 is simple. We mark each value of v and w with the answer of the oracle. Then for each pair (x, y) of values of v and w with $\alpha_{vw}(x)$

and $\alpha_{wv}(y)$ true we check whether $R_{vw}(x, y)$ holds. If this is the case, we stop and delete all values of D_v and D_w other than x and y .

Applying A3 is also easy. For each value x of v we go through all values y of w and check $\alpha_{wv}(y)$ and $R_{vw}(x, y)$. If $R_{vw}(x, y)$ is false for all $y \in D_w$ and there is one y with $\alpha_{wv}(y)$ true, then we remove x from D_v .

It is clear that the algorithms for A2 and A3 are correct and do not require more than $O(d^2)$ time. \square

Lemma 2.11 *There is an algorithm, EI-1, that given an instance $(V, \mathcal{D}, \mathcal{R})$ of a MVS-CSP achieves edge-irreducibility in $O(d^3e)$ time under the conditions stated in Lemma 2.10.*

Proof. The structure of EI-1 is very similar to that of AC-3. First we put all e edges of the variable conflict graph $G(V, E)$ on a stack Q . While Q is not empty we take an edge $\{v, w\}$ from the stack and call $\text{REVISE}(v, w)$. REVISE applies rules A1 to A3 until no further value of v and w can be deleted. Note that our REVISE is symmetric; while the procedure of Mackworth makes the arc (v, w) of the (directed) variable constraint graph \vec{G} consistent, we make the edge $\{v, w\}$ of the (undirected) variable conflict graph G irreducible.

If REVISE eliminates values of v or w , we have to ensure edge-irreducibility of all edges of G that are incident to v and w , respectively—except $\{v, w\}$. Therefore we put these edges on Q and continue by calling REVISE for the following edge on Q .

Actually there is another point where EI-1 differs from AC-3 due to the difference between arc-consistency and edge-irreducibility. Edge-irreducibility induces node-irreducibility; but in the algorithm sketched so far we do not take variables without conflicts into account at all. To each of these variables we must apply the special sub-case of rule A1, i.e. we must remove all of its values except one. Obviously this can be done in $O(dn)$ total time given the variable conflict graph.

The algorithm EI-1 is correct for the following reasons. Due to the initialization of Q each edge $\{v, w\}$ is made irreducible at least once. An edge can only become reducible if (a) the domain of v or w changes or (b) a value of v or w loses its last conflict with values of variables other than v and w . Both kinds of changes are triggered by the removal of a value; namely a value of v , w , or of a variable u that is adjacent to v or w in G . In the latter case it is obvious that EI-1 puts $\{v, w\}$ on Q and makes $\{v, w\}$ irreducible again later. If a value of v or w is removed, REVISE was called for either $\{v, w\}$, $\{v, s\}$, or $\{t, w\}$, where s is a variable adjacent to v and t a variable adjacent to w in G . Lemma 2.9 guarantees that $\{v, w\}$ is made irreducible since we apply our rules A1 to A3 exhaustively. In the other two cases EI-1 puts $\{v, w\}$ on Q since $\{v, w\}$ is incident to $\{v, s\}$ and $\{t, w\}$, respectively. Later, when EI-1 takes $\{v, w\}$ from Q , the irreducibility of $\{v, w\}$ is reestablished.

The time bound of $O(d^3e)$ that Mackworth and Freuder [MF85] gave for

AC-3 applies to EI-1 as well. In Lemma 2.10 we proved that one application of rules A1 to A3 costs $O(d^2)$ time given the oracle mentioned there. Suppose we had such an oracle. We call an application of A1 to A3 *successful* if it leads to the removal of a value of at least one of the two participating variables. The rules are applied at most dn times successfully and for each edge $\{v, w\}$ at most $2d + 1$ times unsuccessfully, namely once for the edge we put on Q during initialization, and once for each of the $2d$ values we potentially remove from v and w . We can assume that G is connected otherwise we can treat each component separately. Thus $e \geq n - 1$, and the runtime of EI-1 sums up to $O(e + (dn + 2de) \cdot d^2) = O(d^3e)$ if we assume the existence of the oracle. \square

It would be simple to implement the required oracle to run in $O(d)$ time if we were willing to accept a storage consumption of $O(d^2e)$. In this case we could compute explicitly the value conflict graph $G_{\mathcal{D}}$, see Definition 2.4. Computing $G_{\mathcal{D}}$ from the given variable conflict graph costs $O(d^2e)$ time and space. Recall that the oracle $\alpha_{vw}(x)$ has to tell whether the value x of the variable v is only in conflict with values of the variable w . Given $G_{\mathcal{D}}$ the oracle's answer is "no" if the length of the adjacency list of $[v, x]$ is greater than the size of the domain of w . Otherwise the adjacency list of $[v, x]$ is short. Thus we simply have to check to which variable each entry of the list refers and answer "yes" if each of these variables is w , "no" otherwise. Since the domain of w has at most d elements, the oracle's answer can obviously be determined in $O(d)$ steps.

However, for large values of d (and n) such an approach would consume too much storage. Instead, we take advantage of ideas that Bessi ere used in order to speed up AC-6 and to lower its space requirements as compared to AC-4 [Bes94]. As mentioned before, AC-6 stores at most *one* support for each arc-value pair, while AC-4 stores *all* supports. Recall that a value x of a variable v has support on an arc $(v, w) \in \vec{E}$ if there is a $y \in D_w$ with $R_{vw}(x, y)$ true. If x has no support on (v, w) , x cannot be in the solution of a classical CSP and is therefore removed from the domain of v .

The data structure that AC-6 uses to keep track of which value has support on which arc works as follows. For each arc-value pair $[(v, w), x]$ with $x \in D_v$ and $(v, w) \in \vec{E}$, AC-6 keeps a list $S_{v,x}$ of all variable-value pairs that $[v, x]$ supports. If x is removed from the domain of v , all $[w, y]$ in $S_{v,x}$ must get new support. If it turns out that there is none, y must be removed from the domain of w . The other "trick" Bessi ere introduced is that he does not go through all values of v when looking for new support for $[w, y]$. He observes that it is useless to check values of v that have been checked before. Instead, he assumes that the domains are given as lists, i.e. with an arbitrary but fixed total order, and only checks those values z of v that succeed x in the domain list of v . Thus, for each arc-value pair $[(w, v), y]$, he can bound the time required for searching new support for y by $O(d)$. Since there are $O(de)$ arc-value pairs, his support data structure can be initialized and maintained in $O(d^2e)$ total time using $O(de)$ space. The following lemma shows how we can use these ideas for a space- and time-efficient oracle data structure for EI-1.

Lemma 2.12 *There is a data structure that implements the oracle of Lemma 2.10 for EI-1. It takes $O(de)$ storage and can be maintained in $O(d^2e)$ total time during the execution of EI-1.*

Proof. For each edge-value pair $[\{v, w\}, x]$ with $\{v, w\} \in E$ and $x \in D_v$, we keep a witness (i.e. a kind of support) $[u, z]$ for the answer “no” of the oracle. The witness testifies that the value x of v is in conflict with the value z of a variable $u \neq w$. Like in Bessière’s case, it is enough to have one such witness per edge-value pair, and it is useless to check twice whether a value is a witness for a given edge-value pair. Thus we can apply his ideas.

We keep a list $W_{u,z}$ with all variable-value pairs for which z is a witness. In addition, we store a Boolean entry with every edge-value pair $[\{v, w\}, x]$ that encodes $\alpha_{vw}(x)$, the answer of the oracle. Suppose all these entries are correct before we remove the value z of a variable u . After the removal, for each $[v, x]$ in the list $W_{u,z}$ we must find a new witness or change its Boolean entry if no further witness exists. We can do this exactly as Bessière’s search for new support, i.e. in $O(d)$ time. The initialization is similar to his as well — except that we do not remove values without witness, but only change their Boolean entry. The Boolean entries require $O(de)$ storage; so do the lists of type $W_{u,z}$. Thus our witness data structure can be maintained in $O(d^2e)$ total time using $O(de)$ space. \square

Now it is clear that EI-1 requires no more than $O(de)$ storage. Combining Lemmas 2.11 and 2.12 yields our main result concerning the algorithm EI-1.

Theorem 2.13 *Given an instance $(V, \mathcal{D}, \mathcal{R})$ of a MVS-CSP, the algorithm EI-1 achieves edge-irreducibility in $O(d^3e)$ time if all predicates R_{vw} in \mathcal{R} can be evaluated in constant time for any $v, w \in V$, $x \in D_v$, and $y \in D_w$. EI-1 requires $O(de)$ storage.*

2.5 A General Label-Placement Algorithm

In this section we suggest a new algorithm for the general label-number maximization problem. Our algorithm is a combination of EI-1 and a heuristic that removes additional candidates. The heuristic chooses a candidate c according to the *conflict number* of c , i.e. the number of candidates of other features that c intersects.

Our algorithm is simple, but has turned out to be very effective. In Section 3.2 we give experimental results obtained in the context of point labeling. Our hope is that EI-1* or other efficient algorithms based on higher degrees of irreducibility will substitute simulated annealing and other iterative methods of gradient descent for the wide variety of problems that fit into the framework of maximum variable-subset constraint satisfaction.

We proceed as follows. Given an instance of a maximum label-number problem (F, \mathcal{D}) , where F is a set of n features and \mathcal{D} contains a set D_f of at

most d candidates for each feature $f \in F$, we first compute the *candidate conflict graph* G_{cand} , the equivalent to the value conflict graph $G_{\mathcal{D}}$ in Definition 2.4. In G_{cand} there is a vertex for each feature-candidate pair and an edge for each pair of intersecting candidates that belong to different features.

We use G_{cand} to maintain the conflict number for each candidate. Our invariant is that the conflict number of a candidate c is always equal to the degree of c in G_{cand} , i.e. to the number of edges incident to c . If we remove c from our label-placement instance, we decrement the conflict number of all candidates whose vertices are adjacent to that of c . Then we delete the vertex $v(c)$ corresponding to c and the edges incident to $v(c)$ from G_{cand} .

Recall that REVISE needs constant-time access to the relation R_{fg} for each pair of candidates of f and g . If an intersection test for a pair of candidates can be done in constant time, we can compute the candidate conflict graph in $O((dn)^2)$. It requires $O(k)$ space where k is the number of edges in the graph. If we are given the *feature conflict graph* (the equivalent to the variable conflict graph in Definition 2.4) we can use a data structure similar to the witness data structure suggested in the proof of Theorem 2.13. With this data structure we can initialize and maintain the conflict number of all candidates in $O(d^2e)$ time using $O(de)$ space. For large values of d this is better than transforming the feature conflict graph into a candidate conflict graph that requires $O(k) \subseteq O(d^2e)$ space.

This observation is useful for high-quality point labeling if each point has a large set of candidates and each candidate is an axis-parallel rectangle¹. If for each point the union of its initial label candidates forms an axis-parallel rectangle¹, one can compute the feature conflict graph in $O(e + n \log n)$ independently of d . An alternative would be to allow an infinite number of candidates and use algorithms for so-called slider models, see Section 3.3.

In case label candidates have more complex shapes with at most s edges and an intersection test needs $f(s)$ time for some function f , computing the candidate conflict graph takes $O(f(s)(dn)^2)$ time in general. To ensure fast access to R_{fg} , the graph can be stored in an adjacency matrix. This requires $O((dn)^2)$ space. However, a careful revision of the proof of Lemma 2.10 shows that constant-time access to R_{fg} is only needed in loops over all pairs of candidates $[b, c]$ of f and g , respectively. Thus representing $G_{\mathcal{D}}$ by ordered adjacency lists suffices. Since time and space for constructing and storing the conflict graph are application-dependent, we assume for the following time and space bounds that G_{cand} is given.

Next we interpret the maximum label-number problem (F, \mathcal{D}) as a MVS-CSP $(F, \mathcal{D}, \mathcal{R})$ by identifying each feature with a variable and each candidate with a value. We set $R_{fg}(b, c)$ to false if candidate b of feature f overlaps candidate c of feature g ($g \neq f$), or if this combination is not desired due to some other application-dependent restriction.

Given $(F, \mathcal{D}, \mathcal{R})$, we use EI-1 to achieve edge-irreducibility, then remove a

¹there are similar results for other shapes of constant complexity

candidate c of some feature f by means of a heuristic, and call $\text{REVISE}(f, g)$ for each feature g that was in conflict with f . This process is repeated until each feature has at most one candidate left and no candidates are in conflict any more.

We suggest the following two heuristics for determining the candidate c that is to be removed next. Both base their decision on the conflict number of c .

RemoveTroubleMaker removes the candidate with the greatest conflict number, either locally, i.e. among the candidates of the current feature, or globally. For the local version, the next feature either is the successor of the current feature in a list containing all features, or it can be a feature that still has the maximum number of candidates (MaxCandNumber).

TakeGoodChild does in a sense the opposite of what **RemoveTroubleMaker** does. Among the candidates of the current feature f this heuristic selects the candidate c with the smallest conflict number, puts c in the solution, and removes all other candidates of f and all those that intersect c . Again, the search for c can be local or global. For the local version, the next feature is either the successor of f in F or a feature with the minimum number of candidates.

Let EI-1^* be the algorithm that combines EI-1 with **RemoveLocalTroubleMakerMaxCandNumber**, i.e. the local version of heuristic **RemoveTroubleMaker** and selection according to MaxCandNumber . EI-1^* operates on a given candidate conflict graph.

Using no data structures other than doubly connected lists, EI-1^* requires $O(d^2n)$ total time to repeatedly select the next candidate to be removed. Removing all of these candidates can cause at most $O(de)$ unsuccessful applications of the rules A1 to A3, using ideas and terminology of the proof of Theorem 2.13. Since the rules are applied at most $O(nd)$ times successfully, and each application requires $O(d^2)$ time, EI-1^* has a time complexity of $O(d^3e)$.

We conclude with the following lemma. It is simple, but important for applying the concept of edge-irreducibility in practice. A formal proof is omitted since the basic ideas have been sketched above. The proof would use Lemma 2.9 and Lemma 2.10.

Lemma 2.14 *Given an instance of a maximum label-number problem (F, \mathcal{D}) , where F is a set of n features and \mathcal{D} contains a set C_f of at most d candidates for each feature f in F , and given the corresponding candidate conflict graph G_{cand} , there is an algorithm, EI-1^* , that finds a solution π of (F, \mathcal{D}) in $O(d^3e)$ time and requires $O(de)$ space, where e is the number of pairs of features with conflicting candidates.*

Let the predicate $R_{fg}(b, c)$ be true iff candidate b of feature f does not intersect candidate c of feature g , and let \mathcal{R} be the set of predicates R_{fg} , one for each pair $\{f, g\} \subseteq F$ of features. Then π is optimal if for the MVS-CSP $(F, \mathcal{D}, \mathcal{R})$ edge-irreducibility implies $|F|$ -irreducibility.

Chapter 3

Point Labeling: Label-Number Maximization

Generally it is assumed that a point label can be seen as an axis-parallel rectangle; the bounding box of the text, see Figure 3.1. Many algorithms for point labeling have been described in the automated cartography literature and in computational geometry. For an extensive bibliography see [WS96].

Good point labeling has two basic requirements. A label should be placed close to the point to which it belongs, and two labels should not overlap each other. For high quality cartographic label placement, further requirements have been formulated [Imh75, Yoe72]. Given the basic requirements, an algorithm can try to either label as many points as possible, or find the largest possible font such that all points can be labeled. In general, both of these problems are NP-hard [FW91, MS91]. In this chapter, we focus on the former problem, i.e. label-number maximization. As in Chapter 2, we are given a set of features (here: points) and for each point a set of label candidates. A solution is a function that maps every point to 0 or to a label from its set of candidates such that no two labels intersect. The size of a solution is the number of points that receive a label. An optimal solution is a solution of maximum size.

Although finding an optimal solution is NP-hard, approximation algorithms have been suggested. For axis-parallel rectangular labels of arbitrary height and width, Agarwal et al. propose an algorithm with an approximation ratio of $1/O(\log n)$ [AvKS98]. Their algorithm is based on divide-and-conquer. If the label height (or width) is fixed, the same paper suggests a line-stabbing algorithm that labels in $O(n \log n)$ time at least half the number of points that are labeled in an optimal solution. For maximizing the *size* of uniform rectangular labels, this approximation factor is optimal, but for maximizing the *number* of fixed-height labels, Agarwal et al. also present a polynomial time approximation scheme (PTAS).

Nearly all of the existing algorithms for point labeling limit the placement of a label with respect to its point to a finite number of label positions. Most algorithms described before allow four label candidates, namely those where a

rectangular label touches its point in one of its four corners [FW91, WW97]. In the automated cartography literature eight candidates per point is also quite common, while the approximation algorithm of Agarwal et al. [AvKS98] allows any constant number as does a heuristic proposed by Kakoulis and Tollis [KT98]. Their algorithm is based on a heuristic for splitting connected components into cliques and uses maximum-cardinality bipartite matching.

We call restrictions of the allowed label positions a *labeling model*. Models that allow a finite number of positions per label are *fixed-position models*, those that allow an infinite number are *slider models*.



Figure 3.1: Rectangular labels of cities of the U.S.A.

This chapter is structured as follows. Section 3.1 introduces six point-labeling models; three fixed-position and three slider models. We analyze how many more labels can be placed in one model than another, in theory.

In Section 3.2 we specialize the general concept of Chapter 2 to the context of point labeling and give the details of three variants of an algorithm based on this concept. Our algorithm has a runtime of $O(k + n \log n)$, where n is the number of points and k the number of intersections among the label candidates. Other than all approximation algorithms suggested so far, our algorithm does not make any assumptions about label shapes and the position of a label relative to its point. Due to this generality we could not expect to prove any approximation factors. However, our algorithm works very well in practice. We compare it experimentally to a number of other methods.

In Section 3.3 we drop the restriction that a label can only be placed at a finite number of positions. Instead, we allow any position where an edge of the label is incident to the point, see Figure 3.1. We show that it is NP-

complete to decide whether a set of points can be labeled with unit squares in the four-slider model. However, each of our three slider models allows a simple factor- $\frac{1}{2}$ approximation algorithm that uses $O(n)$ space and $O(n \log n)$ time. We also give a polynomial-time approximation scheme for each of our slider models. Similar results were already known for fixed-position models [AvKS98]. In order to support the practical relevance of our approximation algorithms for the three slider models, we do a thorough experimental analysis on real-world data and randomly generated point sets.

3.1 Comparing Various Models

This section is joint work with Marc van Kreveld and Tycho Strijk, both Universiteit Utrecht [vKSW98, vKSW99].

In this section we introduce and then compare some common point-labeling models. All of the algorithms we present in the following sections aim to label as many points as possible according to the chosen model.

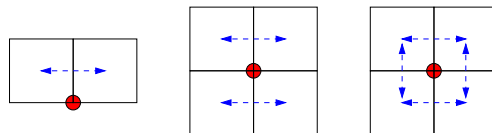


Figure 3.2: top-, two- and four-slider model

Definition 3.1 (point labeling, size of a labeling, optimum labeling)

Given a set P of n points in the plane, and for each point $p \in P$ a set of label candidates L_p , a point labeling is a subset $P' \subseteq P$ and a function λ which maps every point $p \in P'$ to a label $\lambda(p) \in L_p$ such that no two labels intersect. The number of labeled points, i.e. the cardinality of P' , is the size of the point labeling. An optimal labeling is a point labeling which has maximum size among all point labelings.

In this chapter, we restrict ourselves to axis-parallel rectangular label candidates. If we require additionally that a label must be placed such that one of its edges contains the point to be labeled, we get the following labeling models.

Definition 3.2 (slider models) In the four-slider model, a point p must be labeled such that any edge of the label contains p . In the two-slider model, either the label's top or bottom edge has to contain p . In the one-slider or top-slider model, the bottom edge of a label must contain p .

For an illustration of slider models, see Figure 3.2. Note that in all of our models we allow that a label contains other points which then of course cannot be labeled. Our labels are closed, i.e., we disallow touching. One alternative

would be “half-open” labels as in [WW97]. In that paper all edges of a label which are not adjacent to its point are allowed to touch other labels or points. This would make sure that if every label is scaled down by a small amount with its point as scaling center, then all labels are disjoint. When labels do not touch, a map user can more easily match a label and the point to which it belongs. The algorithms could be adjusted to this additional requirement, but intensive case study would be necessary to decide whether a label can be placed when it touches other labels. The bounds of the following comparison of models would still hold, but for the sake of simplicity we keep the number of requirements to a minimum.

One alternative would be to consider labels open and thus allow touching generally. In this case however, we were not able to keep the greedy algorithm’s approximation guarantee of 50%, although the bounds of the comparison below would hold.

We will compare the slider models introduced above to the following fixed-position models.

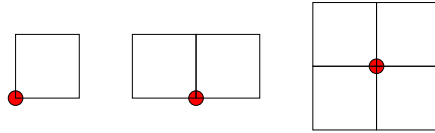


Figure 3.3: one-, two- and four-position model

Definition 3.3 (fixed-position models) *Labeling in the four-position model requires that the a point p is labeled such that one of the label’s corners lies on p . In the two-position model one of the label’s bottom corners must lie on p and in the one-position model the lower left corner of the label must coincide with p .*

For an illustration of fixed-position models, see Figure 3.3. Our measure for comparing the models above is based on optimal labelings of point sets. Some point sets allow a labeling of the whole set in all models. Such point sets are not very interesting for a comparison, so we are mainly interested in point sets where the size of an optimal labeling differs from model to model. We define the *ratio* of two models as follows.

Definition 3.4 (ratio of two models) *Given unit square label candidates and two label-placement models M_1 and M_2 , the (asymptotic) $(M_1 : M_2)$ -ratio is*

$$\lim_{n \rightarrow \infty} \max_{P, |P|=n} \frac{\text{size}(\text{optimal } M_1\text{-labeling for } P)}{\text{size}(\text{optimal } M_2\text{-labeling for } P)}.$$

This measure does *not* take into account aesthetic criteria as listed by Imhof [Imh75]. Since it is a purely quantitative measure and, moreover, only refers

to square labels, it does not apply directly to many practical label placement problems. However, it gives a fair indication of how many more points can be labeled in one model than in another in general.

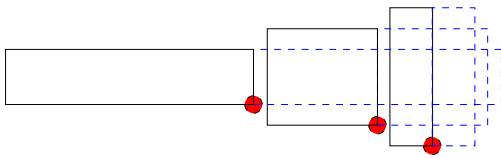


Figure 3.4: The ratio between the two- and the one-position model can become arbitrarily large for labels of different size.

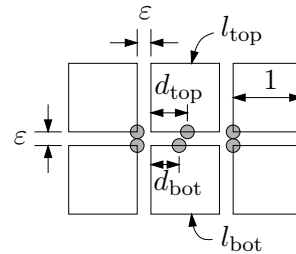


Figure 3.5: $3/2$ is a lower bound on the ratio between the two- or four-slider and any fixed-position model

The reason why we only consider unit square labels in the definition above and in the remainder of this section, is that otherwise some of the ratios between two models would become arbitrarily bad, see Figure 3.4. All points depicted there can be labeled in the two-position model, but only one point can be labeled in the one-position model.

It is worth mentioning that the size of an optimal placement in a slider model cannot be approximated arbitrarily well by a fixed-position model, no matter at how many discrete positions a fixed-position label can be attached to its point. Given such a model, consider all positions in which a unit square label can be attached to its point. W.l.o.g. we may assume that the four corner positions are among them. For each position, mark the place on the edge of the label that the point touches. Choose some $\varepsilon > 0$ to be smaller than half the minimum distance between two markers that both lie either on the top or on the bottom edge of the label. Then there must be points p_{top} and p_{bot} on the top and the bottom label edge, respectively, that are further than ε away from any marker. Let d_{top} and d_{bot} be their respective distances to the label's left edge.

Now consider the six points marked by disks in Figure 3.5. The two leftmost points have vertical distance ε from each other and horizontal distance $1 + 2\varepsilon$ from the corresponding rightmost points. These four points can be labeled in all models that allow any corner of a label to lie on the point to be labeled. The other two points lie at a distance of $\varepsilon + d_{\text{top}}$ and $\varepsilon + d_{\text{bot}}$ to the right of the leftmost points. These two points can be labeled by labels l_{top} and l_{bot} in the two- or four-slider model such that l_{top} and l_{bot} keep a distance of ε to the labels of the left- and rightmost points. However these points cannot be labeled properly in the given fixed-position model since moving l_{top} and l_{bot} by up to ε to the right or left does not make the points coincide with any of the markers at the top or bottom edge of l_{top} and l_{bot} . This is due to our choice of d_{top} and d_{bot} .

In order to get larger point sets as required by Definition 3.4, we simply copy the group of six points depicted in Figure 3.5 at regular intervals along the x -axis. This yields a ratio of $3/2$ between the two- (or four-) slider model and any given fixed-position model.

The labeling models used in this section will be the six introduced in Definition 3.2 and 3.3. All of our comparisons of two such models M_1 and M_2 are based on the following strategy. We want to bound the ratio Ψ by which more labels can be placed in the model with more freedom, say M_1 . We assume an optimal label placement in M_1 . Then we canonically relabel the labeled points by moving every label into a position that is valid in the more restrictive model M_2 . This might cause some labels to intersect. We determine the maximum number δ_{left} of M_2 -labels that intersect the leftmost M_2 -label l . Then we put l into a set S of non-intersecting labels, remove l and all its conflicting labels from the instance and repeat until no labels remain. At the end of the process, S contains at least $k_{\text{opt}}^1/(\delta_{\text{left}} + 1)$ non-intersecting M_2 -labels, where k_{opt}^1 is the size of the assumed optimal M_1 -placement. The size of S is a lower bound for the size of an optimal M_2 -placement, thus $\delta_{\text{left}} + 1$ is an upper bound for the $(M_1 : M_2)$ -ratio. Lower bounds for the ratio Ψ are obtained by giving examples of arbitrary size for which any M_2 -placement is worse by a certain factor than some M_1 -placement.

Since we do not want to compare every two models in isolation, we define three groups. They consist of pairs of models where points with labels in one model can be canonically relabeled such that a certain fraction of points gets labels in the other model.

Definition 3.5 (flipping) *Given two different label placement models M_1 and M_2 , and an axis-parallel vector v of unit length, model M_1 can be flipped into model M_2 by v if any label position in M_1 that is not allowed in M_2 can be translated by v into a valid M_2 -label position.*

Example 3.6 The two-slider model can be flipped into the top-slider model by $(0, 1)$. Analogously, the four-position model can be flipped by $(0, 1)$ into the two-position model, while the two-position model can be flipped by $(1, 0)$ into the one-position model.

Lemma 3.7 *For any two labeling models M_1 and M_2 where M_1 can be flipped into M_2 the $(M_1 : M_2)$ -ratio is 2.*

Proof. Consider an optimal M_1 -labeling of an arbitrary instance of points. Let M_2 be a model into which M_1 can be flipped by a vector v . Then the canonical relabeling mentioned above means translating by v all M_1 -labels that are not valid in M_2 .

We can assume that the vector by which we flip is $(0, 1)$; the case $(1, 0)$ is symmetric. This means that an M_2 -label is either identical to the corresponding M_1 -label or lies one unit above it. Let l_1 be the M_1 -label corresponding to the

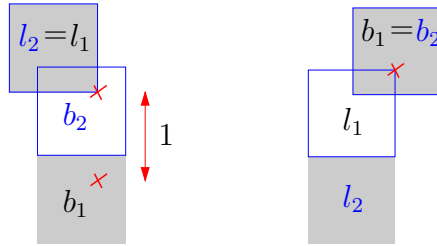


Figure 3.6: If M_1 can be flipped into M_2 then the leftmost M_2 -label l_2 (solid edges) cannot intersect more than one M_2 -label b_2 . (The corresponding non-intersecting M_1 -labels are shaded.)

leftmost M_2 -label l_2 . We show that l_2 can intersect at most one M_2 -label whose M_1 -counterpart is not in conflict with l_1 . As indicated above, this gives us an upper bound of 2 for the $(M_1 : M_2)$ -ratio Ψ .

Suppose that l_2 is identical to the corresponding M_1 -label l_1 ; the other case is symmetric, see the left and right part of Figure 3.6, respectively. Let I_2 be the set of all M_2 -labels intersecting l_2 and let I_1 be the set of their mutually non-intersecting M_1 -counterparts. Then all labels in I_2 must contain the lower right corner of l_2 ; otherwise, either their M_1 -counterparts intersect l_1 , or l_2 is not leftmost. This however forces all labels in I_1 to contain a point at unit distance below that corner (marked by a cross in Figure 3.6) in order not to intersect l_1 . Hence $|I_1| = |I_2| \leq 1$ and $\Psi \leq 2$.

In order to establish the lower bound of 2 for Ψ , just take the four corner points of an axis-parallel square of edge length less than one. For all models M_1 that we are considering and that can be flipped into a model M_2 (see Example 3.6), exactly twice as many of these points can be labeled as in the corresponding M_2 -model. An instance can consist of arbitrarily many of such groups of four points, separated sufficiently. \square

Definition 3.8 (sliding) *Given two different label placement models M_1 and M_2 , and an axis-parallel vector v of unit length, model M_1 can be slid into model M_2 along v if every label position in M_1 can be translated by μv into a valid M_2 -label position for some $\mu \in [0, 1]$*

Example 3.9 The four-slider model can be slid into both the two-slider and the top-slider model along $(0, 1)$. Along $(1, 0)$ we can slide the two-slider into the four-position model and the top-slider into both the two- and the one-position model. Note that the four-slider model cannot be slid into the four-position model.

Lemma 3.10 *Let M_1 and M_2 be two (different) labeling models where M_1 can be slid into M_2 , and let Ψ be the $(M_1 : M_2)$ -ratio. Then $2 \leq \Psi \leq 3$.*

Proof. Again we consider an optimal M_1 -labeling of an arbitrary instance. We assume that we can slide M_1 - into M_2 -label positions along $(0, 1)$; the case $(1, 0)$ is symmetric. We canonically slide all M_1 -labels upwards until we arrive in an M_2 -label position. We show that the leftmost M_2 -label l_2 can then intersect at most two other M_2 -labels. This yields the upper bound of 3 for Ψ .

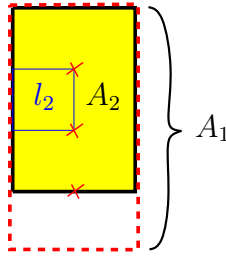


Figure 3.7: If M_1 can be slid into M_2 then the leftmost M_2 -label l_2 cannot intersect more than two M_2 -labels.

M_2 -labels intersecting l_2 can only lie within area A_2 , a rectangle of width two and height three that is placed such that its left edge is centered at the left edge of l_2 , see Figure 3.7. This holds because l_2 is chosen to be leftmost. The corresponding M_1 -labels are restricted to area A_1 , a rectangle of width two and height four obtained by extending A_2 one unit downwards. Every label in A_1 must contain one of the three grid points in the interior of A_1 marked by crosses in Figure 3.7. Thus A_1 can contain only three non-intersecting M_1 -labels including the M_1 -counterpart of l_2 . It follows that l_2 cannot intersect more than two M_2 -labels, and hence that $\Psi \leq 3$.

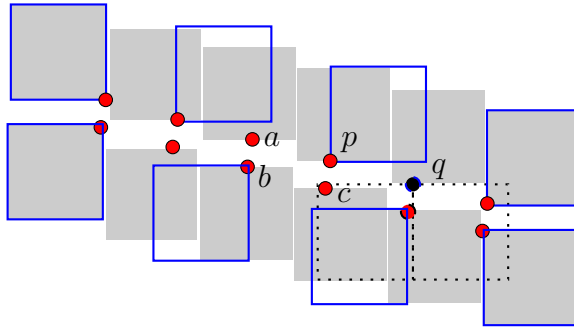


Figure 3.8: If M_1 can be slid into M_2 then the M_1 - M_2 -ratio approaches 2. Here we chose M_1 to be the two-slider model (shaded labels) and M_2 to be the four-position model (solid edges).

For a lower bound that approaches 2 refer to Figure 3.8. There are two rows of n points. Two neighboring points of one row have x -distance $1 - \frac{1}{n-1} + \varepsilon$ and y -distance δ , where $\frac{1}{n-1} > \delta > \varepsilon > 0$. The upper row is a copy of the lower, shifted by the vector $(\delta/2, \delta)$.

Comparing the top-slider model to the one- or two-position model is easy;

just consider one row. In order to compare the four- to the two-slider model, the figure must be rotated by 90 degrees. So let us focus on comparing the two-slider to the four-position model here.

It is obvious that all points can be labeled in the two-slider model. For the four-position model we can argue as follows. Let p be a point of the upper row excluding the last or first two points and let q be the right neighbor of p in the upper row. If a four-position label is attached to p , then either it contains at least one extra point (a , b and c , or q in Figure 3.8), or it makes labeling q difficult. Either q is not labeled, or q 's label is in a lower position and hence q will contain at least two extra points. Since p is the only point whose label can intersect the upper positions of q without intersecting q itself, a failure to label q can be uniquely charged to p . And if p and q are both labeled, we charge the failure to label the two points in q 's label to p and q . In any case, we can charge one point that cannot be labeled to each point that is labeled. For the corresponding points p of the lower row, the same argument holds if we choose q to be the left neighbor of p . \square

Note that the proof above can be simplified for closed labels. We chose to give a proof that carries over to the case of open labels.

The upper bounds for Ψ can be improved to 3 for the pairs of models satisfying the following definition.

Definition 3.11 (corner-sliding) *Given two different label placement models M_1 and M_2 , model M_1 can be corner-slid into model M_2 if every label position in M_1 can be shifted both to the left and to the right such that the point coincides with a corner of the label, and these positions are valid in M_2 . Vertical corner-sliding is defined with left and right replaced by top and bottom.*

Example 3.12 The top-slider model can be corner-slid into the two-position or the four-position model. We can corner-slide the two-slider model into the four-position model. The four-slider model can be vertically corner-slid into the two-slider model. Note that the four-slider model cannot be corner-slid into the four-position model since sliding would be necessary in two directions. The top-slider model cannot be corner-slid into the one-position model since top-slider labels cannot be slid to the left to reach a position in the one-position model.

Lemma 3.13 *Let M_1 and M_2 be two (different) labeling models where M_1 can be corner-slid into M_2 . Then the $(M_1 : M_2)$ -ratio is at most 2. The same holds if M_1 can be vertically corner-slid into M_2 .*

Proof. Again we consider an optimal M_1 -labeling of an arbitrary instance. We assume that we can corner-slide M_1 - into M_2 -label positions; the vertical corner-sliding case is symmetric. We draw a set of vertical lines with unit spacing over the M_1 -labeling such that no line contains a point of the instance, nor a vertical

edge of a label. We count both the number of M_1 -labels that intersect the odd lines and the number of M_1 -labels that intersect the even lines. Assume that the even lines intersect the greater number of squares in the M_1 -labeling; the other case is symmetric. Delete the squares and corresponding points intersecting the odd lines. The remaining squares are now corner-slid into a M_2 -label position such that each label intersects an even line, see Figure 3.9.

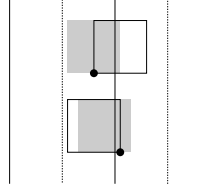


Figure 3.9: After removing M_1 -labels intersecting the odd lines (dashed), the remaining M_1 -labels (shaded) are corner-slid to intersect an even line (solid). This results in a valid M_2 -labeling.

Note that if a given M_1 -label intersects an even line, then the resulting label in the M_2 -position intersects that same even line. Since the spacing between even lines is 2 and the lines are in non-degenerate position, two M_2 -labels intersecting different even lines cannot intersect. Furthermore, two M_2 -labels intersecting the same even line arose from two M_1 -labels intersecting that same even line. Since corner-sliding is done horizontally, the M_2 -labels do not intersect since the M_1 -labels did not intersect.

Since we never remove more than half the M_1 -labels, and the remaining M_1 -labels are all corner-slid to non-intersecting M_2 -labels, the $(M_1 : M_2)$ -ratio is at most 2. \square

Lemma 3.14 *Let $\Psi_{1S,1P}$ be the ratio between the top-slider and the one-position-model. Then $2\frac{1}{4} \leq \Psi_{1S,1P} \leq 3$*

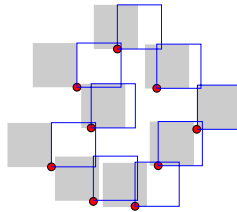


Figure 3.10: Sliding top-slider labels (shaded) to the right into one-position labels (solid edges) can create 9-cycles in the resulting conflict graph of one-position labels.

Proof. With Lemma 3.10 we get $2 \leq \Psi_{1S,1P} \leq 3$. The example in Figure 3.10 raises the lower bound to $2\frac{1}{4}$. \square

Unfortunately we were not able to lower the upper bound of $\Psi_{1S,1P}$ although we can show the following. In order to get from an optimal top-slider labeling to a one-position labeling, we *must* shift all labels to the extreme right. Note that this is not the same as the idea of a canonical relabeling. If we consider the resulting conflict graph of the one-position labels, then this graph is planar, has maximum degree $\Delta = 4$ and all odd cycles have length at least 9. Grötsch's Theorem says that a planar and triangle-free graph is three-colorable, but this gives us only a more graph-theoretic proof of the upper bound of 3.

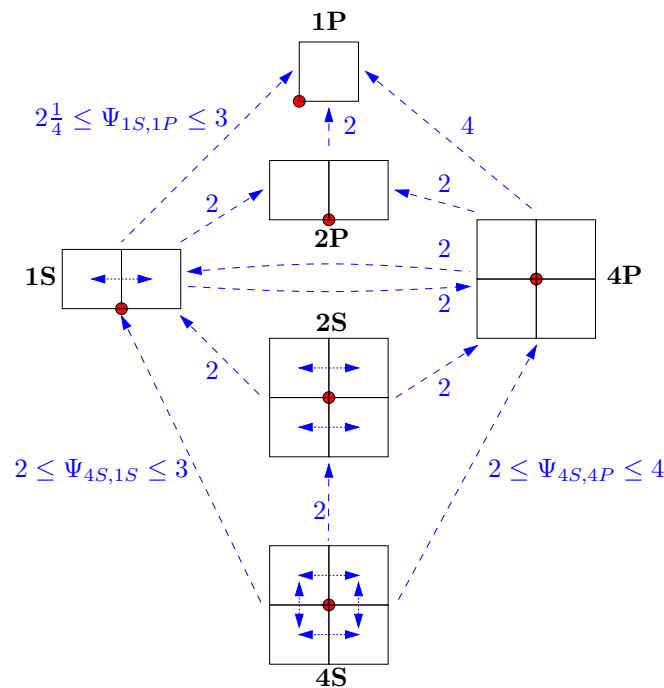


Figure 3.11: Ratios between some label placement models. From top to bottom: the one-position, two-position, top-slider (left), four-position (right), two- and four-slider model

Figure 3.11 summarizes our results. The reason for the upper bound 4 for the ratio $\Psi_{4S,4P}$ between the four-slider and the four-position model is the following. First we convert the four-slider labeling to a two-slider labeling as described in the proof of Lemma 3.13. After that we convert this two-slider labeling to a four-position labeling in the same way. Since both these conversions keep at least half the number of labels, we get $\Psi_{4S,4P} \leq 4$.

3.2 Fixed-Position Models

In this section we specialize the general concept of Chapter 2 to the context of point labeling. First, however, we review some of the previous approaches to maximizing the number of labeled points. For an extensive bibliography on label placement in general, refer to [WS96]. At the beginning of this chapter we have already mentioned a number of approximation algorithms. Let us add the most prominent heuristic methods.

In [CMS95] Christensen et al. compare simulated annealing to gradient descent, to an incremental force-driven algorithm by Hirsch [Hir82], and to 0-1 integer programming suggested by Zoraster [Zor86, Zor90]. The conclusion of their comparison is that simulated annealing is the method of choice for point labeling, see the results in Figure 3.72, page 79. Zoraster later also applied the simulated-annealing algorithm of Christensen et al. to labeling very dense point sets and reported good results [Zor97]. Edmondson et al. showed that the simulated annealing algorithm of Christensen et al. can also be used for general label placement, i.e. for arbitrary feature and label shapes [ECMS97].

Recently, Kakoulis and Tollis suggested another approach to label-number maximization [KT98] that is independent of the shape of the label candidates. The authors compute the *candidate conflict graph* G_{cand} . G_{cand} has a node for each candidate and an edge for each pair of intersecting candidates. In the next step they compute the connected components of G_{cand} . Then Kakoulis and Tollis use a heuristic similar to the greedy algorithm for maximum independent set to split these components into cliques. Finally they construct a bipartite “matching graph” whose nodes are the cliques of the previous step and the features of the instance. In this graph, a feature and a clique are joined by an edge if the clique contains a candidate of the feature. A maximum-cardinality matching yields the labeling. Given G_{cand} , the runtime of their algorithm depends on how the clique check and the matching algorithm are implemented. Practical matching algorithms take $O(k\sqrt{n})$ time; however, our implementation of their heuristic has an asymptotic runtime of $O(kn)$, where k refers to the number of edges in the candidate conflict graph. The authors do not give any time bounds.

The algorithm for label-number maximization we present here unites the following advantages. Our algorithmic approach

- does not depend on the shape of labels,
- can be applied to point, line, or area labeling (even simultaneously) if a finite set of label candidates has been precomputed for each feature,
- is easy to implement,
- runs fast (i.e. in $O(n + k)$ time given the candidate conflict graph), and
- returns good results in practice—at least for labeling points with rectangular labels.

To our knowledge, none of the algorithms described so far shares all of these

characteristics.

The input to our algorithm is the candidate conflict graph of the label candidates. The algorithm is divided into two phases similar to the first two phases of the algorithm for label size maximization described in [WW97]. In phase I, we apply a set of rules to all features in order to label as many of them as possible and to reduce the number of label candidates of the others. These rules do not destroy a possible optimal placement. Then, in phase II, we heuristically reduce the number of label candidates of each feature to at most one. Given the candidate conflict graph, our algorithm takes $O(k+n)$ time and $O(n)$ space.

In Section 3.2.1 we give the details of three variants of the algorithm based on this concept. In Section 3.2.2 we describe the set-up and in Section 3.2.3 the results of our experiments. We compare our algorithm to five other methods, namely simulated annealing, a greedy algorithm (see Section 3.3.2), two variants of the matching heuristic of Kakoulis and Tollis, and a hybrid algorithm that combines our rules with their clique matching.

Part of the examples on which we do the comparison are benchmarks that were already used in [WW97]. We added examples for placing rectangular labels of varying size, both randomly generated and from real world data. Our samples come from a variety of sources; they include the location of some 19,400 ground-water drill holes in Munich, 373 German railway stations, and 357 shops. The latter are marked on a tourist map of Berlin that is labeled on-line by our algorithm. The algorithm is also used by the city authorities of Munich to label drill-hole maps. All example generators, real world data and algorithms are available on the World Wide Web¹.

3.2.1 Algorithm

As an application of the ideas of the previous chapter, we describe three closely related variants of a label-placement algorithm. Although these algorithms do not make any assumptions about the features to be labeled or the label candidates, our description is based on the context of point labeling. This simplifies presentation and experimental evaluation. We used four rectangular, axis-parallel label candidates per point, namely those where one of the label's corners is identical to the point. Our objective is to label as many points as possible.

Each of our algorithms consists of two phases. In phase I, we try to remove as many label candidates as possible without destroying an optimal placement. Then, in phase II, we heuristically pick a candidate, remove it, and fall back on the methods of phase I to further reduce the number of label candidates. We repeat this process until each point has at most one label candidate left, and none of these intersects any of the other remaining candidates. These candidates form our solution.

¹<http://www.math-inf.uni-greifswald.de/map-labeling/general>

While the heuristic in phase II is identical for all algorithms, the way they choose candidates for removal in phase I differs.

Rules applies a set of rules exhaustively to all points. Each rule tries to find a candidate that can be put into the solution and then removes all candidates of the same site or those that intersect the chosen candidate.

EI-1* establishes edge-irreducibility as described in Section 2.4.

EI+L3 establishes edge-irreducibility and additionally applies rule L3 that is described below.

Phase I of Algorithm Rules

This algorithm was joint work with Frank Wagner and Vikas Kapoor, both at Freie Universität Berlin.

In the first phase of algorithm Rules, we apply all of the following rules to each of the points. Let p_i be the label candidate of point p in position i . For each of the rules we supply a sketch of a typical situation in the context of point labeling. We shaded the candidates that are chosen to label their point, and we used dashed edges to mark candidates that are eliminated after a rule's application. We say that two label candidates (of distinct features) are *in conflict* if they intersect. The *conflict partners* of a candidate c are all those candidates that are in conflict with c . Finally let the *conflict number* of c be the number of conflict partners of c .

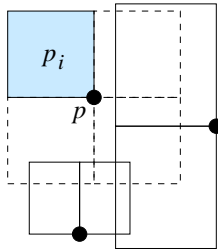


Figure 3.12: Rule **L1**

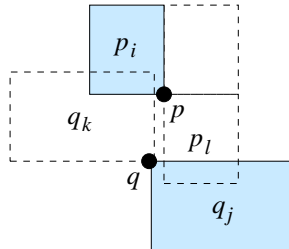


Figure 3.13: Rule **L2**

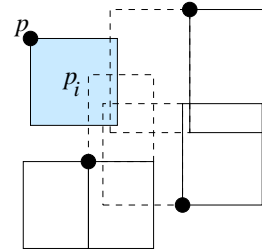


Figure 3.14: Rule **L3**

- (**L1**) If p has a candidate p_i without any conflicts, declare p_i to be part of the solution, and eliminate all other candidates of p , see Figure 3.12.
- (**L2**) If p has a candidate p_i that is only in conflict with some q_k , and q has a candidate q_j ($j \neq k$) that is only overlapped by p_l ($l \neq i$), then add p_i and q_j to the solution and eliminate all other candidates of p and q , see Figure 3.13.
- (**L3**) If p has only one candidate p_i left, and the candidates overlapping p_i form a clique, then declare p_i to be part of the solution and eliminate all candidates that overlap p_i , see Figure 3.14.

We want to make sure that the rules are applied exhaustively. Therefore, after eliminating a candidate p_i , we check whether the rules can be applied in the *neighborhood* of p_i , i.e. to p or to the points of the conflict partners of p_i .

Similar to the rules A1 to A3 in Section 2.4 the rules L1 to L3 have the property that if there is a solution of size t (i.e. t points can be labeled) before applying any of the rules, then this is also the case after the rule's application. This is easy to show, see [WW98]. Note that L1 and L2 are special cases of A1 and A2.

Phase II

If we have not managed to reduce the number of candidates to at most one per point in the first phase, then we must do so in phase II. Since phase II is heuristic, we are no longer able to guarantee optimality. The heuristic RemoveLocal-TroubleMakerMaxCandNumber described in Section 2.5 is conceptually simple and makes the algorithm work well in practice, see Section 3.2.2. The intuition is to start eliminating “troublemakers” where we still have a choice. Speaking more algorithmically, we go through all points p that have the maximum number of candidates, and delete the candidate with the maximum number m of conflicts among the candidates of p if $m \neq 0$. This process is repeated until each point has at most one candidate left and these candidates have no more conflicts. These candidates then form the solution.

As in phase I, after eliminating a candidate, we check whether our rules (i.e. L1 to L3 for Rules, A1 to A3 for EI-1*, and A1 to A3 plus L3 for EI+L3) can be applied in its neighborhood.

Analysis

Other than in Chapter 2 we assume here that the number of candidates per point is a small constant, typically four or eight for point labeling. Let n be the number of points, and k the number of pairs of intersecting candidates in the instance, i.e. the number of edges in the candidate conflict graph. Then EI-1* runs in $O(k)$ time according to Lemma 2.14. In order to bound the running time of Rules and EI+L3, we must analyze the time complexity of rule L3. Rules L1 and L2 are special cases of A1 and A2; thus they can also be checked in constant time per application, see Lemma 2.10.

We use a stack to make sure that our rules are applied exhaustively. After we have applied a rule successfully and eliminated a candidate, we put all points in its neighborhood on the stack and apply the rules to these points. A point is only put on the stack if one of its candidates was deleted or lost a conflict partner.

For rule L3, we have to check whether a candidate is intersected by a clique. In general, this takes time quadratic in the number of conflict partners. Falling back on geometry, however, can help to cut this down. In the case of axis-parallel rectangles, a clique can be detected in linear time by testing whether

the intersection of all conflicting rectangles is not empty. A simple charging argument then yields $O(k^2)$ total time for checking L3.

Note that for L3 other than for the heuristic RemoveLocalTroubleMaker-MaxCandNumber it is not enough to have access to the conflict number of a candidate c ; we actually need access to each conflict partner of c . For implementing rules A1 to A3 one only needs to know whether c is in conflict with candidates of points other than a query point. Of course the candidate conflict graph gives access to all this information.

However, checking L3 can be done in constant time if we apply L3 only to candidates with less than a constant number of conflicts. This makes sense since it is not very likely that the neighborhood of a candidate with many conflicts is a clique. In this case, phase I can be done in $O(n + k)$ time.

In phase II, we can afford to simply go through all points sequentially and check whether they have the current maximum number of candidates. If so, we go through the candidates of the current point and determine the one with the maximum number of conflicts. The amount of time needed to delete this candidate and apply our rules has already been taken into account in phase I. Thus phase II needs only $O(dn)$ extra time.

Putting things together, Rules and EI+L3 take $O(n + k^2)$ time if rule L3 can be checked in linear time, and $O(n + k)$ time if we allow only constant effort for checking L3. In our experiments, we have not bounded this effort, yet this part of the algorithms showed a linear-time behavior. Finally, for axis-parallel rectangular labels, the candidate conflict graph can be determined in $O(k + n \log n)$ time and takes $O(k)$ space.

Thus our algorithms can be implemented to label n points in $O(k + n \log n)$ total time, given a constant number of axis-parallel rectangular label candidates per point and constant effort for checking L3. The algorithms require $O(n)$ storage apart from the candidate conflict graph.

A Hybrid Algorithm

This algorithm was joint work with Tycho Strijk, Universiteit Utrecht.

Since the decisions our algorithms make in phase II are only based on local properties of the candidate conflict graph, these decisions can be made very efficiently. Using more global information is time-costly, but might also improve the quality of the results. Therefore we thought that it would be interesting to combine our set of rules with the global aspect of the matching heuristic of Kakoulis and Tollis [KT98]. The resulting hybrid algorithm proceeds as follows.

As before, we compute the candidate conflict graph. In phase I, we apply our set of rules L1 to L3 exhaustively. In the new phase II, however, we use the heuristic proposed by Kakoulis and Tollis to break up the connected components of the candidate conflict graph into cliques. Recall that in every connected component, they determine the candidate with the highest degree, eliminate it, and repeat this process recursively until each component is a clique. The

choice of the candidate that is to be eliminated has the following exception. If the candidate belongs to a feature that has “few” candidates left, then the candidate with the second highest degree in the current connected component is eliminated, and so on.

Like in the old phase II, after each deletion, we apply our rules in the neighborhood of the eliminated candidate in order to propagate the effect of our heuristical decision.

As soon as all connected components are cliques, we use a maximum-cardinality bipartite-matching algorithm to match as many cliques with features as possible.

The new phase II can be implemented by an extended breadth-first search (BFS). First, we compute all connected components of the candidate conflict graph by common BFS. At the same time, we store the candidate with the highest, second-highest and the lowest degree for each component. To decide whether a component C is a clique, we simply check whether the vertex with minimum degree in C matches the number of vertices in C minus one. If this is not the case, we delete the vertex v_1 with the highest degree. There is one exception. Let a vertex be *important* if it corresponds to a candidate that is the last candidate of a feature. If v_1 is important and the vertex v_2 with the second highest degree in C is not important, then we delete v_2 instead of v_1 .

Now let v be the vertex we deleted. We apply our rules in the neighborhood of v and then the extended BFS recursively to all vertices that were adjacent to v just before its deletion. In each level of the recursion at least one vertex is deleted, and each edge is considered at most twice by BFS. Thus, if each of the n features has a constant number of candidates, we have at most $O(n)$ recursion levels and each takes at most $O(k)$ time. This results in $O(nk)$ time for the new phase II compared to $O(n + k)$ for the previous version.

Computing a maximum-cardinality bipartite matching takes $O(k\sqrt{n})$ in practice.

3.2.2 Experiments

We compare our algorithms **Rules**, **EI-1***, and **EI+L3** to the following five other methods that we all implemented in C++.

Anneal is a simulated-annealing algorithm based on the experiments by Christensen et al.. We follow their suggestions for the initial configuration, the objective function, a method for generating configuration changes, and the annealing schedule [CMS95]. In order to save time, we allowed only 30 instead of the proposed 50 temperature stages in the annealing schedule. This did not seem to influence the quality of the results.

Greedy picks repeatedly the leftmost label (i.e. the label whose right edge is leftmost), and discards all candidates that intersect the chosen label. This simple algorithm has an approximation factor of $1/(H + 1)$, where H is the ratio of the greatest and the smallest label height [vKSW98]. Greedy can be

implemented to run in $O(n \log n)$ time by using a priority-search tree and a heap, see Remark 3.18 in Section 3.3.2 (page 71). However, our $O(n^2)$ -implementation is simply based on lists and uses brute force to find the next leftmost label candidate.

Match refers to the “pure” matching heuristic of Kakoulis and Tollis [KT98]. Our implementation uses the recursive extended BFS of Section 3.2.1 and the maximum-cardinality bipartite-matching algorithm supplied by LEDA [NM90]. It runs in $O(kn)$ time. We did not apply any of our rules and did not do any pre- or post-processing.

Match+L1 is a variant of their algorithm, also proposed in [KT98]. Here rule L1 is applied exhaustively before the heuristic that reduces all connected components to cliques. This does not change the asymptotic runtime behavior.

Hybrid refers to the algorithm sketched in Section 3.2.1. It combines the heuristic by Kakoulis and Tollis that reduces connected components to cliques with our rules L1 to L3. Our implementation uses LEDA’s matching algorithm and requires $O(kn)$ time.

We run our algorithm and those described above on the following instance classes. Figures 3.31 to 3.38 depict an example of each of these classes. The first figure in each caption refers to the number of points in the depicted instance. In some figures, an additional number in parenthesis is given, namely the size of the solution of our algorithm. Rules applied on the depicted instance. Where Rules found a complete labeling no extra number is given.

RandomRect. We choose n points uniformly distributed in a square of size $25n \times 25n$. To determine the label size for each point, we choose the length of both edges independently under normal distribution, take its absolute value and add 1 to avoid non-positive values. Finally we multiply both label dimensions by 10.

DenseRect. Here we try to place as many rectangles as possible on an area of size $\alpha_1\sqrt{n} \times \alpha_1\sqrt{n}$. α_1 is a factor chosen such that the number of successfully placed rectangles is approximately n , the number of points asked for. We do this by randomly selecting the label size as above and then trying to place the label 50 times. If we don’t manage, we select a new label size and repeat the procedure. If none of 20 different sized labels could be placed, we assume that the area is well covered, and stop. For each rectangle we placed successfully, we return its height and width and a corner picked at random. It is clear that all points obtained this way can be labeled by a rectangle of the given size without overlap.

RandomMap and **DenseMap** try to imitate a real map using the same point placement methods as RandomRect and DenseRect, but more realistic label sizes. We assume a distribution of 1:5:25 of cities, towns and villages. After randomly choosing one of these three classes according to the assumed distribution, we set the label height to 12, 10 or 8 points accordingly. The length of the label text then follows the distribution of the German Railway station names (see below). We assume a typewriter font and set the label length to the

number of characters times the font size times $2/3$. The multiplicative factor reflects the ratio of character width to height.

VariableDensity. This example class is used in the experimental paper by Christensen et al. [CMS95]. There the points are distributed uniformly on a rectangle of size 792×612 . All labels are of equal size, namely 30×7 . We included this benchmark for reasons of comparability.

HardGrid. In principle we use the same method as for DenseRect and DenseMap, that is, trying to place as many labels as possible into a given area. In order to do so, we use a grid of $\lfloor \alpha_2 \sqrt{n} \rfloor \times \lceil \alpha_2 \sqrt{n} \rceil$ cells with edge lengths n . Again, α_2 is a factor chosen such that the number of successfully placed squares is approximately n . In a random order, we try to place a square of edge length n into each of the cells. This is done by randomly choosing a point within the cell and putting the lower left corner of the square on it. If it overlaps any of the squares placed before, we repeat at most 10 times before we turn to the next cell.

RegularGrid. We use a grid of $\lfloor \sqrt{n} \rfloor \times \lceil \sqrt{n} \rceil$ squares. For each cell, we randomly choose a corner and place a point with a small constant offset near the chosen corner. Then we know that we can label all points with square labels of the size of a grid cell minus the offset.

MunichDrillholes. The municipal authorities of Munich provided us with the coordinates of roughly 19,400 ground-water drill holes within a 10 by 10 kilometer square centered approximately on the city center. From these points, we randomly pick a center point and then extract a given number of points closest to the center point according to the L_∞ -norm. Thus we get a rectangular section of the map. Its size depends on the number of points asked for. The drill-hole labels are abbreviations of fixed length. By scaling the x-coordinates, we make the labels into squares and subsequently apply an exact solver for label-size maximization. This gives us an instance with a maximum number of conflicts which can just be labeled completely.

In addition to these example classes, we tested the algorithms on the point sets depicted in Figures 3.39 and 3.40, see page 58.

3.2.3 Results

We used examples of 250, 500, . . . , 3000 points. For each of the example classes and each of the example sizes, we generated 30 files. Then we labeled the points in each file with axis-parallel rectangular labels. We used four label candidates per point, namely those where one of the label's corners is identical to the point. We allowed labels to touch each other but not to obstruct points.

Quality

The graphs in Figures 3.15 to 3.22 (see pages 53 and 54) show the performance of the eight algorithms. The average example size is shown on the x-axis,

the average percentage of labeled points is depicted on the y-axis. Note that we varied the scale on the y-axis from graph to graph in order to show more details. The worst and the best performance of the algorithms are indicated by the lower and upper endpoints of the vertical bars. To improve legibility, we give two graphs for each example class; on the left the results of Rules, EI-1*, EI+L3, Anneal, and Hybrid are depicted, while those of Greedy and the two variants of Match are shown in the graph on the right side of each figure.

The example classes are divided into two groups; those that have a complete labeling and those that have not. For the former group, the percentage of labeled points expresses directly the performance ratio of an algorithm. For examples of the latter group, which consists of RandomRect, RandomMap and VariableDensity, there is only a very weak upper bound for the size of an optimal solution, namely the number of labels needed to fill the area of the bounding box of the instance completely. Thus for VariableDensity at most 2539 points can possibly be labeled. Experiments we performed with an exact solver on examples of up to 200 points showed that on an average about 85% of the points in an instance of RandomRect and usually less than 80% in the case of RandomMap can be labeled. Other than VariableDensity, these classes are designed to keep their properties with increasing point number. This is reflected by the fact that the algorithms' performance was nearly constant on these examples. We used the same set of rules as in phase I of our algorithm to speed up the exact solver.

In terms of performance the algorithms can be divided into two groups. The first group consists of simulated annealing, our rule-based algorithms and the new hybrid algorithm; the second group is represented by the greedy method and the two variants of the matching heuristic. The first group outperforms the second group clearly in all but one example class. On RegularGrid data, the second group and Hybrid achieve 100%, followed very closely by the remaining algorithms; note the scale in Figure 3.20. For all example classes (except RegularGrid and MunichDrillholes, where all algorithms performed extremely well), there is a 5- to 10-percent gap between the results of the two groups.

For all examples that have a complete labeling, Rules, EI+L3, EI-1*, and Hybrid label between 95 and 100% of the points. Experiments on small examples hint that the same holds for larger RandomRect and RandomMap examples. For some of the example classes, simulated annealing outperforms our algorithms by one to two percent. However, in order to achieve similarly good results, simulated annealing needs much longer (see below), in spite of the fact that it uses the same fast $O(n \log n)$ algorithm for detecting rectangle intersections (based on an interval tree). It is not surprising that EI+L3 is better than Rules in most cases; recall that the rules L1 and L2 are special cases of A1 and A2. However, we were astonished to see that Hybrid and Rules yield practically identical results in spite of their different approaches. Only for HardGrid and RegularGrid Hybrid was better than Rules—by merely one percent. The similarity of their results suggests that it is the rules which do most of the work. Rules and EI-1* also yield very similar results; for DenseRect, DenseMap, and

RegularGrid EI-1* is slightly better, Rules on the other example classes. The graph for VariableDensity suggests that EI-1* becomes worse than Rules when the density of the candidate conflict graph increases.

In the second group, the greedy algorithm performed well given that it makes its decisions only based on local information. Surprisingly, its results were practically always better than that of the “pure” Kakoulis-Tollis heuristic that relies on a global matching step. Adding rule L1 as a pre-processing step improved the result of the matching heuristic by up to three percent. This variant performed better than the greedy algorithm in most example classes, but was still clearly worse than simulated annealing and our algorithms except on the rather degenerate RegularGrid data.

Time

In Figures 3.23 to 3.30 (see pages 55 and 56) we present the running times of our implementations in CPU seconds on a Sun UltraSparc. We used the SUN-Pro compiler with optimizer flag `-fast`.

Again, to improve legibility, we give two graphs for each example class; on the left the results of the faster algorithms Rules, EI+L3, Hybrid, and the two variants of Match are depicted, while those of Anneal and Greedy are shown in the graph on the right of each figure. Since EI-1* is only slightly faster than EI+L3, and a difference was only perceivable for RandomMap and VariableDensity, we dropped the graphs for EI-1*.

Given heaps and priority search trees, the greedy algorithm would definitively run faster. Our implementation of simulated annealing seems to be slower by a factor of 2 to 3 than that of Christiansen et al. [CMS95]. This difference in running time may be due to the machines on which the times were measured.

On large examples, Rules is faster by a factor of 2 to 10 as compared to the matching heuristic, and by a factor of 30 to 100 with respect to simulated annealing. Applying rule L1 as a pre-processing step speeds up the matching algorithm up to a factor of a third.

EI+L3 (and thus EI-1*) is slower by a factor of 2 as compared to Rules. This is due to the fact that we did not implement REVISE as in Lemma 2.10 but with the brute-force algorithm sketched at the beginning of Section 2.4.

The fact that some of the algorithms are faster on larger than on smaller point sets of VariableDensity, see Figure 3.30 on page 56, is due to the fact that with the increase in density, many label candidates contain points and are therefore eliminated during preprocessing, see also Figure 3.56 on page 60.

Phase I

Since the difference between the three variants of our algorithms (Rules, EI-1*, and EI+L3) does not show very clearly, we also investigated how efficient they were in phase I, i.e. before applying the heuristic RemoveLocalTroubleMaker-

MinCandNumber. Note that EI-1* is identical to EI-1 before phase II.

The graphs in Figures 3.41 to 3.48 (see page 59) show how many percent of the given points are already labeled at the end of phase I. The graphs in Figures 3.49 to 3.56 (see page 60) show how many percent of the label candidates are removed in phase I. The x-axis in these figures shows the initial number of candidates, which is four times the number of points. Recall that we eliminate all candidates that contain points before phase I. These removals are also counted here.

It is not surprising that EI+L3, whose rules are a superset of those of Rules and EI-1, is always better than both Rules and EI-1. However, it is interesting to see that in most of the graphs EI-1 dominates Rules. EI-1 is more effective in labeling points (in all classes except RandomMap and VariableDensity) and in removing candidates (in all but VariableDensity). In our opinion these graphs support the hypothesis that EI-1 represents a considerable progress compared to Rules in attacking label-placement-type problems. It opens the road for other efficient algorithms that achieve higher degrees of irreducibility and will yield even better results since the need for heuristical decisions will decrease with the gain in terms of irreducibility.

On the example class VariableDensity EI-1 is better for small examples, while Rules does better on the larger and thus denser point sets. This seems to be where rule L3 that is used by Rules but not by EI-1 becomes effective. At the same time, this does not influence the runtime behavior of Rules noticeably, see Figure 3.30, page 56.

Quality of Results I

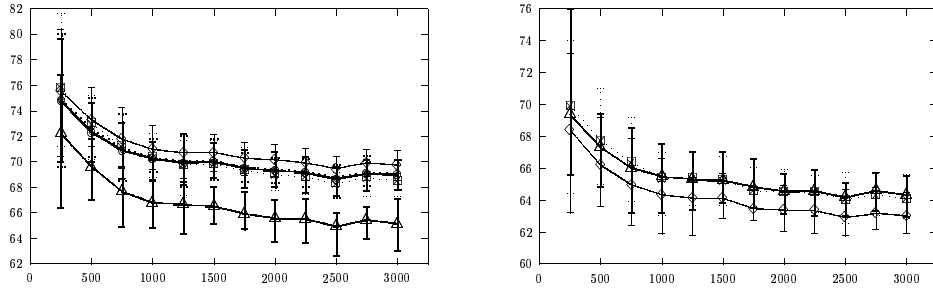


Figure 3.15: RandomMap

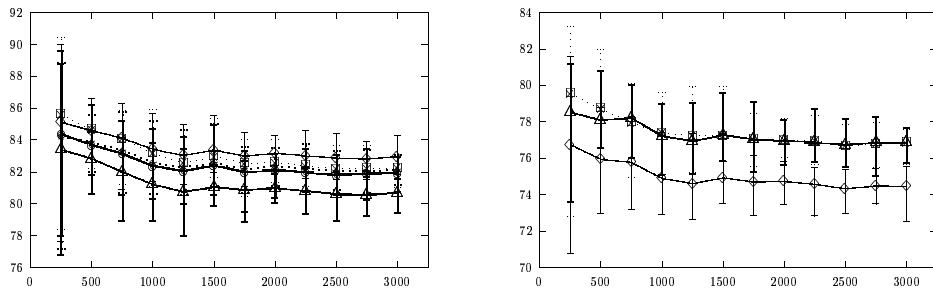


Figure 3.16: RandomRect

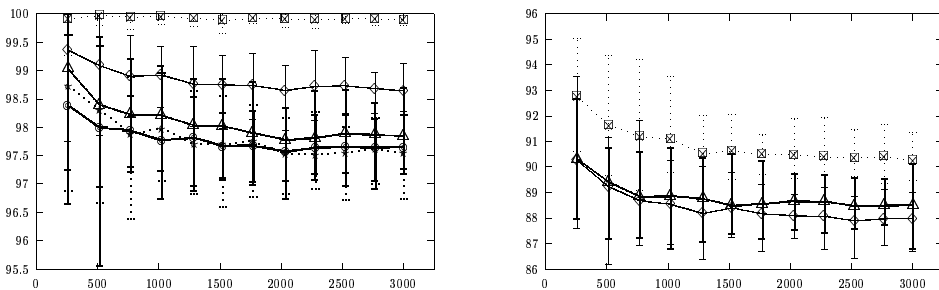


Figure 3.17: DenseMap

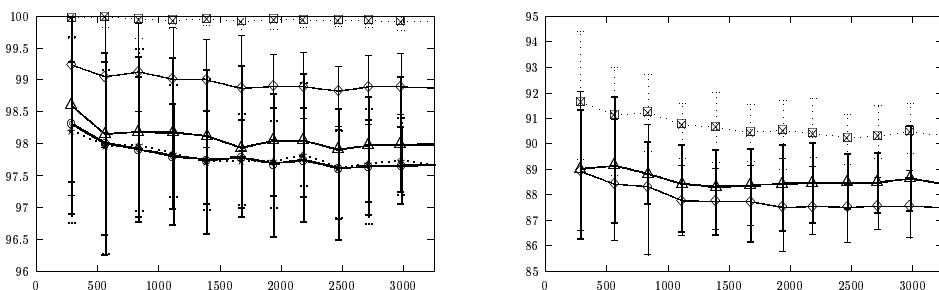
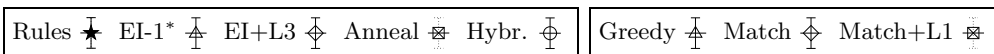


Figure 3.18: DenseRect



Quality of Results II

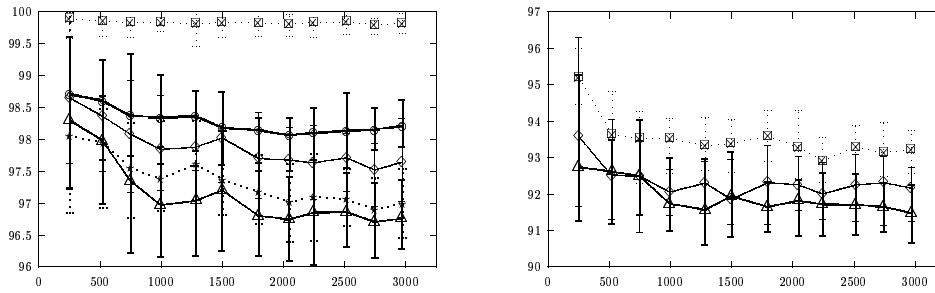


Figure 3.19: HardGrid

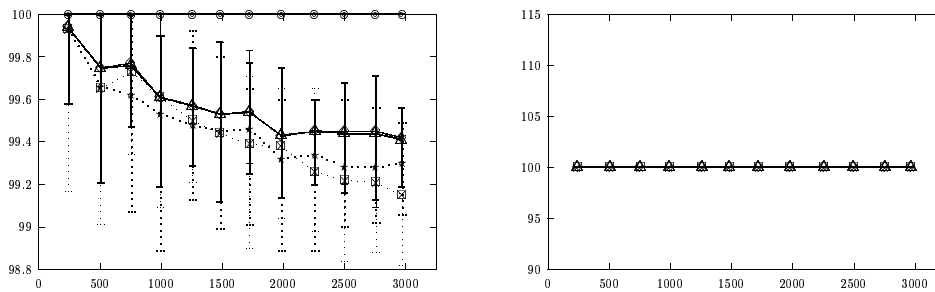


Figure 3.20: RegularGrid

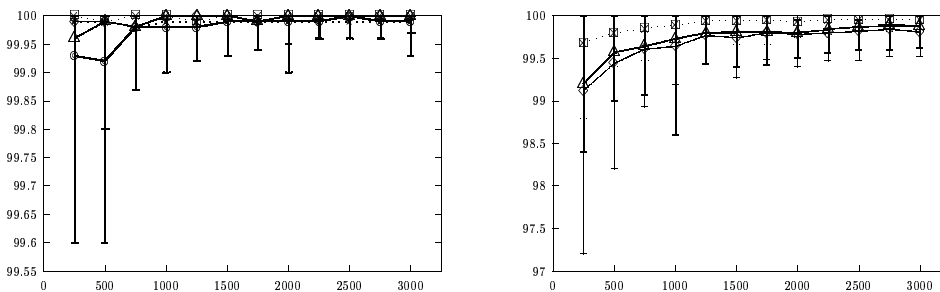


Figure 3.21: MunichDrillholes

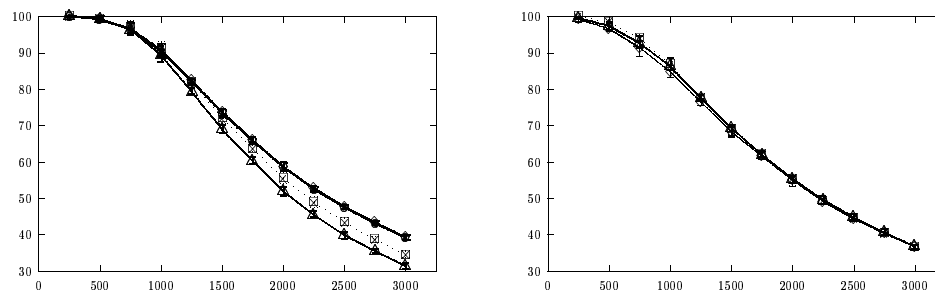
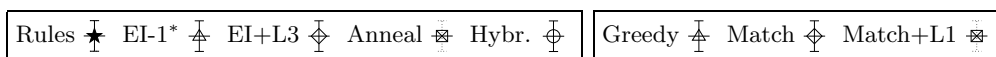


Figure 3.22: VariableDensity



Running Time I

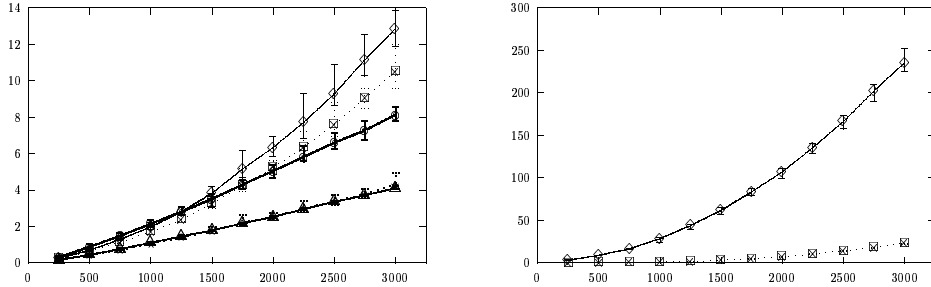


Figure 3.23: RandomMap

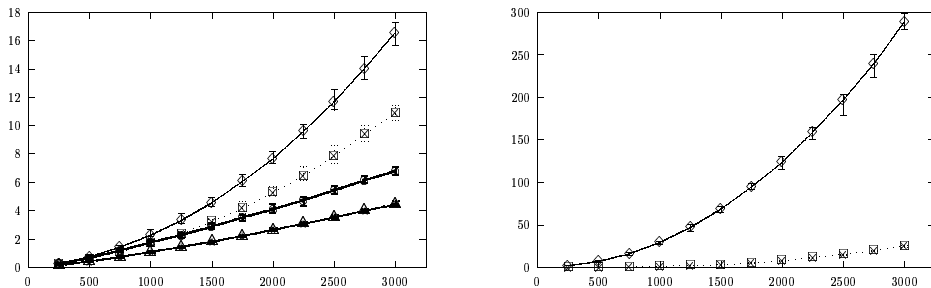


Figure 3.24: RandomRect

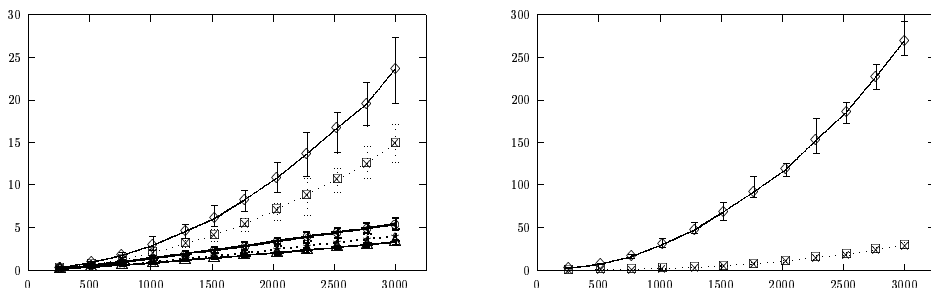


Figure 3.25: DenseMap

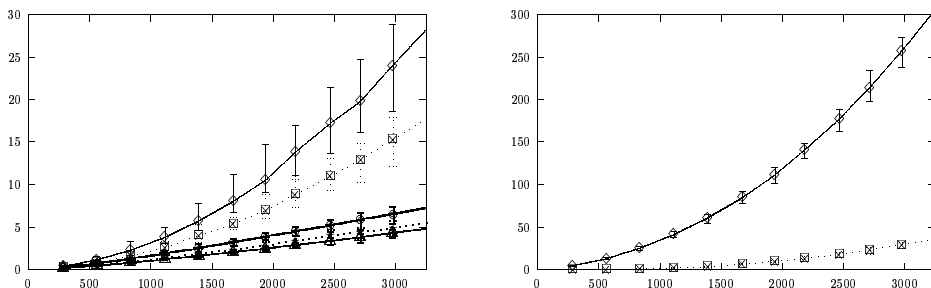


Figure 3.26: DenseRect



Running Time II

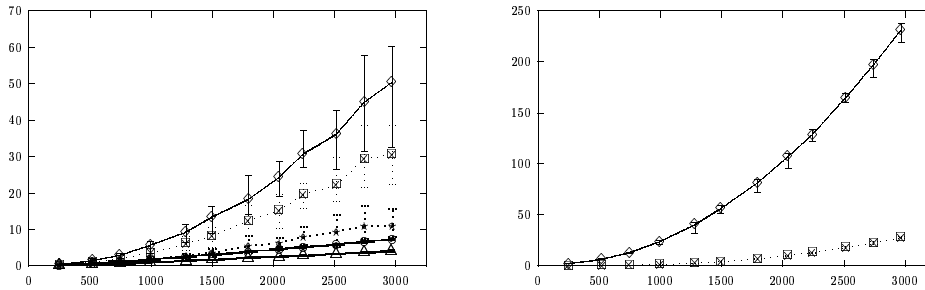


Figure 3.27: HardGrid

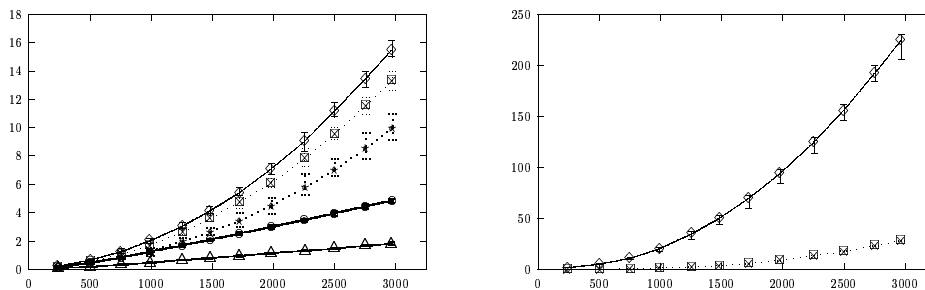


Figure 3.28: RegularGrid

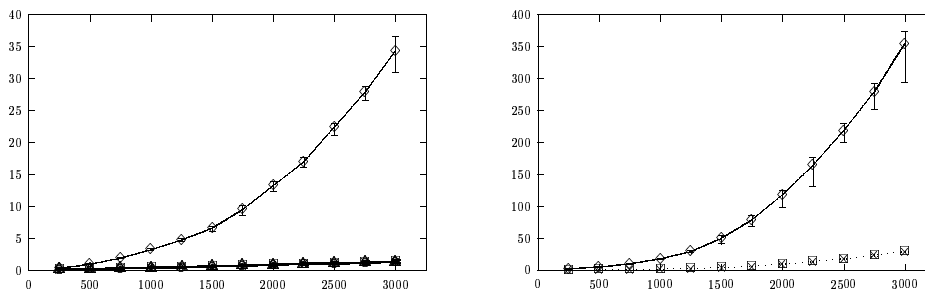


Figure 3.29: MunichDrillholes

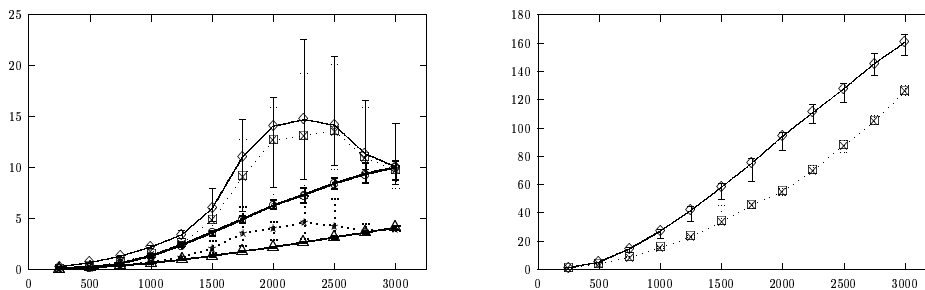
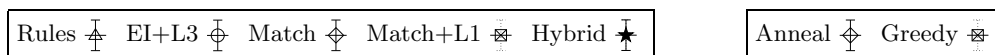


Figure 3.30: VariableDensity



Example Classes

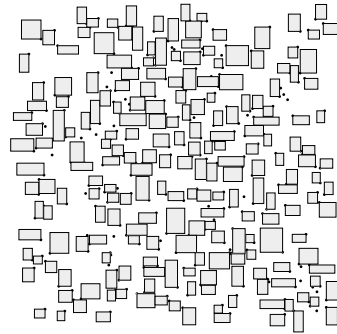
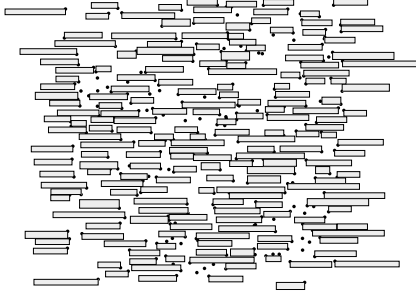


Figure 3.31: RandomMap 250 (193) points Figure 3.32: RandomRect 250 (212) points

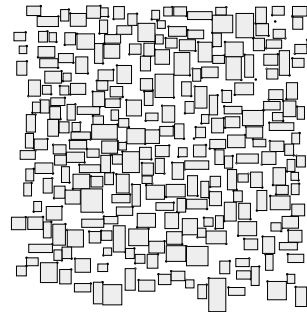
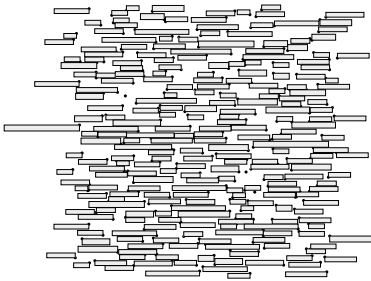


Figure 3.33: DenseMap 253 (249) points Figure 3.34: DenseRect 261 (258) points

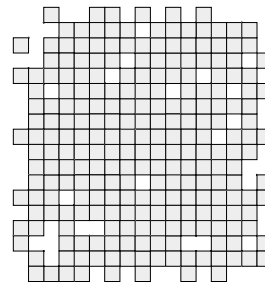
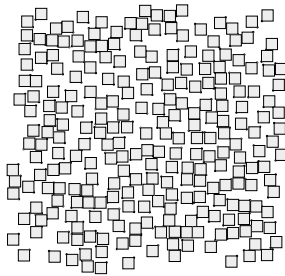


Figure 3.35: HardGrid 253 (252) points

Figure 3.36: RegularGrid 240 points

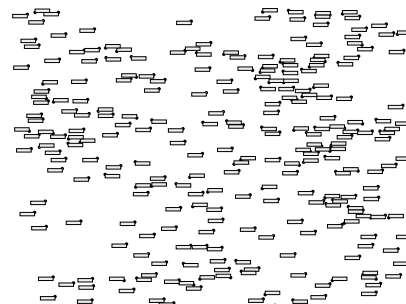
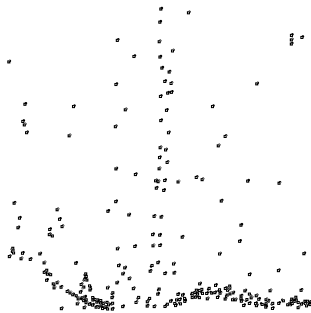


Figure 3.37: MunichDrillholes 250 points

Figure 3.38: VariableDensity 250 points

Two Real-World Examples

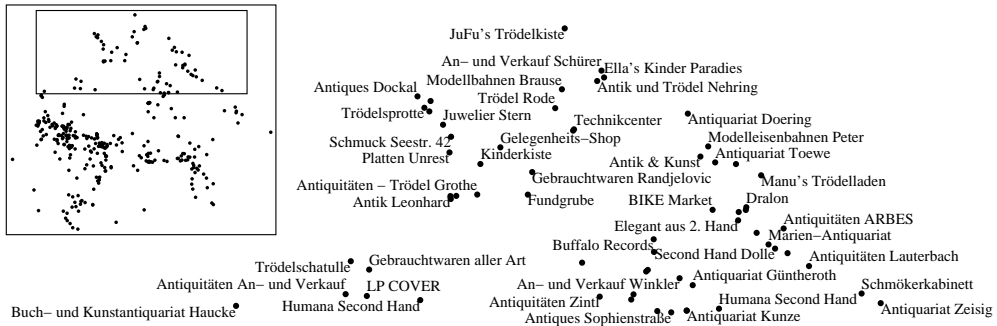


Figure 3.39: left: 357 tourist shops in Berlin, right: 45 of 63 labeled.



Figure 3.40: 373 German railway stations, 270 labeled.

Phase I: Number of points labeled

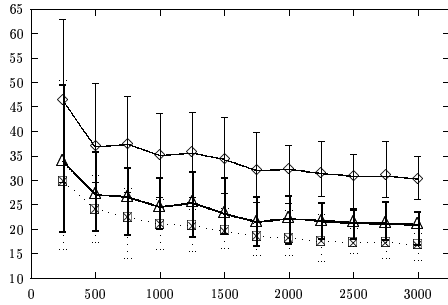


Figure 3.41: RandomMap

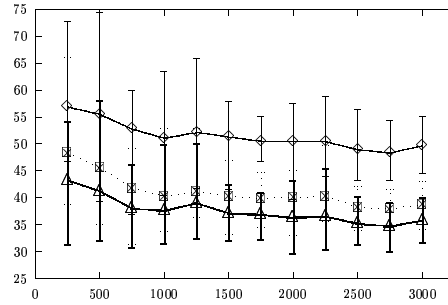


Figure 3.42: RandomRect

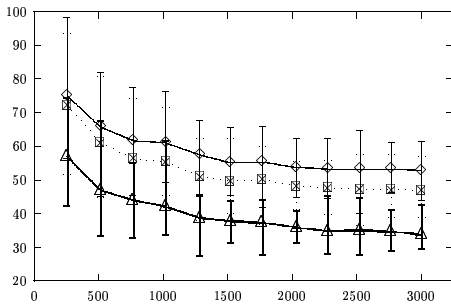


Figure 3.43: DenseMap

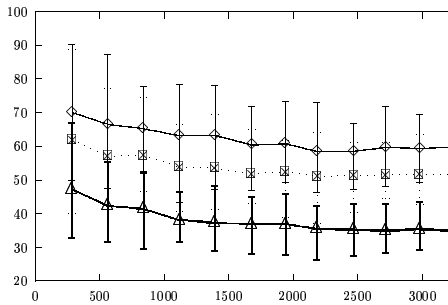


Figure 3.44: DenseRect

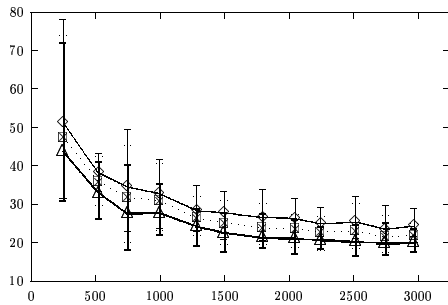


Figure 3.45: HardGrid

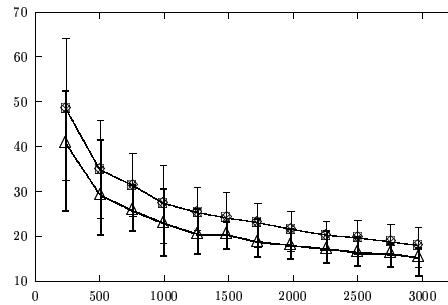


Figure 3.46: RegularGrid

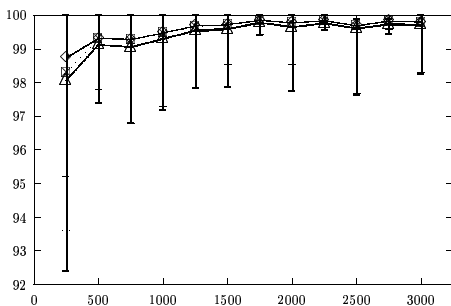


Figure 3.47: MunichDrillholes

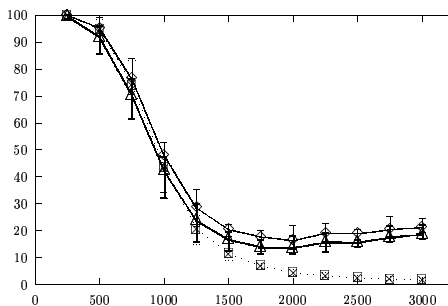


Figure 3.48: VariableDensity

Rules \blacktriangle EI-1 \blacksquare EI+L3 \blacklozenge

Phase I: Number of removed label candidates

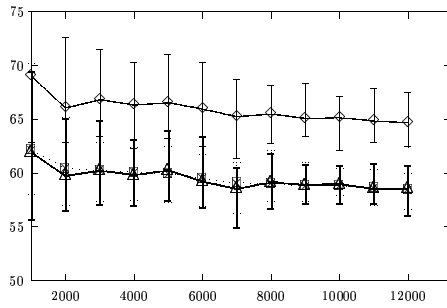


Figure 3.49: RandomMap

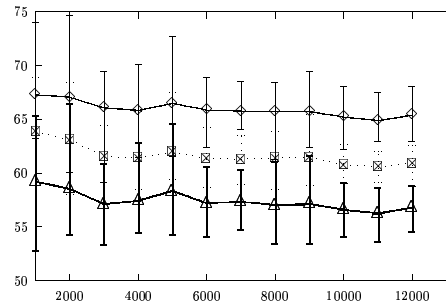


Figure 3.50: RandomRect

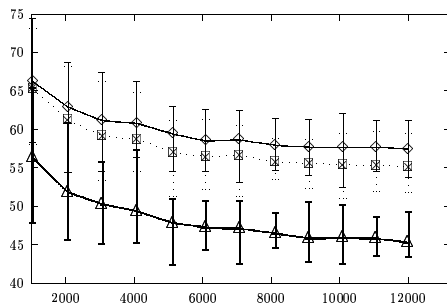


Figure 3.51: DenseMap

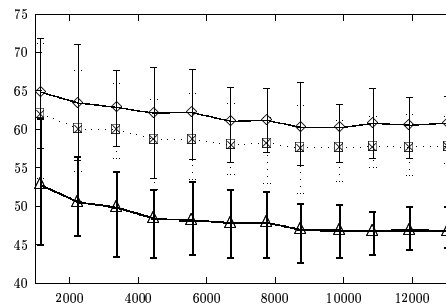


Figure 3.52: DenseRect

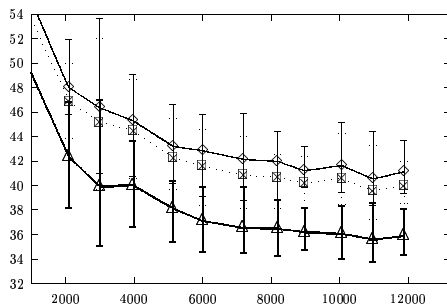


Figure 3.53: HardGrid

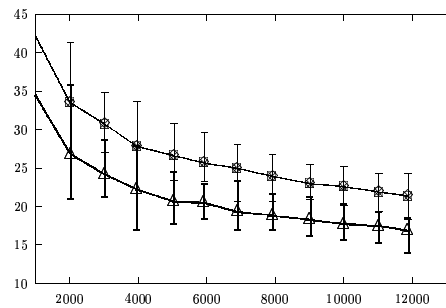


Figure 3.54: RegularGrid

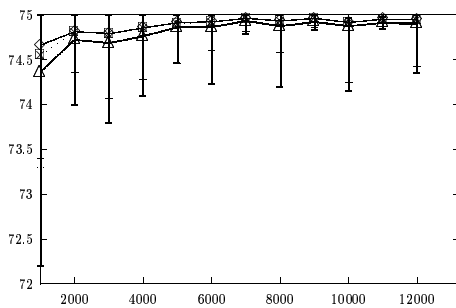


Figure 3.55: MunichDrillholes

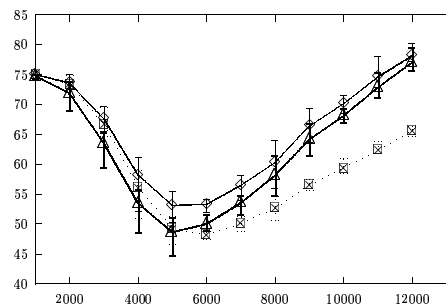


Figure 3.56: VariableDensity

Rules \triangle EI-1 \square EI+L3 \diamond

3.3 Slider Models

This section is joint work with Marc van Kreveld and Tycho Strijk, both Universiteit Utrecht [vKSW98, vKSW99].

In this section we drop the restriction that a label can only be placed at a finite number of positions. Instead, we allow any position on the edges of the rectangle to coincide with the point, see Figure 3.1. Such a model is called a *slider model*. We will study how many more labels can be placed with slider models than with fixed-position models, and to what extent slider models require more difficult algorithms. We generally assume that labels have equal height but not necessarily the same width. This is a natural assumption if labels contain text or numbers of a fixed font size. We consider the rectangle that represents a label to be closed, which implies that labels are not allowed to touch.

Slider models have been used in two previous papers. Hirsch’s paper [Hir82] defines repelling forces for overlapping labels and computes translation vectors for them. After translation, this process is repeated and hopefully, a labeling with few overlaps appears after a number of iterations. This is completely different from our approach, which is combinatorial. The paper by Doddi et al. [DMM⁺97] contains a number of labeling problems and algorithms, each using a different labeling model. One of the problems is solved in a slider model, where each label is allowed to rotate around the point to be labeled. The labels must be equal-size squares (or other regular polygons); the objective is to maximize the label *size*.

Point labeling has long been known to be NP-complete for fixed-position models [FW91, FPT81, KR92, MS91]. However, this does not imply that label placement is also hard for slider models. In Section 3.3.1 we show that this is the case; we prove that it is NP-complete to decide whether a set of points can be labeled in the four-slider model.

In Section 3.3.2, we show that the slider models allow a simple factor- $\frac{1}{2}$ approximation algorithm that uses $O(n)$ space and $O(n \log n)$ time. This was already known for the fixed-position models [AvKS98]. Our algorithm is greedy in that it always places the label whose right edge is leftmost among the right edges of all possible label placements. The algorithm uses a kind of generalized sweep-line in order to select the next label. We remark that our algorithm can be adapted to labels of varying height, but then the approximation factor depends on the ratio of maximum and minimum label height.

In Section 3.3.3, we give a polynomial time approximation scheme for each of our slider models, showing that for any constant $\epsilon > 0$, there is a polynomial time algorithm that labels a fraction of at least $1 - \epsilon$ of the optimal number of labels that can be placed. Again, this result was already known for fixed-position but not for slider models.

In order to support the practical relevance of the greedy algorithm, we do a thorough experimental analysis in Section 3.3.4. We have implemented our

greedy algorithm for the six models. We test it on three data sets from different application areas. One contains 1000 city names of the USA, another contains a data posting with 236 measurements, and the third contains 75 sequence numbers in a scatter plot near a regression line. We give tables showing how many points are labeled in each model for a range of font sizes. It appears that the greedy algorithm produces about 10–15% more labels for a slider model than in the corresponding fixed-position model. This improvement is significant, because more labels are placed in dense areas. We also compare our algorithm to a simulated annealing algorithm proposed by Christensen et al. [CMS95] on a sequence of randomly generated point sets.

3.3.1 NP-Hardness

The complexity of labeling points with axis-parallel rectangular labels from a *finite* set of label candidates is well established in the literature [FW91, FPT81, IL97, KR92, MS91]. Slider models are a generalization of those fixed-position models that force a label to touch the point to be labeled. However, this observation does not yield the NP-hardness of the slider models, since it is not clear how an instance for a fixed-position model can be reduced to an instance of a slider model. Recall for example that the NP-completeness of 0-1-integer programs does not apply to their relaxation. Therefore we show that placing unit square labels in the 4-slider model is NP-complete.

Theorem 3.15 *It is NP-complete to decide whether a set of points can be labeled with axis-parallel unit squares in the 4-slider model.*

Proof. The problem is in \mathcal{NP} for the following reason. We can guess (i.e. compute non-deterministically) a permutation of the points and an integer between 1 and 4 for each point. This number indicates which edge of a label will be attached to the point. Then we go through the points according to the permutation and check for each point whether we can label it such that its label touches it on the chosen edge—given the labels we have already placed. If the new point can be labeled, we move its label into a *canonical* position: Depending on whether the pre-computed edge is horizontal or vertical, we slide the label along this edge as far left or down as possible. If all points can be labeled this way, we accept. Otherwise we discard the subset. The reason why we can reject in this case is the following. If all points could be labeled, we could push all labels in their canonical positions and name a permutations of the points, such that the procedure outlined above would produce the same canonical label placement.

The proof of the NP-hardness is by reduction from planar 3-SAT. Lichtenstein showed that this restriction of 3-SAT is NP-hard [Lic82]. Our proof follows Knuth and Raghunathan’s proof of the NP-hardness of the Metafont-labeling problem [KR92]. We encode the variables and clauses of a Boolean formula ϕ of planar 3-SAT type by a set of points such that all points can only be labeled if ϕ is satisfiable, i.e. if there is a variable assignment such that all

clauses evaluate to true. The advantage of a planar 3-SAT formula is that the variables can always be arranged on a straight line such that they are connected by *non-intersecting* three-legged clauses, see [KR92, Figure 5].

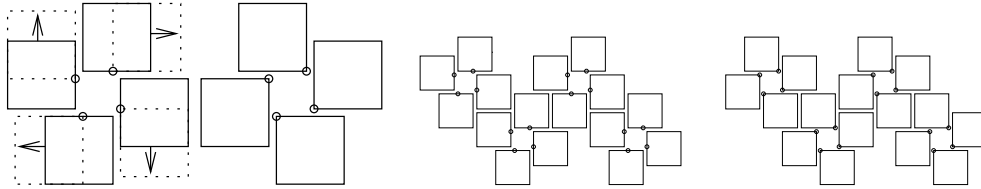


Figure 3.57: Label placements Figure 3.58: Zig-zagging cluster patterns model encoding *true* and *false*.
the labels its Boolean value.

The main observation leading to our proof is the following. Given a cluster of four points (the corner points of a square with edges slightly longer than 1/2 and rotated by a small angle against the axes), there are two fundamentally different ways to label these points, see Figure 3.57. Under the condition that all points have to be labeled, the points can only be labeled as on the left side (which allows some sliding) or on the right side (where the labels are nearly fixed) of Figure 3.57. Note that it is impossible that some points are labeled as on the left and others as on the right side. This gives us a means to encode the Boolean values of a variable in the planar 3-SAT formula ϕ that we want to reduce to a set of points.

The building blocks (or “gadgets”) of our reduction are the clusters for variables, three-legged “combs” for clauses, and adapters connecting variables to clauses. In order to be able to connect a variable to all clauses in which it occurs, we model it not by one but by several four-point clusters in a zig-zag pattern as shown in Figure 3.58. Then still all points have to be labeled according to one of the two schemes mentioned above.

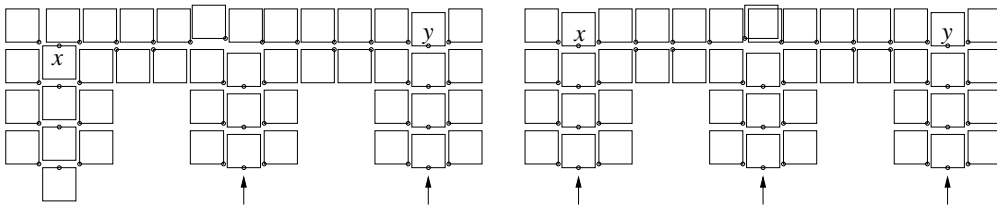


Figure 3.59: Clause with pressure from two variables. Figure 3.60: Clause with pressure from three variables.

We model the clauses by point sets which resemble large combs with three legs, see Figure 3.59. The fourth column of points from the right and the left can be repeated as often as needed to reach the three variables belonging to the clause. The legs can be extended by duplicating the bottom-most row of points. Each leg is connected to a variable by an adapter. An adapter consists of three points a , b , and c . There are two types of adapters, see Figure 3.61

and 3.62. Which type is chosen depends on whether the variable is negated in the clause. If the variable is set to a value which negates the corresponding literal in the clause, the lowest point b in the adapter must be labeled upwards, i.e. the label sticks into the pipe leading to the clause in question. This forces all other points above b to have their label above them as well. Graphically speaking, pressure is transmitted. This is indicated by arrows in Figure 3.59 to 3.62. When the pressure arrives in the top row of points in the representation of a clause, it is transmitted further horizontally, see the labels of the points x and y in Figure 3.59 and 3.60. Note that a variable assignment which fulfills the corresponding literal does not force anything; no pressure is exerted.

If all literals of a clause evaluate to false, then the points of type b in the adapters of the corresponding variables are labeled upwards and pressure is transmitted through all three vertical pipes into the clause. In this case there is a point which cannot be labeled, see Figure 3.60. If, however, there is at least one vertical pipe without pressure, all points belonging to a clause can be labeled, see Figure 3.59.

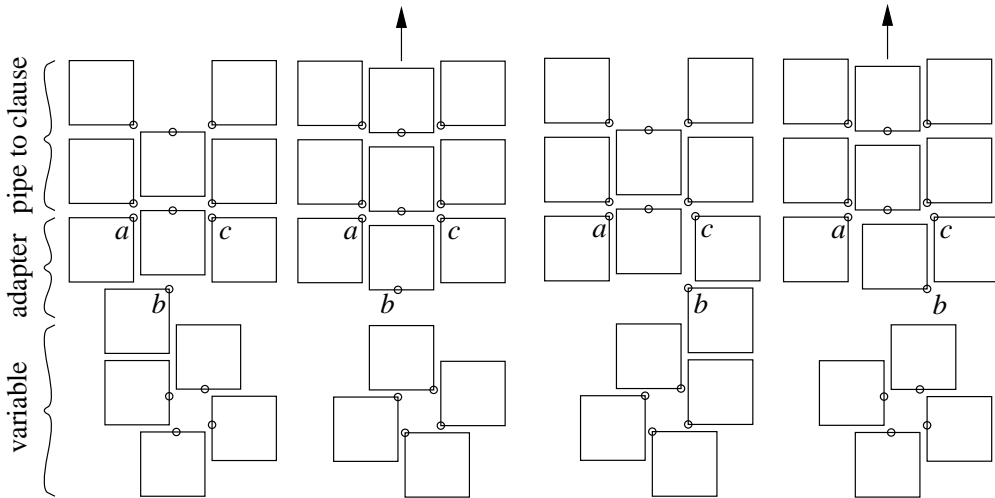


Figure 3.61: Adapters for unnegated literals exert pressure when a variable is set to *false*.

Figure 3.62: Adapters for negated literals exert pressure when a variable is set to *true*.

Hence the question whether ϕ is satisfiable is equivalent to asking whether all points can be labeled in the 4-slider instance to which ϕ is reduced. It is easy to see that the reduction is polynomial: if ϕ consists of m clauses, the instance has $O(m^2)$ points. Their position can certainly be computed in polynomial time. \square

3.3.2 A Greedy Approximation Algorithm

In this section we describe algorithms for point feature labeling in the slider models. They apply to labels of fixed height but arbitrary width. We describe an $O(n \log n)$ time algorithm for the slider models that approximates an optimal solution in the following sense. If the maximum number of labels that can be placed is k_{opt} , then our algorithm places at least $k_{\text{opt}}/2$ labels: a factor- $\frac{1}{2}$ approximation algorithm. In most data sets, however, we expect to come much closer to the optimum.

For the fixed position models, a simple $O(n \log n)$ time, factor- $\frac{1}{2}$ approximation algorithm was described recently by Agarwal et al. [AvKS98]. We obtain the same result for the slider models. We'll only describe the most general four-slider algorithm; it is an extension of the top-slider and two-slider algorithms. It is based on a greedy strategy. For convenience we'll first describe the algorithm with labels allowed to touch, unlike in the previous sections where labels were considered to be closed. Later we show that simple adaptations can be made to obtain non-touching labels.

Given a set of points with labels that have already been placed, and a set of points that don't have a label yet, define the *leftmost label* to be the label whose right edge is leftmost among all label candidates of unlabeled points and that does not intersect previously placed labels.

Lemma 3.16 *Given labels of fixed height and any of the slider models, the greedy strategy of repeatedly choosing the leftmost label finds a labeling of at least half the number of points labeled in an optimal solution.*

Proof. Given a set P of points and a sliding model M , let L_{opt} be an optimum M -labeling. Let L_{left} be the set of labels computed by the greedy strategy. The set L_{left} is maximal in the sense that no label can be added to it without intersecting another label in L_{left} . So any label in L_{opt} must either be in L_{left} as well, or intersect some label in L_{left} , whose right edge is at least as much to the left. Charge each label in $L_{\text{opt}} \setminus L_{\text{left}}$ to a label in L_{left} that lies as least as much to the left and intersects it. For any label in $L_{\text{opt}} \cap L_{\text{left}}$, charge it to itself.

We claim that any label in L_{left} is charged at most twice, from which the lemma follows. For labels in $L_{\text{opt}} \cap L_{\text{left}}$ the claim is obviously true. For any other label $l \in L_{\text{left}}$, observe that a label of L_{opt} that charges l must intersect the closed right edge of l . Since all labels have unit height, and the labels in L_{opt} don't intersect each other, there can only be two labels of L_{opt} that intersect the closed right edge, and hence, charge l . \square

A brute-force algorithm for this simple strategy would need $O(n^3)$ steps. In order to achieve an $O(n \log n)$ time bound, we must use some common geometric data structures.

Let $\{p_1, \dots, p_n\}$ be the set of points that has to be labeled. The label of

p_i is denoted l_i , and the reference point of a label is its lower left vertex. The possible positions of the reference point of a point p_i are represented by four line segments. Two are horizontal, h_{2i-1} and h_{2i} , and two are vertical, v_{2i-1} and v_{2i} . Their position is exactly the position of the edges of the label l_i if it were placed left and below p_i . The width of l_i is denoted w_i , and the height is 1. We can always scale the y -coordinates to this situation.

If a label l_i has been placed, then no reference point position inside l_i is possible. The same holds for reference points inside the rectangle l'_i precisely one unit below l_i (since any label extends one unit above its reference point). The open rectangle that exactly covers l_i and l'_i and their mutual bounding edge is the *extended rectangle* \tilde{l}_i . Since labels are placed from left to right, no reference point positions in nor to the left of \tilde{l}_i will still be accepted by the algorithm. Suppose a subset of the points has already received labels by the algorithm.

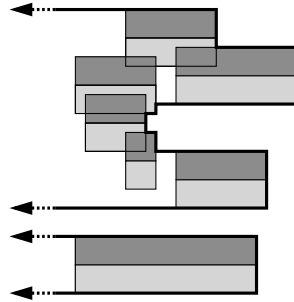


Figure 3.63: Frontier of the placed labels (dark grey) and their lowered copies (light grey).

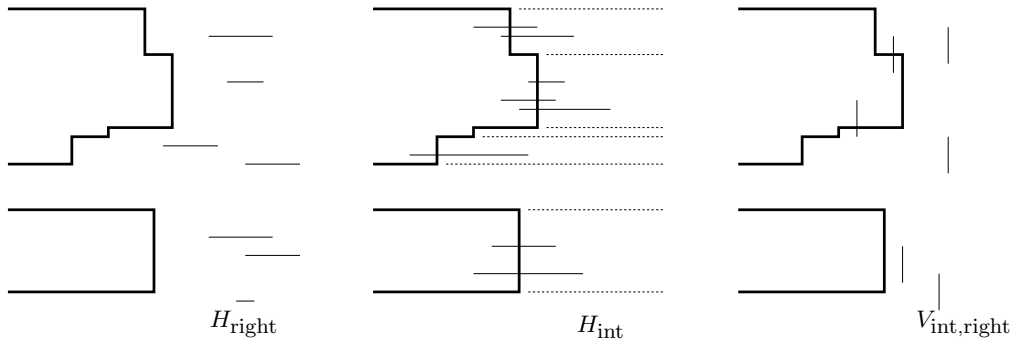


Figure 3.64: The sets H_{right} , H_{int} , and $V_{\text{int,right}}$. The dashed lines in the middle picture separate the segments of H_{int} that are in different red-black trees \mathcal{T}_i .

The *right envelope* of all extended rectangles \tilde{l} for all labels l outlines all reference point positions that are impossible, or cannot occur any more, see the bold line in Figure 3.63. We call this right envelope the *frontier* and denote it by F .

To determine the next leftmost label, we only have to consider the frontier F and the segments $h_{2i-1}, h_{2i}, v_{2i-1}$, and v_{2i} of the points p_i to the right of F that don't have a label yet. Given a horizontal segment h and the frontier F , there are three possibilities: (i) h lies completely left of F . Then h can be discarded; a point on it cannot be a reference point for a label that doesn't overlap another label. (ii) h lies completely right of F . Then the leftmost point on h is a candidate for the next leftmost label. (iii) h intersects F . Then a point just right of the intersection point is the candidate. For a vertical segment v , a similar situation occurs. If v lies left of F , it can be discarded; if v lies right of F , any point on v can be chosen; and if v and F intersect, then any point on v right of F can be chosen as a candidate.

Let H be the set of all horizontal segments that represent reference points of the labels. Similarly, let V be the set of the corresponding vertical segments. Let $H_{\text{right}} \subseteq H$ be the subset of all horizontal segments that lie completely right of F , see Figure 3.64. Let $H_{\text{int}} \subseteq H$ be the subset of all horizontal segments that intersect F . Let $H_{\text{left}} \subseteq H$ be the subset of all horizontal segments that lie completely left of F (these cannot give a valid label any more). Let $V_{\text{int,right}} \subseteq V$ be the subset of all vertical segments that contain at least some point right of F .

To maintain the frontier and the candidates for the best reference point efficiently, we need some data structures. Some of the data structures are used to find the next leftmost label; other data structures are only used to update the former ones efficiently. The data structures are red-black trees \mathcal{T} , heaps \mathcal{H} , and priority search trees \mathcal{P} [McC85]. These are also described in standard textbooks on algorithms [CLR90] and computational geometry [dBvKOS97].

Finding the Leftmost Label

We use three data structures to find the leftmost label position among the ones represented by H_{int} , H_{right} , and $V_{\text{int,right}}$. They are:

1. For each segment in H_{right} we store the x -coordinate of its right endpoint. This corresponds to the right edge of a label whose reference point is the left endpoint of the segment. These values are stored in a heap $\mathcal{H}_{\text{right}}$, where the root stores the minimum.
2. The subset H_{int} is stored as follows. For each vertical segment f_i of F , we maintain a red-black tree \mathcal{T}_i with the segments in H_{int} that intersect f_i (see the middle picture of Figure 3.64). These are stored in the leaves sorted on y -coordinate. With each leaf we also store the width of the corresponding label. We augment each red-black tree by storing at each internal node the minimum width label in the subtree of that node [CLR90]. We use a heap \mathcal{H}_{int} to have fast access to the segment in H_{int} that allows the leftmost label placement. \mathcal{H}_{int} stores for each \mathcal{T}_i the sum of the x -coordinate of f_i and the minimum width of the segments in \mathcal{T}_i . Thus the root of \mathcal{H}_{int} corresponds to the leftmost label among the labels represented by H_{int} .

- 3.** For the vertical segments in V , we don't maintain the set $V_{\text{int, right}}$ but some set V' for which $V_{\text{int, right}} \subseteq V' \subseteq V$. The set V' may contain vertical segments that lie completely left of the frontier; these are removed later. The x -coordinate of each segment of V' is stored in a heap \mathcal{H}_V . After extracting the minimum from \mathcal{H}_V , we test whether it is in $V_{\text{int, right}}$, as described later in **3a**. If not, we discard it and extract the next minimum from the heap, until we find one in $V_{\text{int, right}}$.

We query the three heaps described above. Among their answers, one corresponds to the leftmost label. This is the label we place.

Update assistance structures

After the leftmost label has been determined, we must update the frontier F and several of the data structures described above. This is not so easy. We'll use some more data structures that help to do the updating after the frontier has changed. Let f_{new} be the right edge of the extended rectangle \tilde{l} of the newly placed label l . The new frontier F is the right envelope of the old frontier and f_{new} , see Figure 3.65.

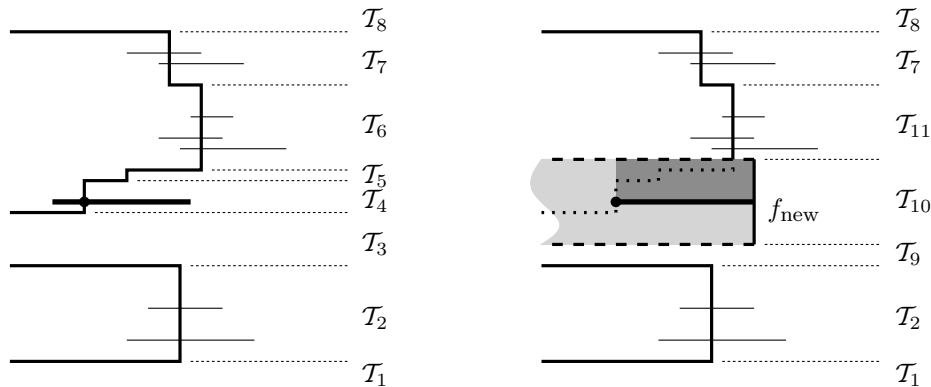


Figure 3.65: When the fat horizontal segment s from H_{int} is chosen, the frontier becomes the right envelope of f_{new} and the old frontier. The new label is dark grey. The grey range (light and dark) is the one with which queries in the priority search trees are done.

- 1a.** To determine which segments move from H_{right} to H_{int} or H_{left} when the frontier changes, we use a priority search tree $\mathcal{P}_{\text{left}}$ on the left endpoints of segments in H_{right} . After placing a label, we query $\mathcal{P}_{\text{left}}$ with the region left of f_{new} (grey in Figure 3.65) to locate the left endpoints of all segments that are no longer in H_{right} . We delete these endpoints from $\mathcal{P}_{\text{left}}$, and we delete the corresponding segments from the heap $\mathcal{H}_{\text{right}}$. For each deleted segment we test whether its right endpoint is right of the frontier. If so, that segment is in H_{int} , and we insert it in the data structures for H_{int} . If not, the segment is in H_{left} and can be discarded.

- 2a.** To determine which segments move from H_{int} to H_{left} when the frontier changes, we use a priority search tree $\mathcal{P}_{\text{right}}$ on the right endpoints of segments in H_{int} . After placing a label, we query $\mathcal{P}_{\text{right}}$ with the region left of f_{new} (grey in Figure 3.65) to locate all right endpoints of segments that have moved from H_{int} to H_{left} . Then we delete the entries corresponding to these segments from the trees \mathcal{T}_i , from the heap \mathcal{H}_{int} and from $\mathcal{P}_{\text{right}}$ itself.

When the frontier changes, we must also reorganize the red-black trees and \mathcal{H}_{int} as a whole. Recall that we use a red-black tree \mathcal{T}_i for each vertical segment of F . At most three new vertical segments can arise when the frontier changes, but many more vertical segments may cease to exist. We use the trees of the destroyed vertical segments of F to assemble the at most three new red-black trees. This is done by the operations SPLIT and CONCATENATE, which are standard for red-black trees. In Figure 3.65 the trees $\mathcal{T}_3, \mathcal{T}_4, \mathcal{T}_5$, and \mathcal{T}_6 are reorganized to the new trees $\mathcal{T}_9, \mathcal{T}_{10}$, and \mathcal{T}_{11} . The heap \mathcal{H}_{int} is updated by removing the value of each destroyed tree, and by inserting the value of each new tree.

- 3a.** Due to the lazy deletion of segments from \mathcal{H}_V , we don't need any additional data structures to update the heap on the vertical segments. However, we need to decide whether an extracted minimum from the heap really is in $V_{\text{int, right}}$. We use an augmented red-black tree \mathcal{T}_V for this test. The leaves of this tree store the vertical segments of the frontier sorted from bottom to top. Each leaf also stores the x -coordinate of its segment. Each internal node is augmented with a value that represents the minimum x -coordinate in its subtree. For any query y -interval, a search in \mathcal{T}_V reports the minimum x -coordinate of the frontier in this y -interval.

Algorithm

While there are still segments in any of the heaps \mathcal{H}_{int} , $\mathcal{H}_{\text{right}}$, or \mathcal{H}_V , do the following steps:

1. Let v be the vertical segment that corresponds to the minimum of \mathcal{H}_V . Search with v in the augmented red-black tree \mathcal{T}_V to see if v has some point right of F . If not, remove v from \mathcal{H}_V and repeat this step.
2. Determine the smallest among the minima of the three heaps \mathcal{H}_{int} , $\mathcal{H}_{\text{right}}$, and \mathcal{H}_V . Remove this minimum from its heap. Let l_i be the label position of point p_i corresponding to this minimum. Choose l_i as the next label to be placed.
3. Determine f_{new} , the right edge of the extended rectangle \tilde{l}_i . Update the frontier F with f_{new} . Update the augmented red-black tree \mathcal{T}_V (from **3a.**) with f_{new} .

Search with the region horizontally left of f_{new} (grey in Figure 3.65) in the priority search trees $\mathcal{P}_{\text{left}}$ and $\mathcal{P}_{\text{right}}$ (from **1a** and **2a**) and update the

structures $\mathcal{H}_{\text{right}}$ (from **1**), $\mathcal{P}_{\text{left}}$ (from **1a**), \mathcal{H}_{int} and the \mathcal{T}_i 's (from **2**), and $\mathcal{P}_{\text{right}}$ (from **2a**) as explained in the description of these structures.

4. Remove all other reference segments corresponding to p_i from the data structures, in which they occur.

Analysis

The basic structures used by the algorithm are heaps, red-black trees, augmented red-black trees, and priority search trees. All of these structures require $O(n)$ space for a set of size n . Also, these structures can be updated in $O(\log n)$ time per insertion or deletion, or extract-min for heaps. Red-black trees allow SPLIT and CONCATENATE in $O(\log n)$ time. The queries on the red-black trees take $O(\log n)$ time, and the queries on the priority search trees take $O(k + \log n)$ time, where k is the number of points found in the query range.

The algorithm's runtime of $O(n \log n)$ follows from the following observations. Any vertical segment f_{new} creates one vertical edge in the frontier F , and shortens at most two of them. It follows that throughout the whole algorithm, at most $3n - 2$ different vertical edges appear in F . Therefore, at most $3n - 2$ vertical edges can be destroyed in the whole algorithm (although many can be destroyed when one vertical segment f_{new} is added to the frontier). This bounds the total number of red-black trees \mathcal{T}_i (from **2**) that can appear, the total number of SPLIT operations, and the total number of CONCATENATE operations by $O(n)$. Since SPLIT and CONCATENATE operations take $O(\log n)$ time each, at most $O(n \log n)$ time is spent on splitting and concatenating. The augmented red-black tree \mathcal{T}_V (from **3a**) can also be maintained in $O(n \log n)$ time for the same reasons.

For each new label placed, one query is done on each of the two priority search trees $\mathcal{P}_{\text{left}}$ and $\mathcal{P}_{\text{right}}$. Such a query takes $O(k + \log n)$ time, where k is the number of points in the range. These points are always deleted from the priority search tree, so the algorithm cannot spend time on reporting these points again later in the algorithm. The priority search trees are initialized with one point for each horizontal segment, and we never add more points to them. So in total, at most $O(n \log n)$ time is spent for initializing, querying and updating the priority search trees.

Closed labels

So far we have only discussed the placement of labels that were allowed to touch at the boundaries, that is, the disjoint placement of open rectangles. How can the ideas be adapted to incorporate closed rectangles as labels? Firstly, we let the frontier represent a closed region where reference points of labels cannot lie any more. But the real problem is that we cannot choose and place the *leftmost* label, because this is not well-defined in the slider model with closed rectangles. The solution is to make a distinction between a placement of a rectangle at

some position with x -coordinate \bar{x} and a placement at some position with x -coordinate arbitrarily close to \bar{x} , but still strictly to the right of it. Such a distinction can be made by using a symbolic value $\epsilon > 0$ that is arbitrarily close to 0. In case of ties in x -coordinates of labels in the heap, one of them may have been moved symbolically to the right, which resolves the tie. If neither or both labels have been moved symbolically, there is a real tie and we can choose either label as the leftmost. When the algorithm finishes and a set of labels has been selected, then the actual positions of these label can be computed.

We conclude:

Theorem 3.17 *Given n points in the plane, and for each point a rectangular label with fixed height and some given width, then for each of the fixed-position and slider models, there is an $O(n \log n)$ time and linear space algorithm which places at least half the optimal number of labels.*

Remark 3.18 For fixed position models, the algorithm can be implemented using only one priority search tree and one heap. We initialize the priority search tree with the reference points of all label positions. In the heap, we store the sum of x -coordinate and label width for each reference point. When the label corresponding to the heap's minimum is chosen, we query in the priority search tree with the appropriate range to find the reference points that are no longer valid. We remove the entries of these reference points from heap and priority search tree, and repeat by selecting the minimum from the heap.

3.3.3 A Polynomial Time Approximation Scheme

In this section we present schemes for approximating the number of points we can label with unit height labels in all slider models. First we will only consider the top-slider model and then show how these results can be generalized to polynomial time approximation schemes for the two- and four-slider model.

Top-slider model

Given a constant $\epsilon \in (0, 1)$ we show that there is an algorithm that finds a top-slider labeling of at least $(1 - \epsilon) \cdot k_{\text{opt}}^{\text{1S}}$ points, where $k_{\text{opt}}^{\text{1S}}$ is the number of labeled points in an optimal top-slider solution. The algorithm has running time $O(n^{4/\epsilon^2})$.

We use line stabbing to split the problem into smaller units as suggested in [AvKS98]. We stab the unit height labels with horizontal lines of spacing strictly greater than 1 such that each label is stabbed by exactly one line. This can be done in $O(n \log n)$ time [AvKS98] and gives us a partition of the set of input points P into disjoint sets P_1, \dots, P_m , where P_i contains all points whose label intersects the i -th line, and m is the number of stabbing lines.

If we want to obtain an approximation ratio better than $1/2$, we cannot afford to discard every second subset P_i of input points. Instead, we have to

look at groups of t consecutive subsets. For $1 \leq i \leq t + 1$, let

$$P^i = P - \bigcup_{j=0}^{\lfloor \frac{m-i}{t+1} \rfloor} P_{i+j \cdot (t+1)}$$

be the set of points that we get from P if we discard every $(t + 1)$ -st subset starting with P_i . This makes sure that if we compute the optimal solution for t consecutive lines, then we get an approximation for P^i since solutions for its blocks of t lines do not interfere with each other. The pigeon hole principle guarantees that one of the $t + 1$ sets of type P^i has an optimal solution of size at least $\frac{t}{t+1} \cdot k_{\text{opt}}^{\text{LS}}$. In [AvKS98] this approach was taken, where the optimal solution for the t -lines problem was solved by dynamic programming. In the case of sliding labels one cannot take this approach because the number of candidate label positions in the discretization is superpolynomial. We will still arrive at a polynomial time approximation scheme for the original problem by *approximating* the t -lines subproblem.

Suppose we find a $\frac{k}{k+t-1}$ -approximation for the t -line problem, then we can approximate the original problem by a factor of $\gamma = \frac{k}{k+t-1} \cdot \frac{t}{t+1}$, which depends on the two parameters t and k . Setting $k = (t+1)(t-1)$ and $t = \lceil 2/\varepsilon \rceil - 2$ then yields $\gamma = t/(t+2) \geq 1 - \varepsilon$, the desired approximation factor. If the instance needs less than $\lceil 2/\varepsilon \rceil - 2$ stabbing lines, the solution of the problem becomes easier. In this case we set $k = (m-1)(\lceil 1/\varepsilon \rceil - 1)$ and approximate the m -line problem directly with a factor of $\gamma = \frac{k}{k+m-1} \geq 1 - \varepsilon$. The running time would then slightly improve to n^{k+1} . So we can assume $t \leq m$ from now on.

It remains to show how we can approximate an optimal solution for t lines by a factor of $\frac{k}{k+t-1}$. The idea is simple and uses the geometrical flavor of the problem. We call a labeling of a set of points *canonical* if all points are labeled and, going through the points from left to right, all labels have been pushed as far left as possible, that is, until they nearly hit another label or have arrived in their leftmost position. (Recall that labels are not allowed to touch each other. As in Section 3.3.2 we treat the distinction between an x -coordinate and a position slightly more to the right symbolically.) Now we just look at *all* canonical label placements of k points. For each such placement we consider the vertical line that goes through the right edge of its rightmost label. We search for the canonical labeling of k points with the leftmost such line ℓ_{left} , see Figure 3.66. (We have *not* visibly drawn the infinitesimally small spaces between the labels.) We call this placement *leftmost* and compare it to the leftmost k labels of the optimum. Let ℓ_{opt} be the vertical line that goes through the k -th leftmost right label edge of the optimal solution, see Figure 3.67. Then we know that ℓ_{left} is at least as far to the left as ℓ_{opt} . We would like to repeat this process with all sets of k points to the right of ℓ_{left} . We must label them under the restriction that their labels can only be placed to the right of ℓ_{left} . If we do so, by how much do we get worse than the optimal solution?

By definition ℓ_{opt} touches one label of the optimal solution and intersects up to $t - 1$ labels on the other $t - 1$ lines. Since ℓ_{left} is not to the right of ℓ_{opt} ,

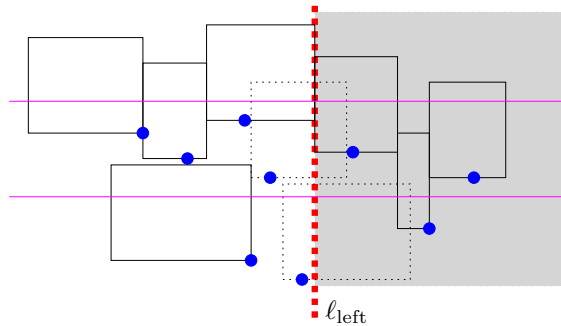


Figure 3.66: Leftmost label placement for a subset of $k = 4$ labels and $t = 2$ lines.

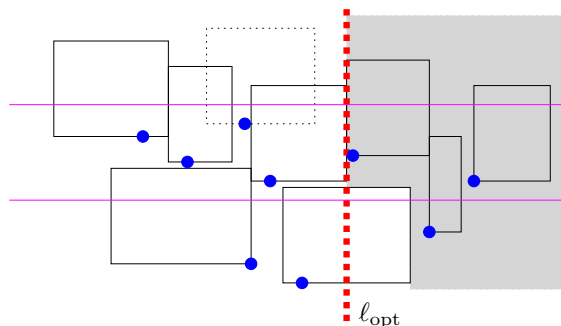


Figure 3.67: An optimal solution for the same points as in the figure above.

the constraint that our leftmost labeling exerts on the next group of k labels is no stronger than the constraint defined by the labels of the optimal solution touching or intersecting ℓ_{opt} , see the gray zones in Figure 3.66 and 3.67. Thus we have placed our first k labels in at most as much ‘space’ as the first $k + t - 1$ labels of the optimal solution. This makes sure that the next line like ℓ_{left} , defined by the next (restricted) leftmost labeling of k points, will again be at most as far to the right as the vertical line through the $(2k + t - 1)$ -st leftmost right label edge of the optimal solution. By repeating this process until all points are used up, we get a $k/(k + t - 1)$ -approximation for the number of labeled points in an optimal solution since we always fit k labels in at most as much space as $k + t - 1$ labels of the optimal solution. This shows that for the appropriate choice of t and k , we obtain a $(1 - \varepsilon)$ -approximation for the whole problem.

Let n' be the number of points whose labels intersect a fixed set of t consecutive lines. What is the time we need to compute the first leftmost placement of k out of these n' points? We enumerate all $\binom{n'}{k}$ choices of these k points. For each choice we have to find its canonical labeling—if there is any. Observe that labeling a point p_1 can constrain the labeling of a point p_2 to its left only by not at all allowing to label it. Since we are only interested in subsets of k points that can be labeled completely, it is enough to go through the points

in lexicographical order and try to place each of them leftmost. We can find a label's leftmost position by going through the list of its predecessors once, so finding a canonical labeling can certainly be done in $O(k^2)$ steps. This means that it takes us $O((n')^k)$ steps to compute the first leftmost labeling. Thus we need $T_{t\text{-line}}(n') = \sum_{j=0}^{\lfloor n'/k \rfloor} O((n' - jk)^k) = O((n')^{k+1})$ time for an approximate solution of the t -line problem. In order to get the total running time T_{total} , we must sum up $T_{t\text{-line}}$ over all possible groups of t consecutive lines. In every group there are at most n points and m , the number of stabbing lines, is at most n as well. Hence $T_{\text{total}}(n) = O(n^{k+2})$. Using $k = (t + 1)(t - 1)$ and $t = 2/\varepsilon - 2$ as above yields $T_{\text{total}}(n) = O(n^{4/\varepsilon^2})$.

Two and four sliders

The scheme for the top-slider model immediately translates into a polynomial time approximation scheme for the two-slider model. For each point of the input set, we simply place a copy at unit distance below it. (To avoid trouble with an original point at the same place, we can move all copies upwards by an infinitesimal amount.) Then only one point of every such pair is labeled in a top-slider solution. Optimal top-slider solutions for this instance correspond one-to-one to optimal two-slider solutions for the original instance. The running time increases only by a constant factor.

In order to use the ideas given above for the four-slider model, we have to do a little more work. Since labels can now move up and down, the use of stabbing lines is not appropriate any more. Instead, we partition the set of input points into m strips of unit height. A strip contains all points between its two bounding horizontal lines and all points that lie on the upper boundary. Similar to our approach above, we will approximate the solution of t consecutive strips. This time, however, we have to drop the points of *two* strips between two blocks to guarantee that solutions of one block do not interfere with solutions of an adjacent block. The pigeon hole principle makes sure that one of the $t + 2$ different sets we get by gluing blocks together has at least cardinality $n \cdot \frac{t}{t+2}$. Suppose we have a $\frac{k}{k+t}$ -approximation for the t -strip problem, then we could approximate an optimal solution of the whole instance by a factor of $\gamma = \frac{k}{k+t} \cdot \frac{t}{t+2}$. Setting $k = t(t + 2)$ and $t = \lceil 3/\varepsilon \rceil - 3$ would then result in $\gamma = t/(t + 3) \geq 1 - \varepsilon$, the desired approximation factor.

The additional difficulty in designing an approximation for the t -strip problem is that we do not know on which of its four sides a label in the optimal solution is attached to its point. We can handle this by considering all four possibilities for each of the k points we have chosen. Now we define a canonical labeling as follows. If a label is to be attached to its point on the top or bottom edge, we again push it as far left as possible. If however its point is going to lie on the right or left edge, we push the label as far down as possible. The idea with considering a special order of the points does not work in this setting, so we try to label the k points in every of the $k!$ possible orders, and for every order we check each of the 4^k possible kinds of placement: left, right, bottom,

or top. In this way we can again find a leftmost labeling and a line ℓ_{left} . The constraint that the leftmost labeling exerts on the next group of k labels is at most as strong as the corresponding constraint of the assumed optimal solution. As above, the constraint of the optimal solution is defined by ℓ_{opt} and the labels of the optimal placement intersected by ℓ_{opt} . Apart from the label whose right edge defines ℓ_{opt} , at most t labels can intersect ℓ_{opt} without intersecting each other since their points have to lie within a vertical strip of height strictly less than t (the bottom borderline is excluded). Hence we have a $\frac{k}{k+t}$ -approximation for the t -strip problem.

In the approximation algorithm for the four-slider model, we need $\binom{n'}{k} k! 4^k k^2$ steps to compute the first leftmost labeling. This still yields an overall running time of $O(n^{4/\varepsilon^2})$.

Theorem 3.19 *For each of the slider models and for any constant $\varepsilon > 0$, there is a polynomial time algorithm which labels at least $(1 - \varepsilon)$ times the maximum number of input points that can be labeled.*

3.3.4 Implementation and Experimental Results

The greedy algorithm of Section 3.3.2 has been implemented for the fixed-position and slider models and tested on three real world data sets from different application areas and on a large sequence of randomly generated point sets. In this section we compare experimentally how many labels are placed in each of the six models.

The algorithms were implemented by Tycho Strijk, Universiteit Utrecht, in C++. For the data structures he made use of the LEDA library [NM90]. He simplified the implementation described in Section 3.3.2 in three respects. Firstly, the red-black trees \mathcal{T}_k can be expected to contain only a few horizontal segments at any moment. So he used simple lists for them. Secondly, LEDA does not have an implementation for priority search trees; he used orthogonal range trees instead. Thirdly, the augmented red-black tree \mathcal{H}_V does not profit much from the augmentation in practice. When searching for the minimum x -coordinate of the frontier F in a y -interval, he simply scans all leaves of the red-black tree in that interval. One can expect to visit only a few leaves, since the y -interval is only twice the unit height.

The first of the three data sets contains 1000 cities of the USA that must be labeled with their name. We used several different font sizes, and determined the bounding boxes of the label text. The tables of Figure 3.68 show the results. The codes 1P, 2P, and 4P are shorthand for the 1-, 2-, and 4-position models. The codes 1S, 2S, and 4S are shorthand for the corresponding slider models. The values in the second table show the results in percentages with respect to the 4-position labeling.

The second data set contains the 236 points of a data posting. The labels are measurement values and come from a book on geostatistics [IS89]. Figure 3.69 shows the labeled data set and the number of labels placed in each model.

font	Number of labels placed						font	Percentage w.r.t. 4-position model					
	model							model					
	1P	2P	4P	1S	2S	4S		1P	2P	4P	1S	2S	4S
5	851	950	971	990	993	999	5	87	97	100	101	102	102
6	777	910	952	967	982	986	6	81	95	100	101	103	103
7	705	852	901	932	964	972	7	78	94	100	103	106	107
8	686	845	896	918	952	958	8	76	94	100	102	106	106
9	607	758	817	836	890	902	9	74	92	100	102	108	110
10	554	704	769	787	853	872	10	72	91	100	102	110	113
11	520	657	721	735	805	831	11	72	91	100	101	111	115
12	500	637	709	719	796	813	12	70	89	100	101	112	114
13	448	570	638	649	716	734	13	70	89	100	101	112	115
14	433	557	624	637	695	712	14	69	89	100	102	111	114
15	382	494	550	556	627	645	15	69	89	100	101	114	117

Figure 3.68: One thousand cities on a large map.

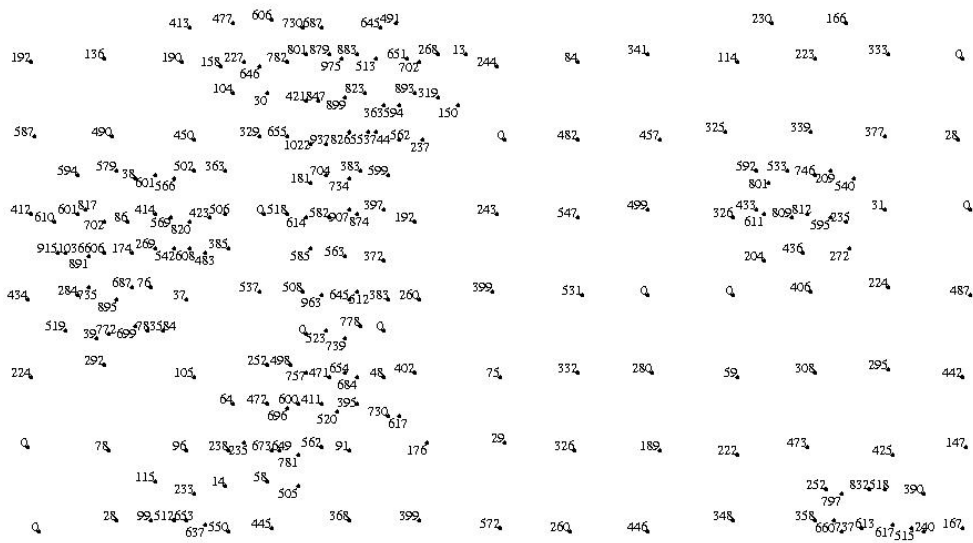
The third data set contains 75 points of a regression analysis. Here the points are clustered near a regression line, and the labels are simply identification numbers. Figure 3.70 shows the labeling.

The bottom tables of Figures 3.69 and 3.70 show that the 4-slider model sometimes places 10–15% more labels than the 4-position model. This improvement is significant, since it is always caused by a better labeling of the areas that are difficult to label. We also created artificial, pseudo-random data sets where all areas are hard to label. These sets were constructed by first placing all points on a grid and after that they were moved randomly a slight distance away from the grid point. Here we indeed found higher improvements: up to 92%.

Efficiency was not the main motivation for these experiments. Still it appeared that the label placement was computed in a few seconds for all data sets we tried, up to 2500 points. A plot shown on a computer screen seldom contains more than 1000 labeled points.

Christensen, Marks and Shieber compared different algorithms using random point sets [CMS95]. Their standard data sets were generated as follows. Inside a grid of size 792 by 612 units, n points were randomly placed and had to be labeled with labels of 30 by 7 units. We considered examples with $n = 100, 250, 500, 750, 1000,$ and 1500 points. For each example size, we generated 25 files. We ran the greedy algorithm for each of our six models on all of the generated files. The average percentages of placed labels over the 25 trials are shown in Figure 3.71. Clearly the labeling model has a big influence on the results.

In Figure 3.72 we extend the comparison of Christensen et al. by the results of our algorithm for the four-position and the four-slider model. Our four-position algorithm is always better than gradient descent, and the denser the map the better it gets in relation to gradient descent. For 1500 points it is almost as good as simulated annealing. The four-slider algorithm yields almost equal results as simulated annealing for less than 750 points and is always better beyond 750 points. The running time of our algorithm is generally only a few



font	Number of labels placed model					
	1P	2P	4P	1S	2S	4S
5	229	236	236	236	236	236
6	216	235	235	236	236	236
7	197	219	230	236	236	236
8	197	219	230	236	236	236
9	185	205	218	235	236	236
10	175	193	207	223	231	230
11	174	189	200	213	221	224
12	174	189	200	213	221	224
13	169	180	188	203	212	212
14	169	180	188	203	212	212
15	157	170	176	192	200	203

font	Percentage w.r.t. 4-position model					
	1P	2P	4P	1S	2S	4S
5	97	100	100	100	100	100
6	91	100	100	100	100	100
7	85	95	100	102	102	102
8	85	95	100	102	102	102
9	84	94	100	107	108	108
10	84	93	100	107	111	111
11	87	94	100	106	110	112
12	87	94	100	106	110	112
13	89	95	100	107	112	112
14	89	95	100	107	112	112
15	89	96	100	109	113	115

Figure 3.69: Labeling of the data posting in 9pt font using the 4-slider model (scaled to fit), and tables with the performance.



font	Number of labels placed					
	model					
	1P	2P	4P	1S	2S	4S
5	75	75	75	75	75	75
6	75	75	75	75	75	75
7	70	74	74	75	75	75
8	70	74	74	75	75	75
9	60	69	70	73	74	74
10	58	65	68	72	72	72
11	55	61	66	66	70	70
12	55	61	66	66	70	70
13	51	58	64	63	68	71
14	51	58	64	63	68	71
15	50	56	61	62	67	68

font	Percentage w.r.t. 4-position model					
	model					
	1P	2P	4P	1S	2S	4S
5	100	100	100	100	100	100
6	100	100	100	100	100	100
7	94	100	100	101	101	101
8	94	100	100	101	101	101
9	85	98	100	104	105	105
10	85	95	100	105	105	105
11	83	92	100	100	106	106
12	83	92	100	100	106	106
13	79	90	100	98	106	110
14	79	90	100	98	106	110
15	81	91	100	101	109	111

Figure 3.70: Labeling of the scatter plot in 11pt font using the 4-slider model (scaled to fit), and tables with the performance.

seconds; even the four-slider algorithm needed just 12 seconds for the largest data sets with 1500 points on a SUN Ultra Sparc. Simulated annealing takes several minutes to label these point sets on the same machine.

model	Percentage of labels placed					
	number of points					
	100	250	500	750	1000	1500
1P	92.60	84.30	73.16	64.56	57.96	48.58
2P	99.56	97.39	90.24	82.22	74.73	62.75
4P	99.84	99.07	95.45	90.47	83.99	71.74
1S	99.72	98.42	93.80	87.80	81.92	71.04
2S	99.92	99.55	97.83	94.85	90.71	80.75
4S	99.96	99.58	98.02	95.37	91.68	82.68

Figure 3.71: Random data sets (results are averaged over twenty-five trials).

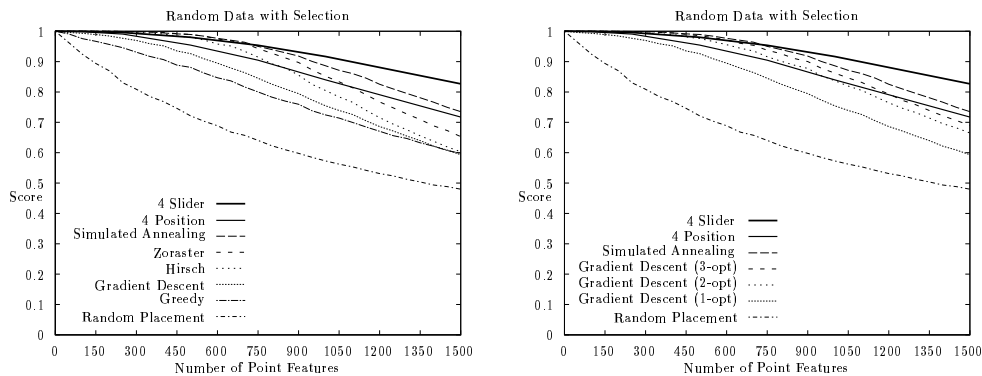


Figure 3.72: Comparison of the four-position and four-slider algorithm to other labeling algorithms.

Chapter 4

Point Labeling: Label-Size Maximization

When placing labels on maps, maximizing the (weighted) number of features that receive a label is certainly the aim that plays the greatest role in practice. However, one might think of other, for instance technical applications where holes must be drilled next to a given number of points on a piece of metal, and the size of these holes is to be maximized. This is an example of the label-size maximization problem that we consider in this chapter.

When comparing the complexity status of label-number and label-size maximization, it is difficult to decide which problem is harder. Label-size maximization can be solved in polynomial time in some special cases: efficient algorithms are known for two label candidates per feature [FW91] and, if the label candidates overlap in a certain way, for any constant number of label candidates per feature [PZC98, SvK99], and even for an infinite number of label candidates per feature [KSY99]. On the other hand the problem of maximizing the number of points with axis-parallel rectangular labels can be approximated arbitrarily well (see Section 3.3.3) while maximizing the size of square labels cannot be approximated better than by a factor of $1/2$ [FW91].

The label-size maximization problems that have been considered so far are the following: labeling points with circles, squares or other regular polygons [FW91, DMM⁺97] and labeling axis-parallel line segments with axis-parallel rectangles that have the same length as the segment they label and must touch or contain the segment [PZC98, SvK99, KSY99]. In this chapter we will consider labeling points with axis-parallel rectangles and with circles.

4.1 Rectangular Labels

Formann and Wagner proposed an approximation algorithm that maximizes the size of uniform axis-parallel square labels. It is optimal in respect to both, its approximation factor of $1/2$ and its running time of $O(n \log n)$ [FW91, Wag94].

Here n refers to the number of points, each of which has four label candidates. For the same problem, there is an algorithm that keeps the theoretical optimality of the approximation algorithm, but performs close to optimal in practice [WW97].

It is obvious that the approximation result of Formann and Wagner also holds for uniform rectangular labels; the coordinate system can always be scaled such that these labels become squares. For rectangles of arbitrary width, however, or for rectangles of arbitrary height and width, no approximation algorithms are known, not even heuristics have been suggested. However, the label-size maximization problem can be reduced to the decision problem if the number of *conflict sizes* can be bounded, i.e. the scaling factors for which label candidates start to intersect. Then one can do a binary search on a sorted list of these conflict sizes. For each step of the search, one would compute the current candidate conflict graph and call an algorithm for the decision problem for the current scaling factor. Any algorithm for the label-number maximization problem could be employed; if it does not find a complete labeling, we return false, and the binary search continues with a lower value, with a higher value otherwise.

In practice, however, the number of available font sizes is usually a small constant, hence a binary search on a list of conflict sizes is not necessary. One can simply start with the smallest font, call label-number maximization, and repeatedly increase the font size as long as the number of unlabeled points is tolerably small.

In [DMM⁺97] Doddi et al. suggest such a bi-criteria algorithm that mediates between the two fundamental optimization problems, namely label-size and label-number maximization. Given an $\varepsilon > 0$, the algorithm labels at least a $(1 - \varepsilon)$ - fraction of the points with axis-parallel uniform square labels of size at least $n_{\text{opt}}/(1 + \varepsilon)$, where n_{opt} is the edge length of the squares in an optimal solution of all points. The algorithm puts $1/\varepsilon$ equidistant markers on each label edge and places the label such that one of the markers coincides with the point to be labeled.

In the same paper, Doddi et al. give approximation algorithms that maximize the size of square labels of arbitrary orientation and of circular labels, again under the restriction that all labels are uniform, i.e. of equal size. The approximation factors of their algorithms are approximately $\frac{1}{37}$ for squares and $\frac{1}{30}$ for circles. In the next section we will improve the approximation factor of their algorithm for circular labels by about 50%.

4.2 Circular Labels

This section is joint work with Tycho Strijk, Universiteit Utrecht.

When labeling points, labels are usually restricted to axis-parallel rectangles which (a) have to touch the point they label, and (b) must not intersect any other label. Condition (a) has often been further restricted in that one of a

label's corners must coincide with the point to be labeled. In this section we restrict ourselves to a different label shape, namely circles of uniform size, while keeping conditions (a) and (b). We *label* a point by attaching a circle to it such that the circle's boundary contains the point. Our objective is to find the largest real d_{opt} , which still allows us to label *all* given points with non-overlapping circles of diameter d_{opt} . We consider our labels to be open circles, thus they may touch other points or labels.

In considering a set of three points in general position, it is clear that approximating the maximum size of circular labels cannot be reduced to the same problem for square labels. While the solution in the former case is bounded (linearly in the diameter of the point set), it is unbounded in the latter.

We show that even deciding whether a set of points can be labeled with unit circles is NP-hard, see Section 4.2.5. This settles an open question raised in [DMM⁺97]. The same proof implies that there is a constant $\delta < 1$ such that it is NP-hard to label points with circles of diameter greater than $\delta \cdot d_{\text{opt}}$. Nevertheless, the maximization problem has already been approximated. Doddi et al. suggested a simple algorithm that labels points with circles whose diameter is at least $1/(4(2 + \sqrt{3})) \approx 1/14.93$ times the optimum and takes $O(n \log n)$ time [DMM⁺97]. However, a careful revision of their proof, see Section 4.2.1, shows that the approximation factor of their algorithm is actually worse by a factor of 2; i.e. the label diameter is guaranteed to be at least $1/(8(2 + \sqrt{3})) \approx 1/29.86$ times the optimum. In this paper, we present an algorithm with an approximation factor of $1/19.59$. While the analysis that yields this factor becomes more involved, the algorithm remains simple.

Both algorithms first determine the smallest diameter D_3 of any three-point subset of the input points. This can be done in $O(n \log n)$ time [DLSS95]. D_3 is needed to compute the diameter of the labels, which in both cases is a constant fraction of D_3 . The observation that no point set of more than two points can be labeled with circles of diameter greater than $2(2 + \sqrt{3})D_3$ yields the respective approximation factors.

Like the algorithm of Doddi et al., when labeling a point our algorithm only needs to know the location of the point's closest neighbor in the set of input points. However, while Doddi et al. maximize the distance between the labels of a pair of closest neighbors, we minimize it in order to use space more efficiently. This implies that they only need to know the *direction* of the closest neighbor while we also need its *distance*. Another difference is that while they label the points in arbitrary order, we exploit this order and process the points in pairs of increasing distance. In order to build and access the data structure that supplies us with this order we need no more than $O(n \log n)$ time in total. Thus our algorithm runs in $O(n \log n)$ time. It requires linear storage.

This section is structured as follows. In Section 4.2.1, we sketch the algorithm of Doddi et al.. In Section 4.2.2 we formalize our main ideas. Then, in Section 4.2.3 we present our algorithm, analyze it in Section 4.2.4, and finally present our NP-hardness proof in Section 4.2.5.

4.2.1 Previous Work

For a (finite) set S of points in the plane, Doddi et al. define the *diameter* $\text{diam}(S)$ the usual way as the maximum distance of any two points in S . They define the *k-diameter* $D_k(S)$ to be the minimum diameter over all k -element subsets of S . Then they make the following two observations. In our description we will abbreviate $D_3(S)$ by D_3 where appropriate.

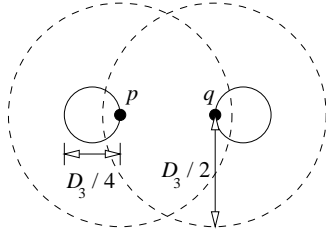


Figure 4.1: Label placement according to the algorithm of Doddi et al..

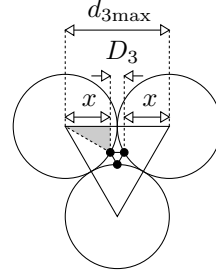


Figure 4.2: Optimal label placement for three points.

1. The open circle centered at a point $p \in S$ with radius $D_3/2$ contains at most one other point $q \in S - \{p\}$. Due to symmetry, an open circle of the same diameter centered at q only contains p and q . This allows p and q to be labeled with labels of diameter $d' = D_3/4$ as in Figure 4.1. Given the distance of p and q to other points in S , it is obvious that labels of other points cannot overlap those of p and q .
2. The maximum label diameter $d_{\text{opt}}(S)$ of any set S of more than two points cannot exceed the maximum label diameter $d_{3\text{max}}$ of three points at pairwise distance $D_3(S)$, see Figure 4.2. Doddi et al. compute $d_{3\text{max}}$ to be $(2 + \sqrt{3})D_3$, but this is incorrect: we have $d_{3\text{max}} = 2x + D_3$, where $x = (d_{3\text{max}}/2) \cdot \cos(\pi/6)$, see the shaded triangle with a right angle and a 30° angle in Figure 4.2. Thus we obtain $d_{3\text{max}} = 2(2 + \sqrt{3})D_3 \approx 7.46 D_3$ as an upper bound for d_{opt} .

Combining these observations yields the approximation factor $d'/d_{\text{opt}} \geq 1/(8(2 + \sqrt{3})) \approx 1/29.86$ of the algorithm of Doddi et al..

4.2.2 Preliminaries

We formalize the idea of the free space around a point as follows.

Definition 4.1 Let $C_{m,r}$ be an open circle with radius r centered at m . We say that two points are a pair of closest neighbors if each is a closest neighbor of the other. Given two points $p, q \in S$, we denote the point-free zone of p and q by $Z_{\text{free}}(p, q) = (C_{p,D_3} \cap C_{q,D_3}) \cup C_{p,d} \cup C_{q,d} \subseteq \mathbb{R}^2$ where $d = d(p, q)$ is the Euclidean distance of p and q .

The definition is illustrated in Figure 4.3. We show that the point-free zone of a pair of closest neighbors $\{p, q\}$ does not contain any other point of S . This will enable us to use part of the zone for labeling p and q .

Lemma 4.2 $Z_{\text{free}}(p, q) \cap S = \{p, q\}$ for any pair $\{p, q\}$ of closest neighbors in S .

Proof. Suppose $Z_{\text{free}}(p, q)$ contains a point $t \in S \setminus \{p, q\}$. Then $t \in C_{p, D_3} \cap C_{q, D_3}$ or $t \in C_{p, d} \cup C_{q, d}$. In the first case the diameter of $\{p, q, t\}$ would be less than D_3 ; a contradiction to the definition of D_3 . The second case would contradict $\{p, q\}$ being closest neighbors. \square

Note that Doddi et al. implicitly also used the concept of a point-free zone, namely the union of the dashed circles $C_{p, D_3/2}$ and $C_{q, D_3/2}$ depicted in Figure 4.1. However, their zone is always contained in our point-free zone Z_{free} , independently of d . This helps us to place larger labels. Let d_{algo} be the diameter of the labels our algorithm is going to place.

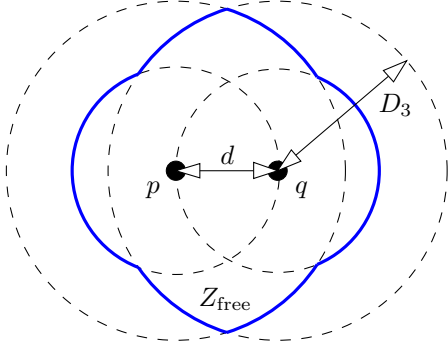


Figure 4.3: The point-free zone Z_{free} of p and q .

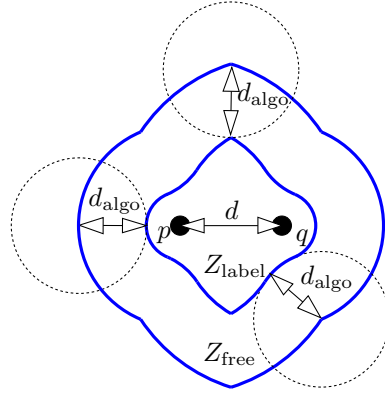


Figure 4.4: The label zone Z_{label} of p and q lies inside Z_{free} .

Definition 4.3 Given two points $p, q \in S$ and a real number $d_{\text{algo}} < D_3$. Then we denote the label zone of p and q by

$$Z_{\text{label}}(p, q; d_{\text{algo}}) = Z_{\text{free}}(p, q) \ominus C_{0, d_{\text{algo}}} = Z_{\text{free}}(p, q) - \bigcup_{x \in \mathbb{R}^2 - Z_{\text{free}}(p, q)} C_{x, d_{\text{algo}}}$$

where \ominus is the Minkowski subtraction operator and 0 is the origin.

In other words, the label zone is the erosion of the point-free zone with a disk of radius d_{algo} . The definition is illustrated in Figure 4.4.

Lemma 4.4 If we label a pair of closest neighbors $\{p, q\}$ with circles of diameter d_{algo} that are contained in the label zone $Z_{\text{label}}(p, q; d_{\text{algo}})$, then these labels cannot overlap the label of any other point $t \in S - \{p, q\}$.

Proof. Suppose the label of t overlaps that of p . Lemma 4.2 tells us that $t \notin Z_{\text{free}}(p, q)$. Then the definition of the label zone ensures that $C_{t, d_{\text{algo}}}$ does not intersect $Z_{\text{label}}(p, q; d_{\text{algo}})$. Observing that t 's label is contained in $C_{t, d_{\text{algo}}}$ and p 's label in $Z_{\text{label}}(p, q; d_{\text{algo}})$ contradicts our assumption. \square

The question is, of course, how large we can choose d_{algo} so that the labels of any pair of closest neighbors fit into their label zone—independently of their distance. This question is dealt with in Section 4.2.4. Let us suppose for the moment that d_{algo} can be expressed as a fraction of D_3 . The next section answers two other important questions, namely how to place the labels, and in which order.

4.2.3 Algorithm

Our algorithm proceeds as described in Figure 4.5. The value of $D_3(S)$ is computed with the algorithm of Datta et al. [DLSS95]. In contrast to Doddi et al. who place the labels of two neighboring points as far apart from each other as possible, we label p and q such that their labels are as close as possible. This means that they will touch each other as in Figure 4.6 if $d(p, q) \leq 2d_{\text{algo}}$. The vectors \vec{p} and \vec{q} denote the coordinates of p and q in the plane. We place the center m_q of q 's label at $m_q = \vec{p}/4 + 3\vec{q}/4 + \vec{a}$. The vector \vec{a} is perpendicular to \overline{pq} and oriented so that it points to the left of $(\vec{p} - \vec{q})$. The length of \vec{a} is $\|\vec{a}\| = \sqrt{d_{\text{algo}}^2/4 - d^2(p, q)/16}$. Correspondingly, the center of p 's label is placed at $m_p = \vec{q}/4 + 3\vec{p}/4 - \vec{a}$. We call the union of these two (open) labels the *label space* of p and q . If there are unlabeled points left after executing the while-loop, we label them arbitrarily.

```

LABEL_POINTS_WITH_CIRCLES( $S$ )
compute  $D_3(S)$ 
 $d_{\text{algo}} := 0.381 D_3(S)$ 
while  $|S| > 1$ 
    choose  $\{p, q\} \subseteq S$  with  $d(p, q)$  minimal
    if  $d(p, q) \geq 2d_{\text{algo}}$  then exit while-loop
    label  $p$  and  $q$  as in Figure 4.6
     $S := S \setminus \{p, q\}$ 
end
for all  $x \in S$  do label  $x$  arbitrarily end
return all label positions

```

Figure 4.5: Our algorithm.

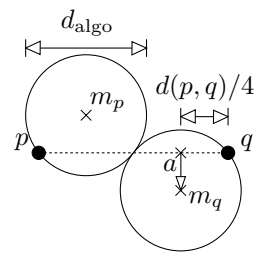


Figure 4.6: Labeling a pair of points.

Lemma 4.5 *Given a set S of n points and a label diameter d_{algo} such that the label space is contained by the label zone for any pair of points in S , then the labels our algorithm places do not intersect.*

Proof. The fact that no two labels overlap follows from the order in which we process the points. It is clear that the first pair $\{p, q\}$ is a pair of closest neighbors. Due to Lemma 4.4, we know that we do not constrain the labeling of any other point in S when we label such a pair within its label zone. In other words, if we remove $\{p, q\}$ from S , then we can ignore p and q as well as their labels for solving the remaining problem. The next pair of points will be a pair of closest neighbors in the reduced set S . Thus Lemma 4.4 applies to this pair as well.

After we leave the while-loop, there may be unlabeled points left. For each such point x there are two possibilities. Either its closest neighbor in the original set is at least $2d_{\text{algo}}$ away. Or all points y with $d(x, y) < 2d_{\text{algo}}$ have been labeled before, since each had a closer neighbor z . In either case the labeling of x is not constrained by any previous label placement. Hence we can label x arbitrarily. \square

Lemma 4.6 *The algorithm can be implemented such that it labels a set S of n points in $O(n \log n)$ time with linear space.*

Proof. Our algorithm labels S in three phases. In the first phase, we compute D_3 in $O(n \log n)$ time [DLSS95]. We need D_3 to compute the diameter $d_{\text{algo}} = 0.381 D_3$ of the labels we are going to place, see Section 4.2.4.

In the second phase, we set up a simple data structure that will answer a limited closest pair query; limited in the sense that we only need to know pairs of points closer than $2d_{\text{algo}}$ in the Euclidean metric. We call these *pairs of relevant neighbors*. An axis-parallel rectangle of size $2d_{\text{algo}} \times d_{\text{algo}}$ contains at most two input points since it fits into a circle of diameter D_3 . Thus an axis-parallel square of edge length $4d_{\text{algo}}$ centered at a point $p \in S$ contains at most twelve input points apart from p . The relevant neighbors of p are obviously among these. This observation enables us to collect all pairs of relevant neighbors with a sweep-line—or rather sweep-window—approach.

As usual we use two data structures: an event-point queue as horizontal structure and a sweep-line status as vertical structure. Our sweep window is a vertical strip of width $2d_{\text{algo}}$ and moves over the plane from left to right. Its right border line r stops at each event point. We have two kinds of events: when an input point $p = (x_p, y_p)$ enters the window (r is at x_p) and when p leaves the window (r is at $x_p + 2d_{\text{algo}}$). When p enters the window we want to report efficiently all points in the window whose y -coordinate is less than $2d_{\text{algo}}$ from y_p . For this purpose the sweep-window status is implemented by a balanced binary tree on the y -coordinates of the points in the window. For later on, we insert each pair $\{p, q\}$ that is reported during the sweep into a priority queue according to its Euclidean distance $d(p, q)$ if $d(p, q) < 2d_{\text{algo}}$. Our sweep takes $O(n \log n)$ time and uses linear space.

In the third phase, we repeatedly extract the minimum $\{p, q\}$ of the priority queue, label p and q with circles of diameter d_{algo} as in Figure 4.6, and delete all pairs containing either p or q from the queue. The remaining points are labeled

arbitrarily in constant time per point. Phase 3 can also be done in $O(n \log n)$ steps.

The running time of the three phases sums up to a total of $O(n \log n)$. The necessary data structures require linear space. \square

4.2.4 Analysis

Given a pair $\{p, q\}$ of closest neighbors in S , our objective now is to compute the maximum radius r of their labels so that the label space is still contained in the label zone $Z_{\text{label}}(p, q; 2r)$ of p and q . Since this radius will only depend on the distance d of p and q , we want to find the minimum r_{min} of $r(d)$, set d_{algo} to a value slightly less than $2r_{\text{min}}$ and run our algorithm. Lemma 4.5 guarantees that no two labels will intersect then.

Since we place the labels of p and q symmetrically, it is enough to analyze the placement of q 's label. We consider two cases depending on the distance of p and q .

In case $d \leq D_3/2$, the point-free zone $Z_{\text{free}}(p, q)$ is the intersection of C_{p, D_3} and C_{q, D_3} . The corresponding label zone $Z_{\text{label}}(p, q; 2r)$ is the intersection of C_{p, D_3-2r} and C_{q, D_3-2r} . As described in the previous section, the label has its center point m_q on a line h , normal to the line connecting p and q . This normal line has distance $d/4$ to q . The distance of m_q to the line \overline{pq} is $\sqrt{r^2 - d^2/16}$ using Pythagoras' rule.

The radius of q 's label is maximized when the label touches the boundary of the label zone, i.e. the circle C_{p, D_3-2r} . We use the property that the touching point of two circles always lies on the line through their centers, see Figure 4.7.

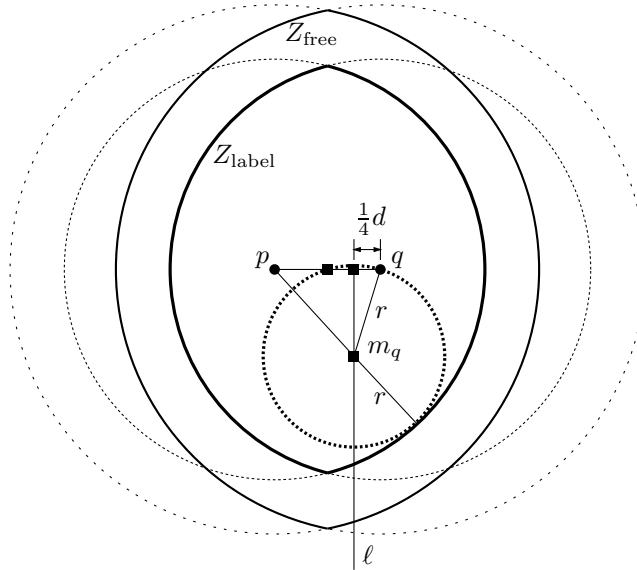


Figure 4.7: Point-free zone Z_{free} and label zone Z_{label} of p and q for $d \leq D_3/2$.

This observation yields the equality

$$d(p, m_q) + r = D_3 - 2r. \quad (4.1)$$

We use that $d(p, m_q) = \sqrt{(\frac{3}{4}d)^2 + (r^2 - d^2/16)} = \sqrt{d^2/2 + r^2}$. The resulting equation

$$\sqrt{d^2/2 + r^2} = D_3 - 3r$$

is quadratic and has two solutions. The solution valid for our problem is

$$r = \frac{3D_3 - \sqrt{D_3^2 + 4d^2}}{8}. \quad (4.2)$$

We will use the notation $\hat{d} = d/D_3$ and $\hat{r} = r/D_3$ to obtain less complicated formulas and to express the fact that the formulas can also be obtained as follows: first scale the point set by a factor $1/D_3$, then determine the optimum label size, and after that scale by a factor D_3 to the original size. As a result Equation (4.2) is simplified to

$$\hat{r} = \frac{3 - \sqrt{1 + 4\hat{d}^2}}{8}. \quad (4.3)$$

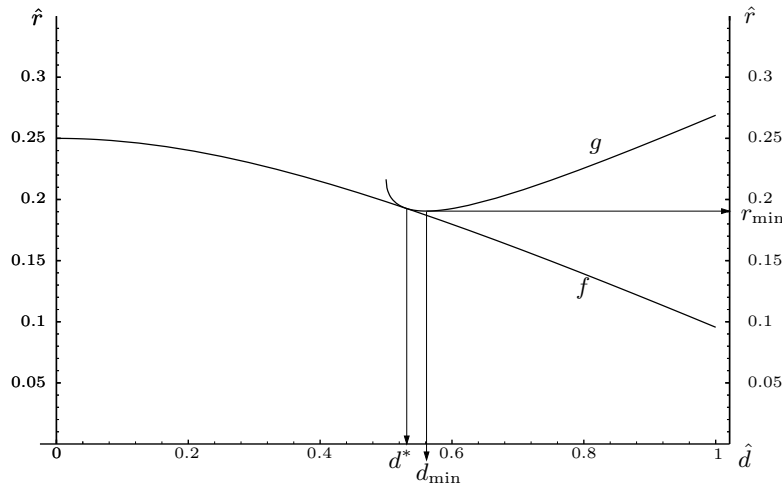


Figure 4.8: The graph of $r(d)$ is determined by the functions f and g . For $d < d^* \approx 0.53 D_3$, the value of r is determined by f , otherwise by g . The minimum $r_{\min} \approx 0.19 D_3$ of $r(d)$ is reached at $d_{\min} \approx 0.56 D_3$.

Graph f in Figure 4.8 depicts Equation (4.3) as a function of \hat{d} .

The case $d > D_3/2$ is more difficult. In this case, the point-free zone $Z_{\text{free}}(p, q)$ is the union of three areas, namely $C_{p, D_3} \cap C_{q, D_3}$, $C_{p, d}$, and $C_{q, d}$, see Figure 4.9.

Accordingly, the boundary of $Z_{\text{label}}(p, q; 2r)$ consists of arc segments that are part of the circles $C_{p, D_3 - 2r}$, $C_{q, D_3 - 2r}$, $C_{p, d - 2r}$, $C_{q, d - 2r}$ and four circles with

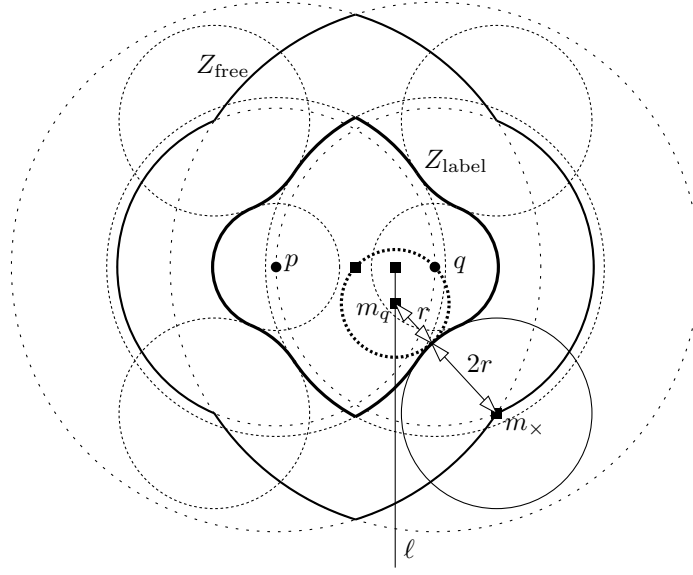


Figure 4.9: The point-free zone Z_{free} and the label zone Z_{label} of p and q for the case $d > D_3/2$. The bold dotted label touches the circle centered at m_x .

radius $2r$ centered at the intersections of C_{p,D_3} with $C_{q,d}$ and at the intersections of C_{q,D_3} with $C_{p,d}$. One of these last four circles is $C_{m_x,2r}$ whose center m_x lies at an intersection of C_{p,D_3} and $C_{q,d}$, see Figure 4.9.

It turns out that a label with maximum radius inside the label zone always touches either C_{p,D_3-2r} or $C_{m_x,2r}$. This depends on the value of d . There is a real $d^* = \frac{\sqrt{1+\sqrt{33}}}{2\sqrt{6}} \cdot D_3 \approx 0.53 D_3$ such that for $d \leq d^*$ the label touches C_{p,D_3-2r} and for $d \geq d^*$ it touches $C_{m_x,2r}$.

The values of r for which the label touches C_{q,D_3-2r} have already been computed, see Equation (4.3). The values, for which the label touches $C_{m_x,2r}$, are computed similarly as follows. The line connecting the center points m_q and m_x intersects the touching point of the two circles. This gives rise to the equation

$$d(m_q, m_x) = 3r, \quad (4.4)$$

see Figure 4.9. If we put the origin of our coordinate system at q and let the negative x -axis contain p , then we get

$$m_x = \left(\frac{1 - 2\hat{d}^2}{2\hat{d}} D_3, -\frac{\sqrt{4 - \hat{d}^{-2}}}{2} D_3 \right), \quad m_q = \left(-\frac{1}{4}\hat{d} \cdot D_3, -\sqrt{\hat{r}^2 - \frac{\hat{d}^2}{16}} \cdot D_3 \right).$$

The equation $d(m_q, m_x) = 3r$ has the following solution for our problem:

$$\hat{r} = \frac{\sqrt{8 - \hat{d}^{-2} + 8\hat{d}^2 - \frac{\sqrt{1-16\hat{d}^2+48\hat{d}^4}}{\hat{d}^2}}}{8\sqrt{2}}. \quad (4.5)$$

Graph g in Figure 4.8 depicts Equation (4.5) as a function of \hat{d} . The graph of $r(d)$ equals $f(d)$ for $d < d^*$ and $g(d)$ otherwise. The minimum r_{\min} of $r(d)$ is reached at the minimum of g since f decreases monotonically for $d \leq d^*$ and g has a negative derivative at $d = d^*$. A numeric computation shows that g has its minimum value when $d \approx 0.56085 D_3$. The corresponding minimum value of g and thus of $r(d)$ is $r_{\min} \approx 0.190526 D_3$.

Theorem 4.7 *Our algorithm labels a finite point set S with circles of diameter $d_{\text{algo}} = 0.381 D_3(S)$. The approximation factor γ is $0.381/(2(2 + \sqrt{3})) \approx 1/19.59$.*

Proof. When we label the points with circles of diameter $2r_{\min}$, we know that the label space of any pair of closest neighbors will be contained in its label zone. Then Lemma 4.5 ensures that for a label diameter $d_{\text{algo}} = 0.381 D_3 < 2r_{\min}$, our algorithm will label all points with non-overlapping labels. The approximation factor is the ratio of the upper bound $2(2 + \sqrt{3})D_3$ (see Figure 4.2) and the diameter d_{algo} of the labels we place. \square

It is clear that any set of congruent equilateral triangles will force our algorithm to produce a labeling with circles of diameter $\gamma \cdot d_{\text{opt}}$ if the triangles are spaced appropriately. However, there are also examples with a smaller optimum labeling where the algorithm performs better. The triangular lattice formed by the centers of a densest disk packing, for example, has an optimal labeling with circles of diameter $d_{\text{opt}} = D_3$. Here our algorithm yields a ratio of $d_{\text{algo}}/d_{\text{opt}} = 0.381$.

4.2.5 NP-Hardness

In this section we show that deciding whether a set of points can be labeled with unit circles is NP-hard. This answers an open question raised by Doddi et al. [DMM⁺97]. Our proof also implies that there is a constant $\delta < 1$ such that it is NP-hard to label points with circles of diameter greater than $\delta \cdot d_{\text{opt}}$. Consequently no polynomial-time approximation scheme exists. The proof is by reduction from *planar* 3-SAT. For a Boolean formula of planar 3-SAT type the variable-clause graph is planar. In this graph, the nodes are the variables and clauses of the formula, and there is an edge between a variable node and a clause node if the variable occurs in the clause. The planarity of the variable-clause graph helps to simplify the proof. The same idea is used in Knuth and Raghunathan's proof of the NP-hardness of the Metafont-labeling problem [KR92].

In the Metafont-labeling problem and other label-placement problems studied previously,[FPT81, FW91, MS91] every label can only be placed in a constant number of positions. In our case, there are infinitely many ways to label a point with a circle. This relaxation could potentially make circle labeling polynomially solvable (even given $\mathcal{P} \neq \mathcal{NP}$) and thus simpler than the discrete label-placement problems studied before, just as real-valued linear programming is simpler than zero-one linear programming. Of course the previous

NP-hardness proofs can be modified to allow a certain continuum of feasible label position in the vicinity of the original discrete positions. In [MS91] this was achieved by adding dummy points that do not receive any label; [IL97] uses a similar strategy.

In our reduction, we do not use dummy points, but restrict the infinite number of potential label positions of all points to at most three by using *immobilizing clusters*. These are special gadgets that consist of three points that must be labeled in a *unique* way. This approach is also different from the other two NP-hardness proofs for label-placement problems with an infinite number of label positions per point [IL97, vKSW99]. We take advantage of the special geometry of circles.

Another major difference to all other label-placement problems we know of is the fact that it is not clear whether circle labeling is in \mathcal{NP} . We do not know whether there is always a polynomial encoding of a solution, even if the input points have rational coordinates.

Theorem 4.8 *It is NP-hard to decide whether a set of points can be labeled with unit circles.*

Proof. We encode the variables and clauses of a Boolean formula ϕ of planar 3-SAT type by a set of points such that all points can be labeled if and only if ϕ is satisfiable, i.e. if there is a variable assignment such that all clauses evaluate to true. Since Lichtenstein showed that planar 3-SAT is NP-hard [Lic82], it follows that circle labeling is NP-hard as well. Note that the variables and clauses of a planar 3-SAT formula can be embedded in the plane as in Figure 4.10 where all variables lie on a horizontal line and all clauses are represented by *non-intersecting* three-legged combs [Lic82].

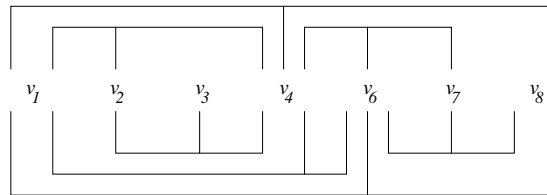


Figure 4.10: Embedding of a planar 3-SAT formula.

The main observation leading to our proof is the following. Given three equidistant points on a line, there are exactly two ways to label these points optimally, see Figure 4.11. Since all labels have diameter 1 here, some basic geometry shows that this distance must be $(1 - \sqrt{2\sqrt{3} - 3})/2 \approx 0.159$. This observation gives us a means to encode the Boolean values of a variable in the planar 3-SAT formula ϕ that we want to reduce to a set of points.

The gadgets of our reduction are the clusters for variables and three-legged combs for clauses. In order to be able to connect a variable v to all clauses in which it occurs, we model v not by one but by several *variable clusters* on a

horizontal line h as in Figure 3.58. Note that the cluster-to-cluster distance of $1 + \sqrt{2\sqrt{3}} - 3$ (from midpoint to midpoint) is chosen such that every second cluster must be labeled the same way. The value of v is represented by the label positions of the leftmost cluster on h (according to Figure 4.11). We call this cluster and every second cluster to its right *odd*. Accordingly, all other clusters are *even*. Then the label positions of all odd clusters encode the value of v and all even clusters that of $\neg v$. This differentiation is important for connecting v to the clauses in which it occurs. Each connection depends on whether v is negated in that clause or not.

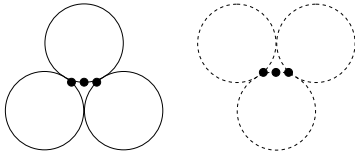


Figure 4.11: A variable cluster and the label placements encoding *true* and *false*.

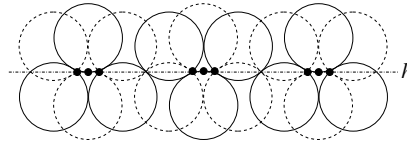


Figure 4.12: Rows of variable clusters model a variable; the label positions of the leftmost cluster determine the variable's Boolean value.

The central idea for modeling the clauses is that we restrict the possible label positions of all points (except one) to a maximum of two. To achieve this, we use *immobilizing clusters* that can only be labeled in one specific way. They consist of three points at pairwise distance $1/(4 + 2\sqrt{3})$, see Figure 4.2. In order to distinguish these auxiliary points from the others, we use the term *active points* for all clause points with at least two possible label positions.

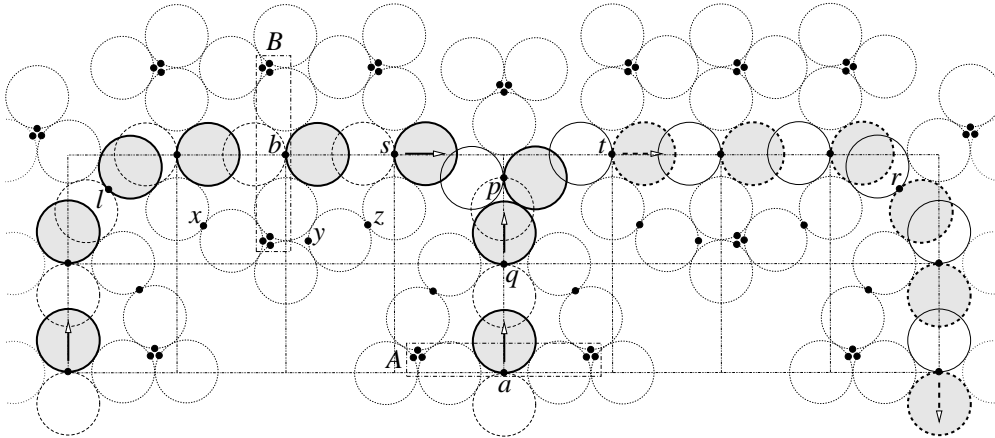


Figure 4.13: Clause with pressure from two variables. The current label positions are marked by shaded labels, alternatives are indicated by solid or dashed circles, and immobile labels are dotted.

We model the clauses by point sets that resemble large combs, see Figure 4.13. Such a comb consists of a horizontal part and three legs. The horizontal part is formed by active points like b in Figure 4.13 and by immobilizing

clusters above and below b that restrict the label of b to two possible positions. All points of type b lie on a horizontal.

The legs consist of points like a in Figures 4.13 and 4.14. These points lie on a vertical line and are also forced into one of two possible positions. Where the legs are joined to the horizontal part of a comb, lack of space does not allow us to use the immobilizing clusters as elsewhere. Instead, we simply attach points like x , y , and z in Figure 4.13 to cluster labels in the vicinity such that the labels of x , y , and z are also immobile and at the same time restrict the label positions of active points (like l , r , s , t , and q) as desired.

Both the horizontal part and the legs of a comb can be extended as far as needed to reach the three variables belonging to the clause. This is done by repeating—at a distance of $\sqrt{3}$ —patterns of seven points like those contained in the boxes A and B in Figure 4.13.

Each leg is connected to the encoding of a variable v . Let g be the vertical on which all points of the leg lie. It is perpendicular to the line h that contains all points encoding v . The lines g and h intersect in the midpoint d of one of the variable clusters on h , see Figure 4.14. The distance of $\sqrt{3}$ between the lowest leg point a and d is chosen to assure that the label of a can only intersect the label of d among the points modeling v . Note that the labels of a and d only intersect if the label of a is placed right below a and that of d right above d .

If variable v is negated in the clause under consideration, we join the leg to an odd cluster of v . Thus the cluster with d is labeled the same way as the leftmost cluster of v , see Figure 4.14. Now if v is set to true, d is labeled upwards. Then a and all other points on line g must also be labeled upwards. To put it graphically, pressure is transmitted upwards. If v is set to false, d can be labeled downwards, and no pressure is transmitted.

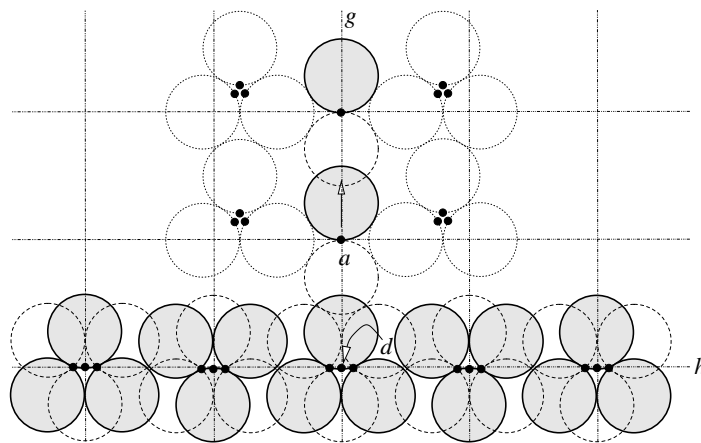


Figure 4.14: We connect the leg of a clause to a variable above the midpoint d of a variable cluster. If the variable is negated in the clause, then we join the leg to an odd cluster (as in this case), otherwise to an even cluster.

On the other hand, if v is not negated in the clause under consideration, then we join the clause's leg to an even cluster of v , which is labeled the opposite way as the leftmost cluster. Then pressure is transmitted if and only if v is set to false.

If all literals of a clause evaluate to false, then pressure is transmitted through all three legs into the clause. In this case there is a point (like p) that cannot be labeled, see Figure 4.13. In case there is at least one leg without pressure, it is obvious that all points belonging to a clause can be labeled. Hence the question whether ϕ can be satisfied is equivalent to asking whether all points in the set resulting from the encoding of ϕ can be labeled with unit circles.

To show that we can in fact connect a variable to several different clauses (below and above), we use a grid of width $\sqrt{3}$ and move all active points to grid points—except points of type l , r , and p , see Figure 4.13. In order to accommodate also the midpoints of the variable clusters on the grid (see Figure 4.14), we slightly increase the distance of neighboring variable clusters to that of immobilizing clusters, i.e. from $1 + \sqrt{2\sqrt{3} - 3} \approx 1.68$ to $\sqrt{3} \approx 1.73$. Then the label positions in every second cluster are still combinatorically equivalent, although labels can slightly wiggle now. Given such a grid embedding of our instance, it is clear that we can connect all variables to clauses according to ϕ .

We still have to ensure that the reduction is polynomial: if ϕ consists of m clauses and $n \leq 3m$ variables, the instance has $O(m^2)$ points. Their position can be computed in polynomial time if we round the grid-cell size and the distance between the three points of the immobilizing clusters to slightly greater rational numbers. The resulting instance is combinatorically equivalent to the one described before. \square

Our gadget proof of the NP-hardness of circle labeling also shows that we cannot expect to approximate this problem arbitrarily well. Formann and Wagner used a similar argument to show that maximizing the size of axis-parallel square labels cannot be approximated beyond a factor of $1/2$ [FW91]. In the formulation of their problem they only allow a constant number of label positions per point (namely four), which makes it easier to determine a good bound for the approximability.

Corollary 4.9 *There is a constant $\delta < 1$ such that it is NP-hard to label points with uniform circles of diameter greater than $\delta \cdot d_{\text{opt}}$.*

Proof. The proof of Theorem 4.8 still holds if the diameter of all labels is slightly reduced to a $\delta < 1$. The reason for this is that though labels have a certain degree of freedom now, every new label position is combinatorically equivalent to exactly one former position. δ must be chosen close enough to 1 to prevent a label from being moved continuously from one old position to another.

If there was a polynomial-time algorithm that could label the point set of the reduction with labels of size δ , we could solve planar 3-SAT in polynomial

time and would thus have $\mathcal{P} = \mathcal{N}\mathcal{P}$. ◻

The bottleneck that determines the minimum value of δ seems to be the encoding of a variable, see Figure 4.12. When the labels (and thus δ) are scaled down gradually, there is a point when two neighboring clusters can have identical instead of alternating label positions, see Figure 4.11. Then the variable's Boolean value is no longer well defined, and the proof collapses.

Chapter 5

Line Labeling

This chapter is joint work with Lars Knipping, Freie Universität Berlin, Marc van Kreveld, Tycho Strijk, both Universiteit Utrecht, and Pankaj K. Agarwal, Duke University [WKvK⁺99].

The interest in algorithms that automatically place labels on maps, graphs, or diagrams has increased with the advance in type-setting technology and the amount of information to be visualized. However, though manually labeling a map is estimated to take fifty percent of total map production time [Mor80], most geographic information systems (GIS) offer only very basic label-placement features. In practice, a GIS user is still forced to invest several hours in order to eliminate manually all label-label and label-feature intersections on a map.

In this chapter, we suggest an algorithm that labels one of the three classes of map objects, namely polygonal chains, such as rivers or streets. Our method is simple and efficient. At the same time, it produces results of high aesthetical quality. It is the first that fulfills both of the following two requirements: it allows curved labels and runs in $O(n^2)$ time, where n is the number of points of the polyline.

In order to formalize what good line labeling means, we studied Imhof's rules for positioning names on maps [Imh62, Imh75]. His well-established catalogue of label placement rules also provides a set of guidelines that refers to labeling linear objects. (For a general evaluation of quality for label-placement methods, see [vDvKSW99].) Imhof's rules can be put into two categories, namely *hard* and *soft* constraints. Hard constraints represent minimum requirements for decent labeling:

- (H1) A label must be placed at least at some distance ϵ from the polyline.
- (H2) The curvature of the curve along which the label is placed is bounded from above by the curvature of a circle with radius r .
- (H3) The label must neither intersect itself nor the polyline.

Soft constraints on the other hand help to express preferences between acceptable label positions. They formalize aesthetic criteria and help to improve the visual association between line and label. A label should

- (S1) be close to the polyline,
- (S2) have few inflection points,
- (S3) be placed as straight as possible, and
- (S4) be placed as horizontally as possible.

We propose an algorithm that produces a candidate strip along the input polyline. This strip has the same height as the given label, consists of rectangular and annular segments, and fulfills the hard constraints. In order to optimize soft constraints, we use one or a combination of several evaluation functions.

The candidate strip can be regarded as a simplification of the input polyline. The algorithm for computing the strip is similar to the Douglas-Peucker line-simplification algorithm [DP73] in that it refines the initial solution recursively. However, in contrast to a simplified line, the strip is never allowed to intersect the given polyline. The strip-generating algorithm has a runtime of $O(n^2)$, where n is the number of points on the polyline. The algorithm requires linear storage.

Given a strip and the length of a label, we propose three evaluation functions for selecting good label candidates within the strip. These functions optimize the first three soft constraints. Their implementation is described in detail in [Kni98]. We can compute in linear time a placement of the label within the strip so that the *curvature* or the *number of inflections* of the label is minimized. Since it is desirable to keep the label as close to the polyline as possible (while keeping a minimum distance) we also investigated the *directed label-polyline Hausdorff distance*. This distance is given by the distance of two points; a) the point p on the label that is furthest away from the polyline and b) the point p' on the polyline that is closest to p . Under certain conditions we can find a label position that minimizes this distance in $O(n \log n)$ time [Kni98]. Here we give a simple algorithm that finds a near-optimal label placement according to this criterion in $O(nk + k \log k)$ time, where k is the ratio of the length of the strip and the maximum allowed discrepancy to the exact minimum Hausdorff distance.

If a whole map is to be labeled, we can also generate a set of near-optimal label candidates for each polyline, and use them as input to general map-labeling algorithms as [ECMS97, KT98, WW98]. Some of these algorithms accept a priority for each candidate; in our case we could use the result of the evaluation function.

In his list of guidelines for good line labeling, Imhof also recommends to label a polyline at regular intervals, especially between junctions with other polylines of the same width and color. River names e.g. tend to change below

the mouths of large tributaries. This problem can be handled by extending our algorithms as follows. We compute our strip and generate a set of the, say ten best label candidates for each river segment that is limited by tributaries of equal type. Then we can view each river segment as a separate feature, and again use a general map-labeling algorithm to label as many segments as possible. Prioritizing each label candidate with its distance to the closer end of the river segment would give candidates in the middle of a segment a higher priority and thus tend to increase label-label distances along the polyline.

This chapter is structured as follows. In the next section we briefly review previous work on line labeling. In Section 5.2 we explain how to compute a buffer around the input polyline that protects the strip from getting too close to the polyline and from sharp bends at convex vertices. In Section 5.3 we give the algorithm that computes the strip and in Section 5.4 we show how this strip can be used to find good label candidates for the polyline. Finally, in Section 6.3 we describe our experiments. Our implementation of the strip generator for x -monotonous polylines and the three evaluation functions can be tested on-line¹.

5.1 Previous Work

In the label-placement literature the problem of automated line labeling has been treated before. In [Coo88, DF92, BL95, AH95, ECMS97, Kra97] only rectangular labels are allowed; curved labels are not considered. In [Fre88] a set of label-placement rules similar to those of Imhof [Imh75] is listed, followed by a rough description of an algorithm. An analysis of Figure 8 in [Fre88] shows that river names are broken into shorter pieces that are then placed parallel to segments of the river. Each piece ends before it would run into the river or end too far from the current river segment.

In [Bar97] curved labels are taken into account. First, the input polyline is split into sections depending on its length and junctions (forks) with other polylines. For details of this step, see [BL95]. Then the polyline is treated with an adaptation of an operator from morphological mathematics, closure, that is a mixture of an erosion and a dilation. This operator yields a baseline for label candidates where the polyline does not bend too abruptly. It is not clear how this is done algorithmically; no asymptotic runtime bounds are given. Finally, simulated annealing is used in order to find a good global label placement, i.e. a placement that maximizes the number of features that receive a label and at the same time takes into account the cartographic quality of each label position.

In [PZC98, SvK99] a more theoretical problem is analyzed; an instance of axis-parallel line segments is labeled with rectangular labels of common height. While the length of each label equals that of the corresponding line segment, the label height is to be maximized.

While the restriction to rectangular labels is acceptable for technical maps

¹<http://www.math-inf.uni-greifswald.de/map-labeling/lines>

or road maps (where roads must be labeled with road numbers), we feel that curved labels are a necessity for high-quality line labeling. The method we suggest is the first that fulfills both of the following two requirements: it allows curved labels and its runtime is in $O(n^2)$. The runtime thus only depends on the number of points of the polyline, and not on other parameters such as the resolution of the output device. Note that the time bound holds even if the approximate Hausdorff distance is used to select good label candidates within the strip as long as we choose the parameter k linear in n .

5.2 A Buffer Around the Input Polyline

In order to reduce the search space for good label candidates, we generate a strip along the input polyline that is (a) likely to contain good label positions and (b) easy to compute. Generating our strip consists of two major tasks. First, we compute a buffer around the polyline that our strip must not intersect. Second, we generate an initial strip and refine it recursively. Each refinement step brings the strip closer to the polyline, but also introduces additional inflections.

The input to our algorithm consists of a polyline $P = (p_1, \dots, p_n)$ with points $p_i = (x_i, y_i)$, a minimum label-polyline distance ε , a maximum curvature $1/r$, and a label height h . It makes sense to choose $r \gg \varepsilon$ but the algorithm does not depend on this. We assume that P is x -monotonous, i.e. $x_1 < \dots < x_n$. Non-monotonous polylines can be split up into monotonous pieces of maximum length in linear time by a simple greedy algorithm. That algorithm goes sequentially through the edges of the polyline. Whenever adding the current edge to the current piece would make that piece non-monotonous, a new piece is started with the current edge.

For ease of presentation we direct P from p_1 to p_n and only label the upper (i.e. left) side of the polyline. We use r -disk (r -arc) as shorthand for a disk (arc) of radius r . We say that p_i is *at a right turn of P* if p_{i+1} lies to the right of the directed line through p_{i-1} and p_i , see p_3 or p_4 in Figure 5.1.

We define the ε - r -buffer $B(P)$ in two steps. First let the ε -buffer be the union of all ε -disks whose center lies on P , see the light-shaded area in Figure 5.1. Second we add certain pieces of r -disks D_i placed at right turns p_i of P . Their task is to bound the curvature of our strip. The center m_i of D_i is placed on the angular bisector b_i of the adjacent edges of P such that D_i touches and contains the ε -disk centered at p_i , see Figure 5.2. Let \overline{D}_i be the part of D_i that is left of the ε -buffer and touches the ε -disk, see the dark-shaded areas in Figure 5.1. Then $B(P)$ is the union of the ε -buffer and the \overline{D}_i for each right turn p_i .

To simplify the calculation of the strip, we also place r -disks D_1 and D_n at the endpoints p_1 and p_n of P , respectively. Let b_n be the normal to the edge $p_{n-1}p_n$ in p_n . Then the center of D_n lies on b_n such that D_n touches and contains the ε -disk centered at p_n , see D_n in Figure 5.2. The placement of D_1 is analogous.

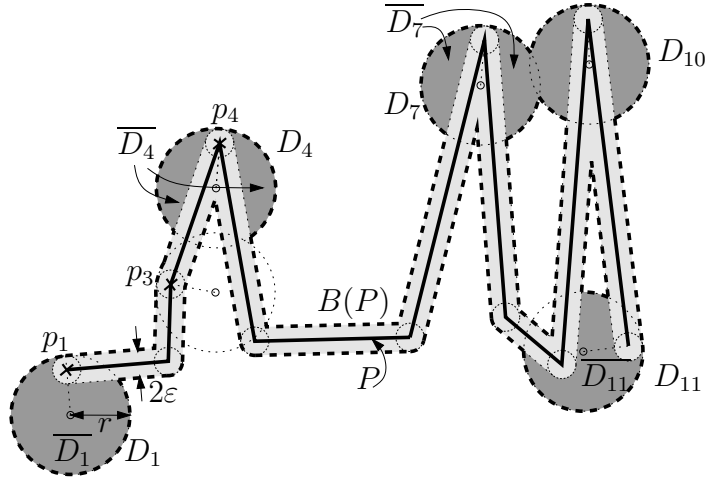


Figure 5.1: The boundary of the ε - r -buffer $B(P)$ (bold dashed line) of the input polyline P (bold solid line).

In order to compute the boundary of the ε - r -buffer we first compute that of the ε -buffer. This is simple since the x -monotonicity of P guarantees that the ε -buffer does not have any holes.

For computing the candidate strip it is important that we have access to the elements of the outer face of the ε - r -buffer in the order in which they occur. We compute the ε - r -buffer in two phases.

In the first phase, for each right turn p_i we follow the boundary of the ε -buffer from t_i to the right until we intersect the boundary of D_i for the first time. This intersection point is denoted by r_i , see Figure 5.2. The arc from t_i to r_i , oriented clockwise, is the *right arc* R_i , one of the two parts of the boundary of D_i we are interested in. The *left arcs* L_i that go counterclockwise from t_i to l_i can be computed analogously. A special case arises if t_i lies in the interior of the ε -buffer. Then R_i or L_i is empty, and we have to follow P from p_i in both directions until we arrive at a point or edge that corresponds to an arc or line segment on the upper part of the ε -buffer. From there, we can continue as usual.

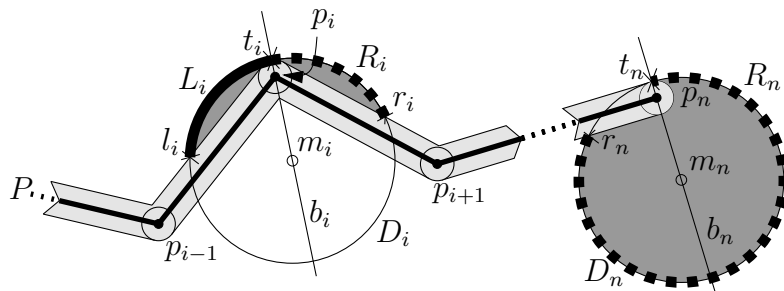


Figure 5.2: Placing r -disks D_i at right turns p_i of the input polyline P .

Clearly, this procedure has a worst-case runtime of $O(n^2)$. The worst case occurs if there are a linear number of right turns p_i where we have to walk over a linear number of segments of the ε -buffer until we hit l_i or r_i , i.e. if r is large compared to the length of the edges of P . However, in practice one can expect to walk only over a constant number of segments of the ε -buffer; then the running time is $\Theta(n)$, see Section 6.3. The worst-case running time can be improved using more sophisticated data structures, but we omit this improvement here as it makes the algorithm more complicated.

In the second phase, we incrementally extend the ε -buffer to the ε - r -buffer using the left and right arcs we just computed. We maintain $\mathcal{B}_{\text{curr}}$, the outer face of the union of the ε -buffer and the areas $\overline{D_i}$ we have processed so far. Initially let $\mathcal{B}_{\text{curr}}$ be the boundary of the ε -buffer and let the interior of $\mathcal{B}_{\text{curr}}$ be the interior of the ε -buffer. Let the r -arc A_i be the union of L_i and R_i . Note that A_i is the part of the boundary of $\overline{D_i}$ that is a potential part of the outer face of the ε - r -buffer. For each right turn p_i we check whether A_i lies completely in the interior of $\mathcal{B}_{\text{curr}}$. If this is not the case we extend $\mathcal{B}_{\text{curr}}$ by using the appropriate parts of A_i .

The boundary of the ε -buffer consists of a linear number of line and arc segments to which we add $O(n)$ arcs of type A_i . One can prove that each of these arcs can contribute at most three pieces to the outer face of $B(P)$. Our implementation does not depend on this result, but it shows that the outer face of $B(P)$ has linear complexity. Due to the incremental construction this is also an upper bound for the size of $\mathcal{B}_{\text{curr}}$.

Given these observations it is easy to devise an $O(n^2)$ -algorithm that computes the boundary of the outer face of $B(P)$. We store $\mathcal{B}_{\text{curr}}$ in a doubly connected list. Since the length of this list is linear we can afford to scan the whole list when we search for intersections with the current arc A_i . If we consider carefully whether we enter or leave the interior of the area delimited by $\mathcal{B}_{\text{curr}}$, we can update $\mathcal{B}_{\text{curr}}$ in linear time for each right turn. We omit details here.

In our implementation of the second phase we use a similar trick as in the first phase to avoid a quadratic runtime in many cases. We exploit the fact that an arc A_i usually spans only a constant number of elements of $\mathcal{B}_{\text{curr}}$.

5.3 A Candidate Strip

Once we have the outer face of the ε - r -buffer, we compute the baseline of the label candidate strip and refine it recursively. We refer to the line and arc segments that delimit the buffer on the upper side between l_1 and r_n as baseline *objects*. We have access to these objects in the order in which they appear on the boundary of the buffer's outer face. We start with an arc A that touches the first and last object O_i and O_k , respectively. We bend A towards the buffer until it hits a third object O_j . There, we split A into two pieces, its children. We connect the children of A with a piece of O_j that initially has

length zero. Then we recursively bend the children further towards the buffer, see Figure 5.4. While we bend, the portion of O_j that connects the children of A is growing. Note that there are two phases: in the first, the radius of the arcs increases while it decreases in the second. The recursion ends where O_i and O_k are adjacent on the buffer (since there is no O_j then) and in the second phase where the curvature of an arc would exceed $1/(r+h)$, h the label height. For the pseudo-code of this algorithm, refer to Figure 5.3.

```

REFINE( $B, i, k, state, G$ )
if  $k = i + 1$  then return
 $\vec{b}_{ik}$  := oriented bisector of  $O_i$  and  $O_k$ :  $\mathbb{R} \rightarrow \mathbb{R}^2$ 
for  $j := i + 1$  to  $k - 1$  do
     $A_j := \text{touching\_arc}(O_i, O_j, O_k, state)$ 
    if  $A_j \neq \emptyset$ 
        then choose  $\beta_j$  such that  $\vec{b}_{ik}(\beta_j) = \text{center}(A_j)$ 
        else  $\beta_j := \infty$ 
    end
end
if  $\min\{\beta_{i+1}, \dots, \beta_{k-1}\} = \infty$  then
    if  $state = 1$ 
        then REFINE( $B, i, k, 2, G$ )
        else return
    end
    choose  $j$  such that  $\beta_j$  minimal
    replace  $O_i - O_k$  in  $G$  by  $A_j$ 
    REFINE( $B, i, j, state, G$ )
    REFINE( $B, j, k, state, G$ )

```

Figure 5.3: pseudo-code for the baseline refinement algorithm

For each level of the recursion, the sequence of arcs we obtain in this way forms a continuous curve L . If we direct L from left to right, it becomes obvious that the radius of all arcs that turn right (i.e. towards the buffer) is at least r and the radius of arcs that turn left is at least $r+h$. By using L as the baseline of our strip of height h we ensure that all arcs that form the upper boundary and the baseline of the strip have at least radius r . Thus the strip fulfills the curvature constraint H2. Since the baseline of the strip cannot intersect the ε -buffer it is clear that the strip also fulfills the distance constraint H1. The non-self-intersection constraint H3 can easily be kept by ending the recursion where the distance between O_i and O_k is less than $2h$.

If the number of inflections is to be kept small, the recursion can also be stopped whenever the directed distance of a strip segment to the polyline is below a given threshold. However this is difficult to check without the Voronoi diagram of the points and (open) edges of P .

It is possible to add two interesting refinement levels. In both, an arc of

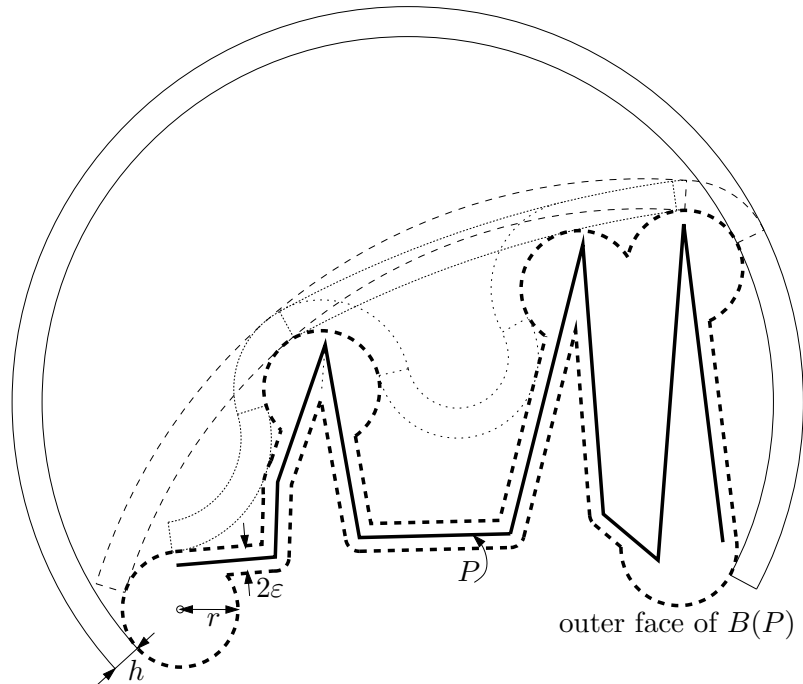


Figure 5.4: refining the candidate strip: first level (solid), second level (dashed), third level (densely dotted), and fourth level (dotted)

the baseline does not necessarily touch three objects on the boundary of the buffer's outer face. For a strip with more rectangular segments one could add a refinement level between level 2 and 3 of the leftmost strip segment in Figure 5.4. Note that the radii of the annular strip segments there increase up to level 2 and then decrease again. Rectangular segments in an additional refinement level can thus be viewed as annular segments with infinite radius. On the other hand, to make the strip follow P as closely as possible, a final refinement level could be added where *all* annular strip segments are delimited by two arcs with radius r and $r + h$. The baseline of this strip is part of the curve on which a disk of radius $r + h$ is rolled around the buffer if the disk must always touch the buffer but not intersect its interior.

In order to determine the third object on an arc, we test each object between the left- and rightmost object in constant time. Thus we need linear time for each level of the recursion. As with the Douglas-Peucker line-simplification algorithm, the number of recursion levels depends on the distribution of the input data and can vary from $\Omega(\log n)$ to $O(n)$. Given the outer face of the ε - r -buffer the strip can hence be computed in $O(n^2)$ time, while the average case can be expected to be in $O(n \log n)$.

5.4 Finding Good Label Positions

In order to satisfy the soft constraints, we evaluate label candidates within the strip according to curvature, number of inflections, or directed label-polyline Hausdorff distance. (We define the curvature of a label as the sum over curvature times length of each label segment. The curvature of a rectangular segment is 0; that of an annular segment with arcs of radius r_1 and $r_2 = r_1 + h$ is $1/r_1$.) For all three evaluation functions, the basic idea is the same. We discretize the space of label candidates such that the discrete space has linear size and contains minima. Then we search the discrete space for a minimum.

For curvature and number of inflections it is easy to see that there is a minimizing label candidate that starts or ends with one of the rectangular or annular segments of the strip. In order to find a minimum, we push a label of the given length through the strip and stop whenever a new segment starts (or ends). To compute the measure of the current candidate, we only have to do a constant number of updates given the value at the previous position. This is how we can find a placement minimizing curvature or number of inflections in linear time.

For Hausdorff distance, the discretization is more difficult. We only take into account the baseline of the strip. In order to compute efficiently the distance between the baseline of a label candidate and the polyline P , we need to know the closest object (point or edge) of P for every point on the whole baseline. Intersecting the baseline with the Voronoi diagram of the objects of P would yield this information and lead to an $O(n \log n)$ algorithm under certain conditions [Kni98].

However, computing the Voronoi diagram for a set of points and line segments is not a trivial task in practice. Therefore we implemented a simpler algorithm that finds a near-optimal label placement as follows. Given an integer k , we split the baseline into k pieces of equal length. Let γ be the length of such a piece. We approximate the distance between each piece and P by the distance of the piece's midpoint from P . This can be done by brute force in $O(nk)$ time with $O(k)$ storage. Then we proceed as above: we push the label through the strip, stop at each midpoint and evaluate the current label position. Its Hausdorff distance to P is within γ from the maximum over the distances of all baseline pieces covered by the label. For fast access to this approximate maximum, we keep the appropriate distances in a priority queue. During the execution of the algorithm, we must insert the distance of each piece at most once into the queue. The same holds for deletions. Each such operation costs $O(\log k)$ time, hence we can compute an optimal placement among all those starting at a midpoint of a baseline piece in $O(nk + k \log k)$ time with $O(k)$ storage. The triangle inequality guarantees that this placement is at most γ further away from P than a placement minimizing the exact directed Hausdorff distance. A detailed description of the implementation of the above evaluation functions can be found in [Kni98] (in German).

5.5 Experimental Results

In order to analyze our line-labeling algorithm, we applied it to synthetic and to real-world data. The latter is taken from the CIA-map data collection², see Figures 5.5 and 5.6. In both figures, labels were placed according to the approximated minimum Hausdorff distance.

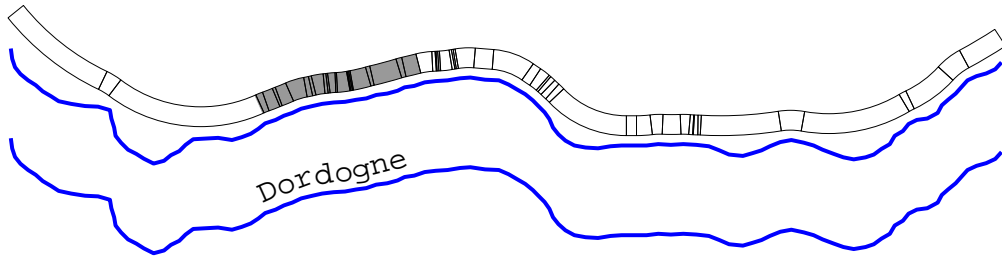


Figure 5.5: A piece of the Dordogne (109 points). Above with candidate strip and label placement (shaded grey), below with lettering

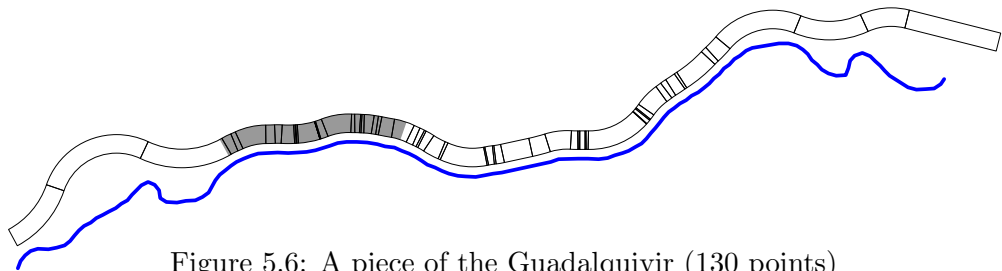


Figure 5.6: A piece of the Guadalquivir (130 points)

The synthetic data belongs to three different example classes. For all three classes we use random numbers Δx_i and Δy_i that we draw from a normal distribution with mean 0 and standard deviation 1. In order to get an x -monotonous polyline we choose the x -coordinates as follows: $x_1 = 0$ and $x_i = x_{i-1} + |\Delta x_i|$. Then we scale all x_i by x_n such that $0 = x_1 < x_2 < \dots < x_n = 1$. The choice of the y -coordinates depends on the example class.

For **RandomWalk** we set $y_1 = 0$ and $y_i = y_{i-1} + \Delta y_i/100$, i.e. we use the same scheme as for the abscissae except we do not take the absolute value of the random number and we scale it with a constant factor.

For **NoiseLine** and **NoiseSine** we use the x -axis and sine as base functions and add some noise: $y_i = f(x_i) + \Delta y_i/100$, where $f(x) = 0$ for NoiseLine and $f(x) = \sin(11\pi x)$ for NoiseSine.

Figures 5.7 to 5.9 depict instances of each of the three example classes. In each figure, the strip of the last refinement level (not counting the second additional refinement level mentioned in Section 5.3) is depicted three times for the same input polyline. The grey regions in the three strips indicate an optimal

²<ftp://gatekeeper.dec.com/pub/graphics/data/cia-wdb/db.tar.Z>

Synthetic Example Classes

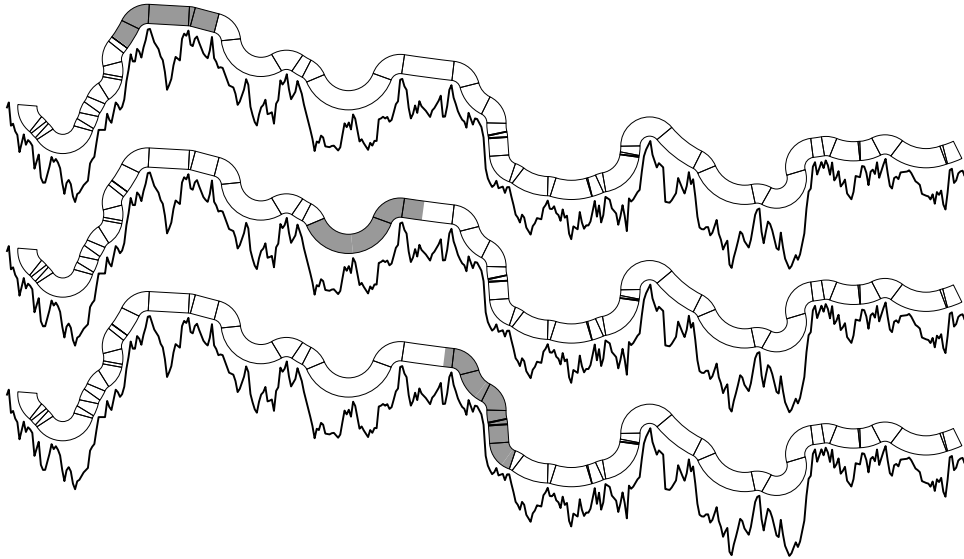


Figure 5.7: RandomWalk with 400 points

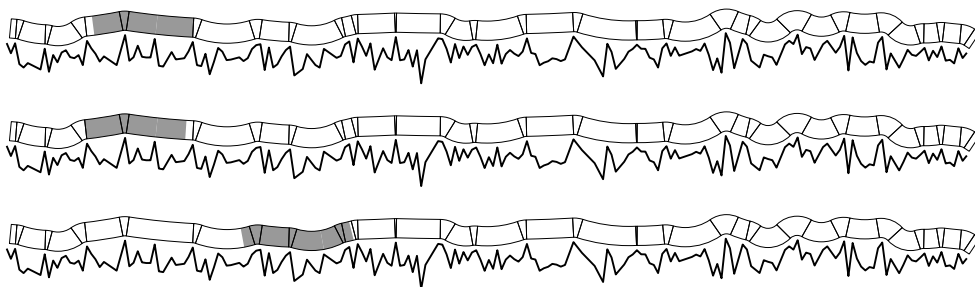


Figure 5.8: NoiseLine with 200 points.

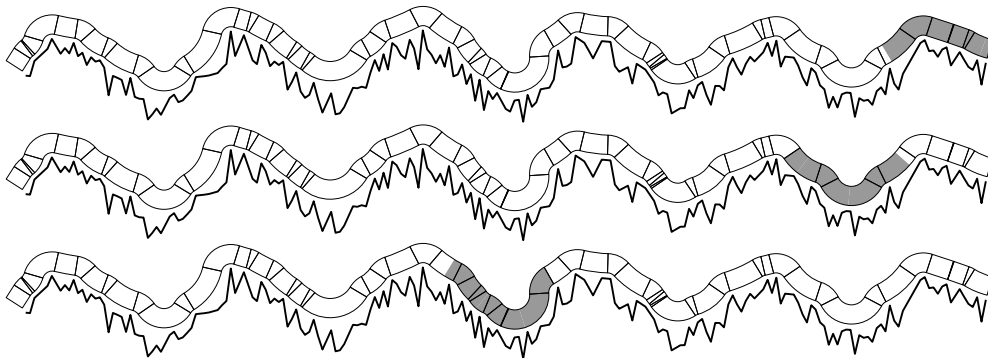


Figure 5.9: NoiseSine with 200 points

label placement within the strip minimizing curvature, number of inflections, and approximative Hausdorff distance (top to bottom). The parameters for the strip computation were minimum label-polyline distance $\varepsilon = 0.005$, curvature bound $r = 0.01$, and label height $h = 0.02$. More examples can be found in [Kni98] or generated on our Web page.

We generated 50 examples with 100, 200, . . . , 1000 points and measured the frequency of some basic operations and the runtime of our C++ implementation, see Figures 5.10 to 5.17. The two additional refinement levels mentioned in Section 5.3 were included in all experiments. In all figures, the x -axis gives the number n of points of the polyline. The points on our graphs give the results averaged over all 50 examples; the extent of the vertical bars indicates the minimum and maximum value among these 50 examples.

Runtime. In Figure 5.10 to 5.12 we depict the running times of the ε -buffer, ε - r -buffer and strip generation for our three example classes. Here the parameters were $r = 8/n$, $\varepsilon = 2/n$, and $h = 10/n$. The y -axis gives the average CPU time (in seconds) on a Sun Ultra-Sparc 250. We used the SunPRO-CC compiler with optimizer flags `-fast -O3`. Note that the three curves in each figure are additive; i.e. the topmost curve corresponds to the total runtime. RandomWalk takes twice as long as the other two example classes, which behave very similarly—as in all following graphs.

In Figure 5.13 we give the runtimes for placing labels within the pre-computed strip according to curvature and approximated Hausdorff distance. We used a label length of $50/n$, and for minimizing the Hausdorff distance we set the approximation parameter γ to $1/(2n)$. We omitted the curves for number of inflections since they were identical to those of curvature; we also omitted those for NoiseSine, which were very similar to those of NoiseLine. Other than in the description in Section 5.4 we used lists instead of priority queues for the approximated Hausdorff distance, hence the quadratic runtime behaviour.

Operation counters. In Figure 5.14 to 5.17 we measured the frequency of some basic operations in order to further analyze our implementation on the three example classes. Figures 5.14 and 5.15 refer to the buffer computation, Figures 5.16 and 5.17 to the strip generation.

In Figure 5.14 we show how many segments of the ε -buffer we visit when computing the extend of the r -arcs A_i . Figure 5.15 shows how many segments of the current outer face of the buffer are visited in $\mathcal{B}_{\text{curr}}$ when computing the outer face of the ε - r -buffer. Both figures show graphs with approximately linear growth as suggested in Section 5.2.

The graphs in Figure 5.16 count the number of tests we do to find the third object O_j between two objects O_i and O_k on the outer face of the ε - r -buffer. The growth rate here seems to be between linear and quadratic. Finally Figure 5.17 gives the number of recursion levels, which grows very slowly.

Graphs for Runtime and Operation Counters

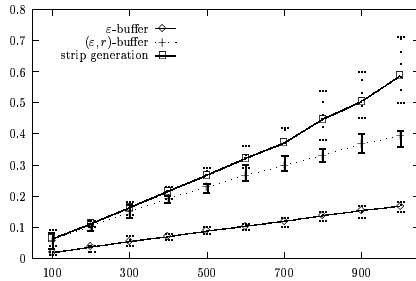


Figure 5.10: Strip generation time for RandomWalk

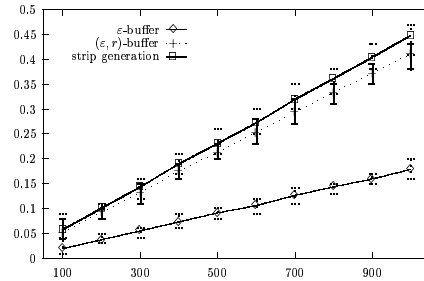


Figure 5.11: Strip generation time for NoiseLine

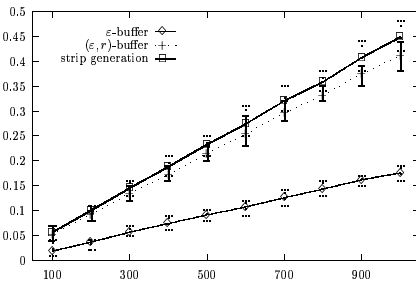


Figure 5.12: Strip generation time for NoiseSine

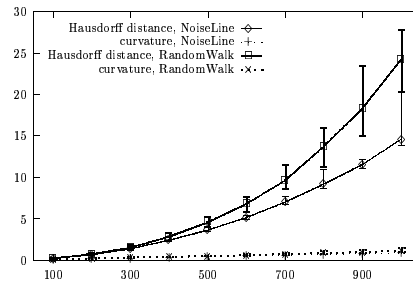


Figure 5.13: Running times for label placement

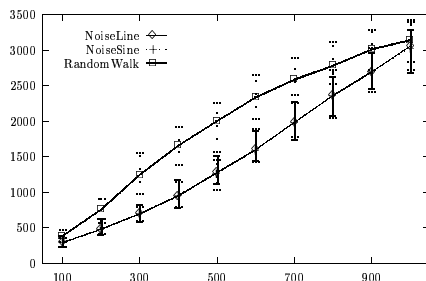


Figure 5.14: Number of Operations for computing r -arcs

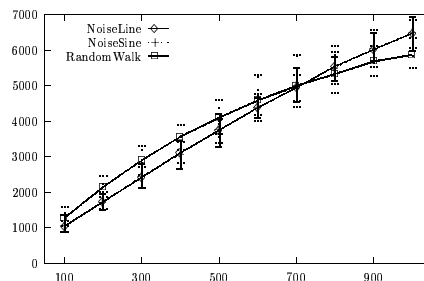


Figure 5.15: Number of Operations for placing r -arcs

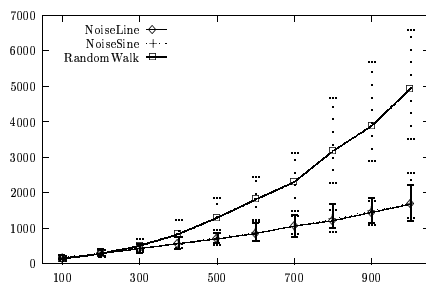


Figure 5.16: Number of Operations for Strip Placement

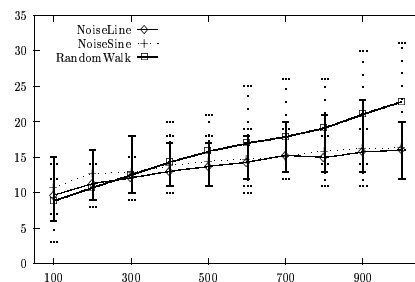


Figure 5.17: Number of refinement levels

5.6 Discussion and Extensions

We have presented a new and conceptually simple method for high-quality line labeling. It is the first that fulfills both of the following two requirements: it allows curved labels and its worst-case runtime is in $O(n^2)$. We introduced a concept of gradual refinement that is similar to the idea of the Douglas-Peucker line-simplification algorithm. This concept allows to introduce additional application-dependent criteria and to stop the refinement when these criteria are met.

An experimental evaluation of our algorithm shows that it usually runs in sub-quadratic time and generally yields good results in practice. However, since we reduce the search space for good label candidates to a one-dimensional strip, it is clear that we cannot hope to find an optimal label placement in every case. As the following example indicates, a more flexible strategy in the buffer construction might help to overcome problems caused by the reduction of the search space.

In Figure 5.18 we depicted all r -arcs at right turns of the input polyline P . The parameter r was chosen large compared to the average segment length of P . As a result, some of the arcs that contribute to the ε - r -buffer are quite distant from the input polyline P . They were caused by right turns incident to two very steep but short edges of P . It would be desirable to remove these arcs. However, we must ensure that the resulting strip does not violate the curvature constraint H2. This can be done as follows. After the first phase of the ε - r -buffer computation we compute the directed Hausdorff distance of each r -arc A_i to the ε -buffer between l_i and r_i . In order of descending distance we check for each A_i whether the corresponding ε -arc lies completely in the area $\overline{D_j}$ of another r -arc A_j . If this is the case, we remove A_i . Then we proceed to the second phase of the buffer computation as usual. Note that the resulting outer face of the buffer still consists exclusively of r -arcs and line segments. Thus the strip will still keep H2.

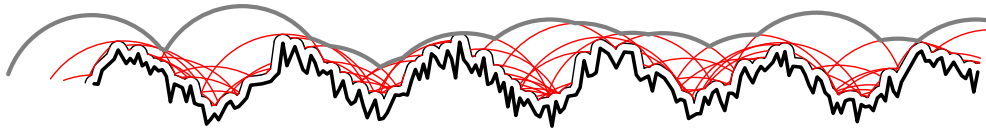


Figure 5.18: Disturbing effects of the definition of the ε - r -buffer. (The upper part of its outer face is marked by bold grey arcs; the input polyline below consists of bold black line segments.)

An alternative approach is as follows. We observed that our placement of the r -disks is good if the adjacent edges of the polyline are long enough. Then the directed Hausdorff distance between the arc A_i and the ε -buffer is minimized. However, in general the placement of the r -disks is too inflexible. It could certainly be improved if we tried to minimize the aforementioned distance during the placement. Then the placement of the r -disks would take into

account not only the adjacent edges of the polyline but all of the polyline (or the ε -buffer) between l_i and r_i .

Finally we would like to acknowledge a simple and elegant idea of Mike Lonergan, University of Glamorgan, Pontypridd. He suggested to put the ε -buffer around the label (and thus simply thicken the strip by 2ε) instead of the polyline. Unfortunately, this does not solve the problem of placing the r -circles.

Chapter 6

Designing Geometric Algorithms

This chapter is joint work with Vikas Kapoor and Dietmar Kühl.

The design phase of an implementation is of utmost importance for its efficiency, flexibility, and ease-of-use. While focusing on flexibility, we demonstrate that there is no reason for sacrificing ease-of-use. Furthermore, we show that a gain in flexibility does not necessarily cause a loss in efficiency.

In this chapter we present a design concept for geometric algorithms that is based on the *generic programming* paradigm. Generic programming is about making implementations more flexible by making them more general. Abstracting from concrete in- or output data representation is an example of generic programming. In contrast to normal programs, the parameters of generic programs are often quite rich in structure. Such parameters might be other programs, types or type constructors, or even other programming paradigms [BS98].

The generic programming paradigm has been so successful that a model—the *Standard Template Library* (STL) [MS96]—was created and added to C++, currently one of the most popular programming languages. The STL is a library of generic components, i.e. of algorithms, data containers, and *iterators* mediating between the former two. Iterators help to decouple algorithms from the type of data container they operate on. While iterators have been known before, the real novelty of the STL was the introduction of a requirements-based taxonomy of iterators, which gives a guideline for full decoupling, and an implementation of this taxonomy using the C++ template mechanism. By becoming part of the C++ standard, the STL has attracted considerable attention and has itself set a standard for good design.

After the introduction of the STL further concepts such as *data accessors* have been suggested in the C++ literature to help programmers make their implementations even more generic [Küh96, Wei97]. Data accessors are a means to further decouple an implementation from the representation of in- and output data [KW97]. So far, these extensions have been applied predominantly to graph problems [NW96]. Exceptions such as [Wei98, Ket98] deal with the rep-

resentation of geometric objects, not with the design of geometric *algorithms*, our main interest here. In order to show the relevance of STL-style generic programming including later extensions as data accessors for geometric algorithms, we investigate a simple rectangle-intersection algorithm that follows the well-known sweep-line paradigm.

Using this example we lead the reader step by step from an inflexible, naive interface to a truly flexible interface that supports code reuse. These steps reflect our own change of perspective during the implementation of our label-placement algorithms. Note that reusability helps to lower implementation costs in the long run and to achieve correctness—for the simple reason that more users mean more testing. The reader should be familiar with the programming language C++ that we chose to demonstrate our concept.

In order to support our concept with experimental data, we implemented a sweep-line algorithm for the rectangle intersection problem in C++ in two ways; (a) using straight-forward object orientation and (b) following our design concepts. We compare the runtime of both implementations, as well as the size of their source code and executables. The data we used for the runtime comparison stems from random generators as well as from real world instances. The example classes have been used for an experimental analysis of map-labeling algorithms and are described in detail in [WW98]. The source code of our implementations, the documentation of the interfaces, the test data and its description are accessible via the WWW¹.

While the ingredients of our concept have already been known, to our knowledge this is the first time that they are applied so rigorously to a geometric problem, that they are made accessible in the form of a tutorial and that they are accompanied by a thorough experimental analysis.

Other than the STL, the C++ Library of Efficient Data Types and Algorithms (LEDA) [NM90] was designed having mostly ease-of-use rather than genericity in mind. The resulting short-comings in terms of interfacing with user-defined data structures are investigated in [Küh96]. This has led to the implementation of a LEDA extension package for graph iterators [NW96].

A recent report about the design of CGAL, the Computational Geometry Algorithms Library [Ove96], also a C++ library, includes a section about the pros and cons of generic versus object-oriented programming [FGK⁺98].

This chapter is structured as follows. In Section 6.1, we describe our example algorithm for the rectangle-intersection problem. In Section 6.2, we present a naive interface for this algorithm, investigate its disadvantages and modify it step-by-step to a generic and thus flexible interface. Finally, in Section 6.3, we compare the implementations of the naive and the most flexible interface of the previous section.

¹see <http://www.math-inf.uni-greifswald.de/map-labeling/design/>

6.1 Algorithm

We demonstrate our design concepts at the example of a sweep-line algorithm for detecting all intersections among a set of axis-parallel rectangles in the plane [Ede80]. Our sweep line will be a vertical line sweeping the plane from left to right. As usual, the sweep line is supported by two data structures, the *event-point schedule* and the *sweep-line status*.

Event points are the x -values where the sweep line must stop because either its status changes or intersections have to be reported. In our case all event points are known before the sweep begins: they are the x -values of the left and right edges of the rectangles. Thus a sorted list suffices to implement our event-point schedule. An event point must be stored in such a way that it is clear whether it refers to a left or right edge of a rectangle.

The sweep-line status stores intervals corresponding to the intersections of the sweep line with the given rectangles. The endpoints of the intervals are the y -values of the lower and upper edges of the input rectangles. Initially the sweep-line status is empty. When a left edge of a rectangle is encountered during the sweep, the interval corresponding to the edge is inserted into the sweep-line status. A rectangle is reported if its interval is currently in the sweep-line status and intersects the new interval. When a right edge of a rectangle is encountered, the corresponding interval is deleted from the sweep-line status.

This reduces the rectangle intersection problem to the problem of maintaining a set of intervals such that intervals can be efficiently inserted and deleted, and interval-intersection queries can be answered quickly. To achieve this, we implement our sweep-line status with the interval-tree data structure [Ede80]. We have implemented a semi-dynamic version, which must be initialized with the endpoints of all intervals it is going to contain during the sweep. The pre-processing, namely sorting the points and building up an empty balanced tree, takes $O(n \log n)$ time, where n is the number of intervals. Inserting intervals can then be done in $O(\log n)$ time, deleting in constant time, while a query takes $O(k + \log n)$ steps, where k is the number of intervals reported. Since every interval is stored only once in the interval tree, the storage consumption is linear.

6.2 Step by Step Towards Good Design

In this section we start with a naive interface for the algorithm described in the previous section. Then we consider the limitations of this approach and show how these can be overcome. We will improve the straight-forward solution in several steps, each of which is independent of the others. Note that the naive interface could easily be implemented without explicit data conversion by all of the subsequent solutions—if that was our goal.

6.2.1 The Naive Approach

How would a naive programmer interface an algorithm for the rectangle intersection problem described above? He might implement a function, whose input would be an array of rectangles and their number. The output could either consist of a list of pairs of intersecting rectangles, or, more user-friendly, of a so-called conflict graph. In this graph, each rectangle is represented by a node, and two nodes are connected by an edge, if the corresponding rectangles intersect. This is the interface described in Program 6.2.1.

```
class Rectangle;
class Graph;
Graph* rectangle_intersection(Rectangle* rectangle_array, int n);
```

Program 6.2.1: a naive functional interface

This interface would force the programmer to implement the data structures `Rectangle` and `Graph`. On the other hand, it would force the user in most cases to convert his rectangles into those required by the interface, and, after calling `rectangle_intersection`, extract the desired information from the conflict graph. In order to do so, the user would have to study the interfaces of the in- and output data structures. Another disadvantage of the naive interface is that it would not even allow the user to hand over rectangles in a *container* different from an array, like a list or a set.

6.2.2 Decoupling Algorithm and Data Organization

The most obvious disadvantage of our previous interface is the tight link between the algorithm and its in- and output data structures. In order to decouple algorithm and data organization, we must solve two problems.

- (P1) *container independence*, i.e. we do not want to force the user to hand over an *array* of rectangles.
- (P2) *representation independence*, i.e. we should not require the use of a fixed representation of rectangles or graphs, but rather accept any representation fulfilling certain requirements.

The first problem can be solved with the help of *iterators*. Iterators are a generalization of pointers; they are light-weight objects that point to other objects. As the name suggests, iterators are used to iterate over a range of objects: if an iterator points to an element in a range, it can be incremented so that it points to the next element or to an end-of-range marker. Iterators can also be tested for equality, e.g. to test whether the end of a range is reached. They represent an extremely versatile link between containers and algorithms. If an algorithm's interface takes iterators as arguments, then the algorithm can be applied to any container that provides access to its elements via iterators. This is the central concept introduced by the STL [MS96].

Consequently, our next interface will expect iterators to manage the input. Handling the output via iterators is not so simple, as the conflict graph is not a linear structure. This problem is attacked in the following section. For the time being, we ask the user to provide his definition of a graph as a template parameter to our interface. Of course, this definition must fulfill some requirements. The implementation expects the member functions for inserting nodes and edges.

```
class User_Rectangle;
class User_graph<User_Rectangle*>;
typedef User_graph<User_Rectangle*> Graph;
// requirements
typedef Graph::node_type node;
node Graph::insert_node (User_Rectangle*);
void Graph::insert_edge (node&, node&);
```

Note that the user is not forced to write his own graph data structure; he can use that of any library and write a simple wrapper that implements our requirements with the help of the library graph.

In order to solve the second problem, we use so-called *data accessors*. While iterators realize access to objects, data accessors are used to access the data associated with these objects [KW97]. Data accessors have two parts, a data accessing function, which is responsible for the actual access, and a light-weight object. This object is also referred to as the data accessor. It encapsulates the data type to be accessed and is used to select the correct data accessing function. Our next interface will require the user to provide such data accessors for his representation of the input data. Assume that the user declares the following `User_Rectangle` type.

```
typedef CoordType double;
struct User_Rectangle { CoordType llx, lly, urx, ury; };
typedef User_Rectangle* User_iterator;
```

where `llx`, `lly`, `urx`, and `ury` represent the coordinates of the lower left and upper right corner of the rectangle, respectively. The data accessor for the x -coordinate of the lower left corner can then be defined as follows.

```
// data accessor
struct LLXDA { typedef CoordType value_type };
// data accessing function
CoordType get(LLXDA const& da, User_iterator const& it)
{ return (*it).llx; }
```

All our algorithm needs to know about the user's rectangles are the coordinates of their corners. This is exactly what the data accessors will supply. Note the interplay between data accessing function and its arguments, namely the data accessor and the iterator, in Program 6.2.2.

```

template < class Iterator, class Graph,
           class LLXDA, class LLYDA, class URXDA, class URYDA >
void rectangle_intersection(Iterator begin, Iterator end,
                           Graph& graph,
                           LLXDA llxda, LLYDA llyda,
                           URXDA urxda, URYDA uryda)
{
    for (Iterator rect_it = begin; rect_it != end; ++rect_it)
        typename LLXDA::value_type lower_x = get(llxda, rect_it);
    // ...
    Iterator rect_it1, rect_it2;    // ...
    typename Graph::node_type node1 = graph.insert_node(rect_it1),
        node2 = graph.insert_node(rect_it2);
    graph.insert_edge(node1, node2); // ...
}

```

Program 6.2.2: a functional, data-organization independent interface

6.2.3 Tightening Control

Suppose the user of our implementation is only interested in a tiny fraction of the output, like the number of intersections or all rectangles intersecting a given rectangle. Or suppose he would like to abort the execution of the algorithm when the sweep line reaches a certain x -value or if another condition becomes true. With the interface suggested in the previous subsections, he would not be able to take advantage of such a situation. It is clear that a functional implementation cannot provide such a degree of interaction between the user and the algorithm. Thus we will switch to a class interface, which allows us to have a state and offer the user more information about the algorithm's progress.

The key to more control is the *loop kernel* [Küh96, Wei97]. The loop kernel is a method which encapsulates the body of the central loop of the algorithm. It can be advanced in single steps and informs the user about the current state of the algorithm. The loop kernel makes the whole algorithm look like an iterator that can be incremented until the execution is finished.

For our example algorithm, we would define the following states: **none** at the beginning, **done** at the end, **rectangle_begin** when the sweep line hits the left edge of a rectangle, and **rectangle_end** when a right edge is reached. We implement the loop kernel by a member function **step()** that advances the sweep line to the next event point and returns the current state, see Program 6.2.3.

The concept of the loop kernel would be incomplete without the idea of *full logical inspectability*. An algorithm is fully logically inspectable if the user can access all important intermediate results during the execution. This is important for animation, interaction or simply for debugging. In our example this would mean access to the content of the whole sweep-line status, not just to those rectangles that intersect the current one. Hence we offer two pairs of iterators, one referring to the whole sweep-line status, and one marking just the range of current intersections. Their type is discussed in Section 6.2.5.

```

template < class Iterator,
           class LLXDA, class LLYDA, class URXDA, class URYDA >
class Rectangle_intersection {
public:
    // constructor
    Rectangle_intersection(Iterator begin, Iterator end);
    // return the result in user supplied graph
    template <class Graph> void run(Graph& graph);
    // loop kernel
    enum state { none, rectangle_begin, rectangle_end, done };
    bool valid() { return (state != done); }
    state step();
    // full logical inspectability
    Iterator current(); // rectangle represented by curr. event point
    typedef /* ... */ Solution_iterator;
    Solution_iterator begin();
    Solution_iterator end();
    // queries: report all rectangles intersecting the curr. rectangle
    Solution_iterator current_begin();
    Solution_iterator current_end();
};

```

Program 6.2.3: a class interface

Note that the user can still get the output in a graph representation as before, but the existence of a graph data structure is no longer a prerequisite to using the algorithm. (This can also be achieved without templated member functions like `run()`, which are standard conform, but not yet supported by all C++ compilers.)

6.2.4 Influencing Critical Decisions

Another important question is the following. Which definition of “intersection” do we implement? Does touching already imply an intersection? What about inclusion? Of course, the answers to these questions will differ from application to application. Instead of trying to cover all possible interpretations, we leave the definition of an intersection to the user. To do so, we must isolate the decision making parts of our implementation such that no information local to the algorithm is needed. Then the user can provide *function objects*, with which the algorithm is parameterized.

Our rectangle intersection algorithm has two basic data structures, the event-point schedule and the sweep-line status. The order of event points decides, whether the projections of the corresponding rectangles intersect on the x -axis. Similarly, a query of the sweep-line status returns rectangles, whose projections intersect that of the current rectangle on the y -axis. To differentiate between these two categories of intersections, we require the user to provide two function objects, one that enables us to sort the event-point schedule accordingly and the other for determining the behavior of the interval tree that

implements the sweep-line status.

In the following we give examples of function objects that view rectangles as topologically open and thus do not report rectangles touching each other. To sort the event points accordingly, we just have to make sure that an event point e_l corresponding to the left edge of a rectangle will be inserted into the schedule after all event points that belong to right edges with the same x -coordinate. Then all of the latter rectangles are already removed from the sweep-line status and will not be reported when we reach e_l .

Since an event point is internal to our algorithm, it cannot be accessed directly by the user. Thus we have to isolate the information needed to sort the event points. If two event points have the same x -value, we need to know whether each corresponds to a left or right edge of the respective rectangle. This information can easily be obtained via the x -coordinate of the event point plus the iterator pointing to the corresponding rectangle. Their types are of course known to the user. Thus we require a function object, which realizes a comparison between two pairs of the corresponding types, see Program 6.2.4.

```
class Compare_x {
public:
    typedef pair<CoordType, User_iterator> Pair;
    bool operator()(const Pair& p, const Pair& q) {
        bool before = true;
        if (p.first == q.first)
        {
            // p is the x-coordinate of the left edge of a rectangle
            if (p.first == p.second->llx) before = false;
        }
        else before = (p.first < q.first);
        return before;
    }
};
```

Program 6.2.4: compare function object for sorting the event-point schedule

The function object for the interval tree is quite simple, see Program 6.2.5.

```
class Compare_y {
public:
    bool operator()(CoordType const y1, CoordType const y2)
    { return (y1 < y2); }
};
```

Program 6.2.5: compare function object for querying the sweep-line status

Note the difference of this technique to the approach of implementing the most general definition of intersection and then filtering out all undesired information. Our function objects have the potential to reduce the complexity not just of the *output*, but also of the *computation*.

6.2.5 The Complete Interface

Program 6.2.6 shows the interface resulting from our successive improvements. At first sight it might look more complicated than the naive interface. To guarantee a smooth learning curve for users not familiar with generic programming and the algorithm we implemented, a good library would provide a concrete representation of rectangles and graphs, say `Our_rectangle` and `Our_graph`, as well as defaults for the other template parameters. This would make it possible to use our algorithm as in Program 6.2.7.

Note that the constructor has been parameterized by objects of each of the class's template parameters. This allows the user to instantiate our algorithm with objects that may be constructed other than by their default constructor. The data accessors `URXDA` and `URYDA` for the coordinates of the upper right corner of the input rectangles might for example be parameterized with a given height and width of the rectangles, which could change from instantiation to instantiation of the intersection algorithm.

Of course, we have also implemented the interval tree for the sweep-line status with the concepts presented here. This is the point where the flexibility of our algorithms bears fruit, since we do not have to convert any rectangles into intervals to construct the interval tree. This is due to the fact that the endpoints of the intervals we want to store in the tree correspond to the y -coordinates of the corners of our input rectangles. Thus we just parameterize the nested interval tree class with a subset of the template parameters of the class `Rectangle_intersection`. The required parameters are the types of the rectangle iterator and the compare function object for y -coordinates as well as the data accessors `LLYDA` and `URYDA`, see the private definition of the type `Interval_tree` in Program 6.2.6. So in a way, the class `Interval_tree` considers our input rectangles to be nothing but intervals, namely the rectangles' projection on the y -axis.

The iterator `Solution_iterator` needed to traverse the sweep-line status is provided by the class `Interval_tree`. Dereferencing a `Solution_iterator` supplies the user with an iterator of the type, with which he has parameterized the class `Rectangle_intersection`.

Program 6.2.8 shows how the data structures required by our algorithm could be declared. Our example demonstrates one of the advantages of using data accessors. If the input consists of squares of common size, the user has to store only the coordinates of their lower left corners. When our algorithm needs a coordinate of the opposite corner, the corresponding data accessing function computes it on the fly. This reduces the storage consumption of the input by 50%.

Note that not all compilers support the pointer-to-member mechanism for template parameters we used here. The obvious workaround is to declare explicitly all four data accessors and accessing functions required by the generic implementation.

The intersection algorithm for squares can then be declared as in Program 6.2.9.

```

class Our_rectangle; class Our_graph;
class Our_lxda; class Our_lyda; class Our_hxda; class Our_hyda;
class Our_compare_x; class Our_compare_y;

template < class Iterator = Our_rectangle*,
           class LLXDA = Our_lxda, class LLYDA = Our_lyda,
           class URXDA = Our_hxda, class URYDA = Our_hyda,
           class CompareX = Our_compare_x,
           class CompareY = Our_compare_y >
class Rectangle_intersection
{
    // type of the sweep-line status
    typedef Interval_tree<Iterator, LLYDA, URYDA, CompareY>
        Sweep_line_status;
public:
    // type of rectangle coordinates
    typedef typename LLXDA::value_type value_type;
    // type of iterator used to return intersections
    typedef typename Sweep_line_status::iterator Solution_iterator;
    // constructor
    Rectangle_intersection(Iterator begin, Iterator end,
                          LLXDA lxda = LLXDA(), LLYDA lyda = LLYDA(),
                          URXDA hxda = URXDA(), URYDA hyda = URYDA(),
                          CompareX comp_x = CompareX(),
                          CompareY comp_y = CompareY() );

    // loop kernel
    enum state { none, rectangle_begin, rectangle_end, done };
    bool valid() const { return (state != done); };
    state step();
    // full logical inspectability
    Iterator current(); // rectangle represented by current event point
    value_type current_sweep_line_position() const;
    Solution_iterator begin();
    Solution_iterator end();
    // queries: report all rectangles intersecting the curr. rectangle
    Solution_iterator current_begin();
    Solution_iterator current_end();
    // miscellaneous member functions
    int number_of_intersections() const;
    // return conflict graph of input rectangles
    template <class Graph> void run (Graph& graph);
};

```

Program 6.2.6: flexible interface of the class Rectangle_intersection


```

Our_rectangle rects[10];
Our_graph our_graph;
// ...
// declare and run the rectangle intersection algorithm
Rectangle_intersection rectangle_intersection(rects, rects+10);
rectangle_intersection.run(our_graph);

```

Program 6.2.7: Ease-of-use: applying `Rectangle_intersection` to library-supplied data structures

```

// representation of a square
typedef int CoordType;
const CoordType length = 50;
struct Square { CoordType x, y; };

// data accessors for the above representation
template<CoordType Square::*member>
struct CoordLowDA { typedef CoordType value_type; };
template<CoordType Square::*member>
struct CoordHighDA { typedef CoordType value_type; };

// data accessing functions
template <CoordType Square::*member>
inline CoordType get(CoordLowDA<member> const&,
                    User_iterator const& it)
{ return (*it).*member; };

template <CoordType Square::*member>
inline CoordType get(CoordHighDA<member> const&,
                    User_iterator const& it)
{ return (*it).*member + length; };

// compare function objects as defined above
class Compare_x; class Compare_y;

```

Program 6.2.8: Flexibility: user-supplied types for using the class `Rectangle_intersection`

```

// algorithm
typedef Rectangle_intersection< User_iterator,
                             CoordLowDA<&(Square::x)>,
                             CoordLowDA<&(Square::y)>,
                             CoordHighDA<&(Square::x)>,
                             CoordHighDA<&(Square::y)>,
                             Compare_x, Compare_y >
    Square_intersection_algo;

// iterator for access to the solution
typedef typename Square_intersection_algo::Solution_iterator
    Solution_it;

```

Program 6.2.9: declaration of the class `Rectangle_intersection` for squares of common size

Program 6.2.10 shows how the types declared in Program 6.2.8 and 6.2.9 are plugged into our interface.

```

main()
{
  Square squares[10]; // input of squares...
  Square_intersection_algo my_algo(squares, squares+10);
  while (my_algo.valid())
    if (my_algo.step() == Square_intersection_algo::rectangle_begin)
    {
      Square* curr = my_algo.current();
      // assuming output operator for Square
      cout << *curr << " intersects: " << endl;
      Solution_it end = my_algo.current_end();
      for (Solution_it it = my_algo.current_begin(); it != end; ++it)
        cout << *it << endl;
    }
}

```

Program 6.2.10: a toy application for user-supplied data structures

6.3 Experiments

We compare two different implementations of the rectangle intersection problem in terms of runtime. The implementations are characterized as follows.

1. the *object-oriented* approach encapsulates the interface of Section 6.2.1 in a class. Its implementation requires a fixed type of rectangle. Similarly, the underlying interval tree is a class requiring a fixed type of interval.
2. the *generic* approach implements the generic interface of Section 6.2.5. All data structures are implemented according to the concepts suggested in this chapter.

6.3.1 Example Classes

We ran both implementations on the following eight example classes. These benchmarks have also been used to compare the quality of map-labeling algorithms experimentally, see Section 3.2. They are available from our Web page.

RandomRect. We choose n points uniformly distributed in a square of size $25n \times 25n$. Each point corresponds to the lower left corner of a rectangle. To determine the size of each rectangle, we choose the length of both edges independently under normal distribution, take its absolute value and add 1 to avoid non-positive values. Finally we multiply both rectangle dimensions by 10.

DenseRect. Here we try to place as many rectangles as possible on an area of size $\alpha_1\sqrt{n} \times \alpha_1\sqrt{n}$. α_1 is a factor chosen such that the number of successfully

placed rectangles is approximately n , the number of sites asked for. We do this by randomly selecting the rectangle size as above and then trying to place the rectangle 50 times. If we don't manage, we select a new rectangle size and repeat the procedure. If none of 20 different sized rectangles could be placed, we assume that the area is well covered, and stop. For each rectangle we placed successfully, we return its height and width. To generate (a limited amount of) intersections, we randomly choose a corner and use that as the position of the lower left corner of the rectangle we return.

RandomMap and DenseMap. These example classes try to imitate a real map using the same methods as `RandomRect` and `DenseRect` for placing the lower left corner of the rectangles, but more realistic rectangle sizes. We assume a distribution of 1:5:25 of cities, towns and villages. After randomly choosing one of these three classes according to the assumed distribution, we set the rectangle height to 12, 10 or 8 points accordingly. The length of the rectangle text then follows the distribution of a set of 377 German Railway station names. We assume a typewriter font and set the rectangle length to the number of characters times the font size times $2/3$. The multiplicative factor reflects the ratio of character width to height.

VariableDensity. This example class was suggested in an experimental comparison of map-labeling algorithms by Christensen et al. [CMS95]. There, the points are distributed uniformly on a rectangle of size 792×612 . All rectangles are of equal size, namely 30×7 .

HardGrid. In principle we use the same method as for `DenseRect` and `DenseMap`, that is, trying to place as many rectangles as possible into a given area. In order to do so, we use a grid of $\lfloor \alpha_2 \sqrt{n} \rfloor \times \lceil \alpha_2 \sqrt{n} \rceil$ cells with edge lengths n . Again, α_2 is a factor chosen such that the number of successfully placed squares is approximately n . In a random order, we try to place a square of edge length n into each of the cells. This is done by randomly choosing a point within the cell and putting the lower left corner of the square on it. If it overlaps any of the squares placed before, we repeat at most 10 times before we turn to the next cell. Finally, we choose a random corner of the square we placed and use that as the lower left corner of the square we return.

RegularGrid. We use a grid of $\lfloor \sqrt{n} \rfloor \times \lceil \sqrt{n} \rceil$ square grid cells. For each cell, we randomly choose a corner and place a point with a small constant offset near the chosen corner. On this point, we place a square with an edge length of grid cell size minus the offset.

MunichDrillholes. The municipal authorities of Munich provided us with the coordinates of roughly 19,400 ground-water drill holes within a 10 by 10 kilometer square centered approximately on the city center. From these sites, we randomly pick a center point and then extract a given number of sites closest to the center point according to the maximum norm. Thus we get a rectangular section of the map. Its size depends on the number of points asked for. The drill-hole labels are abbreviations of fixed length. By scaling the x -coordinates, we make these rectangular labels into squares and subsequently apply an exact solver for label size maximization. The label size determined in

this way is the size of the squares we return. We place them with their lower left corner on the scaled drill holes.

Figures 6.17 to 6.24 show an important parameter of these example classes, namely their average number of intersections. We did not count pairs of *touching* rectangles, like in the example implementation of the compare function objects in Section 6.2.4. We used examples of approximately 250, 500, . . . up to 3000 rectangles. For each of the example classes and each example size, we averaged the runtime and the number of intersections over 30 files.

6.3.2 Results

In Figures 6.1 to 6.24, the average number of rectangles over the 30 files for each example size is shown on the x -axis. On the y -axis, Figures 6.1 to 6.8 show the average runtime for both implementations for the interesting special case that the user is merely interested in the number of intersections. This is slightly favorable for the generic implementation since there, no information about which rectangles are in fact intersecting, has to be stored. In this setting, the running times were nearly identical.

Figures 6.9 to 6.16 show the runtime when all intersections are stored in adjacency lists during the execution of the two programs. Here the generic implementation took between 0 and 60% longer than the object-oriented version. Note however that this gap is smaller—not more than 20%—when the two example classes RegularGrid and MunichDrillholes with the lowest density are ignored, i.e. those with the smallest ratio of the number of rectangles and the number of intersections, see Figure 6.22 and 6.23.

The average runtime is given in CPU seconds. It was measured on a Sparc-Ultra-1 machine; the programs were compiled with the SUN CC-4.2 compiler with optimizer option `-fast`. On our Web page, we provide graphs for the same test suite run on an SGI IP27 with the mipsPRO CC-7.1 compiler. The results were comparable.

While the generic and the object-oriented implementation do differ much in their source code size, we listed the sizes of executables for identical test programs for both implementations in Table 6.1. The source code is available via our Web page as well.

<i>test program for . . .</i>	<i>size of executable in KBytes</i>
object-oriented interval tree	63
generic interval tree	74
object-oriented rectangle intersection	128
generic rectangle intersection	120

Table 6.1: sizes of the executables

It is interesting to note that although the executable of a simple test program for the generic version of the interval tree is slightly larger than that of its object-oriented counterpart, it is opposite for the corresponding versions of the rectangle intersection data structure. The reason for this seems to be that the generic implementation does not need to convert and store the input data for the interval tree explicitly. This difference in the sizes of the executables may become substantial in case of larger class hierarchies.

6.3.3 Evaluation

We have presented a toolbox of concepts which helps to turn inflexible into generic and thus reusable interfaces. We have exemplified this transition at a geometric algorithm, namely a sweep-line algorithm for the rectangle intersection problem. On the road from a naive to a flexible interface for this algorithm, we suggested to decouple algorithms from the organization of their in- and output data. Then we presented the loop kernel as an important means of gaining control over the execution of an algorithm. Full logical inspectability introduced additional transparency. Finally, we came up with function objects as a way to parameterize algorithms with information that can be used to influence critical decisions.

In our experiments, we compared a generic to an object-oriented implementation of the rectangle intersection algorithm. We investigated the runtime of the two implementations on eight example classes from random and real world sources and in two different settings. In the first setting, which was favorable for the generic implementation, the running times were nearly identical. In the second setting, the generic implementation was just 20% slower than its competitor on all example classes but the two least dense. We do not think that this is too high a price for the gain of flexibility achieved by the generic interface.

The slowdown caused by an object-oriented library like LEDA is of a different order of magnitude. In [MN92], the runtime of Dijkstra's algorithm using LEDA Fibonacci heaps is compared to an implementation using special integer Fibonacci heaps. For this example, the authors report a slowdown by a factor of 3.

Runtimes for Computing the Number of Intersections

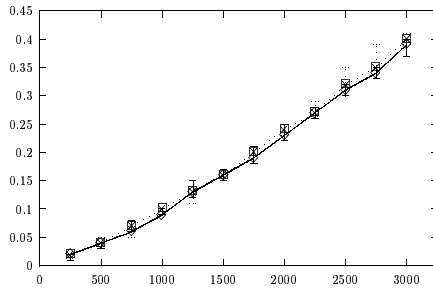


Figure 6.1: RandomMap

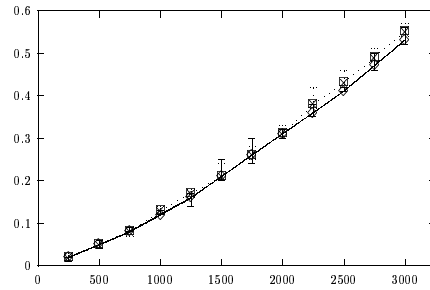


Figure 6.2: RandomRect

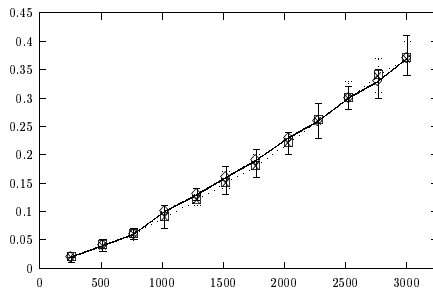


Figure 6.3: DenseMap

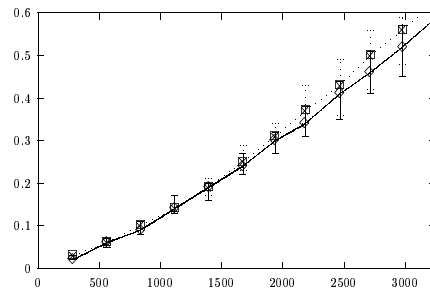


Figure 6.4: DenseRect

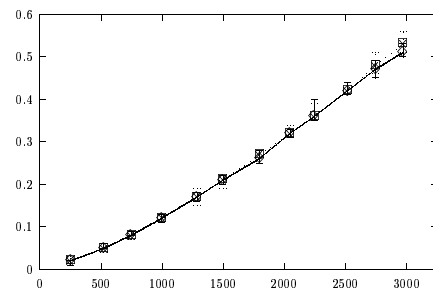


Figure 6.5: HardGrid

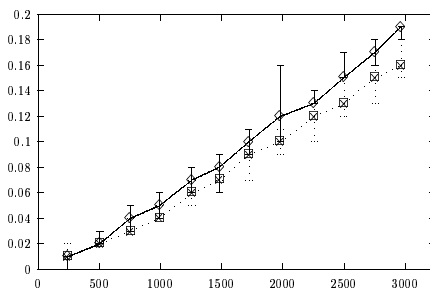


Figure 6.6: RegularGrid

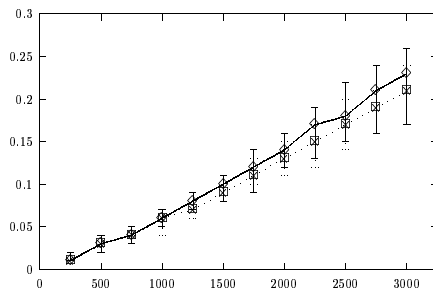


Figure 6.7: MunichDrillholes

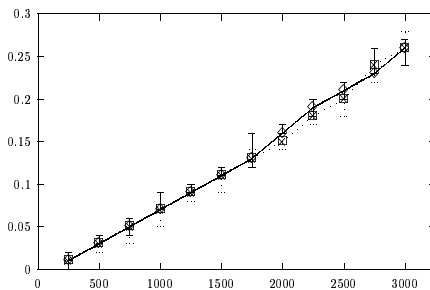


Figure 6.8: VariableDensity

Generic Implementation



Object-Oriented Implementation



Runtimes for Generating Adjacency Lists

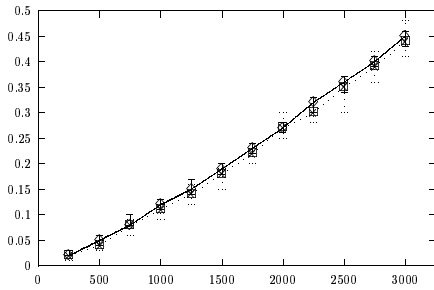


Figure 6.9: RandomMap

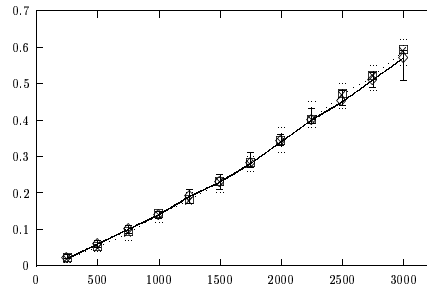


Figure 6.10: RandomRect

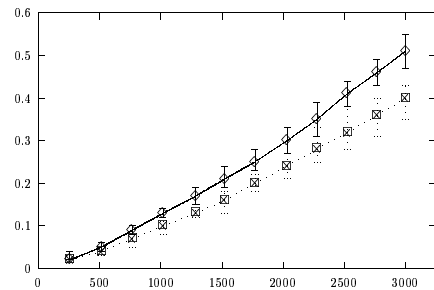


Figure 6.11: DenseMap

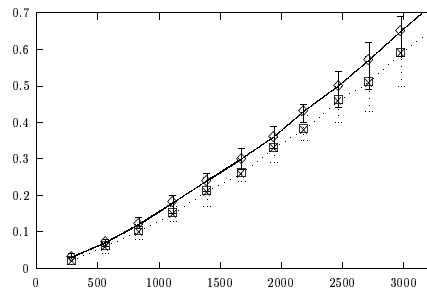


Figure 6.12: DenseRect

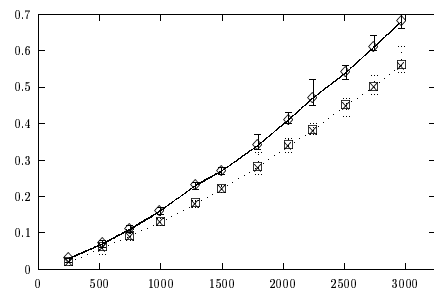


Figure 6.13: HardGrid

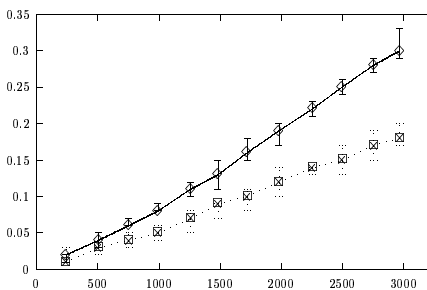


Figure 6.14: RegularGrid

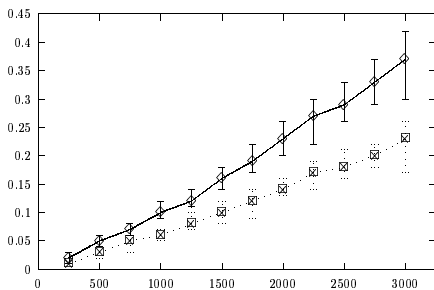


Figure 6.15: MunichDrillholes

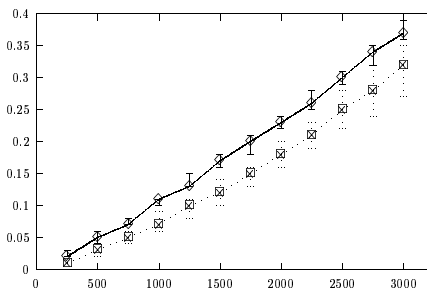




Figure 6.16: VariableDensity

Generic Implementation 

Object-Oriented Implementation 

Numbers of Intersecting Rectangles

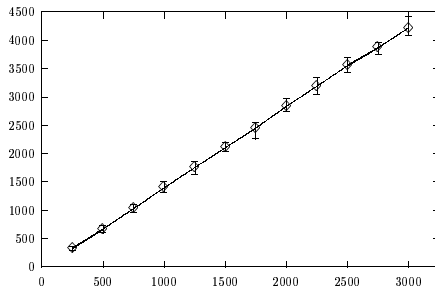


Figure 6.17: RandomMap

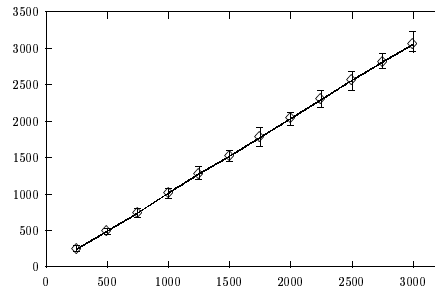


Figure 6.18: RandomRect

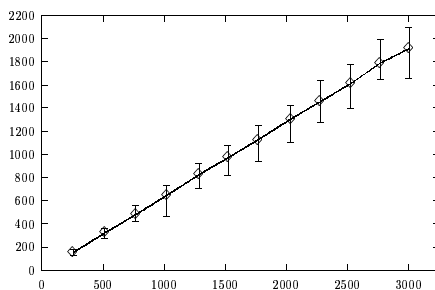


Figure 6.19: DenseMap

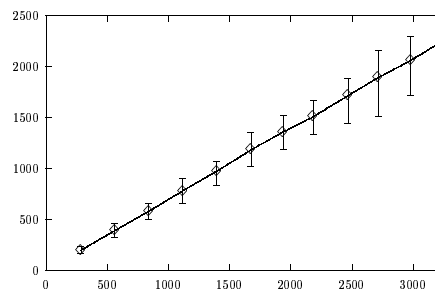


Figure 6.20: DenseRect

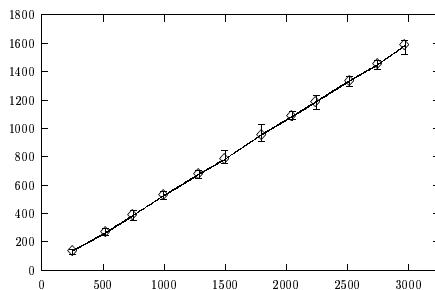


Figure 6.21: HardGrid

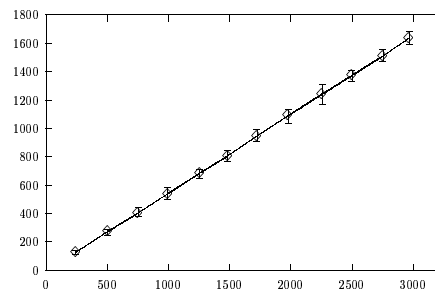


Figure 6.22: RegularGrid

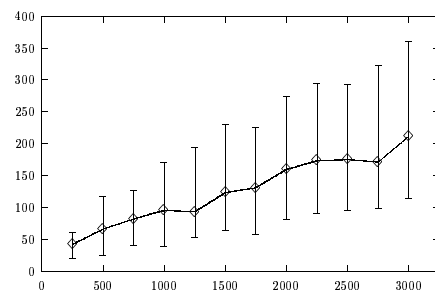


Figure 6.23: MunichDrillholes

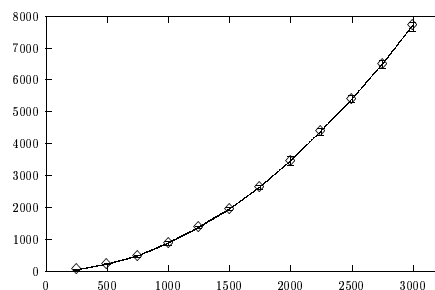


Figure 6.24: VariableDensity

Conclusion

In this thesis we have presented research on the problem of attaching labels to features in graphs, networks, diagrams, or cartographic maps. We have approached the problem from two sides. On the one hand we have developed a general framework for maximizing the number of features that receive a label by extending the classical constraint-satisfaction framework to maximum variable-subset constraint satisfaction. The features and label candidates of a label-placement problem are called variables and values, respectively, in the context of constraint satisfaction. The fact that two candidates intersect and thus should not both appear in a solution is expressed by symmetric binary constraints. Our new framework is of interest not only for label-placement problems, but for any overconstraint system where it is possible to drop some of the variables in order to find an assignment for the remaining variables that satisfies all constraints. We have proposed an algorithm, EI-1, that achieves edge-irreducibility, a new form of local consistency similar to arc-consistency in classical constraint satisfaction. EI-1 considers pairs of variables and removes their redundant values, i.e. values whose removal does not reduce the size of an optimal solution. We have also given an efficient algorithm, EI-1*, that combines EI-1 with a simple heuristic and proved to be very effective for labeling point sets.

It would be interesting to extend EI-1 to problems where constraints are not necessarily symmetric or binary, and to take priorities into account. The rules that EI-1 checks in order to achieve edge-irreducibility for all pairs of variables can easily be generalized to any constant number r of variables. However, it is not clear whether the generalized rules achieve r -irreducibility and whether there are efficient and practical algorithms that can apply these rules for a constant $r > 2$ exhaustively.

Our hope is that EI-1* or other efficient algorithms based on higher degrees of irreducibility will substitute simulated annealing and other iterative methods of gradient descent for the wide variety of problems that fit into the framework of maximum variable-subset constraint satisfaction.

In this thesis we have also investigated special cases of the label-placement problem. For labeling points with squares or with circles we have shown that the NP-hardness for picking labels from finite candidate sets carries over to infinite candidate sets, more precisely to the case where we require that the boundary of a label contains the point to be labeled. For rectangles of fixed

height we have given a corresponding positive result, namely a polynomial-time approximation scheme (PTAS) for label-number maximization. Since this scheme uses stabbing lines of equal spacing, it is not clear how to extend our result to arbitrary rectangles. Even for choosing a non-intersecting subset of maximum cardinality from n (fixed-position) rectangles only a factor- $O(\log n)$ approximation algorithm is known [AvKS98].

We have shown that one cannot expect to find a PTAS for maximizing the size of uniform circular labels. Although we proposed an efficient approximation algorithm for this problem, there is still a large gap between the approximation factor of about $1/20$ that we have shown for our algorithm and the factor $1/2$ that we conjecture to be best possible.

While the hardness results and approximation schemes above are predominantly of theoretical interest, the algorithm EI-1* for features with a constant number of label candidates and the greedy algorithm for sliding rectangular labels, our method for labeling polygonal chains, and our toolbox for designing flexible geometric algorithms can and should be applied in practice.

The point-labeling algorithm Rules, which is described in Section 3.2, is actually used by the city authorities of Munich for labeling ground-water drill holes. We also cooperated with a company that wants to label tourist shops in Berlin. This company plans to offer the following on-line service². A user can select a subset of the shops according to what articles (s)he is interested in. After pressing the submit button, the corresponding shops in the current map area are labeled with their names. More information on a shop can be obtained by clicking on its label.

²<http://von-kunst-bis-krimskrams.de/>

Zusammenfassung

Bei der Visualisierung von Information auf Landkarten, in Graphen oder Diagrammen spielt die Beschriftung von Gestaltungselementen wie Punkten, Linienzügen oder Kanten eine große Rolle. So schätzt man, dass bei der manuellen Erstellung einer Landkarte ungefähr die Hälfte der Zeit für die Beschriftung benötigt wird. Dieser Umstand erklärt das Interesse daran, den Prozess des Beschriftens möglichst weitgehend zu automatisieren. Das Beschriftungsproblem lässt sich ganz allgemein als kombinatorisches Entscheidungsproblem ausdrücken: Gegeben eine Menge von zu beschriftenden Elementen einer Grafik und zu jedem Element eine Menge von Beschriftungskandidaten, ist es möglich, jedem Element einen Kandidaten aus der betreffenden Menge zuzuordnen, ohne dass sich zwei der gewählten Kandidaten schneiden?

Leider muss man davon ausgehen, dass dieses Problem im allgemeinen nicht effizient zu entscheiden ist. Daher haben fast alle bisherigen Arbeiten zu diesem Thema heuristische oder approximative Verfahren für entsprechende Optimierungsprobleme vorgeschlagen, etwa um eine Beschriftung einer möglichst großen Teilmenge der Grafikelemente zu finden. Dieses Problem bezeichnet man als das Anzahlmaximierungsproblem.

In meiner Dissertation näherte ich mich dem Beschriftungsproblem von zwei Seiten. Einerseits stelle ich ein theoretisches Gerüst auf, mit dessen Hilfe man das Anzahlmaximierungsproblem für endliche Kandidatenmengen gut formulieren und effiziente Algorithmen angeben kann, die den Suchraum für eine optimale Lösung erheblich verkleinern. Dieses Gerüst verallgemeinert das klassische Constraint-Satisfaction-Problem. Ich gebe einen solchen Algorithmus sowie eine einfache Heuristik an, die auf diesen Algorithmus aufbaut und in der Praxis sehr gute Resultate liefert.

Andererseits beschäftige ich mich mit Spezialfällen des Beschriftungsproblems. Ich habe mich zuerst mit der Beschriftung von Punkten mit achsenparallelen Rechtecken befasst, wobei für jeden Punkt die vier klassischen Positionen zugelassen wurden, also die, bei denen eine Ecke der Beschriftung den betreffenden Punkt berühren muss. Auf dieses Spezialproblem habe ich den erwähnten allgemeinen Algorithmus angewandt und mit anderen Verfahren experimentell verglichen. Dann habe ich mich mit Beschriftungsmodellen beschäftigt, in denen eine unendliche Anzahl von Beschriftungskandidaten zugelassen wird. Ich konnte zeigen, dass man nicht erwarten kann, obiges Entscheidungsproblem für quadratische oder kreisförmige Beschriftungen effizient zu lösen, falls jede Be-

schriftung ihren Punkt berühren muss. Trotzdem habe ich effiziente Algorithmen für Optimierungsversionen beider Probleme gefunden. Für achsenparallele rechteckige Beschriftungen gleicher Höhe stelle ich ein Approximationsschema für das Anzahlmaximierungsproblem vor. Andererseits zeige ich, dass man nicht erwarten kann, ein solches Schema für die Maximierung der Größe kreisförmiger Beschriftungen zu finden.

Ausser der Beschriftung von Punkten untersuche ich das Problem, wie man Linienzüge, also Flüsse oder Straßen, qualitativ hochwertig beschriften kann. Dies ist im Gegensatz zur Punktbeschriftung, wo die Zielfunktion meist klar ist, eher ein Modellierungsproblem, bei dem man die Anforderungen der Kartographen erst herausfinden muss. Ich habe diese dann in harte und weiche Anforderungen eingeteilt und einen effizienten Algorithmus vorgeschlagen, der garantiert, die harten Anforderungen zu erfüllen, und der innerhalb eines Teils des Suchraums die weichen Anforderungen soweit wie möglich optimiert.

Um die Praxisrelevanz meiner Untersuchungen zu unterstreichen, sind die meisten angegebenen Algorithmen implementiert worden. Dabei bin ich auf das schon bekannte Paradigma des generischen Programmierens gestoßen, das es ermöglicht, Programme sehr allgemein und flexibel zu halten. Ich zeige, wie man damit erfolgreich geometrische Algorithmen entwirft.

Curriculum Vitae

- 30.11.67 born in Stuttgart, Germany
- 2.6.87 A-levels at Mörike Gymnasium Ludwigsburg
- 1.7.87 – 28.2.89 alternative service
- 1.10.89 enrolment at Albert Ludwig University Freiburg;
major in mathematics, minor in computer science
- 1.10.91 enrolment at Freie Universität Berlin;
study focus: theoretical computer science
- 19.12.95 graduation from Freie Universität Berlin;
title of Master's Thesis: "Map Labeling"
- 1.6.96 – 31.5.99 research assistant of Dr. Frank Wagner at Freie Universität
Berlin; work on project "Efficient Algorithms for Map La-
beling" funded by the German Science Foundation (DFG)
- 14.7. – 10.8.97 and
- 22.3. – 31.3.99 research guest of Dr. Marc van Kreveld, Utrecht University,
The Netherlands
- 10.2.99 completion of PhD thesis
- 26.4. – 30.4.99 research guest of Prof. Peter Widmayer, ETH Zurich,
Switzerland
- 28.5.99 doctoral defense, talk on "Parametrized Complexity—
a New Approach for Hard Problems"

Thanks

Thanks to Sabine.

Thanks to Frank for giving me a great research topic.

Thanks to Vikas for implementing our algorithms at day and night.

Thanks to Sven for being such a great office mate at the institute.

Thanks to Christian for answering my technical questions.

Thanks to Marc for giving me the opportunity to do research in Utrecht.

Thanks to Tycho for all the mathematical lessons he taught me.

Thanks to Jack for suggesting Lemma 3.13 and its proof.

Thanks to Frank Schumacher for waiting as patiently as a spider in the Web.

Thanks to Rudi Krämer and Karsten for real-world data.

Thanks to Ulrike for pointing me to the topic of my defense talk.

Thanks to Karla Thiede for steering me smoothly through administration.

Thanks to the German Science Foundation (DFG) for funding the project.

Bibliography

- [Ach95] Alf-Christian Achilles. The collection of computer science bibliographies. <http://liinwww.ira.uka.de/bibliography/>, 1995.
- [AF84] John Ahn and Herbert Freeman. A program for automatic name placement. *Cartographica*, 21(2–3):101–109, 1984.
- [AH95] David H. Alexander and Carl S. Hantman. Automating linear text placement within dense feature networks. In *Proc. AutoCarto 12*, pages 311–320. ACSM/ASPRS, Bethesda, 1995.
- [AIK89] Hiromi Aonuma, Hiroshi Imai, and Yahiko Kambayashi. A visual system of placing characters appropriately in multimedia map databases. In *Proceedings of the IFIP TC 2/WG 2.6 Working Conference on Visual Database Systems*, pages 525–546. North-Holland, 1989.
- [Ali62] Georges Alinhac. *Cartographie Théorique et Technique*, chapter IV. Institut Géographique National, Paris, 1962.
- [AvKS98] Pankaj K. Agarwal, Marc van Kreveld, and Subhash Suri. Label placement by maximum independent set in rectangles. *Computational Geometry: Theory and Applications*, 11:209–218, 1998.
- [Bar97] Mathieu Barrault. An automated system for name placement which complies with cartographic quality criteria: The hydrographic network. In *Proceedings of the Conference on Spatial Information Theory (COSIT'97)*, volume 1329 of *Lecture Notes in Computer Science*, pages 499–500, Pittsburgh, PA, 1997. Springer-Verlag.
- [Bes94] Christian Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.
- [BFR95] Christian Bessière, Eugene C. Freuder, and Jean-Charles Régim. Using inference to reduce arc-consistency computation. In *Proc. International Joint Conference on Artificial Intelligence*, Montréal, August 1995.

- [BL95] Mathieu Barrault and François Lecordix. An automated system for linear feature name placement which complies with cartographic quality criteria. In *Proc. Auto-Carto 12*, pages 321–330, Charlotte, NC, March 1995. ACSM/ASPRS, Bethesda.
- [Boy73] A. Raymond Boyle. Computer aided map compilation. Technical report, Department of Electrical Engineering, University of Saskatchewan, Canada, 1973.
- [Boy74] A. Raymond Boyle. Report on symbol and name manipulation and placement. Technical report, Department of Electrical Engineering, University of Saskatchewan, Canada, 1974.
- [BS98] Roland Backhouse and Tim Sheard. Call for Participation of “Workshop on Generic Programming”, Marstrand, Sweden. <http://www.cs.uu.nl/people/johanj/wgp98.html>, June 1998.
- [C⁺96] Bernard Chazelle et al. Application challenges to computational geometry: CG impact task force report. Technical Report TR-521-96, Princeton Univ., April 1996.
- [CFMS97] Jon Christensen, Stacy Friedman, Joe Marks, and Stuart Shieber. Empirical testing of algorithms for variable-sized label placement. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry (SoCG'97)*, pages 415–417, 1997.
- [CK89] L. Paul Chew and Klara Kedem. Placing the largest similar copy of a convex polygon among polygonal obstacles. In *Proceedings of the Fifth Annual Symposium on Computational Geometry, Saarbrücken*, pages 167–174, New York, 5–7 June 1989. ACM, ACM Press.
- [CLR90] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [CMS95] Jon Christensen, Joe Marks, and Stuart Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 14(3):203–232, 1995.
- [Coo88] A.C. Cook. *Automated Cartographic Name Placement Using Rule-Based Systems*. PhD thesis, Polytechnic of Wales, 1988.
- [dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.
- [DF92] Jeffrey S. Doerschler and Herbert Freeman. A rule-based system for dense-map name placement. *Communications of the ACM*, 35:68–79, 1992.

- [Djo94] Y. Djouadi. Cartage: A cartographic layout system based on genetic algorithms. In *Proc. EGIS'94*, pages 48–56, 1994.
- [DLSS95] Amitava Datta, Hans-Peter Lenhof, Christian Schwarz, and Michiel H. M. Smid. Static and dynamic algorithms for k -point clustering problems. *J. Algorithms*, 19:474–503, 1995.
- [DMM⁺97] Srinivas Doddi, Madhav V. Marathe, Andy Mirzaian, Bernard M.E. Moret, and Binhai Zhu. Map labeling and its generalizations. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 148–157, New Orleans, LA, 4–7 January 1997.
- [DMR97] Karen Daniels, Victor Milenkovic, and Dan Roth. Finding the largest area axis-parallel rectangle in a polygon. *Computational Geometry: Theory and Applications*, 7:125–148, 1997.
- [DP73] David H. Douglas and Thomas K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10(2):112–122, December 1973.
- [ECMS97] Shawn Edmondson, Jon Christensen, Joe Marks, and Stuart Shieber. A general cartographic labeling algorithm. *Cartographica*, 33(4):13–23, 1997.
- [Ede80] Herbert Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Report F59, Inst. Informationsverarb., Tech. Univ. Graz, Graz, Austria, 1980.
- [EIS76] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5:691–703, 1976.
- [Fei96] Uriel Feige. A threshold of $\ln n$ for approximating set cover. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 314–318, 1996.
- [FGK⁺98] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. Research Report MPI-I-98-007, Max-Planck Institute for Computer Science, Saarbrücken, 1998.
- [FPT81] Robert J. Fowler, Michael S. Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Information Processing Letters*, 12(3):133–137, 1981.
- [Fre88] Herbert Freeman. An expert system for the automatic placement of names on a geographic map. *Information Sciences*, 45:367–378, 1988.

- [FW91] Michael Formann and Frank Wagner. A packing problem with applications to lettering of maps. In *Proc. 7th Annu. ACM Sympos. Comput. Geom. (SoCG'91)*, pages 281–288, 1991.
- [FW92] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.
- [Hir82] Stephen A. Hirsch. An algorithm for automatic name placement around point data. *The American Cartographer*, 9(1):5–17, 1982.
- [IL97] Claudia Iturriaga and Anna Lubiw. NP-hardness of some map labeling problems. Technical Report CS-97-18, University of Waterloo, Canada, 1997.
- [Imh62] Eduard Imhof. Die Anordnung der Namen in der Karte. In *International Yearbook of Cartography*, pages 93–129, Bonn Bad Godesberg, 1962. Kirschbaum.
- [Imh75] Eduard Imhof. Positioning names on maps. *The American Cartographer*, 2(2):128–144, 1975.
- [IS89] Edward H. Isaaks and R. Mohan Srivastava. *An Introduction to Applied Geostatistics*. Oxford University Press, New York, 1989.
- [Jam96] Michael B. Jampel. *Over-Constrained Systems in CLP and CSP*. PhD thesis, Dept. of Comp. Sci. City University, London, September 1996.
- [JC89] Christopher B. Jones and Anthony C. Cook. Rule-based name placement with Prolog. In *Proc. Auto-Carto 9*, pages 231–240, 1989.
- [JFM96] Michael Jampel, Eugene Freuder, and Michael Maher, editors. *Over-Constrained Systems*. Number 1106 in LNCS. Springer, August 1996.
- [JT98] Joseph R. Jones and Paul Thurrott. *Cascading Style Sheets: a primer*. MIS Press, P. O. Box 5277, Portland, OR 97208-5277, USA, Tel: (503) 282-5215, 1998.
- [Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [Ket98] Lutz Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 146–154, June 1998.
- [KMPS93] Ludek Kučera, Kurt Mehlhorn, Bettina Preis, and Erik Schwarzenacker. Exact algorithms for a geometric packing problem. In *Proc. 10th Sympos. Theoret. Aspects Comput. Sci.*, volume 665

- of *Lecture Notes in Computer Science*, pages 317–322. Springer-Verlag, 1993.
- [Kni98] Lars Knipping. Beschriftung von Linienzügen. Master’s thesis, Fachbereich Mathematik und Informatik, Freie Universität Berlin, November 1998.
- [KR92] Donald E. Knuth and Arvind Raghunathan. The problem of compatible representatives. *SIAM J. Discr. Math.*, 5(3):422–427, 1992.
- [Kra97] Joshua C. Kramer. Line feature label placement for ALPS5.0. unpublished manuscript, available at <http://paul.rutgers.edu/~jckramer/academics/Report/>, 1997.
- [KSY99] Sung Kwon Kim, Chan-Su Shin, and Tae-Cheon Yang. Labeling a rectilinear map with sliding labels. Technical Report HKUST-TCSC-1999-06, Hongkong University of Science and Technology, July 1999.
- [KT98] Konstantinos G. Kakoulis and Ionnis G. Tollis. A unified approach to labeling graphical features. In *Proc. 14th Annu. ACM Sympos. Comput. Geom. (SoCG’98)*, pages 347–356, June 1998.
- [Küh96] Dietmar Kühl. Design patterns for the implementation of graph algorithms. Master’s thesis, Technische Universität Berlin, 1996.
- [KW97] Dietmar Kühl and Karsten Weihe. Data access templates. *C++ Report*, 9(7):18–21, July 1997.
- [Lic82] David Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.
- [Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
- [McC85] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.
- [MF85] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence.*, 25:65–74, 1985.
- [MH86] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.
- [Mil94] William Mills. Practical considerations in name placement: A defence of Pinhas Yoeli. *Cartographica*, 31(4):58–62, 1994.
- [MN92] Kurt Mehlhorn and Stefan Näher. Algorithm design and software libraries: Recent developments in the LEDA project. In Jan van Leeuwen, editor, *Proceedings of the IFIP 12th World Computer*

- Congress. Volume 1: Algorithms, Software, Architecture*, pages 493–508, Amsterdam, The Netherlands, September 1992. Elsevier Science Publishers.
- [Mor80] Joel L. Morrison. Computer technology and cartographic change. In D.R.F. Taylor, editor, *The Computer in Contemporary Cartography*. J. Hopkins Univ. Press, New York, 1980.
- [MS91] Joe Marks and Stuart Shieber. The computational complexity of cartographic label placement. Technical Report TR-05-91, Harvard CS, 1991.
- [MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, MA, 1996.
- [NM90] Stefan Näher and Kurt Mehlhorn. LEDA: A library of efficient data types and algorithms. In *Proc. Internat. Colloq. Automata Lang. Program.*, pages 1–5, 1990.
- [NW96] Marco Nissen and Karsten Weihe. Combining leda with customizable implementations of graph algorithms. Technical Report 17, Fakultät für Mathematik und Informatik, Universität Konstanz, October 1996. ISSN 1430-3558.
- [Ove96] Mark H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 53–58. Springer-Verlag, 1996.
- [Pre93] Bettina Preis. Ein NP-vollständiges Plazierungsproblem. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, February 1993.
- [Pre98] Mike Preuß. Solving map labeling problems by means of evolution strategies. Master's thesis, Fachbereich Informatik, Universität Dortmund, February 1998.
- [PZC98] Chung Keung Poon, Binhai Zhu, and Francis Chin. A polynomial time solution for labeling a rectilinear map. *Information Processing Letters*, 65(4):201–207, 1998.
- [Rai98] Günther Raidl. A genetic algorithm for labeling point features. In *Proc. of the Int. Conference on Imaging Science, Systems, and Technology*, pages 189–196, Las Vegas, NV, July 1998.
- [Rai99] Günther Raidl. An evolutionary approach to point-feature label placement. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, page 807. Morgan Kaufmann, July 1999.

- [Rum98] Wolfgang Rumlmaier. Optimierung von Labelanordnungen mit Genetischen Algorithmen und Simulated Annealing. Master's thesis, Institute of Computer Graphics, Vienna University of Technology, April 1998.
- [Sch95] Erik Schwarzenecker. *Ein NP-schweres Plazierungsproblem*. PhD thesis, Technische Fakultät der Universität des Saarlandes, Saarbrücken, 1995.
- [SFV95] Thomas Schiex, Hélène Fargier, and Gérard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proc. International Joint Conference on Artificial Intelligence*, Montréal, August 1995.
- [SvK99] Tycho Strijk and Marc van Kreveld. Labeling a rectilinear map more efficiently. *Information Processing Letters*, 69(1):25–30, 1999.
- [SW98] Sven Schönherr and Alexander Wolff. MAKEIT! – Generating and maintaining makefiles automatically. In Roberto Battini and Alan A. Bertossi, editors, *Proc. Workshop on Algorithms and Experiments (ALEX98), Trento, Italy*, pages 165–174. Università di Trento, 9–11 February 1998.
- [SZ97] Phil Stephens and Ray Zhang. Archaeologists claim finding world's oldest scaled map. *China News Digest*, 20 November 1997. <http://www.herbaria.harvard.edu/china/focnews/October-97/0011.html>.
- [VA99] Bram Verweij and Karen Aardal. An optimisation algorithm for maximum independent set with applications in map labelling. In *Proc. 7th Annu. Europ. Symp. on Algorithms (ESA '99)*, volume 1643 of *Lecture Notes in Computer Science*, pages 426–437, Prague, 16–18 July 1999. Springer-Verlag.
- [vDTdB99] Steven van Dijk, Dirk Thierens, and Marc de Berg. On the design of genetic algorithms for geographical applications. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, pages 188–195. Morgan Kaufmann, July 1999.
- [vDvKSW99] Steven van Dijk, Marc van Kreveld, Tycho Strijk, and Alexander Wolff. Towards an evaluation of quality for label placement methods. In *Proceedings of the 19th International Cartographic Conference (ICA '99)*, pages 905–913, Ottawa, 14–21 August 1999. Int. Cartographic Association.
- [vKSW98] Marc van Kreveld, Tycho Strijk, and Alexander Wolff. Point set labeling with sliding labels. In *Proc. 14th Annu. ACM Sympos. Comput. Geom. (SoCG'98)*, pages 337–346, 7–10 June 1998.

- [vKSW99] Marc van Kreveld, Tycho Strijk, and Alexander Wolff. Point labeling with sliding labels. *Computational Geometry: Theory and Applications*, 13:21–47, 1999.
- [vR89] Jan W. van Roessel. An algorithm for locating candidate labeling boxes within a polygon. *The American Cartographer*, 16(3):201–209, 1989.
- [VWS97] Oleg Verner, Roger Wainwright, and Dale Schoenefeld. Placing text labels on maps and diagrams using genetic algorithms with masking. *INFORMS Journal on Computing*, 9(3):266–275, 1997.
- [Wag94] Frank Wagner. Approximate map labeling is in $\Omega(n \log n)$. *Information Processing Letters*, 52(3):161–165, 1994.
- [WB91] Chyan Victor Wu and Barbara Pfeil Buttenfield. Reconsidering rules for point-feature name placement. *Cartographica*, 28(1):10–27, 1991.
- [Wei97] Karsten Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, volume 32, 10 of *ACM SIGPLAN Notices*, pages 34–48, New York, October 1997. ACM Press.
- [Wei98] Karsten Weihe. Using templates to improve C++ designs. *C++ Report*, 10(2):14–21, 1998.
- [Wil73] W.T. Wilkie. Computerized cartographic name processing. Master’s thesis, Department of Electrical Engineering, University of Saskatchewan, Canada, 1973.
- [WKvK⁺99] Alexander Wolff, Lars Knipping, Marc van Kreveld, Tycho Strijk, and Pankaj K. Agarwal. A simple and efficient algorithm for high-quality line labeling. In David Martin and Fulong Wu, editors, *Proc. GIS Research UK 7th Annual Conference (GISRUK’99)*, pages 146–150, Southampton, 14–16 April 1999. Department of Geography, University of Southampton.
- [WS96] Alexander Wolff and Tycho Strijk. A map labeling bibliography. <http://www.math-inf.uni-greifswald.de/map-labeling/bibliography/>, 1996.
- [WW97] Frank Wagner and Alexander Wolff. A practical map labeling algorithm. *Computational Geometry: Theory and Applications*, 7:387–404, 1997.
- [WW98] Frank Wagner and Alexander Wolff. A combinatorial framework for map labeling. In Sue H. Whitesides, editor, *Proceedings of the Symposium on Graph Drawing (GD’98)*, volume 1547 of *Lecture*

- Notes in Computer Science*, pages 316–331. Springer-Verlag, 13–15 August 1998.
- [Yoe72] Pinhas Yoeli. The logic of automated map lettering. *The Cartographic Journal*, 9:99–108, 1972.
- [Zor86] Steven Zoraster. Integer programming applied to the map label placement problem. *Cartographica*, 23(3):16–27, 1986.
- [Zor90] Steven Zoraster. The solution of large 0-1 integer programming problems encountered in automated cartography. *Operations Research*, 38(5):752–759, 1990.
- [Zor97] Steven Zoraster. Practical results using simulated annealing for point feature label placement. *Cartography and GIS*, 24(4):228–238, 1997.