The NP-Hardness of Covering Points with Lines, Paths and Tours and their Tractability with FPT-Algorithms

by

Apichat Heednacram
B. Eng. (1^{st} Class Hons), Griffith University, 2001

Submitted in fulfilment of the requirements of the degree of
Doctor of Philosophy

Institute for Integrated and Intelligent Systems
Science, Environment, Engineering and Technology
Griffith University

March 2010

Abstract

Given a problem for which no polynomial-time algorithm is likely to exist, we investigate how to attack this seemingly intractable problem based on parameterized complexity theory. We study hard geometric problems, and show that they are fixed-parameter tractable (FPT) given an instance and a parameter k. This allows the problems to be solved exactly, rather than approximately, in polynomial time in the size of the input and exponential time in the parameter.

Although the parameterized approach is still young, in recent years, there have been many results published concerning graph problems and databases. However, not many earlier results apply the parameterized approach in the field of computational geometry. This thesis, therefore, focuses on geometric NP-hard problems. These problems are the Line Cover problem, the Rectilinear Line Cover problem in higher dimensions, the Rectilinear Minimum-Links Spanning Path problem in higher dimensions, the Rectilinear Hyperplane Cover problem, the Minimum-Bends Traveling Salesman Problem and the Rectilinear Minimum-Bends Traveling Salesman Problem, in addition to some other variants of these problems.

The Rectilinear Minimum-Links Spanning Path problem in higher dimensions and the Rectilinear Hyperplane Cover problem had been the subject of only conjectures about their intractability. Therefore, we present the NP-completeness proofs for these problems. After verifying their hardness, we use the fixed-parameter approach to solve the two problems.

We focus on solving the decision version of the problems, rather than solving the optimizations. However, with the Line Cover problem we demonstrate that it is not difficult to adapt algorithms for the decision version to algorithms for the optimization version. We also implement several algorithms for the Line Cover problem and conduct experimental evaluations of our algorithms with respect to previously known algorithms. For the remaining problems in the thesis, we will establish only the fundamental results. That is, we determine fixed-parameter tractability of those problems.

Statement of Originality

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

(Signed) A. Heednacram

Apichat Heednacram

Approval

Name:	Apichat Heednacram
Student No:	964128
Degree:	Doctor of Philosophy
Thesis Title:	The NP-Hardness of Covering Points with Lines, Paths and Tours and their Tractability with FPT-Algorithms
Submission Date:	01 March 2010

Principal Supervisor: Professor Vladimir Estivill-Castro
School of Information and Communication Technology, Griffith University
Australia

Principal Supervisor: Dr. Francis Suraweera
School of Information and Communication Technology, Griffith University
Australia

Acknowledgements

I would like to thank the Australian Government, Department of Education, Employment and Workplace Relations (DEEWR) for awarding me the Endeavour Postgraduate Award. The scholarship was generously supplied for three years, thus enabling me to enjoy my research with sufficient financial support.

I express my sincere gratitude to my two supervisors, Dr Francis Suraweera and Professor Vladimir Estivill-Castro. Without their support, this thesis would not have been possible. I cannot thank them enough for their guidance over the years, in particular, their commitment and time in making weekly discussions extremely useful.

I would also like to acknowledge Professor Vladimir Estivill-Castro for his financial support for my travels overseas and for the many opportunities he has provided. Thanks for providing the sponsorship to conferences such as ISAAC08, CATS08, CATS09 and particularly, a visit to the Yahoo Research and the Barcelona Media Innovation Centre (FBM-UPF) at the Universitat Pompeu Fabra (UPF) in Barcelona, Spain. Thank you Professor Josep Blat, Director of Departament de Tecnologia for providing a working space for several weeks at the UPF. I also thank Professor Abdul Sattar, Director of the Institute for Integrated and Intelligent Systems (IIIS) for helping me on several occasions with the VISA application and travel-documents. I would like to thank the Enterprise Information Infrastructure (EII) for providing sponsorship to the Theoretical Computer Science Day in Sydney and the EII PhD School. Thanks also to Kathleen Williamson, The University of Queensland, for making each trip as convenient as possible.

Thanks to Mike Fellows and Frances Rosamond for the few meetings we had and thank you for keeping me updated about the FPT news. I would like to thank Rod Downey for providing some interesting papers during the ACSW2009 in New Zealand. I thank Joel Fenwick and Artak Amirbekyan for helping me with the initial stages of the PhD. Thank you Mahdi Parsa, my fellow PhD student, who shares the same passion about parameterized complexity theory. Finally, thanks to my family and friends for their unconditional love.

To Lord Buddha who told us that our innate wisdom and virtuous abilities are not truly lost, just not yet uncovered.

List of Outcomes Arising from this Thesis

International journals with papers fully refereed

1. V. Estivill-Castro, A. Heednacram, and F. Suraweera. Reduction Rules Deliver Efficient FPT-Algorithms for Covering Points with Lines. *ACM Journal of Experimental Algorithmics*, 14:1.7–1.26, November 2009.

2. V. Estivill-Castro, A. Heednacram, and F. Suraweera. NP-completeness and FPT Results for Rectilinear Covering Problems. *Journal of Universal Computer Science*, 16(5):622–652, May 2010.

3. V. Estivill-Castro, A. Heednacram, and F. Suraweera. FPT-algorithms for Minimum-Bends Tours. *International Journal of Computational Geometry and Applications*, August 2010 (Accepted with minor corrections).

International conferences with papers fully refereed

1. V. Estivill-Castro, A. Heednacram, and F. Suraweera. The Rectilinear k-Bends TSP. In, M. T. Thai and S. Sahni, editors, *Proceedings of the 16th Annual International Combinatorics Conference (COCOON)*, volume 6196 of Lecture Notes in Computer Science, pages 264–277. Springer, Berlin, July 19–21, 2010.

Notation

S — a set of input points.

n — the size of S.

S' — a subset of S; that is, $S' \subseteq S$.

n' — the size of S' where $n' \leq n$.

P — the class of polynomial-time solvable problems.

Q — a parameterized problem.

k — a positive integer and a parameter of parameterized problems.

d — the number of dimensions.

ϕ — the number of orientations.

L_3 — the set of all lines that cover at least 3 points in S.

$L_{\lceil n/k \rceil}$ — the set of all lines that cover at least $\lceil n/k \rceil$ points in S.

L_{k+1} — the set of all lines that cover at least $k+1$ points in S.

$|L|$ — the cardinality of a set L.

$cover(L)$ — all points in S covered by some line in L.

$lines(T)$ — the set of lines used by the line-segments in a tour T.

$\overline{p_1, p_2}$ — the line through points p_1 and p_2.

$\overline{p_1 p_2}$ — the line-segment between points p_1 and p_2.

$d(p_1, p_2)$ — the distance from point p_1 to p_2.

$\mathcal{A}(L)$ — the arrangement of lines induced by L.

$dual(p)$ — a line in dual space to the point p in primal space.

$\mathbb{R}^d$ — the real coordinate space of d-dimensions.

$\Pi(x, *, *)$ — the yz-plane.

$\Pi(*, y, *)$ — the xz-plane.

$\Pi(*, *, z)$ — the xy-plane.

Glossary of Terms

Confidence Interval: A confidence interval indicates that the expected running time of the algorithm on an instance has 95% probability of falling inside the interval. A confidence interval is defined by the following formula: 95%C.I. $= M \pm (1.96 * SD/\sqrt{N})$ where M is the sample mean, SD is the standard deviation, and N is the number of samples. If the confidence intervals are disjoint, then we have statistical significance that one algorithm's CPU time is lesser than the other.

Direction: We use this term for the direction of travel of a line-segment in a tour; for example, a vertical line-segment can be traveled North-South or in the opposite direction, South-North.

Dual Space: The dual-space mapping assigns a line l given by $y = mx + b$ in a 2-dimensional space to the point $p_l = (m, -b)$, and a point $p = (p_x, p_y)$ to a line l_p given by $y = p_x x - p_y$. This mapping has the property that two lines l_p and l_q in dual space intersect at a point p_l, which, in primal space, is the line l through the points p and q that are images of the two lines.

Fan-out: The number of children nodes in a parent node in a tree.

FPT: This denotes the class of fixed-parameter tractable parameterized problems.

Hard Problems: The problems for which no polynomial-time algorithms are known. In this thesis, the problems discussed are in the class of NP-hard.

Hyperplane: A generalization of the concept of a plane in geometry into a larger number of dimensions.

Line: An unbounded infinite set of points with zero-width that is convex and contains the shortest path between any of two points in it. Alternatively, a linear vector space of dimension 1.

Line-segment: The bounded portion of a line that constitutes the shortest path between the extremes of the segment.

Line Cover: A line cover is a set of lines that cover the points in S.

Link Length: The number of line-segments in a given path.

Orientation: The slope of a line with respect to the x-axis and, when applied to a line segment, it refers to the orientation of the line hosting it.

Plane: A two-dimensional subspace in a three-dimensional space.

Recilinear Line: An axis-parallel line (in two dimensions, this line is either horizontal or vertical).

Recilinear Tour: A tour that visits a set of input points and is made up of rectilinear lines. In this thesis, it refers to a solution of the Rectilinear k-Bends Traveling Salesman Problem.

Spanning Path: A path that passes through a set of input points.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the fields of mathematics and computer science, there are some problems that cannot be solved practically by any computer when the input size n of these problems becomes large. The exponential function 2^n and the factorial function $n!$, for example, it would take about 3.5 years and 9.6×10^{44} years for a computer making one trillion (10^{12}) operations per second to execute 2^{50} and 50! operations, respectively. The problems that cannot be solved in polynomial time are called *intractable* or *hard* problems; they have been around for many decades [Lev07, p. 393], [Nie06, p. 53]. To classify these problems by their degree of hardness, mathematicians have developed the field of computational complexity theory.

The theory of computational complexity seeks to classify problems according to their inherent difficulty. The scientific community categorizes these problems into complexity classes. The complexity class P is the set of decision problems that can be solved by a deterministic machine in polynomial time. The complexity class NP is the set of decision problems that can be solved by a non-deterministic machine in polynomial time [GJ79, p. 23-31].[1] In an equivalent definition, the class NP consists of those problems that are verifiable in polynomial time. That is, given a certificate of a solution, one could verify that the certificate is correct in polynomial time in the size of the problem's input.

[1] This is less formal than the definition in the source, but it will suffice for our purposes.

The most important open question in complexity theory is whether the complexity class P is the same as the complexity class NP [GJ79, CLRS01, Lev07]. It was first posed in 1971, but so far, nobody has been successful in finding a mathematical proof of P $\neq$ NP [CLRS01, p. 966]. Figure 1.1 depicts the two alternatives to this open question. The discovery of a polynomial-time algorithm for any of the NP-complete problems would imply that P = NP. However, after

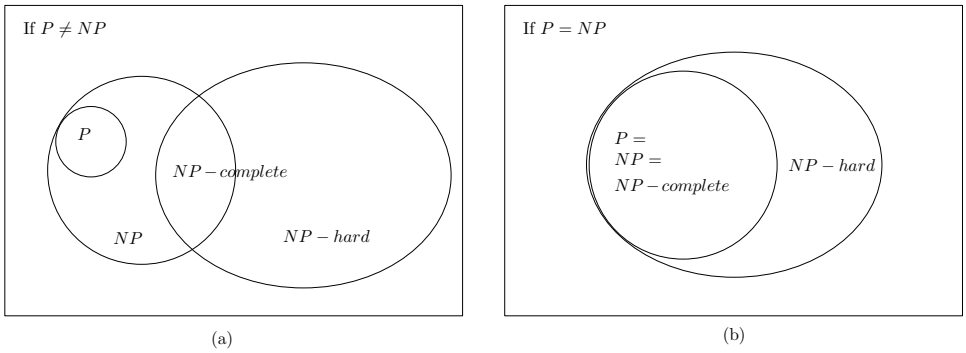


Figure 1.1: Diagram for P, NP, NP-complete and NP-hard problems [GJ79].

nearly four decades of research have passed, Fortnow argues that it is not likely for the P versus NP problem to be resolved in the near future [For09].

Although some problems are difficult to solve, they are very important in many fields of applications and we cannot just forget or do nothing. Many NP-hard problems, problems that are at least as hard as the hardest problems in NP, arise in diverse domains such as, Boolean logic, graphs, sets and partitions, storage and retrieval, sequencing and scheduling, network design, mathematical programming, algebra and number theory, computational geometry, games and puzzles, automata and language theory, program optimization, biology, chemistry and physics [Lev07, CLRS01]. Hence, if we can solve some of these problems or demonstrate how we can obtain algorithms for these problems, it will be beneficial for researchers and programmers who require such algorithms to solve their problems.

In the past, there were well known techniques for coping with NP-hard problems, for example, approximation algorithms, randomized algorithms, heuristic algorithms and so on. In the 1990s, fixed-parameter algorithms were introduced as an alternative for solving hard combinatorial problems [DF99]. It is a

new and fast developing field of research with many options for exploration, as claimed by Niedermeier, the author of the book "Invitation to Fixed-Parameter Algorithms" [Nie06]. There are numerous examples where fixed-parameter techniques have proved themselves to be quite useful in coping with the hardness of NP-hard problems. One popular example of NP-hard problems is the Vertex Cover problem. Niedermeier [Nie06] points out that the k-Vertex Cover (finding a vertex cover of size k) has an algorithm with a running time $O(2^k n)$ following directly from the search tree method described in Mehlhorn's 1984 textbook [Meh84b, p. 216]. Hence, this problem is fixed-parameter tractable (FPT). Establishing that a problem is fixed-parameter tractable has inspired researchers [CKJ01, CKX06] to deliver several improvements that frequently result in practical methods. For example, in 2006, Chen et al. provided an $O(kn + 1.2738^k)$ time algorithm to solve the Vertex Cover problem [CKX06]. This now enables implementations that solve this for problem instances with millions of vertices and covers of several hundreds.

In this thesis, we use the fixed-parameter approach to design FPT-algorithms for hard problems in computational geometry. Computational geometry is the branch of computer science that studies algorithms design and analysis for solving geometric problems. It has applications in a very large scale integration (VLSI) design, geographic information systems (GIS), image processing, computer graphics, statistics and robotics [dBvKOS00]. We concentrate on geometric problems because the survey of Giannopoulos et al. [GKW08] shows that there are only a few results regarding fixed-parameter tractability in computational geometry. Moreover, Venkatesh Raman reports in the latest issue of the "Parameterized Complexity Newsletter" [Ros09] that there seems to be little research on parameterized techniques for geometric problems. Hence, we have taken up the challenge to "push the envelope" of parameterized complexity when applied to geometric problems.

1.2 Research Aims

This thesis intends to solve some hard geometric problems, especially those that have a connection with the Traveling Salesman Problem and its geometric variants. Our approach is to use fixed-parameter complexity. Our research considers an instance of a hard problem that also includes a positive integer k,

besides the size of the input n. We obtain fixed-parameter algorithms because a problem instance is now seen in light of these two variables. Two central problems addressed here had only been the subject of conjectures about their intractability. We will take this opportunity to, firstly, present their hardness results, in particular, the corresponding NP-completeness proofs. Then we will design fixed-parameter algorithms that deliver the exact solutions (not merely the approximate ones). One of our selling points is also the simplicity of the techniques used to prove fixed-parameter tractability for hard problems. Although each problem is unique, they are all linked because they are about covering points with lines, paths or tours optimizing lines (or line-segments). In several cases, we show how to progress from the solution of a problem to the next related problem. The hard geometric problems that will be investigated in this research are the Line Cover problem, the Rectilinear Line Cover problem in higher dimensions, the Rectilinear Minimum-Links Spanning Path problem in higher dimensions, the Rectilinear Hyperplane Cover problem, the Minimum-Bends Traveling Salesman Problem and the Rectilinear Minimum-Bends Traveling Salesman Problem, in addition to some other variants of these problems.

1.3 Related Research

Although approximation algorithms have proved useful in coping with the NP-hard problems [SW01, Lev07, CLRS01], they do not guarantee optimal solutions. Alternatively, we may use exact algorithms to solve NP-hard problems to optimality, but exact algorithms [Woe03] often come with a large complexity. The field of parameterized complexity has, essentially, identified two types of exact algorithms — those exact algorithms where the exponential explosion on input size is seemingly unavoidable and contrastingly, those exact algorithms where the exponential complexity can be narrowed to a small parameter (see Figure 1.2). The problems that require exponential time in the size of the input are informally referred to as a class XP.

The theory of parameterized complexity was developed in the 1990s by Rod Downey and Michael Fellows. Their 1999 monograph [DF99] presented an introduction to the field. Since then, there have been many new results in the field of exactly-solving hard combinatorial problems. Parameterized complexity

theory provides methods for proving problems to be in the fixed-parameter tractable class. The fixed-parameter approach can lead to algorithms that are both efficient and guaranteed to find optimal solutions.

Figure 1.2: Parameterized complexity attempts to confine the combinatorial explosion [DF99].

Niedermeier's paper [Nie04] on ubiquitous-parameterization provides a good introduction to this rewarding field of fixed-parameter algorithms. Sloper has shown how parameterized complexity and techniques in parameterized algorithm design can be used to solve various graph problems [Slo01, Slo05]. The techniques used include Win/Win, Bounded-Search-Tree, Greedy Localization, Crown Decomposition, Modeled Crown Decomposition, Extremal Method and Reduction Rules. McCartin presented both algorithmic aspects and structural aspects of parameterized complexity [McC03]. She introduced a general framework for parameterized counting problems, extending the framework developed by Downey and Fellows for decision problems. Saurabh developed the technique in designing exact algorithms for both parameterized and optimization versions of the graph problems [Sau07]. He gave the fastest known algorithms for a number of NP-hard problems.

Focusing on solving NP-hard problems in practice, Hüffner, Niedermeier and Wernicke [HNW08] surveyed three main techniques for developing practical fixed-parameter algorithms, namely kernelization (data reduction with provable performance guarantee), bounded-search-tree and a new technique called iterative compression. Prieto demonstrated the use of the method of extremal structure to design FPT-algorithms [Pri05]. Here, several NP-complete problems were effectively solved by the method of extremal structure that operates on two main lemmas, namely, the boundary lemma and the kernelization lemma. The ideas that constitute the core of the method of extremal structure appeared in 2000 in a paper by Fellows, McCartin, Rosamond and Stege [FMRS00].

Their approach is called the method of coordinatized kernels. This method was based on kernelization techniques and extremal combinatorics. Hüffner also suggested various techniques for designing fixed-parameter algorithms for hard graph problems [HÜ07]. Hüffner's thesis shows that the concept of fixed-parameter tractability and algorithmic techniques, whose development was driven by this concept, can lead to practical and useful programs for solving real-world problems exactly. Hüffner's thesis involved not only the design of the fixed-parameter algorithms but also the implementation and experimental evaluation of the algorithms. Therefore, we conclude that parameterized complexity has expanded the approaches to deal and tackle hard problems.

In this thesis, we focus on developing exact algorithms for geometric problems, especially those that are NP-hard. We believe that once an exact solution for one problem is established, it will produce an impact on several other related problems in the field. Hence, we also advance the field of computational geometry in this way. To give an example, the Line Cover problem, or the problem of covering a set S of n points in the plane with the minimum number of straight line-segments possible, is relevant to other problems such as the Minimum-Bends Traveling Salesman Problem [SW01], the N-line traveling salesman problem [Rot92], and the convex-hull and k lines traveling salesman problem [DvDR94, DW96]. The Line Cover problem, has been known to be NP-hard [MT82] for over 20 years. Langerman and Morin proved [LM01, LM02, LM05] that this problem is fixed-parameter tractable and produced two exact algorithms for covering n points with k lines. Grantson and Levcopoulos then used these results to obtain the approximation and exact algorithms for covering n points with the least number of lines. There are a number of problems that require efficient solutions for covering points with the least number of lines. The Minimum-Bends Traveling Salesman Problem is a problem where, given a set of points in the plane, we want to find a tour through the points, consisting of the least number of straight lines. The Minimum-Bends Traveling Salesman Problem is significant in computer science, because it has many applications in the design of VLSI, the movement of heavy machinery, milling out an area, mowing a lawn, surveying an area by aircraft and several other applications where bends are considered very costly and one want to minimize the number of those bends. We note that although there exists approximation algorithms [SW01, Wag06] for the Minimum-Bends

Traveling Salesman Problem, there has been little progress in exact algorithms for these problems. Therefore, one of the aims for this research is to come up with exact algorithms and establish the fixed-parameter tractability for these problems.

1.4 Methodology

For most research in the design of algorithms, one establish the claims mathematically, and thus the methodology is the logic derivation of a mathematical proof. However, when designing a new or improved algorithm that is fixed-parameter tractable, several tools are available. Fixed-parameter algorithmic methods that have appeared in the literature [Nie06, Pri05, HÖ7] are a method of reduction to a problem kernel, a method of extremal structure, bounded-search-tree, dynamic programming, iterative compression, color-coding, tree decompositions of graphs, and so on.

In this research, we use the method of reduction to a problem kernel and the bounded-search-tree technique to prove fixed-parameter tractability results for several NP-hard problems. Note that the method of reduction to a problem kernel is also known as *kernelization*. For example, we use the kernelization approach with reduction rules to design our FPT-algorithms for the Line Cover problem. We use the bounded-search-tree technique to solve the Rectilinear Line Cover, the Rectilinear Minimum-Links Spanning Path Problem and the Rectilinear Hyperplane Cover in higher dimensions. We also use kernelization in combination with the bounded-search-tree technique to prove the correctness of the Minimum-Bends Traveling Salesman Problem.

Note that both the classical theory of computation and the theory of parameterized complexity focus on the decision problems rather than the optimization problems[2]. Here, we present two compelling reasons why this has been the case. First, a negative result for the decision problems would imply a negative result for the optimization problems. Second, if we are able to establish the complexity results for the decision problems, then we acquire fundamental results that

[2]A decision problem is a problem where the answer is always YES or NO. An optimization problem is one in which feasible solutions have an associated value that must be minimized or maximized [CLRS01, p. 972].

will further advance our knowledge. Consequently, we will follow the standard approach and focus on the decision problems. However, we are aware that, in practice, what is common is dealing with the optimization problems. We demonstrate both versions (the decision problems and the optimization problems) in Chapter 2 as examples. It is important to note that for the other chapters we intend to establish the fundamental results only.

1.5 Basic Concepts and Definitions

In this section, we explain in more detail the concepts of computational complexity theory in addition to parameterized complexity theory. Also, we provide a formal introduction to the Traveling Salesman Problem and the relevant geometric definitions in use throughout the thesis.

1.5.1 Computational Complexity Theory

Earlier, we have seen that the complexity class P is contained in NP. However, NP contains many important problems, called NP-complete problems, for which no single polynomial-time algorithm is known. To define what an NP-complete problem is, we must first explain the notion of polynomial-time reductions.

Definition 1 (Polynomial-time Reductions). *A decision problem D_1 is said to be polynomially reducible to a decision problem D_2 if there exists a function f that transforms instances of D_1 to instances of D_2 such that*

1. *f maps all YES-instances of D_1 to YES-instances of D_2 and all NO-instances of D_1 to NO-instances of D_2.*
2. *f is computable by a polynomial-time algorithm [GJ79, p. 35].*

Definition 2 (NP-complete). *A decision problem D is NP-complete if:*

1. *D is in NP, and*
2. *Every problem in NP is polynomially reducible to D [GJ79, p. 37].*

To prove that an NP problem D is, in fact, an NP-complete problem, it is sufficient to show that an already-known NP-complete problem reduces to D. A problem satisfying Property 2 above, but not necessarily Property 1, is said to

be NP-hard. A consequence of this definition is that if we had a polynomial-time algorithm for D, we could solve all problems in NP in polynomial time.

Definition 3 (NP-hard). *A problem D_2 is NP-hard if, for some NP-complete problem D_1, D_1 is polynomial-time reducible to D_2 [GJ79, p. 113].*[3]

NP-hard problems are not all NP, despite having NP as the prefix in their class name. Examples of well-known NP-hard problems are:

- the Hamiltonian cycle problem,
- the Traveling salesman problem,
- the Knapsack problem,
- the Subset sum problem,
- the Vertex cover problem and
- the Independent set problem.

The above well-known problems have been studied thoroughly for the past 40 years. Note that optimization problems that are NP-hard can easily be restated in terms of a decision version. As an example, consider the optimization question, "what is the shortest tour?". This question is NP-hard. However, this problem can be restated as the decision problem, "is there a tour-length less than k?". This problem is NP-complete [GJ79, p. 114]. Since polynomial-time algorithms providing exact answers are not known for NP-complete problems, sometimes an approximation is satisfactory.

Definition 4 (Approximation Algorithm). *An approximation algorithm is an algorithm that returns near-optimal solutions. If such an algorithm achieves an approximation ratio of $\rho(n)$, then it is called a $\rho(n)$-approximation algorithm, where the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution. That is,*

$$max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \quad \leq \quad \rho(n)$$

[CLRS01, p. 1022].

[3]The notion of an NP-hard problem in the source is defined more formally by using a special type of reduction called a polynomial-time Turing reduction. NP-hard problems may be decision problems, search problems or optimization problems [GJ79, p. 114].

The definition above applies for both minimization and maximization problems. For a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is greater than the cost of an optimal solution. For a maximization problem, $0 < C \leq C^*$, the ratio C^*/C gives the factor by which the cost of the approximate solution is less than the cost of an optimal solution. Some researchers felt that it should be feasible to obtain solutions that are as close as possible to the optimum, with additional cost. The suggestion was an approximation scheme where the computational requirements are proportional to the quality of the approximation. This notion is actually formalized as follows:

Definition 5 (Approximation Scheme). *An approximation scheme is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1+\epsilon)$-approximation algorithm [CLRS01, p. 1023].*

Definition 6 (PTAS). *An approximation scheme is a polynomial-time approximation scheme (PTAS) if, for every $\epsilon > 0$, there is a polynomial-time $(1+\epsilon)$-approximation algorithm and the algorithm runs in $O(n^{f(1/\epsilon)})$ time for some arbitrary function f and the size of the input n [Mar08].*

Using PTAS with a smaller ϵ may result in more computation time. That is, there is a trade-off between the computation time and the quality of the approximation.

Definition 7 (EPTAS). *An approximation scheme is an efficient polynomial-time approximation scheme (EPTAS) if it is a PTAS with a running time of the form $O(f(1/\epsilon)n^{O(1)})$ [Mar08].*

Definition 8 (FPTAS). *An approximation scheme is a fully polynomial-time approximation scheme (FPTAS) if it is an EPTAS and the algorithm runs in $O((1/\epsilon)^{O(1)}n^{O(1)})$ time [Mar08].*

Note that there is a connection between PTAS and fixed-parameter tractability [Mar08, Mar05]. We will discuss this later in the thesis, once we have introduced the notion of fixed-parameter tractability. Sometimes NP-complete problems are also challenging by the fact that they may not even afford approximation algorithms. This is formalized with the definitions of APX, APX-hard [MS00, KMSV98].

Definition 9 (APX). *The class APX is the set of NP optimization problems such that, for some $\epsilon > 1$, there exists a polynomial-time ϵ-approximation algorithm for that problem [MS00].*

Definition 10 (APX-hard). *A problem is APX-hard if there is a PTAS reduction from every problem in APX to that problem [MS00].*

In this research, we are interested in hard geometric problems and the problems in connection with the Traveling Salesman Problem. Therefore, we provide a brief overview of the Traveling Salesman Problem, its variations and its computational complexity. Note that problems like the Euclidean Traveling Salesman Problem potentially deal with real data of coordinates of points in the plane. However, Turing machines, one of the models of computation many computer scientists are most familiar with, are not real RAM (Random Access Machine). The reader should be cautious that the main model of computation in computational geometry is real RAM, the model that is equipped with real-valued arithmetic operations [GKW08, PS85]. Brattka and Hertling [BH98] described a theoretical implementation of a real RAM based on a Turing machine. Their modified real RAM model can express exactly the computational power of Turing machines on real numbers. It is a research topic of its own right to investigate how to extend complexity theory to accommodate other models of computation. An introduction to this area is for instance the book by Weihrauch [Wei00].

1.5.2 Traveling Salesman Problem: A Formal Introduction

Given a set of cities and the cost of travel between each pair of cities, the Traveling Salesman Problem (TSP) is to find the cheapest way of visiting each city exactly once and then returning to the starting city. The research on the TSP and its variants has a long history dating back to the 1800s and the Irish mathematician, Sir William Rowan Hamilton. Since then, it has become one of the most studied problems in algorithms, operations research, optimization, and computational complexity [ABCC06, GP07].

The TSP is closely related to the Hamiltonian Cycle Problem in an undirected graph. The Hamiltonian Cycle Problem asks for a tour that

visits each vertex exactly once and finishes at the starting vertex. This problem is NP-complete and thus, the TSP is NP-complete because we can reduce the Hamiltonian Cycle Problem to the TSP [GJ79, p. 211].

The Symmetric Traveling Salesman Problem (STSP) is when the cost of traveling between two nodes in the TSP network is the same in both directions. The case where the cost of traveling from city i to city j is not the same as the cost from city j to city i is called the Asymmetric Traveling Salesman Problem (ATSP). The STSP can be viewed as a special case of the ATSP. We quote the following mathematical definition of the standard TSP or the STSP.

Definition 11 (TSP [GP07, p. 754]). *Let $G = (V, E)$ be a complete graph with a prescribed cost c_e for each edge $e \in E$. Then the problem is to find a least cost hamiltonian cycle in G. This problem is characterized by the triplet $(I, \mathbb{F}, f)$ where*

1. *$I = \{G = (V, E) : G$ is a complete graph with prescribed cost c_e for each $e \in E\}$;*
2. *$\mathbb{F}(G) =$ Set of all hamiltonian cycles in G;*
3. *For all $H \in \mathbb{F}(G)$, we have $f(G, H) = \sum_{e \in H} c_e$.*

The optimization problem is NP-hard; however, the decision problem (given the costs and a number k, decide whether there is a round-trip route cheaper than k) is NP-complete [GP07, p. 6], [GJ79].

The Euclidean Traveling Salesman Problem (ETSP) is when, given n nodes in $\mathbb{R}^2$ (more generally, in $\mathbb{R}^d$), we desire the minimum cost salesman tour for these nodes, where the cost of the edge between node (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Papadimitriou [Pap77] proved that the ETSP is NP-complete.

The Multiple Traveling Salesmen Problem [KB06, SMT99, SH73] is the same as the standard TSP, except that we have more than one salesman. Given m salesmen and n cities, the problem is to visit every city exactly once (except the home city) with exactly one salesman, so that the total distance traveled by all the salesmen is the minimum. The MTSP is NP-complete because

it includes the TSP as a special case. It is suggested [SH73] that the MTSP is appropriate for the problem of bank messenger scheduling, where a crew of messengers pick up deposits from branch banks and return them to the central office for processing. Several techniques are used to solve the MTSP, such as integer linear programming formulations [KB06], neural networks [SMT99], branch and bound [SH73]. For more variants on TSP, we recommend the reader to the book "The Traveling Salesman Problem and Its Variations" [GP07].

We now look at the computational complexity of the TSP. The most direct approach to solve the TSP optimally would be to try all the permutations and see which one is the cheapest. Each possible tour is a permutation of $1, 2, 3 \ldots n$, where n is the number of cities; therefore, the number of tours is $n!$. This solution rapidly becomes impractical. Using the techniques of dynamic programming, Held and Karp in 1962 solve the problem in time $O(n^2 2^n))$ [HK62]. Although this is exponential, it is still the best-known guarantee on the running time of a general solution method for the problem and has survived over 40 years of challenges [ABCC06, p. 50]. In the TSP with triangle inequality and the distance measured is symmetric, Christofides' algorithm [Chr76] provided a TSP tour of at most 1.5 times the optimal. This is believed to be the best-known approximation algorithm for TSP [Aro96]. For the special case of the TSP with distances one and two, the best-known polynomial-time approximation algorithm finds a tour of length at most 1/7 more than optimal [BK06].

1.5.3 Parameterized Complexity Theory

There are several hard problems that require exponential runtime when the complexity is measured in terms of the input size only, but they can be computed in a polynomial runtime in the input size and exponential in a (small) parameter, k. Hence, if the exponential term depends only on the parameter k and the value of k is small, such problems are still tractable. Parameterized complexity theory measures complexity not only in terms of the input size, but additionally in terms of a parameter k that is a numerical value that may depend on the input in an arbitrary way [FG06, p. 4-5].

Parameterized Problems

Many problems have (or can be re-written in) the following form, namely, given an instance I and a small positive integer k, does I have some property that depends on k? Consider three illustrative examples of basic graph problems below.

Vertex Cover
Instance: A graph $G = (V, E)$ and a positive integer k.
Parameter: k.
Question: Is there a subset of vertices $V' \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in V'?

Independent Set
Instance: A graph $G = (V, E)$ and a positive integer k.
Parameter: k.
Question: Is there a subset of vertices $V' \subseteq V$ with k or more vertices that form an independent set, that is, there are no two vertices adjacent in V'?

Dominating Set
Instance: A graph $G = (V, E)$ and a positive integer k.
Parameter: k.
Question: Is there a subset of vertices $V' \subseteq V$ with k or fewer vertices such that every vertex $v \in V$ is contained in V' or has at least one neighbor in V'?

Figure 1.3 illustrates instances and their corresponding optimal solution sets marked by dark shading (opposite to the white shading of the remaining vertices). Since these problems are NP-complete and in XP, we aim for an algorithm that is exponential only in k, and not in the input size. Such problems are classical examples of parameterized problems.

Definition 12 (Parameterized Problem). *A parameterized problem is a problem whose input also includes a positive integer that measures some aspects*

of the input. Thus, inputs are pairs (I, k) *where* I *is an instance of the problem and* k *is a positive integer with* $k \leq |I|$.[4]

In this thesis, we consider parameterized decision problems. All problems have a non-negative integer as parameter.

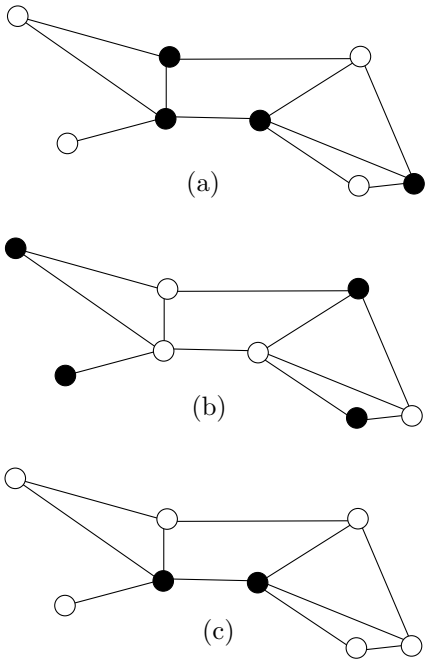


Figure 1.3: The examples of three parameterized problems, (a) Vertex Cover, (b) Independent Set and, (c) Dominating Set [Nie06].

Fixed-Parameter Tractability

The central notion of parameterized complexity theory is fixed-parameter tractability. Intuitively, a parameterized problem Q is fixed-parameter tractable if the complexity to solve each instance (I, k) is of the form $f(k) \cdot n^{O(1)}$, where n is the size of the input I, k is the parameter and f is an arbitrary function depending only on k. When this notion is made formal and languages are

[4] A more formal definition can be provided using the standard method for identifying a decision problem with languages, and mapping the issue of solving decision problems to the issue of the recognition of languages [Nie06, p. 23].

identified with decision problem, the class FPT denotes the corresponding complexity class [Nie06, p. 23]. For the purpose of this thesis, we do not require such formality.

One of the interesting and fundamental properties of fixed-parameter tractability is that the definition is unchanged if we replace $f(k) \cdot n^{O(1)}$ with $f(k) + n^{O(1)}$. The proof of this can be derived from the inequality $a \cdot b \leq a^2 + b^2$ [FG06, p. 26]. An algorithm that runs in time $f(k) \cdot n^{O(1)}$ or $f(k) + n^{O(1)}$ in order to solve a given problem Q, is called a *fixed-parameter algorithm* [Nie06, p. 2], or an FPT-algorithm, for short [FG06, p. 5]. In Chapter 2, we actually illustrate how an FPT-algorithm for the decision problem leads to tractable algorithms for the corresponding optimization problem where the parameter is small.

We mentioned earlier that there is a connection between PTAS and parameterized complexity theory. The papers by Marx [Mar08, Mar05] discuss many more aspects of the interplay between the theories of parameterized complexity and approximation algorithms, but the main idea is that if an optimization problem has an EPTAS, then its parameterized version is in FPT. Therefore, by showing that the parameterized version of an optimization problem is not FPT, we provide strong evidence that this optimization problem does not admit an EPTAS. However, if an optimization problem has no PTAS, it does not necessarily mean that the problem is not FPT. For example, the Vertex Cover problem is FPT; however, the problem does not have a PTAS [Mar08].

Finally, it is interesting to note that by showing that a problem is fixed-parameter tractable does not necessarily lead to an efficient or practical algorithm [Nie06, p. 27]. Since the function $f(k)$ may take large values such as

$$f(k) = 2^{2^{2^{2^{2^k}}}}.$$

A running time of $f(k) \cdot n^{O(1)}$ here cannot be considered tractable, even for $k = 1$. In this sense, we must distinguish two aspects of fixed-parameter tractability, namely, the theoretical part and the practical part. In this thesis, we deal with both practical fixed-parameter algorithms and the theoretical fixed-parameter algorithms.

Method of Reduction to a Problem Kernel

There are various techniques for designing FPT-algorithms. For example, before we solve a hard problem, we may use a polynomial-time preprocessing algorithm to shrink the given input data as much as possible. This idea gives rise to the *method of reduction to a problem kernel* [DF99, p. 39]. The method of reduction to a problem kernel reduces a problem instance I to an equivalent instance I', where the size of I' is bounded by some function of the parameter k. We call an instance that has been sufficiently reduced, a *kernel*. The main tool to facilitate the reduction is *reduction rules* [Nie06, p. 24]. This technique of using reduction rules to transform the instances of a parameterized problem into equivalent instances, with the size bounded by a function of the parameter, is also known as *kernelization* [Nie06, p. 55].

Definition 13 (Kernelization). *Let Q be a parameterized problem with inputs (I, k), where I is a problem instance and k is a parameter. The reduction to a problem kernel then means replacing instance (I, k) by a reduced instance (I', k') such that*

$$k' \leq f(k), \qquad |I'| \leq g(k)$$

where f and g are arbitrary functions depending only on k and

$$(I, k) \in Q \quad \text{if and only if} \quad (I', k') \in Q.$$

The reduction from (I, k) to (I', k') must be computable in polynomial time [Nie06, p. 55].

The problem kernel is the transformed instance (I', k') defined above. The function $g(k)$ is called the size of the problem kernel. If g in the above definition is a linear function, we say that we have a *linear kernel*. Likewise if g is a quadratic function, we say that we have a *quadratic kernel* [Slo05, p. 21]. It is also important to note the following connection between FPT and kernelization.

Lemma 1. *A decision problem is FPT if and only if it is kernelizable [Nie06].*

The idea of kernelization can be illustrated using the Vertex Cover problem (a decision version of the problem finds if there exists a vertex cover of size no larger than k). Hüffner et al. [HNW08] state three simple data reduction rules to reduce the input size of a given instance of the problem (also known as Buss's reduction for vertex cover).

Reduction Rule VC1: Remove all isolated vertices.

Reduction Rule VC2: For degree-1 vertices, put their neighbouring vertex into the cover.

Reduction Rule VC3: If there is a vertex of degree at least $k+1$, put this vertex into the cover.

The first rule is correct, because isolated vertices have no adjacent edges. The second rule is correct, because to cover an edge in the graph, one of its two endpoints must be in the vertex cover. If one of these is a degree-1 vertex, then the other endpoint has the potential to cover more edges than the degree-1 vertex. The third rule is correct, because if we did not take v into the cover, then we would have to take every single one of its $k+1$ neighbours into the cover to cover all edges adjacent to v. This is not possible, because in a decision version of the problem the maximum allowed size of the cover is k.

After exhaustively performing the rules VC1-VC3, no vertex in the remaining graph has a degree higher than k. The remaining graph can have at most k^2 edges if it is to have a solution. By rules VC1 and VC2, every vertex has degree at least two, which implies that the remaining graph contains at most $k(k+1)$ vertices. Thus, we have the problem kernel of size $O(k^2)$.

Note that most of the reduction rules in this thesis are monotonic to the kernel (and not using gadgets). That is, the resulting problem is a sub-problem of the previous one. The advantages of this approach are its simplicity and that it preserves the monotone properties as introduced by Gutner [Gut09].

Method of Extremal Structure

The *method of extremal structure* is a systematic approach to FPT-algorithm design based on kernelization techniques and extremal combinatorics [Pri05]. The method operates following a paradigm presented by two lemmas, namely, the kernelization and boundary lemmas. Consider a maximization problem, the boundary lemma is used to find the theoretical bounds on the maximum size of an instance reduced under an adequate set of reduction rules, and the kernelization lemma is invoked to decide the instances which are larger than $f(k)$ for some function f depending only on the parameter k.

When kernelizing, the intent is to transform in polynomial time instances (I, k) of a parameterized problem Q into smaller instances (I', k'). These transformed or reduced instances have the property that (I, k) is a YES-instance for Q if and only if (I', k') is a YES-instance for Q. Once the instances are no longer susceptible to reduction rules, two lemmas are invoked. We can state these two lemmas formally as follows:

Lemma 2 (Boundary Lemma). *Let (I', k') be reduced and be a YES-instance and $(I', k' + 1)$ be a NO-instance for a maximization problem Q. Then the size of I' is less than a bounding function $f(k')$ [Pri05].*

Lemma 3 (Kernelization Lemma). *If (I', k') is a reduced instance of a maximization problem Q and $|I'| \geq f(k')$, then (I', k') is a YES-instance of the problem [Pri05].*

Sometimes, we need to algorithmically preprocess the instance to identify or apply certain reduction rules. The algorithmic version of the method of extremal structure operates by first establishing a preprocessing algorithm. Once the instance is preprocessed, the reduction rules can be applied and the following two lemmas hold:

Lemma 4 (Algorithmic Boundary Lemma). *If $|I| > f(k)$ for some function f, then the preprocessing algorithm will either find a solution for Q or it will reduce I [Pri05].*

Lemma 5 (Algorithmic Kernelization Lemma). *Any instance (I, k) of Q can be reduced to a problem kernel of size $f(k)$ [Pri05].*

Prieto demonstrated [Pri05] how the method of extremal structure can be used to effectively solve several NP-complete problems, namely, Max Cut, Max Leaf Spanning Tree, Nonblocker, s-Star Packing, Edge-Disjoint Triangle Packing, Max Internal Spanning Tree and Minimum Maximal Matching. The method assumes a solution (or witness structure) in an extremal sense: no better solutions exist. The witness structure is chosen to provide a framework to identify the reduction rules. If all reduction rules can no longer be applied, we identify and prove structural claims. When all claims are proved, we bound the size of the witness structure in the boundary lemma. The kernelization lemma is then invoked to decide the instances that are larger than the bound. If they are smaller than the bound, then the method finds if there exists a solution in a brute force manner.

Bounded-Search-Tree

A brute-force way to find an optimal solution for combinatorial problems is to exhaustively search the entire solution space. This can be done in a tree-like fashion. For the bounded-search-tree approach, the idea is to create search trees such that the depths of these search trees depend only on the parameter k. These search trees are nothing but the tree of recursive calls of the corresponding recursive algorithm, where the depth of the recursion tree is bounded by the small parameter value.

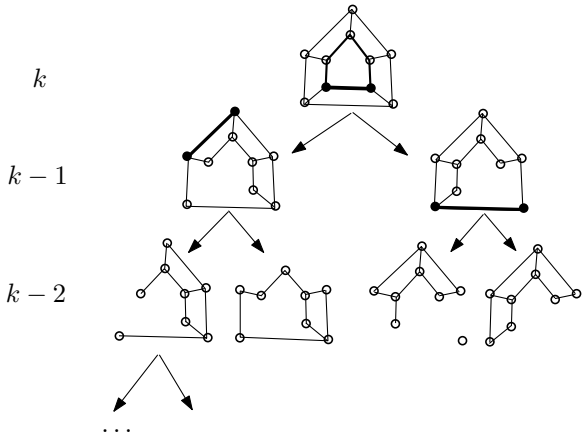


Figure 1.4: Search tree for finding a vertex cover of size at most k [HNW08].

We now demonstrate how to use this method on the Vertex Cover problem. Recall that the Vertex Cover problem takes a given graph G and asks if there is a size k set of vertices that covers all the edges. A simple search tree algorithm [HNW08] to solve this problem on a graph G starts by choosing an arbitrary edge $e = \{u, v\}$ (we know that u or v must be part of the vertex cover), and recursively searching for a vertex cover of size $k-1$ both in $G-u$ and $G-v$. That is, the algorithm branches into two subcases knowing one of them must lead to a solution of size at most k (if such a solution should exist). As shown in Figure 1.4, these recursive calls can be seen as a tree structure. Since the depth of the recursion is bounded by the parameter k and the search always branches into two subcases, the size of this tree is bounded by $O(2^k)$. Note that the depth of the search tree depends only on the parameter k and not the size of the initial input instance.

Although, the introduction of fixed-parameter tractability in the literature often uses the Vertex Cover problem as an example, it is believed that not all NP-complete problems are fixed-parameter tractable. In fact, the Dominating Set problem is not FPT [DF99, p. 463], [DF95a] but belongs to some class of the W-hierarchy, unless an unlikely collapse occurs in the parameterized complexity theory. We discuss this parameterized intractability in the next section.

Parameterized Intractability

For some parameterized problems, there is no known method that enables us to demonstrate fixed-parameter tractability. In such cases, we are concerned with the proof of parameterized intractability. The class $W[1]$ is the lowest class of parameterized intractability. With a concept of a parameterized reduction[5], Niedermeier [Nie06, p. 25] defines the notion of $W[1]$ using the k-Step Halting problem as follows.

Short Turing Machine Acceptance
(or k-Step Halting problem)
Instance: A nondeterministic Turing Machine M, an input word x, and a positive integer k.
Parameter: k.
Question: Does M accept x in a computation of at most k steps?

Definition 14 (Class W[1]). *The class $W[1]$ contains all problems that can be reduced to k-Step Halting problem by a parameterized reduction. A parameterized problem is called $W[1]$-hard if the k-Step Halting problem can be reduced to it by a parameterized reduction. A problem in $W[1]$ that is $W[1]$-hard is called $W[1]$-complete [Nie06, p. 25].*

The borderline between fixed-parameter tractable and fixed-parameter intractable lies between FPT and this $W[1]$. We can solve problems in FPT in $f(k)\cdot n^{O(1)}$ time, but for $W[1]$-hard problems we only know of exact solving algorithms with running time $n^{\Theta(k)}$ unless FPT=$W[1]$ [Nie06, p. 212].

Definition 15 (Class XP). *A parameterized problem Q is in XP if it can be determined in $f(k) \cdot n^{g(k)}$ time whether $(I, k) \in Q$ where n is the size of the*

[5] A parameterized reduction uses the same reducibility concept as Definition 13 except that it only requires $k' \leq f(k)$, but does not require $|I'| \leq g(k)$ [Nie06, p. 36].

input I, k is the parameter, f and g are arbitrary functions only depending on k [Nie06, p. 211].

We can draw an overview of parameterized complexity hierarchies as follows:

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \ldots \subseteq W[Sat] \subseteq W[P] \subseteq XP$$

This sequence is commonly known as the W-hierarchy. The "W" stands for the *weft* of Boolean formulae and circuits [Nie06, p. 211]. We refer the reader to the book by Downey and Fellows [DF99, Chapter 12] for a more in-depth study of the classes $W[1], W[2], W[Sat], W[P]$ and XP.

Examples of problems that are known to belong to a class but not known to belong to the perhaps smaller class are as follows, Vertex Cover is in the class FPT, Independent Set is $W[1]$-complete and Dominating Set is in the class $W[2]$-complete [Nie06, p. 23], [BG93, DF95b, DF95a]. Note that from the point of view of standard unidimensional complexity, these problems are all NP-complete; however, parameterized complexity suggests they are not all equally hard. The conjecture that $FPT \neq W[1]$ is analogous to the conjecture that $P \neq NP$ in classical complexity theory. A parameterized reduction from any of $W[1]$-complete problems to FPT would lead to a collapse of the classes $W[1]$ and FPT.

1.5.4 Geometric Definitions

We briefly explain some geometric definitions that are used in this thesis. Details of these definitions can be found in the standard computational geometry books [PS85, Ede87, dBvKOS00].

The geometric objects are normally sets of points in Euclidean space. We will consider, besides finite sets of points, the line containing two given points, the line-segment defined by its two extreme points, the plane containing three given points, and the polygon defined by a sequence or points.

We denote the *d-dimensional Euclidean space* [PS85, p. 17] by $\mathbb{E}^d$, that is, the Cartesian product $\mathbb{R} \times \mathbb{R} \times \ldots \times \mathbb{R}$ of d copies of $\mathbb{R}$ which is the set of all ordered sequences $(x_1, x_2, \ldots, x_d)$, with $x_i \in \mathbb{R}$, for $1 \leq i \leq d$ where $\mathbb{R}$ denotes the set of real numbers. The length of a vector x is defined as $(\Sigma_{i=1}^{d} {x_i}^2)^{1/2}$ [PS85,

p. 17]. A *point* in $\mathbb{E}^d$ is specified by the vector of its d coordinates in a Cartesian system [Ede87, p. 399]. Every point p corresponds to the vector anchored at the origin that has p as its endpoint.

Let $S = \{p_0, p_1, \ldots, p_k\}$ be a finite set of points in $\mathbb{E}^d$. A point x is a *linear combination* of S if $x = \Sigma_{i=0}^{k} \lambda_i p_i$ for suitable real numbers λ_i [Ede87, p. 399]. If $\Sigma_{i=0}^{k} \lambda_i = 1$, then x is also called an *affine combination* of S, and furthermore, if $0 \leq \lambda_i \leq 1$ for $0 \leq i \leq k$, then x is a *convex combination* of S [Ede87, p. 399]. The set S is *linearly dependent* if there is a point p_i in S which is a linear combination of $S - \{p_i\}$, otherwise S is *linearly independent* [Ede87, p. 399].

Given two distinct points p_1 and p_2 in $\mathbb{E}^d$, the linear combination $\lambda p_1 + (1 - \lambda)p_2$ is a *line*, the convex combination $\lambda p_1 + (1 - \lambda)p_2, 0 \leq \lambda \leq 1$ is a *line-segment* [PS85, p. 18]. In $\mathbb{E}^3$ a line is a one-dimensional hyperplane defined by two distinct points; a *plane* is a two-dimensional hyperplane defined by three points, no three of the points can be co-linear (on a single straight line). Generally, a *hyperplane* defines a k-dimensional subspace in $\mathbb{E}^d$ by $k + 1$ linearly independent points where $k < d$ [Ede87, p. 400].

A subset S of the plane is a *convex set* if and only if for any two points p_i and p_j in S, the line-segment $\overline{p_i p_j}$ is entirely contained in S. The *convex hull* of a set of points S is the smallest convex set containing S [dBvKOS00, p. 2]. A *polygon* is defined by a finite set of segments such that every segment extreme is shared by exactly two edges and no subset of edges has the same property [PS85, p. 18]. The segments are the *edges* and their extremes are the *vertices* of the polygon. A *simple polygon* has no pair of nonconsecutive edges sharing a point [PS85, p. 18]. A simple polygon partitions the plane into two disjoint regions, the *interior* and the *exterior* that are separated by the polygon (*Jordan curve theorem*). A simple polygon is a *convex polygon* if its interior is a convex set [PS85, p. 18].

Consider a line $y = mx + b$ in a 2-dimensional space, called the *primal space*, which is defined by two parameters m and b [dBvKOS00, p. 170]. To represent this line through this pair of parameters, we can simply use the point $(m, -b)$ in another 2-dimensional space, called the *dual space*. Similarly, a point in the primal space (p_x, p_y) defines a line $y = p_x x - p_y$ in the dual space. This mapping is one form of *dual-space transformation* [dBvKOS00, p. 169]. Note that, as shown in Figure 1.5, three points on a line in primal space become three lines through

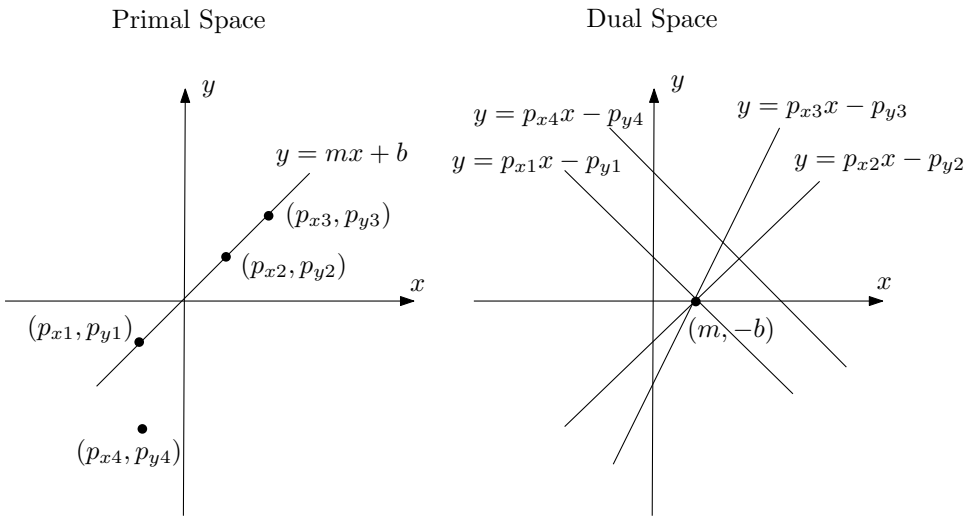


Figure 1.5: The mapping between the primal space and the dual space.

a point in dual space. Thus, one useful property of dual-space transformation is that, if in the primal space, a point (p_x, p_y) is on a line $y = mx + b$, then in the dual space the corresponding line $y = p_x x - p_y$ passes through the corresponding point $(m, -b)$, and vice versa.

Given a set $L = \{l_1, \ldots, l_n\}$ of n lines in the plane, L subdivides the plane in several regions, called faces. The edges of this subdivision are line-segments, the vertices are intersection points between two lines of L. Some of the edges and faces are unbounded. The *arrangement of lines* $\mathcal{A}(L)$ is the subdivision of the plane into vertices, edges, and faces induced by L [dBvKOS00, p. 172]. Often a problem on a set of points is dualized and turned into a problem on arrangement of lines. Here, a line through a pair of points in the primal space becomes a vertex in the dual-line arrangement.

1.6 Problem Descriptions

In this section, we give an overview of the problems studied in this thesis. We explain the problem-definitions and the motivation for researching each of these problems. In general, when we know that a problem is NP-hard, the first step in deciding fixed-parameter tractability for the problem is usually to check if the problem is in XP, because if we can prove it is not in XP, then we know it is

not in FPT. If the problem is in XP, then we can investigate whether or not this parameterized problem belongs to the class FPT.

1.6.1 The Line Cover Problem

In Chapter 2, we study both the decision version and the optimization version of the Line Cover problem.

Decision Version

We define the decision version of the Line Cover problem formally as follows:

Instance: A set S of n points in the plane, a positive integer k.
Parameter: k.
Question: Is it possible to cover these n points in S with at most k lines?

The answer YES means that it is possible to cover n points in the plane with k or fewer lines. Otherwise, the answer is NO, it is not possible to cover them with k lines. This problem has been known to be NP-hard [MT82] for over 20 years (in fact, APX-hard [KAR00]) and, therefore, considered to be intractable. Nevertheless, this problem emerges with direct connections to variations of the Traveling Salesman Problem (TSP). The decision version of the Line Cover problem was shown to be fixed-parameter tractable by Langerman and Morin [LM05]. However, their approach is mostly theoretical. We present an alternate approach and provide an implementable solution that is also practical. We evaluate experimentally several variants of the algorithms and show that our approach substantially improves the execution time when compared with previously known algorithms. Based on these decision-algorithms, we then provide algorithms for the optimization version of the problem.

Optimization Version

We present three different optimization-algorithms. The optimization version of the Line Cover problem is defined as follows:

Instance: A set S of n points in the plane.
Question: What is the minimum number of straight line-segments to cover these n points?

We show that these algorithms are easy to implement without having to rely on other algorithms.

1.6.2 The Rectilinear Line Cover Problem in Higher Dimensions

The first part of Chapter 3 discusses the Rectilinear Line Cover problem in higher dimensions. We define the Rectilinear Line Cover formally as follows:

Instance: A set S of n points in $\mathbb{R}^d$, a positive integer k.
Parameter: k.
Question: Is it possible to cover these n points in S with at most k lines where the lines are restricted to be axis-parallel?

The problem of finding the minimum number of rectilinear lines required to cover a set of points in two dimensions is polynomially solvable. However, in three dimensions (or higher), this problem is NP-complete [HM91]. Therefore, we will focus on the Rectilinear Line Cover problem in higher dimensions. We present the solution to this problem in d dimensions and show how our approach delivers efficient fixed-parameter algorithms. Our first algorithm uses the bounded-search-tree technique. The second algorithm uses the kernelization approach and has a smaller time complexity than the first one.

1.6.3 The Line Cover Restricted to a Finite Number of Orientations

The second part of Chapter 3 extends the result of the Rectilinear Line Cover problem to solve the following problem.

Instance: A set S of n points in $\mathbb{R}^d$, a positive integer k and a positive integer ϕ.
Parameter: k or ϕ (one can choose k, or ϕ, or both as the parameter).
Question: Is it possible to cover these n points with at most k lines, where these lines are restricted to the lines having a finite number of orientations, ϕ, where $\phi \geq 2$?

This is the variant of the Line Cover problem where the set of k lines must now have their orientations taken from a given finite set $R = \{r_1, r_2, \ldots, r_\phi\}$. We provide a fixed-parameter algorithm for this problem, based on the bounded-search-tree technique.

1.6.4 The Rectilinear k-Links Spanning Path Problem in Higher Dimensions

The third part of Chapter 3 examines the Minimum-Links Spanning Path Problem where every line-segment in the path is axis-parallel. Note that previously, only conjectures were enunciated over several years for the NP-completeness of this problem [BBD$^+$07, BBD$^+$09]. We provide the proof that the Rectilinear Minimum-Links Traveling Salesman Problem in higher dimensions is NP-complete. Based on this result, we show that the Rectilinear Minimum-Links Spanning Path Problem higher dimensions is NP-complete. Thus, we tackle this newly-proved intractability with fixed-parameter tractability.

Instance: A set S of n points in $\mathbb{R}^d$, a positive integer k.
Parameter: k.
Question: Is there a piecewise linear path through these n points in S having at most k line-segments (links) where every line-segment in the path is axis-parallel?

We show that the Rectilinear Minimum-Links Spanning Path Problem is FPT. Our fixed-parameter algorithm is based on the bounded-search-tree technique.

1.6.5 The k-Links Spanning Path Restricted to a Finite Number of Orientations

The last part of Chapter 3 extends further the result of the Rectilinear Minimum-Links Spanning Path Problem to solve the following problem.

Instance: A set S of n points in $\mathbb{R}^d$, a positive integer k and a positive integer ϕ.
Parameter: k or ϕ.

Question: Is there a piecewise linear path through these n points having at most k links where these k links are restricted to the lines having a finite number ϕ of orientations, where $\phi \geq 2$?

This variant of the Rectilinear Minimum-Links Spanning Path Problem now requires a set of k links to have their orientations taken from a given finite set $R = \{r_1, r_2, \ldots, r_\phi\}$. We prove that this problem is also FPT based on the bounded-search-tree technique.

1.6.6 The Rectilinear Hyperplane Cover Problem

In Chapter 4, we study a closely related problem of the Rectilinear Line Cover problem. We call this problem, the Rectilinear Hyperplane Cover problem. Before our research, the hardness of the problem and its time complexity were unknown. Thus, we first establish the hardness of this problem. Therefore, we provide the NP-completeness proof of the problem. This suggests dealing with such intractability with fixed-parameter tractability.

Instance: A set S of n points in $\mathbb{R}^d$, a positive integer k.
Parameter: k.
Question: Is it possible to cover these n points in S with at most k axis-parallel hyperplanes of $d-1$ dimensions?

We design an FPT-algorithm for this problem using the bounded-search-tree technique. We also provide another FPT result when the hyperplanes of $d-1$ dimensions are restricted to a finite number of orientations.

1.6.7 The k-Bends Traveling Salesman Problem

In Chapter 5, after examining the problems of covering points with lines and covering points with paths in the previous chapters, we study the problem of covering points with tours in two-dimensional Euclidean space (2D for short). We define the k-Bends Traveling Salesman Problem formally as follows:

Instance: A set S of n points in 2D, a positive integer k.
Parameter: k.
Question: Is there a piecewise linear tour through the n points in S with at most k bends?

Both, Arkin et al. [AMP03] and Wagner [Wag06] gives reductions from the Line Cover problem that prove this problem is NP-complete. Although the problems are hard to solve, they have important applications. For example, turns are considered very costly in the movement of heavy machinery, so we want to have as few turns as possible. Also it is applicable in real-world problems such as surveying an area by aircraft, traffic routing and space travel [Wag06, Chapter 2]. We prove that this problem is FPT.

1.6.8 The Rectilinear k-Bends Traveling Salesman Problem

In Chapter 6, we study a variant of the k-Bends Traveling Salesman Problem in which we restrict the line-segments in the tour to be rectilinear (that is, axis-parallel). We define the Rectilinear k-Bends Traveling Salesman Problem formally as follows:

Instance: A set S of n points, a positive integer k.
Parameter: k.
Question: Is there a piecewise linear tour through the n points in S with at most k bends where every line-segment in the path is either horizontal or vertical?

We prove that the Rectilinear Minimum-Links Traveling Salesman Problem in three dimensions is NP-complete. However, the hardness of the problem in two dimensions remains open. The rectilinear version of the problems received considerable attention during the 1990's [LCY90, dBvKNO92, LYW96] and also recently [Col04, ABD+05, DSW05, Wag06, WDS09, BBD+09]. Much of the interest in the rectilinear version has been motivated by applications in VLSI design. For example, the number of bends in a path affects the electric resistance and hence the accuracy of the expected timing and the voltage in the chips [LYW96]. Therefore, it is desirable to minimize the number of bends. We solve three variants of the problem. The first variant requires that one line-segment covers all the points on the same line in the tour, while the second variant requires the same line-segment orientation to cover points on the same line. The third variant does not have any of the above constraints. We prove that these three variants are FPT.

1.7 Summary of Results

The main contributions of this thesis are as follows:

1. We suggest alternate techniques to solve some of the hard geometric problems. These techniques are useful illustrations to cope with other NP-hard problems.

2. We design new FPT-algorithms for solving the Line Cover problem. We implement six new algorithms and four other algorithms that have appeared in the literature.

3. We give two FPT-algorithms for the Rectilinear Line Cover problem in higher dimensions — one based on the bounded-search-tree technique and the other based on the kernelization approach.

4. We prove that the decision versions of the Rectilinear Minimum-Links Traveling Salesman Problem and Rectilinear Minimum-Links Spanning Path Problem in three dimensions are NP-complete.

5. We prove that the parameterized version of the Rectilinear Minimum-Links Spanning Path Problem in higher dimensions is FPT.

6. We prove that the problem of covering points in three dimensions with axis-parallel planes of two dimensions is NP-complete.

7. We provide an FPT-algorithm for the Rectilinear Hyperplane Cover problem based on the bounded-search-tree technique.

8. We prove that the k-Bends Traveling Salesman Problem in two dimensions and some of its variants are FPT with respect to the number of bends k.

9. We prove that the Rectilinear k-Bends Traveling Salesman Problem is also FPT. We also give two more FPT results for two different variants of this problem.

Chapter 2

The Line Cover Problem

In this chapter, we present efficient algorithms to solve the Line Cover problem exactly. In this NP-hard problem, the inputs are n points in the plane and a positive integer k, and we are asked if we can cover these n points with at most k lines. Our approach is based on fixed-parameter tractability and, in particular, kernelization. We propose several reduction rules to transform instances of Line Cover into equivalent smaller instances. Once the instances are no longer susceptible to these reduction rules, we obtain a problem kernel whose size is bounded by a polynomial function of the parameter k and does not depend on the size n of the input. Our algorithms provide exact solutions and are easy to implement. We also describe the design of the algorithms to solve the corresponding optimization problem exactly. We experimentally evaluated ten variants of the algorithms to determine the impact and the trade-offs of several reduction rules. We show that our approach provides tractability for a larger range of values of the parameter and larger inputs, improving the execution time by several orders of magnitude with respect to the earlier algorithms that use fewer rules.

2.1 Introduction

There are applications where covering with lines is necessary, because, turns are considered very costly. For example, robots collecting balls, helicopters dropping supplies, and highway construction are illustrative, because moving objects pick up speed when traveling in a straight line. The Line Cover problem also appears in the Minimum-Bends Traveling Salesman Problem [SW01],

where, given a set of points in the plane, the problem is to find a tour through the points, consisting of the least number of straight lines. Minimizing the number of turns in the tour is desirable in VLSI and in the movement of heavy machinery [LYW94, LYW96, AFM00, ABD+05, DSW05, WDS09].

These applications demand practical implementations. Parameterized complexity offers alternatives to those problems regarded as intractable from the perspective of classical complexity theory [DF99, Nie06, FG06]. The decision version of the Line Cover was shown to be fixed-parameter tractable by Langerman and Morin [LM01, LM02, LM05], who also provided two FPT-algorithms. They call their first algorithm BST-Dim-Set-Cover, because it uses a bounded-search-tree. It has $O(k^{2k}n)$ time complexity. They refer to their second as Kernelize, because it uses kernelization. This second algorithm has $O(n^3 + k^{2(k+1)})$ time complexity, but they suggest using the algorithm from Guibas et al. [GOR96] to obtain the $O(nk + k^{2(k+1)})$ time complexity. Later, Grantson and Levcopoulos [GL06] use both of these algorithms to obtain theoretical improvements and to solve the optimization version. They provide algorithms to compute the smallest number k of straight lines that cover n points approximately in time $O(n \log k + k^4 \log k)$ and exactly in $O(n \log k + (\frac{k}{2.22})^{2k})$ time. However, there is little evidence that these approaches have delivered practical implementations. These algorithms use several repetitions of invocations of each other. They also use significantly laborious machinery from computational geometry. Therefore, when actually implemented, the theoretical algorithms become rather inefficient (if at all feasible to implement). This would suggest the parameterized complexity approach has theoretical merit but little practical impact.

There are many approaches showing that a problem is FPT. We have already mentioned kernelization. Kernelization can be achieved through the reduction rules. These rules shrink the problem instances into a kernel. The kernel can then be decided by a kernel lemma or can be solved using some exhaustive search techniques [HNW08]. We introduce two new reduction rules that lead to practical FPT-algorithms for the Line Cover problem.

2.2 Related Work and Our Contribution

We describe the earlier algorithms demonstrating that covering points with the least number of lines is in FPT (fixed-parameter tractable). We will take advantage of the reduction rules provided by the previous algorithms for developing our practical alternatives.

Langerman and Morin [LM05] gave the FPT-algorithm called BST-Dim-Set-Cover. With an input set S of n points and an integer k, BST-Dim-Set-Cover outputs whether S can be covered with k lines based on the bounded-search-tree method. Essentially, the algorithm exhaustively explores all sets of k pairs of points. A pair of points defines a line, and k pairs are a potential cover. An improved version applies kernelization (Kernelize [LM05]) and solves the reduced instance with the bounded-search-tree method. In this algorithm, all input points are assumed to be distinct.

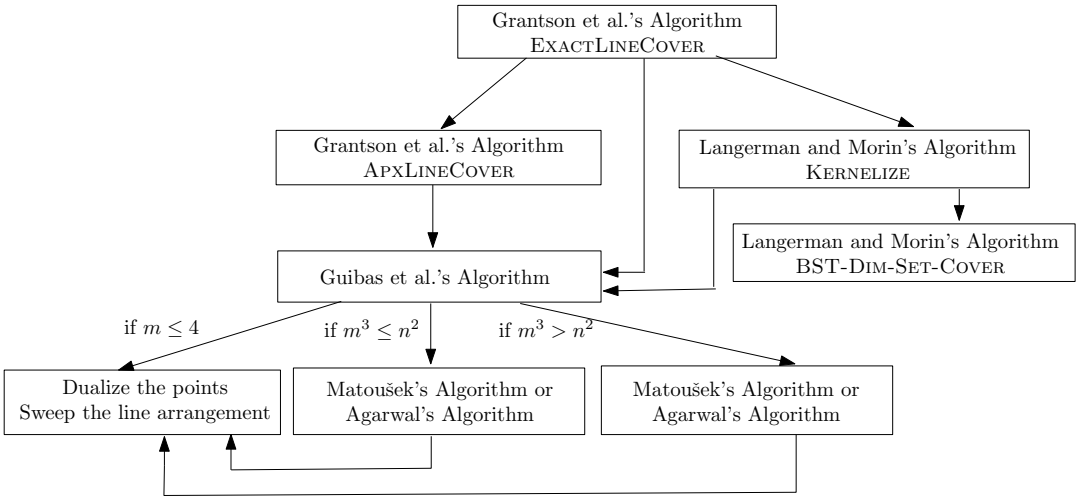


Figure 2.1: Grantson et al.'s algorithms invoke several other known algorithms.

Grantson et al. [GL06] proposed approximate and exact algorithms to solve the minimum line-covering problem using Langerman and Morin [LM05] and again Guibas et al. [GOR91, GOR96]. Their approximation algorithm is called ApxLineCover and their exact algorithm is called ExactLineCover. The building blocks are called in more frequently with a careful selection of parameters, as they introduce more reduction rules than before. Figure 2.1 shows

that Grantson et al.'s algorithms, which are for theoretical purposes, invoke several other known algorithms and use significantly laborious machinery from computational geometry. Guibas et al.'s algorithm [GOR91, GOR96] is used repeatedly by the approximation algorithm ApxLineCover. The exact algorithm ExactLineCover uses the approximation algorithm ApxLineCover. The approximation is sufficiently close for the exact algorithm to use Guibas et al.'s algorithm once more to find all the lines covering at least $k+1$ points (these lines are in the optimal solution). Finally, the algorithm calls the original Langerman and Morin's algorithm Kernelize to cover the remaining points.

To search efficiently for the approximation value, Grantson et al's ApxLineCover considers three schemes to explore the different values of k; in each scheme, k is incremented differently. Each value of k is submitted to a test to prove that the set S cannot be covered with k lines (and thus, k must be incremented). The process demonstrating that a cover will not be found detects all the lines covering at least $k+1$ points, using Guibas et al.'s algorithm, and then greedily covers the remaining points.

Note that a core component in Guibas et al.'s algorithm finds a line covering the maximum number of points of a set S of n points (this is also known as the exact fitting problem). Guibas et al. [GOR91, GOR96] achieve this in $O(\min\left\{\frac{n^2}{m}\log\frac{n}{m}, n^2\right\})$ time, where m corresponds to the maximum number of points in a line. To obtain this algorithm, Guibas et al. present another algorithm to find all the lines covering at least m points also in $O(\min\left\{\frac{n^2}{m}\log\frac{n}{m}, n^2\right\})$ time. The latter algorithm calls three subroutines depending on the values of m. If $m \leq 4$, the first subroutine transforms m points into m lines in the dual space and use the topological line sweep algorithm to find all vertices incident to m lines. If $m^3 \leq n^2$, the set S is recursively divided into two equal subsets until m is small enough, then the first subroutine is invoked to find all the candidate lines. Each of these candidate lines is then tested using the Matoušek [Mat90] algorithm or the Agarwal [Aga90] algorithm. These algorithms determine the incidences between a set of lines L and a set of points S (that is, all occurrences of a point $p \in S$ lying on a line $l \in L$)[1]. If $m^3 > n^2$ (the third subroutine),

[1]Note that in the late 1980s through to 1990, there were several independent and incremental improvements to the incidence problem by both Matoušek and Agarwal with a significant use of machinery from computational geometry.

the set S is recursively divided into m^2/n equal subsets until m is small enough, before calling the first subroutine.

In this thesis, we introduce new reduction rules that lead to practical FPT-algorithms for the Line Cover problem. We discuss their implementation and describe experiments for direct comparisons with the above-mentioned algorithms. Our experiments show that, when these new algorithms are carefully implemented, the parameterized approach (in particular, the reduction rules) leads to algorithms that can handle even large instances. The main contribution is the fast implementation of the known and the two new reduction rules that we apply in the preprocessing phase to shrink the input instances. We show that simple reduction rules have several advantages. Their correctness is transparent and thus, they are easy to implement correctly. They also cascade and may deliver smaller kernels than the theoretical guarantee. We show this result for the decision and the optimization versions of the problem.

2.3 Simple Reduction Rules

The concept of fixed-parameter tractability by kernelization is an approach that considers the simplification (or the recursive design of algorithms) in the current instance into a smaller instance of the same decision problem. Since in parameterized complexity, we have two variables (n, k) for the size of the decision problem, we would typically like to reduce n in polynomial time even if we require the exploration of many values of k. Eventually, the instance would be so small in n (polynomial in k) that it can be solved by an exhaustive search (or a similar approach), resulting in an overall complexity, polynomial in n although, perhaps, exponential in k. This characterizes the class of fixed-parameter algorithms.

We start by illustrating how the reduction-rules approach provides a simple proof that the decision problem is fixed-parameter tractable. In particular, if k is large with respect to n, for example $k \geq n = |S|$, the answer would be trivially YES, because we could use one line for each data point in S. In fact, we can do better, because we can always use one line for covering two points.

Reduction Rule 1. *If $k \geq \lceil n/2 \rceil$, then the answer is YES [GL06].*

This rule is applied by taking an instance and, in $O(1)$ time, comparing the value of k with $\lceil n/2 \rceil$; if an explicit solution is required, we would pass one line for each pair of points (potentially one more line if n is odd) in $O(n)$ time.

Consider the simplest case when $k = 1$, we would need to have all n points in S in a line.

Reduction Rule 2. *If $k = 1$, then the answer is YES if and only if all points in S are co-linear [GL06].*

This reduction rule requires $O(n)$ time.

If the input S has repeated points, then one simple reduction is to remove the duplicated points.

Reduction Rule 3. *If S has repeated points, then S can be simplified to a set with no repetitions and the answer for the simplified set is an answer for the original input [LM05].*

This reduction rule can be applied by sorting in $O(n \log n)$ time. It should be pointed out that while BST-Dim-Set-Cover [LM05] can handle duplicated points in S, Grantson et al. [GL06] assume that the input to their algorithms is always a proper set without repetitions (otherwise, their algorithm fails). From now on, we will also consider S to be a set.

Reduction Rule 4. *If there is a set of $k + 1$ or more co-linear points, place the line through them in the cover and remove them from further consideration [LM05].*

The rule is correct because, if the line through the $k + 1$ different points was not in the cover, then we would need more than k lines to cover just these points. Thus, if the set accepts a cover with k lines, any line with k or more points must be in the cover. The implementation of this rule demands a test for finding lines with many co-linear points and may be achieved using Guibas et al.'s [GOR91, GOR96] algorithm. However, we implement this rule slightly differently (we discuss this in Section 2.4.2). Grantson and Levcopoulos [GL06] use this rule to decide NO-instances as follows:

Lemma 6. *If there is a subset S_0 of S so that $|S_0| \geq k^2 + 1$, and the largest number of co-linear points in S_0 is k, then the answer is NO.*

This result leads to a quadratic-size kernel because, by repeated application of Reduction Rule 4, any instance (S, k) of the LINE COVER can be reduced to a problem kernel of size at most k^2. When the rule cannot be applied, every line covers at most k points; thus, any cover with k lines would cover at most k^2 points.

However, we now present the first of a series of new reduction rules that provide an alternative reduction and simpler kernel finding. In what follows, we let L_3 be the set of all lines that cover at least three or more points in S. If L is a set of lines, we let $cover(L)$ be all points in S covered by some line in L. With this notation, we can describe the next reduction rule that can also be taken as a structural lemma.

Reduction Rule 5. *Let $p_1 \neq p_2$ be two points in $S \setminus cover(L_3)$. Let $S' = S \setminus \{p_1, p_2\}$. Replace (S, k) by $(S', k-1)$.*

The rule essentially says that if we can find two different points that are never covered by a line that covers three or more points, then we can reduce to a smaller problem that ignores these two points and uses one less line. Alternatively, there is always an optimal solution that uses one of its lines to cover these two points.

The correctness proof of Reduction Rule 5 goes as follows. Consider an instance where $\{p_1, p_2\} \subset S \setminus cover(L_3)$. If an optimal cover C uses $\overline{p_1, p_2}$ (that is, the line through p_1 and p_2), then $C \setminus \{\overline{p_1, p_2}\}$ covers $S \setminus cover\{\overline{p_1, p_2}\}$ optimally. Suppose C does not include the line $\overline{p_1, p_2}$. We will show that there is a cover using the same number of lines, and this other cover uses $\overline{p_1, p_2}$. The cover C must use a line $l_i \notin L_3$ to cover p_i, for $i = 1, 2$. If l_i does not cover another point besides p_i, then l_i can be removed and replaced by $\overline{p_1, p_2}$, resulting in a cover of size $|C|$. Otherwise, there is at most another point $q_i \in S$ covered by l_i besides p_i, for $i = 1, 2$. Consider a cover C' that does not have l_i, for $i = 1, 2$, but instead has $\overline{p_1, p_2}$ and $\overline{q_1, q_2}$. The cover C' has two new lines but two fewer old lines, so C' has the same number of lines as C. Any other point in S besides $\{p_1, p_2, q_1, q_2\}$ is covered in C' by the same line that covers it in C. The cover C' has the same size as C and includes the line $\overline{p_1, p_2}$ as required. See Figure 2.2 for an illustration.

Clearly, this rule can be applied repeatedly until the size of $S \setminus cover(L_3)$ is no more than one point. A very similar reasoning gives our next reduction rule.

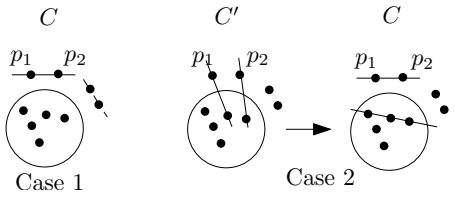


Figure 2.2: Two cases for the proof of Reduction Rule 5.

Reduction Rule 6. *Let $p_1 \neq p_2$ be two points in cover(L_3). Suppose that no other line in L_3 besides $\overline{p_1, p_2}$ covers p_1 or p_2 (that is, $L_3 \cap \{l | l$ covers $p_1\} = L_3 \cap \{l | l$ covers $p_2\} = \{\overline{p_1, p_2}\}$). Let $S' = S \setminus$ cover$\{\overline{p_1, p_2}\}$. Replace (S, k) by $(S', k-1)$.*

Reduction Rule 6 is correct. Again, this suggests that optimal solutions must use the line $\overline{p_1, p_2}$. If a cover C did not use $\overline{p_1, p_2}$, it must use two other lines that are not in L_3 to cover p_1 and to cover p_2. Similar to the previous proof, a line $l_i \in C$ that covers p_i can cover at most another point q_i, thus $C \cup \{l_1, l_2\} \setminus \{\overline{p_1, p_2}, \overline{q_1, q_2}\}$ is a cover of the same cardinality that uses $\overline{p_1, p_2}$.

The two new reduction rules above do not improve the quadratic worst-case size of the kernel. However, we believe that, in practical circumstances, the incorporation of these new rules results in a smaller kernel than k^2, leading to faster kernel solving. Hence, we present variants of the algorithms where the decision about which rules are involved is different, and we test our claim experimentally.

2.4 Algorithms for the Decision Problem

We will focus on the decision problem first. Specifically, we assume that we have a set S and an integer k as inputs and we are asked if S can be covered with k lines or fewer. Later, we discuss the issue of actually finding a cover. We will apply the reduction rules one after the other until none of the rules applies. We call this phase the Preprocessing Algorithm, and part of our contribution is to suggest the ordering of application as well as algorithms and data structures for their application. Once the instances are no longer susceptible to the reduction rules, we will invoke the kernel lemma (Lemma 6). We may need to invoke a

bounded-search-tree approach to solve the kernel, but we will also suggest how to set the scene heuristically for the exhaustive search.

Reduction rules have different complexities and trim the instance differently. Some enable a rule that previously could not be applied. For example, after removing lines with more than k points from consideration (Reduction Rule 4), the reduced instances now may be covered using one line for each pair of points (Reduction Rule 1 is now applicable). The cascading effects of reduction rules is important for the algorithmic engineering of FPT-algorithms [Nie06].

The PREPROCESSING Algorithm below resolves the instance (S, k) or returns a kernelized instance (S', k'), where $S' \subseteq S$, $k' \leq k$ and $|S'| \leq O(k^2)$. In it, all rules except our last are attempted (later, we explore the effect of the last rule). If a rule determines the type of an instance (determines it is a YES-instance, like Reduction Rule 1 or Reduction Rule 2, or determines it is a NO-instance, like Reduction Rule 4 with Lemma 6), then we can halt. Reduction Rule 5 is computationally costly, thus we execute this rule last, when the size of our instance is smaller.

PREPROCESSING ALGORITHM(S, k)

if $|S| \leq 2k$, (*Reduction Rule 1 applies*)
  then answer YES and halt.
if all points in S are co-linear, (*Reduction Rule 2 applies*)
  then answer YES and halt.
if not REDUCTIONRULE_KPLUS(S, k, S', k'), (*Reduction Rule 4 applies*)
  then answer NO and halt.
if $|S'| \leq 2k'$, (*Reduction Rule 1 applies*)
  then answer YES and halt.
Construct L_3, a set of all lines through 3 or more points in S'.
while there exist $\{p_1, p_2\} \in S' \setminus cover(L_3)$, (*Reduction Rule 5 applies*)
  do $S' \leftarrow S' \setminus cover\{\overline{p_1, p_2}\}$; $k' \leftarrow k' - 1$;
Repeat all the steps until every rule can no longer be applied.

Our ordering of the rules has been the result of the experimental evaluation of the trade-off of the cost of the application of the rule with respect to the overall running time of the algorithm. For example, we recommend that Reduction Rule 4 operates after Reduction Rule 2. Note that if Reduction Rule 2 applies,

the instance is resolved and we do not need to check if Reduction Rule 1 applies. However, if Reduction Rule 4 applies, it is worth checking if Reduction Rule 1 is now applicable, because this is a constant-time check before the more costly construction of L_3.

2.4.1 Details of ReductionRule_kPlus

The Boolean function, ReductionRule_kPlus, checks if Reduction Rule 4 is applicable. Grantson et al. [GL06] present a function (they call Lines) to find lines covering more than k points in the plane. This Lines function [GL06, p. 9] uses Guibas et al.'s algorithm [GOR96] and each invocation of Guibas et al.'s algorithm finds one line through at least $k+1$ points. Guibas et al.'s algorithm itself also calls another subroutine and relies on other known algorithms. Therefore, we believe that Guibas et al.'s algorithm is not for use in practical settings. Nevertheless, we use their idea to create what we call a "simple version of Guibas" where we directly transform points into lines in the dual space and sweep the dual-line arrangement to find all vertices incident to at least $k+1$ lines. The idea behind this dual-space transformation is that the $k+1$ co-linear points on a line in primal space become the $k+1$ lines passing through a single point in dual space. Thus, finding this intersection point (vertex) in the dual space is the same as finding lines through at least $k+1$ points in primal space, but has a smaller time complexity. Our direct use of the dual-space makes the implementation of ReductionRule_kPlus relatively easy. Although our function ReductionRule_kPlus resembles the Lines function [GL06], there are several distinctive points.

1. Our function ReductionRule_kPlus cascades Rule 4 on itself. That is, once we find a line covering more than k points, we also look for a line covering one point less and iterate this step until no line is found. In the Lines function, the value of k remains constant.

2. We cascade rules on themselves, so our Preprocessing Algorithm finds settings where Lemma 6 applies when the original Lines would not. Also, our preprocessing has more instances where it produces a smaller kernel than that produced with Lines.

3. To find lines through more than k points, we use the dual-space transforma-

tion [dBvKOS00, p. 169]. Guibas et al.'s algorithm [GOR96] performs this transformation when $k \leq 3$. Since k is usually small, our direct use of the dual-space does not significantly penalize the observable performance in the (now simpler) implementation of the function ReductionRule_kPlus. Moreover, we perform the transformation and construction of the arrangement only once.

4. Finally, the Lines function [GL06] implicitly requires a test for the k lines found to cover S (and a test that S is empty). While the test for the emptiness of S is presented in the last two lines of the original pseudocode [GL06], it was not explicitly mentioned in the complexity analysis (although it can be performed within the same complexity, but all this results in a larger hidden constant in the O-notation). Our function ReductionRule_kPlus ensures that $S \setminus cover\{l_1, \ldots, l_s\}$ is computed explicitly and the exit points of the loop are clearer.

```
ReductionRule_kPlus(S, k, S', k')
 1  S' ← ∅; L_k ← ∅; arrangement A ← ∅        (*initialization*)
 2  while (S ≠ ∅),
 3        do choose p ∈ S; S ← S \ {p}
 4           if p not in cover(L_k),
 5              then S' ← S' ∪ {p}
 6                   Transform p into a line dual(p) in dual space
 7                   A ← A.insert(dual(p));
 8                   while there is vertex v in A incident to more
 9                   than k − |L_k| lines and |L_k| ≤ k,
10                        do L_k ← L_k ∪ {dual(v)}; k' ← k − |L_k|
11                           for each p' ∈ S' ∩ cover(dual(v))
12                                do A ← A.delete(dual(p'));
13                           S' ← S' \ cover(dual(v))
14                   if (|S'| > ((k − |L_k|)^2), (*Lemma 6*)
15                      then return false
16  return true                              (*return S' and k' by reference*)
```

In the pseudocode above, the arrangement of the dual-lines is denoted by $\mathcal{A}$. The insertion of a line $dual(p)$ into $\mathcal{A}$ is denoted by $\mathcal{A}.insert(dual(p))$.

Similarly, $\mathcal{A}.delete(dual(p))$ refers to the deletion of a line $dual(p)$ from $\mathcal{A}$. The function ReductionRule_kPlus scans the original set S and examines one point at a time. As an invariant of this iteration, it maintains a set of lines $L_k = \{l_1, l_2, \ldots, l_s\}$ where it knows that each l_i must be in a cover of S with k or fewer lines (thus $s \leq k$). It also maintains a set $S' \subset S$ of points so that $cover(L_k) \cap S' = \emptyset$ (that is, the points in S' are not covered by any of the lines found to be in a cover). Initially, the invariant is satisfied, because $L_k = \emptyset$ and $S' = \emptyset$. If it finds that $S' \geq (k-s)^2 + 1$, then the hypothesis of Lemma 6 is satisfied and it exits immediately with the value `false`. Otherwise, when S is empty, the invariant ensures that S' is a kernel and $k' = k - s$ is the new value for a cover because of s successive (cascading) applications of the reduction rule.

When the next point $p \in S$ is examined, it is removed from S. If $p \in cover(L_k = \{l_1, l_2, \ldots, l_s\})$, then nothing needs to be done to maintain the invariants. However, when $p \notin cover(L_k)$, then $S' \leftarrow S' \cup \{p\}$. This addition of p to S' may now trigger the reduction rule in S'. Thus, we enter another inner loop. In this inner loop, we investigate if there is a line with $k + 1 - s$ or more points in S'. If such a line is found, it must belong to any cover that uses k or fewer lines because $s = |L_k|$ lines have already been found to belong to the cover. Therefore, in this inner loop, we enlarge L_k and set $S' \leftarrow S' \setminus cover(L_k)$ until no line with $k + 1 - s$ points is found in S'. At this point, the invariant of the outer loop is satisfied. We make the observation that if a line through $(k-s)+1$ points is found in S' and all points covered by this line are removed, then S' will not satisfy the conditions of Lemma 6 (that is, our function will never return `false` at this stage). However, if S' becomes empty and there are still points in S, the outer loop will continue. Moreover, if we reach the case when $s = k$ and there are still points in S, eventually it may be that a single point in S' triggers Lemma 6 for the function to return `false`.

2.4.2 Cascade-And-Sort

When preprocessing cannot decide an instance, it returns a kernel that is a YES-instance if and only if the original instance was also a YES-instance. The structure of our first algorithm is as follows.

Algorithm Cascade-And-Sort(S, k)

1 $(S', k') \leftarrow$ The Preprocessing Algorithm(S, k)
2 **for** $p \in S'$,
3 **do** $Weight(p) \leftarrow \max_{l_i \in L_3 \wedge p \in l_i}\{|\{q \in S' | q \in l_i\}| + \frac{i}{|L_3|}\}$
4 Sort S' in descending $Weight$ order
5 **return** BST-Dim-Set-Cover(S', k') [LM05]

With an input set S of n points and an integer k, BST-Dim-Set-Cover outputs whether or not S can be covered with k lines. Before calling BST-Dim-Set-Cover, our algorithm sorts the points in S' according to their weights. The weight is a measure how many points a line covers, and is the instrument to guide this greedy approach [CLRS01]. The points that are covered by the same line in L_3 receive the same weight. The weight is higher if the line covers more points in S'. Experimentally, we have found BST-Dim-Set-Cover benefits from this sorting of the weights. This is because the points from the same candidate lines are now adjacent in the search tree. Moreover, the candidate lines with more points are tested before the lines with fewer points (in practice, we often find that lines covering many points are likely to be in the optimal solution).

Running Time Analysis

Let us first analyze the running time of the Preprocessing Algorithm. Reduction Rule 1 can be performed in constant time or $O(n)$ time for an explicit solution. Reduction Rule 2 can also be applied in $O(n)$ time.

We now consider the running time of the function ReductionRule_kPlus that is the cascading application of Reduction Rule 4. For each point chosen from S, we need to check whether it lies on an already-found line or not. For the purpose of this point-location query, we construct an arrangement by incrementally inserting a line into the arrangement (at most k times). The time required to insert a line l_i is linear in the complexity of its zone. According to the Zone Theorem [ESS93], the complexity of the zone of a line in an arrangement of m lines is $O(m)$, and the construction of the m lines-arrangement takes $O(m^2)$ time. Therefore, each insertion of a line $l_i \in L_k$ into the arrangement requires $O(k)$ time and we can construct the k-lines arrangement in $O(k^2)$ time. A query, whether a chosen point lies on any of the lines, can be answered in $O(\log k)$ time

using Mulmuley's point-location algorithm (CGAL's library) [CGA07]. We perform such a query at most n times; therefore, it takes $O(n \log k + k^2)$ to perform all point-location queries and construct the k-lines arrangement.

The next part of ReductionRule_kPlus transforms points into lines in the dual space and incrementally inserts the dual lines one by one into the arrangement $\mathcal{A}$. Note that not every point is transformed into a line, only the points that are not in $cover(L_k)$ are inserted as the dual lines into $\mathcal{A}$. Due to Lemma 6, there are, at most, $k^2 + 1$ points in S' being examined in each inner loop whether or not there is a line with more than $k - |L_k|$ points and $|L_k| \leq k$. Therefore, there are at most $k^2 + 1$ dual lines in $\mathcal{A}$ each time we sweep $\mathcal{A}$ for a vertex that is incident to more than $k - |L_k|$ lines. Now, let us analyze the time taken in each step. First, transforming at most $k^2 + 1$ points into $k^2 + 1$ dual lines takes $O(k^2)$ time and we do this at most k times hence, this step requires $O(k^3)$ time. Second, we insert at most $k^2 + 1$ dual lines into $\mathcal{A}$ in $O(k^4)$ time or $O(k^5)$ time when we account for the no more than k repetitions. Third, sweeping for all vertices incident to more than $k - |L_k|$ lines can be done in $O(k^4)$ time in each inner loop or $O(k^5)$ time if we consider all the executions of the inner loop. Finally, once a line with more than $k - |L_k|$ co-linear points is found, removing the points on the line takes $O(k^2)$ time (there are at most $k^2 + 1$ points in each inner loop) or $O(k^3)$ time for $|L_k| \leq k$ lines. Thus, the total running time of the ReductionRule_kPlus function is $O(n \log k + k^2 + k^3 + k^5 + k^5 + k^3) = O(n \log k + k^5)$ time.

It takes constant time to check if we can apply Reduction Rule 1 again or $O(k^2)$ if an explicit solution is required. Our next rule, Reduction Rule 5, finds two different points that are never covered by a line that covers three or more points by first constructing L_3. Note that, here, the size of S' is at most k^2, so we build L_3 by transforming at most k^2 points into k^2 lines in the dual space and sweeping the dual-line arrangement to find all vertices incident to more than two lines. We construct the arrangement in $O(k^4)$ time and build L_3 in $O(k^5)$ time because there are at most $O(k^4)$ lines in L_3 and each line contains at most k points. Removing two different points that are never covered by L_3 can be done in $O(k^2)$. Hence, Reduction Rule 5 can be carried out in $O(k^4 + k^5 + k^2) = O(k^5)$ time.

We repeatedly apply all the above rules at most $k-1$ times, because the application of a rule resolves the instance or reduces the value of k. Thus, this iteration totals another $O(k^6)$ time. Thus, combining the running time of each reduction rule and the time of iterative calls, the PREPROCESSING Algorithm can be performed in $O(n+n+(n\log k+k^5)+k^2+k^5+k^6) = O(n\log k+k^6)$ time.

The CASCADE-AND-SORT Algorithm involves weighting the points in S', sorting them, and calling BST-DIM-SET-COVER. We assign a weight to each point in the kernel (this step can be made together with the construction of the set L_3 for Reduction Rule 5) and sorting the points in the kernel requires $O(k^2\log k)$ time. BST-DIM-SET-COVER requires $O(k^{2k}n')$ time where $n' = |S'|$. Thus, this part of the main algorithm can be carried out in $O(k^2\log k+k^{2k+2}) = O(k^{2k+2})$ time. Therefore, the total running time for CASCADE-AND-SORT is $O(n\log k+k^{2k+2})$.

2.4.3 Cascade Further

We now present CASCADE FURTHER, whose objective is to evaluate the impact of the last reduction rule. Therefore, this algorithm offers the following new feature. We incorporate the last of our reduction rules (Reduction Rule 6) after the previous kernelization process and, naturally, before solving the problem kernel. The preprocessing phase remains the same as CASCADE-AND-SORT. The structure of the main algorithm is as follows:

ALGORITHM CASCADE FURTHER(S,k)

1 (S',k') ← THE PREPROCESSING ALGORITHM(S,k)
2 Construct L_3, a set of all lines through three or more points in S'.
3 **while** there exist $\{p_1,p_2\} \in cover(L_3)$ such that no other line in L_3 besides $\overline{p_1,p_2}$ covers p_1 or p_2, (*Reduction Rule 6 applies*)
4 **do** $S' \leftarrow S' \setminus cover\{\overline{p_1,p_2}\}$; $k' \leftarrow k'-1$;
5 **for** $p \in S'$,
6 **do** $Weight(p) \leftarrow \max_{l_i \in L_3 \wedge p \in l_i}\{|\{q \in S' | q \in l_i\}| + \frac{i}{|L_3|}\}$
7 Sort S' in descending $Weight$ order
8 **return** BST-DIM-SET-COVER(S',k') [LM05]

Running Time Analysis

The additional work with respect to Cascade-And-Sort is the exhaustive applications of Reduction Rule 6. This involves constructing L_3 and finding if there exist two different points $\{p_1, p_2\} \in cover(L_3)$ where no more than one line in L_3 can cover any of these points. Since $k' \leq k$ and $n' \leq n$ where $n' = |S'|$, we over estimate the cost if we replace k' with k and also if we replace n' with n in the running time analysis. We saw earlier that the set L_3 has at most $O(k^4)$ lines and each line contains at most k points. The set L_3 can be built in $O(k^5)$ time while also storing the information on how many lines pass through each particular point. Checking if the rule applies on each line $l_i \in L_3$ and each point on l_i takes $O(k^5)$ time. Removing p_1, p_2 together with the points on the line $\overline{p_1, p_2}$ requires at most $O(k^2)$ time. Hence, Reduction Rule 6 can be carried out in $O(k^5 + k^2) = O(k^5)$ time. Thus, the entire preprocessing phase and the application of Reduction Rule 6 can be performed in $O(n \log k + k^6)$ time.

The running time for weighting the points in S', sorting them and calling BST-Dim-Set-Cover is the same as in Cascade-And-Sort, that is, $O(k^{2k+2})$. Thus, the total running time for Cascade Further is $O(n \log k + k^{2k+2})$.

2.4.4 Prioritize Points in Heavy Lines

The variant we now introduce solves the kernel differently. There would be no need to sort the points, but to prioritize over lines around points. We solve the problem kernel using the idea that if we have a YES-instance, one of the lines must cover as many points as the average coverage provided by the covering lines. That is, if S can be covered with k lines and $|S| = n$ for $n > k$, then there is one of the k lines in the cover of S covering at least $\lceil n/k \rceil$ points. Let $L_{\lceil n/k \rceil}$ be the set of all lines covering at least $\lceil n/k \rceil$ points in S and $cover(L_{\lceil n/k \rceil})$ be all points in S covered by a line in $L_{\lceil n/k \rceil}$. Grantson et al. [GL06] used this idea to control the number of calls when solving the kernel with Langerman and Morin's algorithm. While this idea ensures that we can find one of the covering lines, the problem remains in that there could be a large number of candidate lines in $L_{\lceil n/k \rceil}$. To illustrate this, consider Figure 2.3. In this example, $l_1, l_2, \ldots, l_8$ are all candidate lines, because they are in $L_{\lceil n/k \rceil}$. Thus, testing all the candidate lines will give us at least one of the k lines that belongs to the solution set. Grantson

et al. showed that there are at most $3k^2/2$ lines covering the average number of points. Thus, the idea of finding lines with the average number of points results potentially in a quadratic number of recursive calls.

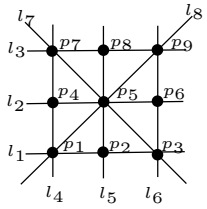


Figure 2.3: Instance (S, k) where $|S| = 9$ and $k = 3$.

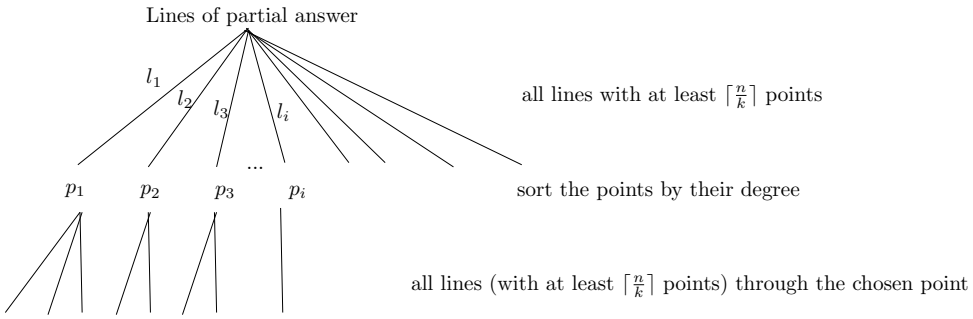


Figure 2.4: A search tree in each recursive call based on prioritizing points in heavy lines.

Our strategy is to turn the focus of attention to the points that are in the candidate lines $l_i \in L_{\lceil n/k \rceil}$. Consider such a point p in $cover(L_{\lceil n/k \rceil})$. Since p must be covered somehow in any cover, we can make recursive calls for all lines $l_i \in L_{\lceil n/k \rceil}$ that cover p. This significantly reduces the branching factor of the recursive calls from the perspective of p. Grantson et al. show that there are at most $3k/2$ candidate lines passing through a given point in the plane [GL06]. We use this observation to create a new bounded-depth search tree having a virtual structure, as shown in Figure 2.4.

The actual approach is to first identify $L_{\lceil n/k \rceil}$ and then choose a point p from S in one of the lines in $L_{\lceil n/k \rceil}$. For the correctness of the algorithm, we explore all candidate lines with at least $\lceil n/k \rceil$ points. However, we expect that in practice, the branching factor will amortize to linear. The strategy to prioritize

the points p on lines in $L_{\lceil n/k \rceil}$ is important. Our scheme chooses p with the largest number of lines in $L_{\lceil n/k \rceil}$, that is, most of the candidate lines meet at p. This is to increase the chance of having optimal lines passing through p. We also use a hashing scheme to avoid exploring a candidate line twice (an already tested line because of another point). The AroundThePoints function solves the problem kernel using this strategy.

Algorithm Prioritize Points in Heavy Lines(S, k)

```
1  (S',k') ← The Preprocessing Algorithm(S, k)
2  Construct L_3, a set of all lines through three or more points in S'.
3  while there exist {p_1, p_2} ∈ cover(L_3) such that no other line in L_3
   besides p_1p_2 covers p_1 or p_2                    (*Reduction Rule 6 applies*)
4        do S' ← S' \ cover{p_1p_2}; k' ← k' − 1;
5  return AroundThePoints (S', k')
```

AroundThePoints(S', k')

```
 1  n' ← |S'|; T ← ∅
 2  if n' ≤ 2k',
 3     then return true
 4  if k' > 0,
 5     then L_⌈n'/k'⌉ ← the set of all lines covering at least ⌈n'/k'⌉ points
 6          C ← cover(L_⌈n'/k'⌉)
 7          Sort C in descending order of degree where
 8          degree (p) =def |{l ∈ L_⌈n'/k'⌉ | p ∈ l}|
 9          for each p ∈ C in descending degree,
10               do
11                  for each l ∈ L_⌈n'/k'⌉, p ∈ l, and l ∉ T
12                       do
13                          T ← T ∪ {l}   (*lines that were tested before*)
14                          if AroundThePoints(S' \ cover(l), k' − 1),
15                             then return true
16  return false
```

Running Time Analysis

We saw earlier in Cascade Further that, the entire preprocessing phase and the application of Reduction Rule 6 can be performed in $O(n \log k + k^6)$ time.

We now perform a worst-case analysis of the asymptotic running time of the AroundThePoints function. The search tree in Figure 2.4 has a depth of at most k (it is a bounded-search-tree). For worst-case analysis, we can use $k \geq k'$ and $n' \leq k^2$ where $n' = |S'|$. The branching factor of the tree depends on the number of the candidate lines covering at least $\lceil n'/k' \rceil$ points. Grantson et al. [GL06] show that there are at most $3k^2/2$ candidate lines. In fact, the analysis of the size of (number of nodes in) the tree emulates the analysis of Grantson et al. [GL06], because we apply a hashing scheme to avoid testing candidate lines that are already tested (the variable T in the pseudocode). Thus, we can consider $3k^2/2$ nodes in the first level of recursion. In the next recursive call, we will have at most $(3/2)^2[k(k-1)]^2$, and $(3/2)^3[k(k-1)(k-2)]^2$ nodes in the third level. After k recursive calls, we will have at most $(3/2)^k(k!)^2$ nodes and, therefore,

$$\text{number of nodes} \leq \left[\frac{3}{2}\right]^k (k!)^2 \approx \left[\frac{3}{2}\right]^k \left(\frac{k}{e}\right)^{2k} < \left(\frac{k}{2.22}\right)^{2k}$$

(also using Stirling's approximation [GL06]).

It remains to analyze the workload at each node. This consists of the time required to compute $L_{\lceil n'/k' \rceil}$, a set of all lines covering at least $\lceil n'/k' \rceil$ points, the time required to sort the points p in lines $l_i \in L_{\lceil n'/k' \rceil}$ and the time required to build a list of candidate lines around each chosen point. We find all lines covering at least $\lceil n'/k' \rceil$ points in the same way as before, that is, we transform at most k^2 points to k^2 lines in the dual space, and we sweep the dual-line arrangement to find all vertices incident to at least $\lceil n'/k' \rceil$ lines. This process can be performed in $O(k^4)$ time. Sorting the points in $cover(L_{\lceil n'/k' \rceil})$ in descending order of their degree requires $O(k^2 \log k)$ time, because there are at most k^2 points in $cover(L_{\lceil n'/k' \rceil})$. Finally, to build a list of candidate lines around each chosen point takes $O(k^3)$ time, because there are at most k^2 points in S' and there are at most $3k/2$ candidate lines for any given point. Thus, the work performed at each node requires $O(k^4 + k^2 \log k + k^3) = O(k^4)$ time. The product of the number of nodes with the workload at each node is the worst-case complexity of the new bounded-search-tree algorithm and is given by $O\left(k^4(k/2.22)^{2k}\right)$ time.

Thus, the total running time of Prioritize Points in Heavy Lines is $O\left(n \log k + k^4(k/2.22)^{2k}\right)$.

2.5 Algorithms for the Optimization Problem

2.5.1 Search with Increments of One

We now present the FPT-algorithms to solve exactly the optimization version of the problem. The value of k in this case is not given as an input, but k is the minimum number of lines used to cover n points. Our first algorithm calls Prioritize Points in Heavy Lines with k_0, where $k_0 \leq k$; if this fails, we increment the value of k_0 until we succeed. Initially, k_0 is set to 1.

Algorithm One increments(S)

```
1  k_0 ← 1; success ← false
2  while (not success),
3        do
4           success ← call Prioritize Points in Heavy Lines(S, k_0)
5           if (not success), then k_0 ← k_0 + 1
6  return (k ← k_0)
```

Running Time Analysis

To estimate the number of calls to Prioritize Points in Heavy Lines we consider two cases.

1. The first case is when the call to Prioritize Points in Heavy Lines fails (the set S cannot be covered with k_0 lines) and this was discovered by a reduction rule and not by analyzing the kernel. In this case, Prioritize Points in Heavy Lines takes $O(n \log k + k^5)$ time.

2. The second case is when the call to Prioritize Points in Heavy Lines fails or is successful because of a call to the kernel. For example, the last call to Prioritize Points in Heavy Lines found k, the minimum number of lines used to cover S. The successful call to Prioritize Points in Heavy Lines runs in $O(n \log k + k^4(k/2.22)^{2k})$ time and it happens once (we stop when we have found the answer).

The sum over the time complexities considering both cases is

$$\left[\sum_{k_0=1}^{k-p}(n \log k_0 + k_0^5)\right] + \sum_{k_0=k+1-p}^{k}\left[n \log k_0 + k_0^4(k_0/2.22)^{2k_0}\right].$$

That is, the last p tests for the value of k required that a kernel be examined (and were not determined by a reduction rule). In our experimental evaluation, the value of p is typically very small (it is already the case that $p < k$). Moreover, if the smallest cover has cardinality k different from the initial values of k_0 (say $k_0 \in \{1, 2, 3\}$), the reduction rules act promptly and no kernel is tested. This is because the set constructed for the rule that applies Lemma 6 needs to have size $k_0^2 + 1$ and if $k_0 = 3$, this is merely a set of size 10. Moreover, this is where the cascading effect of ReductionRule_kPlus, as opposed to Lines, also has a profound effect, determining the NO-instance by the rules, rather than by kernel analysis.

Since

$$\sum_{k_0=1}^{k}(n \log k_0 + k_0^5) = n \sum_{k_0=1}^{k} \log k_0 + \sum_{k_0=1}^{k} k_0^5 \approx nk \log k + (1/6)(k^6)$$

we conjecture that the time complexity of One increments, in practice, is $O(nk \log k + k^4(k/2.22)^{2k})$.

Although, in practice, we expect only a constant number of calls where a kernel is resolved, in theory we still have to consider the worst case where, for every call, a kernel is analyzed. Therefore, we now perform the worst-case analysis. The overall time complexity of One increments is clearly the sum over the time complexities for calling Prioritize Points in Heavy Lines with the values of $k_0 = 1$ until $k_0 = k$, that is,

$$\sum_{k_0=1}^{k}\left[n \log k_0 + k_0^4(k_0/2.22)^{2k_0}\right] = O(nk \log k + k^5(k/2.22)^{2k}).$$

2.5.2 Guessing a Lower Bound

We explore an alternative method of searching for a value k_0 (closer to the optimal value k), where $k_0 \leq k$. The algorithm calls Prioritize Points in Heavy Lines to decide if we can cover n points with k_0 equal to 1. If this fails, we increment the value of k_0. If k_0 is no longer small ($k_0 > 5$), we find a new value for k_0 that will, potentially, bring us closer to the optimal value of k. We will call such a value, k^*; it can be set as the number of lines to which Reduction Rule 6 can be applied. This rule finds two different points $\{p_1, p_2\} \in cover(L_3)$

where no more than one line in L_3 can cover any of these points. Hence, any lines we detect here must be in the solution set. The number of lines we detected cannot be greater than k for a YES-instance, hence $k^* \leq k$.

```
Algorithm Take a guess(S)
 1  k_0 ← 1; k* ← −1; success ← false
 2  while (not success),
 3        do
 4           if k_0 > 5 and k* = −1,          (*value of k* is computed once*)
 5              then k* ← 0
 6                   construct L_3, a set of all lines with three points or more
 7                   while there exist {p_1, p_2} ∈ cover(L_3) such that no
                     other line in L_3 besides p_1p_2 covers p_1 or p_2,
 8                        do k* ← k* + 1   (*Reduction Rule 6 applies*)
 9                   if k* > k_0, then k_0 ← k*
10  success ← call Prioritize Points in Heavy Lines(S, k_0)
11  if (not success), then k_0 ← k_0 + 1
12  return (k ← k_0)
```

Running Time Analysis

The additional work with respect to One increments is counting the number of lines where Reduction Rule 6 applies. Similar to the earlier computation, the set L_3 has at most $O(n^2)$ lines and each line contains $O(n)$ points. The set L_3 can be built in $O(n^3)$ time while also storing the information on how many lines pass through each particular point. Checking if the rule applies in each line $l_i \in L_3$ and each point on l_i takes $O(n^3)$ time.

Although, in practice, Reduction Rule 6 works very well, we assume the worst case when it fails to improve the value of k^*. Here, we improve the value of k_0 by incrementing it (that is, $k_0 \leftarrow k_0 + 1$ in the pseudocode). We saw earlier that in the worst case the sum over the time complexities for calling Prioritize Points in Heavy Lines with the values of $k_0 = 1$ until $k_0 = k$ is $O(nk \log k + k^5(k/2.22)^{2k})$ time.

Therefore, the overall time complexity of Take a guess is $O(n^3 + n \log k + k^5(k/2.22)^{2k}) = O(n^3 + k^5(k/2.22)^{2k})$.

2.5.3 Binary Search

An alternative method of searching for the k value is to use a binary search. This technique can search for the desired value between two bounds (left and right) in logarithmic time. However, we need to provide the values of the lower bound (left) and the upper bound (right) for the search. We can double the value of k_0 (initially $k_0 = 1$) until we find $2k_0$ that can cover all the points in S. The idea is to use k_0 as the lower bound and $2k_0$ as the upper bound to search for the optimal k value (that is, the minimum number of lines used to cover S). We refer the reader to the classical sorting and searching book [Meh84a] for more details about this technique.

ALGORITHM BINARY SEARCH(S)

```
1  k_0 ← 1
2  success ← false
3  while (not success),
4        do
5           success ← call PRIORITIZE POINTS IN HEAVY LINES(S, k_0)
6           if (not success),
7              then k_0 ← 2k_0
8  k_yes ← k_0
   (*binary search for the minimum number of lines*)
9  Lo ← (k_yes/2) + 1; Hi ← k_yes − 1;
10 while (Lo ≤ Hi),
11       do
12          Mid ← ⌊(Lo + Hi)/2⌋
13          success ← call PRIORITIZE POINTS IN HEAVY LINES(S, Mid)
14          if (success),
15             then Hi ← Mid − 1
16             else  Lo ← Mid + 1
17 if (success),
18    then k ← Mid
19    else  k ← Mid + 1
20 return k
```

The algorithm above performs a binary search between two bounds. We call them Lo and Hi. The search begins at Mid, the middle value between Lo and

Hi. Whenever the call to PRIORITIZE POINTS IN HEAVY LINES is successful, we make $Hi = Mid - 1$ (our intention is to find the minimum value that results in a YES-instance), otherwise we shift the lower bound to $Mid + 1$. The search stops when $Lo > Hi$ and the value of k is returned. This is the value such that (S, k) is a YES-instance, but $(S, k-1)$ is a NO-instance.

Running Time Analysis

The running time consists of the time for finding the upper bound for the binary search ($2k_0$) and the time of searching for the minimum number of lines used to cover S (binary search).

The first loop stops when $2k_0$ results in a successful call to PRIORITIZE POINTS IN HEAVY LINES. In such a case, the number of calls to PRIORITIZE POINTS IN HEAVY LINES is $1 + \log 2k_0$ (note that $k \leq 2k_0 < 2k$). The total running time in the worst case is, therefore,

$$(1 + \log 2k_0)\left[n \log 2k_0 + (2k_0)^4 (2k_0/2.22)^{4k_0}\right]$$

$$= O(n \log^2 k + k^4 \log k (k/1.11)^{4k}).$$

The second loop (binary search) is executed $\log k_0$ times. The running time in each loop is bounded by $O(n \log 2k_0 + (2k_0)^4 (2k_0/2.22)^{4k_0})$. We assume the worst-case scenario here. The running time in this second loop is thus $O(n \log^2 k + k^4 \log k (k/1.11)^{4k})$.

Therefore, the overall time complexity of Algorithm BINARY SEARCH is $O(n \log^2 k + k^4 \log k (k/1.11)^{4k})$.

2.6 Evaluation

We now evaluate the algorithms presented here, comparing them with the alternatives in the literature. We consider six algorithms for solving the decision version of the problem and another four algorithms for solving the optimized version. Table 2.1 summarizes the time complexities of the 10 algorithms. The CASCADE-AND-KERNELIZE Algorithm is the same as the KERNELIZE Algorithm, except that the reduction rule, removing a line having more than k points, is used in a cascading effect. That is, once we find a line covering more

Table 2.1: Summary of the time complexities of 10 algorithms.

Algorithms	Time Complexity	Problem	Reference
Cascade-And-Sort	$O(n \log k + k^{2k+2})$	Decision	(Sec. 2.4.2)
Cascade Further	$O(n \log k + k^{2k+2})$	Decision	(Sec. 2.4.3)
Prioritize Points in Heavy Lines	$O(n \log k + k^4(k/2.22)^{2k})$	Decision	(Sec. 2.4.4)
BST-Dim-Set-Cover	$O(k^{2k}n)$	Decision	[LM05]
Kernelize	$O(n^3 + k^{2k+2})$	Decision	[LM05]
Cascade-And-Kernelize	$O(n^3 + k^{2k+2})$	Decision	[LM05]
One increments	$O(nk \log k + k^5(k/2.22)^{2k})$	Optimization	(Sec. 2.5.1)
Take a guess	$O(n^3 + k^5(k/2.22)^{2k})$	Optimization	(Sec. 2.5.2)
Binary Search	$O(n \log^2 k + k^4 \log k(k/1.11)^{4k})$	Optimization	(Sec. 2.5.3)
ExactLineCover	$O(n \log k + k^{2k+2})$	Optimization	[GL06]

than k points, we also look for a line covering one point less and repeat this step until no line is found. The reduced instance is then solved with the bounded-search-tree method. Note that, in the original Kernelize [LM05], the value of k remains constant. We believe that this strategy might slightly improve the performance of Kernelize. ExactLineCover is described in Grantson and Levcopoulos's article [GL06]. However, in the actual implementation of this algorithm, we replace the calls to Guibas et al.'s algorithm [GOR96] with the calls to the simple version of Guibas, as described earlier. Since these are the worst-case complexities, they include some constant factors that are nontrivial in any implementation and which may also exhibit very different performances in a variety of instances. Therefore, we conducted an experimental evaluation. We implemented the ten algorithms,[2] using the GNU C++ Compiler version 4.1.2 and the CGAL's library [CGA07]. All experiments are performed on a computer with an Intel Pentium 4 processor, at 2.40GHz with 512MB of RAM. We consider this machine to be a standard PC. We read the input points from a file, and we shuffle (with all permutations being equally likely) the input points once, before we execute the algorithms. We do not include the time for reading the input points; that is, we specifically measure the running time when the computation of the covering actually starts. Note that the running time is measured using a timer class in CGAL that measures user process time (CPU) [CGA07] and not real time.

[2]The source code and the data of the problem instances are available upon request.

2.6.1 Performance on Random Instances

Algorithms for the Decision Problem

We focus first on the performance for instances where the set of points can be covered with k lines. Thus, we define the following procedure to generate instances with this property. We create k random lines by first generating k random points $\{s_1, \ldots, s_k\}$ and transforming them to dual space. These k points then become k lines that will be used as the cover. If we want to generate a set S of n points, we repeat n times the random selection of one of the k lines (with equal probability) and place a point p_j on this line. The point p_j is chosen by selecting the x-coordinate uniformly in a bounded range and the y-coordinate is determined by the fact that p_j belongs to l_i. Once the file is generated, the points are shuffled once with all permutations equally likely. Since each of the n points lies at least on one of the k lines, the instance generated this way is always a YES-instance. For a NO-instance, we simply reduce the value of k in the YES-instance by one. We generate 10 files in this manner and we present the average running times after the executions of these files. We tested four combinations for the values of k and n, with the results being given in Table 2.2 and Table 2.3. These results are also presented graphically in Figure 2.5 and Figure 2.6 as an illustration. Note that all results regarding the observed averages in this thesis are reported with 95% confidence intervals.

Discussion

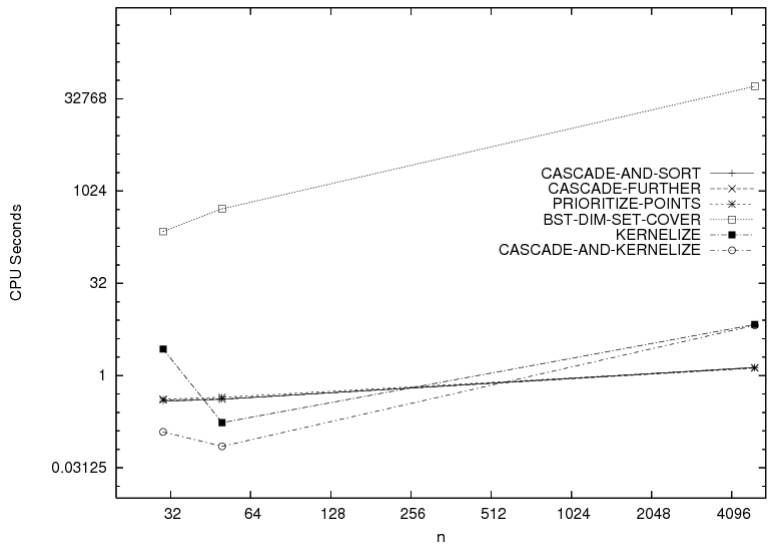
One could easily appreciate the emerging pattern. Simply, the implementation of our algorithms requires a CPU time in the order of seconds, while the theoretical algorithms without our simple reduction rules require a CPU time in the order of hours! Consider, first, the YES-instances where $n > k^2$. Here, the kernelization phase reduces the size of the instance effectively, because the points are assigned to lines by the generation process with equal probability, thus the expected number of points on any given line is greater than k. Therefore, lines with $k+1$ or more points are not uncommon and the kernelization phase is effective, playing an important role in reducing the size of the instance. The cascading is also effective; we find that the number of lines having more than k' points are detected and removed several times (reducing k' further and making it more likely to find another line in the solution).

Table 2.2: Average running times for the decision problem with 95% confidence intervals (10 random YES-instances for each value of k and n).

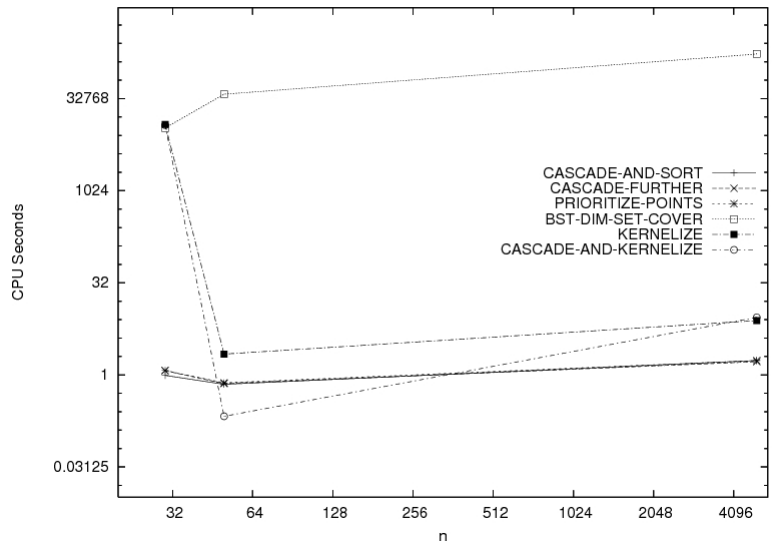
		Running Time		
	k	6		
Algorithm	n	30	50	5,000
Cascade-And-Sort		0.38 ± 0.03s	0.41 ± 0.05s	1.36 ± 0.07s
Cascade Further		0.40 ± 0.04s	0.41 ± 0.04s	1.33 ± 0.08s
Prioritize Points in Heavy Lines		0.41 ± 0.04s	0.44 ± 0.05s	1.34 ± 0.07s
BST-Dim-Set-Cover		223 ± 90s	525 ± 126s	≈ 15hr
Kernelize		2.71 ± 1.58s	0.17 ± 0.09s	6.81 ± 0.76s
Cascade-And-Kernelize		0.12 ± 0.06s	0.07 ± 0.03s	6.59 ± 0.61s
		Running Time		
	k	7		
Algorithm	n	30	50	5,000
Cascade-And-Sort		0.98 ± 0.14s	0.70 ± 0.05s	1.71 ± 0.12s
Cascade Further		1.17 ± 0.19s	0.72 ± 0.05s	1.64 ± 0.10s
Prioritize Points in Heavy Lines		1.18 ± 0.20s	0.74 ± 0.07s	1.72 ± 0.13s
BST-Dim-Set-Cover		≈ 3hr	≈ 11hr	≫ 24hr
Kernelize		≈ 3hr	2.19 ± 1.99s	7.72 ± 0.51s
Cascade-And-Kernelize		≈ 3hr	0.21 ± 0.11s	8.69 ± 1.05s

Table 2.3: Average running times for the decision problem with 95% confidence intervals (10 random NO-instances for each value of k and n).

		Running Time		
	k	5		
Algorithm	n	30	50	5,000
Cascade-And-Sort		0.25 ± 0.03s	0.24 ± 0.03s	0.27 ± 0.04s
Cascade Further		0.24 ± 0.03s	0.24 ± 0.03s	0.26 ± 0.04s
Prioritize Points in Heavy Lines		0.23 ± 0.02s	0.24 ± 0.03s	0.26 ± 0.03s
BST-Dim-Set-Cover		239 ± 5.23s	454 ± 7.23s	≈ 15hr
Kernelize		0.23 ± 0.05s	0.11 ± 0.08s	7.67 ± 0.60s
Cascade-And-Kernelize		0.05 ± 0.02s	0.04 ± 0.02s	6.57 ± 0.31s
		Running Time		
	k	6		
Algorithm	n	30	50	5,000
Cascade-And-Sort		≈ 1hr	0.48 ± 0.05s	0.54 ± 0.07s
Cascade Further		0.83 ± 0.24s	0.46 ± 0.05s	0.58 ± 0.07s
Prioritize Points in Heavy Lines		0.82 ± 0.23s	0.49 ± 0.06s	0.53 ± 0.08s
BST-Dim-Set-Cover		≈ 4hr	≈ 8hr	≫ 24hr
Kernelize		≈ 2hr	0.33 ± 0.11s	8.48 ± 1.12s
Cascade-And-Kernelize		≈ 2hr	0.13 ± 0.06s	8.14 ± 1.07s

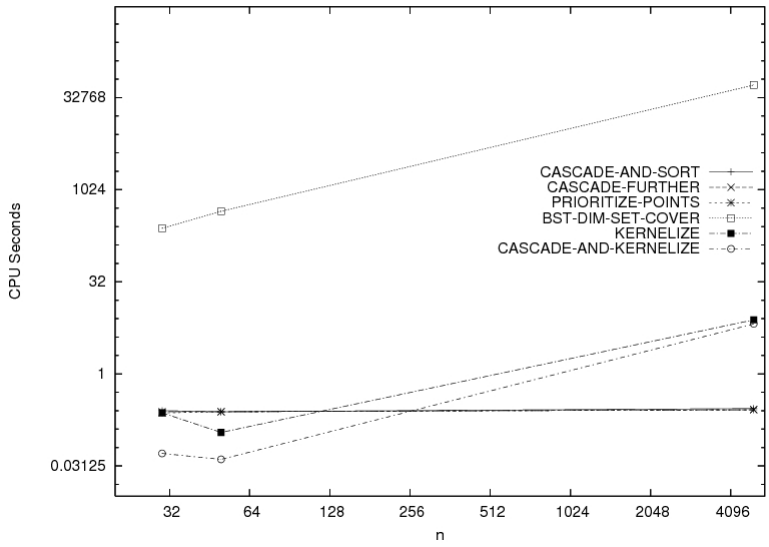


(a) $k = 6$

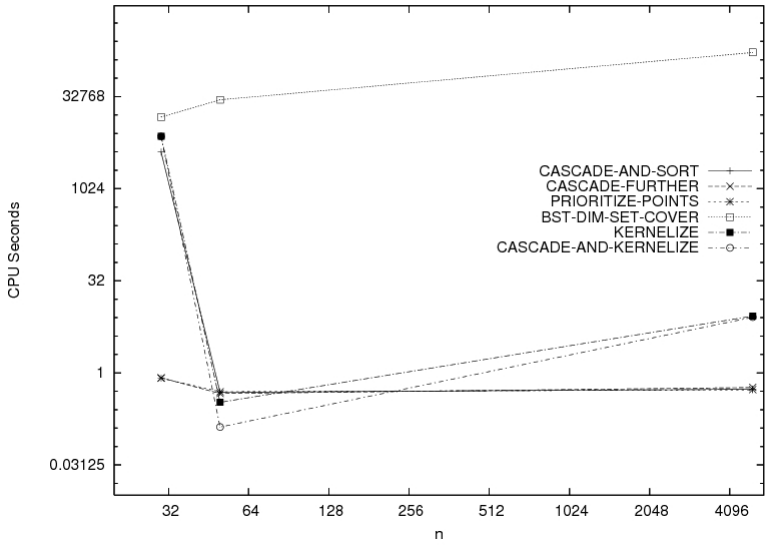


(b) $k = 7$

Figure 2.5: Running time versus the instance size for random YES-instances.



(a) $k = 5$



(b) $k = 6$

Figure 2.6: Running time versus the instance size for random NO-instances.

Table 2.4: Average kernel's sizes and average running times for practical limits of k and n with 95% confidence intervals (10 random instances for each value of k and n).

		Kernel's Size		Running Time (min)	
	k	30	5	30	5
Algorithm	n	900	100,000	900	100,000
Cascade-And-Sort		0	0	5 ± 0.12	0.4 ± 0.01
Cascade Further		0	0	5 ± 0.21	0.4 ± 0.01
Prioritize Points in Heavy Lines		0	0	5 ± 1.13	0.4 ± 0.01

Naturally, our three algorithms perform slightly worse than Kernelize and Cascade-And-Kernelize for very small values of k and n (for example, $k = 6, n = 50$). This is because our algorithms carry overhead to ensure the exhaustive application of the reduction rules in the kernelization phase. However, the kernelization phase of Kernelize and Cascade-And-Kernelize requires $O(n^3)$ time; hence, they only perform well when n is also small.

When $n < k^2$ and k is sufficiently large, our three algorithms solve random YES-instances relatively fast (see the instance with $k = 7$ and $n = 30$, for example). Since $n < k^2$, the expected number of points on a given line is less than k. While the kernelization part of Kernelize and Cascade-And-Kernelize do not apply at all, Reduction Rule 5 and Reduction Rule 6 of our algorithms work really well. This is why the introduction of more reduction rules proposed here results in large improvements with respect to earlier algorithms. With instances $k = 6, n = 30$ and $k = 7, n = 30$, the three known algorithms perform quite differently, depending on the values of (n/k). If the average number of points on any given line is much less than k, then the probability that their kernelization process can reduce the size of these random instances is very low. For $n > k^2$, Cascade-And-Kernelize performs better than Kernelize. Again, this is because several lines that have more than k' points are detected and removed (further reducing k' and making it more likely to find another line in the solution). This illustrates again the power of using more reduction rules, even if they seem simple and do not produce a theoretically smaller kernel. Similar results are obtained in the case of the NO-instances except that Cascade-And-Sort does not perform as well as our other two algorithms. This is not surprising, because Cascade-And-Sort invokes BST-Dim-Set-Cover after the kernelization phase has failed to detect the NO-instances.

Table 2.5: Average running times for the optimization problem with 95% confidence intervals (10 random instances for each value of k and n).

		Running Time			
	k	6		7	
Algorithm	n	30	50	30	50
One increments		0.9 ± 0.07s	0.8 ± 0.04s	2.5 ± 0.39s	1.7 ± 0.10s
Take a guess		1.3 ± 0.06s	2.0 ± 0.04s	2.7 ± 0.40s	2.4 ± 0.09s
Binary Search		2.1 ± 0.24s	1.6 ± 0.10s	3.6 ± 0.46s	2.3 ± 0.18s
ExactLineCover		2.4 ± 0.42s	5.1 ± 0.18s	≈ 7hr	6.4 ± 0.29s

The previously-described results show that implementation of the previous algorithms is hardly practical, even for a small value of k ($k = 7$). Thus, we also explore the practical limits for the value of k in our algorithms (see Table 2.4). The random instances here are generated with the same procedure as before. The value of k used for the instances generation is the same as the parameter k of the inputs, that is, these are the tests on the YES-instances. We note the average size of the kernels together with the average running times. The result shows that our algorithms solve the instances in five minutes, even when k is as large as 30. Experimentally, we also found that reduction rules perform essentially all the work and the average size of the kernels is zero in all cases. In particular, Reduction Rule 4 and Reduction Rule 5 in the Preprocessing Algorithm work extremely well in these random instances. In the next section, we generate structured instances and create the situations that disallow some of these reduction rules to work to see how our algorithms perform (we discuss this later).

Algorithms for the Optimization Problem

All the algorithms presented so far are for the decision version of the problem. Naturally, we use the instances generated in this random model above to also evaluate the algorithms for the optimization version of the problem. We organized the experiments as before. We considered 10 random instances with different values of k and n, and we used our implementations to explore the practical limits of values for k and n. Recall that k in this case is not given as an input, but k is the minimum number of lines used to cover n points.

The results for the optimization problem are shown in Table 2.5. Again, our implementations are several orders of magnitude more CPU-time efficient than

Table 2.6: Average running times for practical limits of k and n with 95% confidence intervals (10 random instances for each value of k and n).

		Running Time (min)		
	k	30	10	5
Algorithm	n	900	1,600	100,000
One increments		38 ± 1.54	0.15 ± 0.01	0.43 ± 0.02
Take a guess		11 ± 0.27	19 ± 0.05	0.46 ± 0.03
Binary Search		22 ± 1.32	0.33 ± 0.02	1.2 ± 0.03

the previous algorithms, which are not practical for values such as $k = 7$ and $n = 30$. Exploring the limits of how far we can take our implementations, we also considered larger values of k and n. Table 2.6 shows the corresponding running times.

Discussion

The result is obvious. Our three algorithms outperform ExactLineCover. All algorithms perform significantly better when $n > k^2$ than when $n \leq k^2$, because the algorithms detected a number of lines that have more than k points and removed these lines early.

The performance of our three algorithms is rather similar when the value of k is still small. The binary search technique of the Binary Search Algorithm actually pays off when k is big (see Table 2.6). It reduces the running time of the One increments Algorithm by 42%, when $k = 30, n = 900$. Nevertheless, the One increments Algorithm still runs reasonably fast for k up to 30, and n up to 100,000 or higher. When $k \leq 5$, the One increments Algorithm and the Take a guess Algorithm are technically the same algorithms. When $k > 5$, Take a guess performs twice as well as Binary Search and 3.5 times better than One increments (see $k = 30, n = 900$). However, Take a guess requires $O(n^3)$ time to provide an initial guess for the value of k; hence, it performs very poorly when n is big (that is, Take a guess runs 126 times slower than One increments when k is 10 and n is 1600) while One increments and Binary Search can take a very large value of n (up to one million points). In summary, One increments is ideal for applications with a small k value (less than 20). Binary Search is suitable when k is large.

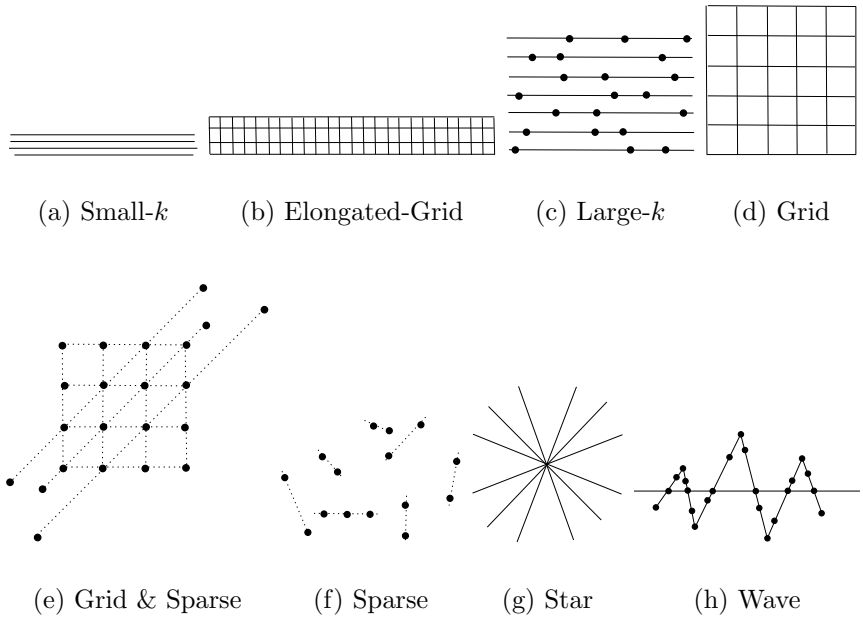
These results show that our algorithms work extremely well in the previously described types of instances, because the reduction rules perform essentially

all the work. Rarely, the algorithm is required to solve a kernel. Therefore, we decided to construct instances where the reduction rules would have little opportunity or instances where the rules do not apply, together with instances where the rules do apply efficiently. We call them structured instances, because they are generated under a certain prototype.

2.6.2 Performance on Structured Instances

Algorithms for the Decision Problem

We first evaluate the performance of the algorithms for the decision problem. The instances here will be generated along eight profiles. Figure 2.7 illustrates the profiles of these instances.



(a) Small-k (b) Elongated-Grid (c) Large-k (d) Grid

(e) Grid & Sparse (f) Sparse (g) Star (h) Wave

Figure 2.7: Profiles for structured instances.

In this experiment, the input points are not randomly distributed in the Euclidean space; instead, they are somewhat structured. In fact, each input file is manually generated by deliberately placing each point such that the cover of these points looks exactly like the profiles drawn in Figure 2.7. As a representative of each profile, we typically generate one input file, because any linear

transformation of its shape (the profile) preserves the answer; that is, these points can still be covered by the same number of lines even if the profile shape is slanted or deviates from the horizontal or vertical plane. We now describe the eight profiles and how they are motivated.

1. The Small-k profile has data points that lie on k horizontal lines (Figure 2.7(a)). The lines were chosen manually, but the points on the lines are generated automatically. In this profile, the value of n can be increased without increasing the value of k; therefore, we make sure that each line covers as many points as possible, such that $n >> k^2$. Typically, we will test our algorithms in at least three files in this profile (for a different value of k). Here, we expect kernelization to perform all the work, so we essentially compare the overhead of all the algorithms using this profile.

2. The Elongated-Grid (Figure 2.7(b)) profile is similar to the above Small-k instance profile and also has $n >> k^2$. However, the profile has data points in each vertex of an elongated grid with k horizontal lines and n/k vertical lines. This profile tests the case where a single point can be covered by several lines in L_3.

3. The Large-k profile represents a test when the input size is small ($n < k^2$), and here the kernelization stops, because the input size is already smaller than the square of the parameter (Figure 2.7(c)).

4. The Grid profile has data points in each vertex of a square grid with k horizontal and k vertical lines; thus, $n = k^2$ (Figure 2.7(d)). We test this profile when k is 5, 6, and 7 (and n is 25, 36, and 49, respectively). This is usually what we obtain as a quadratic kernel; each point could lie on several lines, all of which could be in a solution. Hence, this profile tests the situation where the kernel is k^2 and there are a large number of candidate lines to be tested.

5. The Grid and Sparse profile (Figure 2.7(e)) is a case where, in addition to the points in the grid, we add points outside the grid in p of the diagonal lines, making sure they do not share any x or y coordinates. This makes kernels that are less than quadratic, because the p lines are removed by Reduction Rule 6 and the grid part becomes incomplete (in the diagonal),

so we expect our algorithms to perform slightly better here than in the grid.

6. The Sparse profile is a case where most of the lines cover just a few points (Figure 2.7(f)). The input size n is chosen such that $2k < n < 3k$.

7. The Star profile (Figure 2.7(g)) is similar to the grid, that is, we test the situation where $n = k^2$. However, here the points do not share any x or y coordinates, thus, no point can be covered by several candidate lines as it can in the grid.

8. The Wave profile (Figure 2.7(h)) tests the case where one line that covers most of the points in the plane is not actually in the optimal cover.

We look at values of k no greater than 7. We create one input file for each profile (except for Small-k, Elongated-Grid and Grid, we create three files of these profiles). We tested each algorithm on both the YES-instances and the NO-instances. For the YES-instances, we make sure that all generated instances discussed earlier are covered by a minimum number of lines that is equal to the value k used as input for the algorithm. In total, we create 14 input files (instances), each of these instances corresponds to the 14 columns in Table 2.7. For the NO-instances, we reduce the value of k in the corresponding YES-instances by one. For each file, we shuffle the input points 10 times and we present the average running times in seconds. The results for the YES-instances are as shown in Table 2.7; and for the NO-instances, the results appear in Table 2.8. Note that all results regarding observed averages in this section are computed with 95% confidence intervals in the same way as we did in the previous section. However, for the sake of readability, we omit them from the tables.

While we evaluate our algorithms with respect to those with theoretical merits, we also look at the difference in performance among our three algorithms (see Table 2.9). We made tests on the grid-type instances, but the results are not sufficiently obvious for us to comment on the three algorithms. Therefore, we select a new structured instance, called Incomplete-Grid (see Figure 2.8(a)), to test our algorithms. In this type of instance, we expect that none of the reduction rules will apply, leaving a large kernel that may be solved well with the heuristic approach of each algorithm.

Table 2.7: Average running times on structured instances for the decision problem (10 shuffles on the same input file for a YES-instance).

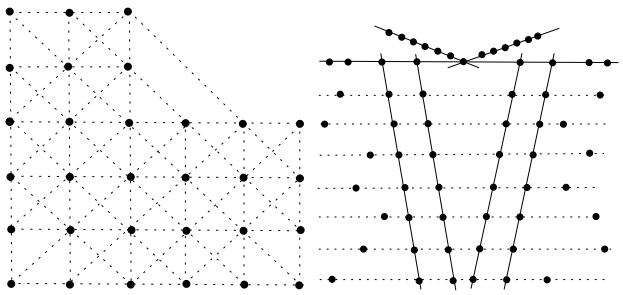
		Input Profile						
		Small-k			Large-k	Elongated-Grid		
	k	4	4	4	7	4	4	4
Algorithm	n	100	1,000	10,000	21	100	1,000	10,000
Cascade-And-Sort		0.1s	0.2s	1.5s	0.4s	0.1s	0.2s	1.5s
Cascade Further		0.1s	0.2s	1.6s	0.7s	0.1s	0.2s	1.6s
Prioritize Points in Heavy Lines		0.1s	0.3s	1.6s	0.7s	0.1s	0.2s	1.6s
BST-Dim-Set-Cover		1.0s	9s	96s	≈ 3hr	1.1s	10s	58s
Kernelize		0.03s	0.4s	25s	≈ 3hr	0.02s	0.35s	23s
Cascade-And-Kernelize		0.02s	0.4s	23s	≈ 3hr	0.03s	0.42s	23s

		Input Profile (cont.)						
		Grid			Grid & Sparse	Sparse	Star	Wave
	k	5	6	7	6	7	6	6
Algorithm	n	25	36	49	22	15	36	23
Cascade-And-Sort		0.5s	1.1s	2s	0.4s	0.2s	1.2s	168s
Cascade Further		1s	1.6s	3s	0.6s	0.2s	2s	0.8s
Prioritize Points in Heavy Lines		1.4s	2.5s	4.5s	0.7s	0.2s	2s	0.9s
BST-Dim-Set-Cover		3.5s	293s	≈ 5hr	107s	939s	336s	125s
Kernelize		3.5s	180s	≈ 5hr	90s	1,260s	416s	124s
Cascade-And-Kernelize		2.7s	185s	≈ 5hr	83s	1,214s	366s	126s

Table 2.8: Average running times on structured instances for the decision problem (10 shuffles on the same input file for a NO-instance).

		Input Profile						
		Small-k			Large-k	Elongated-Grid		
	k	3	3	3	6	3	3	3
Algorithm	n	100	1,000	10,000	21	100	1,000	10,000
Cascade-And-Sort		0.05s	0.05s	0.05s	≈ 2hr	0.05s	0.05s	0.05s
Cascade Further		0.05s	0.05s	0.05s	0.65s	0.05s	0.05s	0.05s
Prioritize Points in Heavy Lines		0.05s	0.05s	0.05s	0.67s	0.05s	0.05s	0.05s
BST-Dim-Set-Cover		0.76s	7.8s	78s	≈ 2hr	0.75s	7.6s	76s
Kernelize		0.02s	0.42s	24s	≈ 2hr	0.02s	0.37s	23s
Cascade-And-Kernelize		0.03s	0.35s	24s	≈ 2hr	0.02s	0.24s	22s

		Input Profile (cont.)						
		Grid			Grid & Sparse	Sparse	Star	Wave
	k	4	5	6	5	6	5	5
Algorithm	n	25	36	49	22	15	36	23
Cascade-And-Sort		0.12s	0.28s	0.52s	0.2s	0.22s	0.31s	0.27s
Cascade Further		0.14s	0.28s	0.57s	0.2s	0.23s	0.27s	0.27s
Prioritize Points in Heavy Lines		0.14s	0.25s	0.53s	0.2s	0.23s	0.27s	0.27s
BST-Dim-Set-Cover		4s	274s	≈ 7hr	138s	≈ 1hr	311s	157s
Kernelize		0.02s	0.07s	0.16s	2.4s	≈ 1hr	0.02s	3s
Cascade-And-Kernelize		0.01s	0.01s	0.02s	0.04s	≈ 1hr	0.01s	0.13s



(a) Incomplete-Grid (b) Bad-Heavy-Lines

Figure 2.8: Two special structured-instances for testing the features of our algorithms.

We also justify the earlier claim that Prioritize Points in Heavy Lines performs better when sorting points p (on lines with at least $\lceil n/k \rceil$ points) according to their degree (number of lines through p). We test this claim with an instance in Figure 2.8(b) where, in this instance, we set the scene such that the many lines that cover most of the points in the plane are not actually in the optimal solution, hence the algorithm is likely to make a bad choice of p (without sorting the point). The experiment will show that, by this heuristic sorting, we have a better chance of cutting out branches of the search tree; hence Prioritize Points in Heavy Lines runs even faster. This result appears in Table 2.10.

Table 2.9: Comparison of our three algorithms for the decision problem (10 shuffles on the Incomplete-Grid instance).

	Running Time	
Algorithm	YES-instance ($k = 6, n = 30$)	NO-instance ($k = 5, n = 30$)
Cascade-And-Sort	50.87 ± 46s	0.25 ± 0.03s
Cascade Further	107.64 ± 67s	0.25 ± 0.03s
Prioritize Points in Heavy Lines	2.16 ± 0.3s	0.25 ± 0.02s

Experimentally, we also found that when we have the grid profiles, the preprocessing phase of Prioritize Points in Heavy Lines fails to reduce the input instance (see Table 2.11). Therefore, we consider grid-type instances to be a difficult instance for Prioritize Points in Heavy Lines, and we investigated how far we could stretch k in this case. We aim for the value of k that

can solve the YES-instance in about 30 minutes on a standard PC. Table 2.12 shows that the size of the kernel is quadratic and, in that case, the kernel has a workload where the practical limit for k is about 30.

Table 2.10: Average running times of Prioritize Points in Heavy Lines with and without sorting feature (10 shuffles on the Bad-Heavy-Lines instance).

Algorithm Prioritize Points in Heavy Lines	Running Time YES-instance ($k = 10, n = 63$)	Running Time NO-instance ($k = 9, n = 63$)
with sorting	22.5 ± 3s	145.8 ± 0.6s
without sorting	77.4 ± 22s	145.2 ± 0.6s

Table 2.11: Kernel's size of Prioritize Points in Heavy Lines for each input.

Input Profile	k	n	Kernel's Size	
			YES-instance	NO-instance
Small-k	4	100	0	0
Small-k	4	1000	0	0
Small-k	4	10000	0	0
Large-k	7	21	0	0
Elongated-Grid	4	100	0	0
Elongated-Grid	4	1000	0	0
Elongated-Grid	4	10000	0	0
Star	6	36	0	0
Wave	6	23	0	0
Grid	5	25	25	0
Grid	6	36	36	0
Grid	7	49	49	0
Grid & Sparse	6	22	6	0
Sparse	7	15	0	0

Table 2.12: Practical limits for k with Algorithm Prioritize Points in Heavy Lines.

Grid k by k	Kernel Size	Approx. Running Time
$k = 29$	841	27min
$k = 30$	900	31min

Discussion

First, let us discuss the case of the YES-instances. Our three algorithms can handle larger values of k and n than the previously known algorithms. Our algorithms achieve the best results in the grid-type instances. Note that the

reduction rules do not apply in these kinds of structures; this means that the cost of solving the problem kernel in our three algorithms is also less than than those of the known algorithms. For the same values of k and n, we notice that Prioritize Points in Heavy Lines performs better in the Star instance-type than the Grid type, because there are fewer candidate lines to test in the Star instance-type. In the Grid instance type, Cascade-And-Sort runs faster than Prioritize Points in Heavy Lines. However, Cascade-And-Sort performs quite poorly on the Wave instance type (because lines with many points do not necessarily belong to the solution set). The Cascade Further Algorithm, which has one more reduction rule than Cascade-And-Sort, solves the Wave instance type 200 times faster than Cascade-And-Sort. This reveals the power of our Reduction Rule 6. When we consider the NO-instances, the performance of Prioritize Points in Heavy Lines is rather similar across the board. That is, Prioritize Points in Heavy Lines requires less than a second to decide the NO-instances.

Let us consider the differences in performance among our three algorithms. The Incomplete-Grid instance type in Figure 2.8(a) represents the situation when the sorting strategy does not work really well, because several lines that cover most of the points are not part of the cover. The result shows that Prioritize Points in Heavy Lines outperforms the other two algorithms (see Table 2.9). Moreover, there is a large variance in the performance of Cascade-And-Sort and Cascade Further, while the variance is very small in the Prioritize Points in Heavy Lines Algorithm. In Prioritize Points in Heavy Lines, our strategy of attacking the kernel is also better than those of Cascade-And-Sort and Cascade Further, which are based solely on sorting the kernel prior to calling BST-Dim-Set-Cover. The Prioritize Points in Heavy Lines Algorithm also incorporates another strategy, which is sorting the point p on lines with at least $\lceil n/k \rceil$ points according to their degree (number of lines through p). The result in Table 2.10 confirms that this strategy indeed improves the performance of this algorithm for the YES-instance. Finally, Prioritize Points in Heavy Lines has the practical limit for k of about 30; whereas, other known algorithms perform well up to $k = 6$.

Table 2.13: Average running times on structured instances for the optimization problem (10 shuffles on the same input file).

		Input Profile						
		Small-k			Large-k	Elongated-Grid		
	k	4	4	4	7	4	4	4
Algorithm	n	100	1,000	10,000	21	100	1,000	10,000
One increments		0.2s	0.3s	1.7s	2.2s	0.2s	0.3s	1.7s
Take a guess		0.2s	0.3s	1.7s	1.8s	0.2s	0.3s	1.7s
Binary Search		0.2s	0.3s	1.7s	2.1s	0.2s	0.3s	1.7s
ExactLineCover		0.9s	1.2s	12s	≈ 6hr	0.9s	1.3s	12s

		Input Profile (cont.)						
		Grid			Grid & Sparse	Sparse	Star	Wave
	k	5	6	7	6	7	6	6
Algorithm	n	25	36	49	22	15	36	23
One increments		1.4s	3s	6s	1s	1s	2.4s	1.3s
Take a guess		1.5s	3.6s	7s	1.2s	1.1s	3s	1.6s
Binary Search		3.8s	5.3s	10s	1.7s	0.7s	4s	2s
ExactLineCover		4.2s	290s	≈ 4hr	161s	≈ 2hr	283s	188s

Algorithms for the Optimization Problem

We have tested the decision version of our algorithms with profiles such as those in Figure 2.7. When we look at the optimization version with the same profiles, we get the results shown in Table 2.13.

Discussion

Our three algorithms outperform Algorithm ExactLineCover in all cases. The ExactLineCover Algorithm is no longer practical when k is 7, because it solves the instances Sparse, Grid, and Large-k in two hours, four hours, and six hours, respectively; our three algorithms solve them in less than 10 seconds.

2.6.3 Implementation Issues

In our experiments, when we read the input points, we removed the duplicated points. Note that this is Reduction Rule 3. This step was performed in all algorithms to ensure that all algorithms were working under the same assumptions. This extra work requires $O(n^2)$ time or $O(n \log n)$ time if we sort the input. This step was not reflected in the timings; however, if we also were to consider the time taken to clean the input from duplicates, then all algorithms would have a running time with an additional $O(n \log n)$ term.

In CGAL, we can either construct the line arrangement using an incremental insertion function or an aggregated insertion function. Theoretically, construct-

ing the arrangement using an incremental insertion function can be performed in $O(n^2)$ time, where n is the number of lines [CGA07]. Constructing the arrangement using the aggregated insertion function can be performed in $O((n+k)\log n)$ time, where k is the total number of intersection points [CGA07]. Although the running time of the second function is asymptotically better than the first one when the arrangement is relatively sparse (a small number of intersection points), we observed that the first function performed better for the construction of the dual-line arrangements. This is because the dual-line arrangement is largely filled with intersection points. If we have 1,000 points, then the 1,000 lines in dual space have close to 1 million intersection points already (at least one intersection for every two points).

Duality mapping is not defined for the vertical lines. Therefore, the implementation of our algorithm and Grantson and Levcopoulos's algorithm (and any algorithms that use Guibas et al. [GOR96]) need to pay attention to this aspect. The solution is to rotate the scene so that there are no vertical lines [dBvKOS00]. We did not include this work in our timings.

2.7 Chapter Summary

Our algorithms provide exact solutions. In this thesis, approximation algorithms or probabilistic algorithms that give an answer with a high probability of correctness were not considered. We looked at the performance of our algorithms in comparison with the other algorithms. We acknowledge that the purpose of Langerman and Morin's algorithm was to produce theoretical results. On the other hand, our aim was to show that the parameterized approach can lead to algorithms that can handle large instances in practice. We also proved experimentally that the new simple reduction rules presented here have significant benefits in implementation, although theoretically, they do not produce a smaller kernel. We solved the kernel using a bounded-search-tree technique with a new heuristic that examines each candidate line around a given point. Although this idea did not improve the theoretical time complexity (asymptotically, the O-notation is the same), it significantly improved the running time in practice.

Chapter 3

The Rectilinear Covering Problems in Higher Dimensions

This chapter discusses two rectilinear (that is, axis-parallel) covering problems in d dimensions and their variants. The first problem is the Rectilinear Line Cover where the inputs are n points in $\mathbb{R}^d$ and a positive integer k. We are asked if we can cover these n points with at most k lines where these lines are restricted to be axis-parallel. We show that this problem has efficient fixed-parameter algorithms. The second problem is the Rectilinear k-Links Spanning Path Problem where the inputs are also n points in $\mathbb{R}^d$ and a positive integer k, but here we are asked if there is a piecewise linear path through these n points, having at most k line-segments (links) where these line-segments are axis-parallel. We prove that this problem is FPT under the assumption that no two line-segments share the same line. Previous to this result, only conjectures were stated over several years on the NP-completeness of the second problem. We provide the proof that the Rectilinear Minimum-Links Traveling Salesman Problem and the Rectilinear Minimum-Links Spanning Path Problem are NP-complete by a reduction from the One-In-Three 3-SAT problem. Thus, we deal with the intractability just discovered with fixed-parameter tractability. Moreover, if we extend our problems to a finite set of orientations, our approach proves these problems remain FPT.

3.1 Introduction

A *rectilinear line* is an axis-parallel line that is either horizontal or vertical. When the Line Cover problem is restricted to only rectilinear lines, we call this the Rectilinear Line Cover problem. Consider a path that is made up of more than one line-segment, the number of the line-segments in a given path is often called the *link length*. A path having a minimum number of line-segments or links is called a *minimum-links path* [Wag06].

The minimum-links path is equivalent to a path having the minimum number of bends (or links) and it could be self intersecting. Therefore, the Minimum-Links Path Problem is sometimes referred to as the Minimum-Bends Path Problem. However, the number of bends in the path is actually one less than the number of line-segments that make up the path. In general, the Minimum-Bends Path Problem attempts to find a path between two distinct points that has a minimum number of bends. Finding the path between two points (usually with obstacles) is different from finding the path that passes through a set of points. The latter path is called a *spanning path* [Col04, BBD+09] and the problem of finding the spanning path that covers a set of points, such that this path has the minimum number of links, is known as the Minimum-Links Spanning Path Problem. This problem is NP-complete by a reduction from the Edge Embedding on a Grid [BBD+09]. An alternate proof of NP-completeness was given by Arkin et al. [AMP03] who showed that the problem of finding a minimum-links tour is NP-complete by a reduction from the Line Cover.

When the Minimum-Links Spanning Path Problem is restricted to allow only rectilinear lines, we call this the Rectilinear Minimum-Links Spanning Path Problem. It was conjectured in 2007 to be NP-complete [BBD+07], and again, the conjecture appeared in 2009 [BBD+09]. Bereg et al. [BBD+09] suspected that this problem is NP-complete, because Hassin and Megiddo [HM91] show that the Rectilinear Line Cover problem in three dimensions (or higher) was NP-complete, but they also left this issue open. The problem of traversing a set of points and minimizing the number of links in the tour where the links are restricted to being rectilinear lines is called the Rectilinear Minimum-Links Traveling Salesman Problem. Wagner raised and

left open whether this problem is NP-complete or if it is polynomially solvable [Wag06]. Thus, one goal of this thesis is to provide the NP-completeness proofs of the Rectilinear Minimum-Links Spanning Path Problem and of the Rectilinear Minimum-Links Traveling Salesman Problem.

There are applications in 2D and 3D where covering while minimizing turns is necessary, because turns are considered very costly. We have mentioned the applications in VLSI design and the movement of heavy machinery in the previous chapter. In this chapter, we study the Rectilinear Line Cover and the Rectilinear Minimum-Links Spanning Path Problem in $d \geq 3$ dimensions. We acknowledge that when $d > 3$ dimensions, the problems considered are interesting from, mainly, a theoretical point of view.

If we reformulate the problem of covering points with the minimum number of rectilinear lines as a decision problem with parameter $k > 0$, then we have to answer whether we can cover S with k or fewer rectilinear lines. We refer to this problem as the Rectilinear Line Cover problem. This problem was shown to be fixed-parameter tractable by Langerman and Morin [LM05] who provided the algorithms for the general Line Cover. We prove that the Rectilinear Line Cover in d dimensions is FPT, based on the alternative idea. This new approach improves the time complexity of the previous algorithms [LM05]. We also look at the parameterized version of the Minimum-Links Spanning Path Problem and we refer to this as the Rectilinear k-Links Spanning Path Problem. We study the Rectilinear k-Links Spanning Path Problem in three dimensions and we extend our new result to the general d dimensions, based on the FPT design technique called the bounded-search-tree.

Derick Wood and his colleagues studied many computational geometric problems with restricted orientations [RW88, RW91, FW96, FW04]. Here, we study the Line Cover and the Minimum-Links Spanning Path Problem problems, where a set of k lines is restricted to the lines having a finite number of orientations.

3.2 Related Work

We review three similar classes of the covering problems. The first class is the Line Cover problem in higher dimensions, that finds a minimum number of

lines used to cover all the input points. The second class finds a path between two distinct points and the third class finds a path that passes through a set of points (spanning path). Moreover, we have a restricted version of the problems. That is, the rectilinear version requires all lines and lines on objects to be parallel to the coordinate system; whereas, the unrestricted or general version (non-rectilinear) allows arbitrary orientations of the lines. We review the three classes of the problems in both general and rectilinear versions.

3.2.1 Lines Covering a Set of Points in Higher Dimensions

Although Grantson and Levcopoulos [GL06] provided the approximate algorithm and the exact algorithm to compute the smallest number k of straight lines that cover n points, their algorithms only solve the problem in two dimensions. In the higher dimensions, the decision problem of the Line Cover was shown to be fixed-parameter tractable by Langerman and Morin [LM05], who provided two FPT-algorithms. The first uses the bounded-search-tree technique and has $O(k^{dk}n)$ time complexity, while the second one uses kernelization and has $O(k^{d(k+1)} + n^{d+1})$ time complexity, where d is the number of dimensions. Their algorithms are presented in an abstract setting that does not restrict the lines to be either horizontal or vertical, but can be adapted to the rectilinear case with the same complexity.

3.2.2 Path Between Two Distinct Points

Without obstacles, finding a path between two distinct points can, trivially, be performed in polynomial time. Therefore, the challenge in path problems is to avoid obstacles that block the direct path. Obstacles are typically required to have axis-parallel edges if the problem is set in a rectilinear domain where the path also must be orthogonal.

- **Shortest Path:** One can compute the shortest path among a set of polygonal obstacles in polynomial time. The complexity of the problem was $O(n^2)$ time in the worst-case and later was improved to subquadratic time [Mit93, KMM97]. Note that n is the number of vertices of obstacles. The shortest rectilinear path can also be solved in polynomial time.

For example, the algorithm of Lee et al. [LCY90] computes the shortest rectilinear path among rectilinear weighted obstacles in $O(n \log^{3/2} n)$ time, where n denotes the number of vertices of obstacles.

- **Minimum-Links Path:** Finding a minimum-links path with obstacles between two distinct points is also polynomial in both the general and the rectilinear versions. In the general version, Arkin et al. [AMS92] show that this problem in a simple polygon can be answered in $O(\log n)$ time with $O(n^3)$ time for preprocessing the data structure, where n is the number of vertices in the polygon. In the rectilinear version, de Berg et al. [dBvKNO92] solve a minimum-links path between two distinct points in $O(n^d \log n)$ where $d \geq 3$ is the number of dimensions and n is the number of axis-parallel boxes in d-space.

3.2.3 Path Covering a Set of Points

This class of the problems finds a spanning path that covers a set of points. The main objective is to minimize the number of turns in the path.

- **Minimum-Links Spanning Path:** The general version of the Minimum-Links Spanning Path Problem is NP-complete. However, as we explained earlier, for the rectilinear version, it was not previously known whether it is NP-complete [BBD+09]. For the rectilinear version (2D and 3D), Collins [Col04] improves the upper and lower bounds on the number of line-segments. Bereg et al. [BBD+09] improve these bounds further and also provide a constant-factor approximation algorithm for the Minimum-Links Spanning Path Problem in the rectilinear d-dimensional case.

3.2.4 Our Contribution

We saw earlier that the Shortest Path and the Minimum-Bends Path Problem can be solved in polynomial time, but there are interesting variants that are NP-complete. The Rectilinear Line Cover in three dimensions (or higher) is already known to be NP-complete, but only conjectures exist for the status of the Rectilinear Minimum-Links Spanning Path Problem as we mentioned in the Introduction. Thus, we are also interested in establishing the complexity of this problem. We make the following contributions.

- We improve the previously-known FPT result for the Rectilinear Line Cover in higher dimensions. We give two FPT-algorithms, one based on the bounded-search-tree technique and the other based on the kernelization approach.
- We prove that the Line Cover problem, when restricted to a finite number of orientations, is FPT.
- We prove that the decision version of the Rectilinear Minimum-Links Traveling Salesman Problem is NP-complete.
- We then prove that the decision version of the Rectilinear Minimum-Links Spanning Path Problem is also NP-complete.
- We give an FPT-algorithm for solving the Rectilinear Minimum-Links Spanning Path Problem in higher dimensions under the assumption that no two line-segments share the same line.
- We prove that the Minimum-Links Spanning Path Problem, when restricted to a finite number of orientations, is FPT.

3.3 Rectilinear Line Cover Problem

3.3.1 Using a Bounded-Search-Tree

In this section, we will use the bounded-search-tree technique to show that the Rectilinear Line Cover belongs to the class FPT. In this problem, given n points in $\mathbb{R}^d$ and a positive integer k, we are asked whether it is possible to cover these n points with at most k lines where these lines are restricted to being axis-parallel. First we discuss the problem in three dimensions. Consider the search tree in Figure 3.1. We choose a point $p \in S$ and we explore the possibilities of a cover at this given point. In a YES-instance, this point must lie on at least one of the lines NS, EW or UD[1]; thus, we can construct a search tree with fan-out of three. The task is to explore exhaustively these three candidate orientations. At each node of the tree, we select a point p not already covered. We expand three branches corresponding to the NS-orientation, the EW-orientation and

[1]N, S, E, W, U, and D stand for North, South, East, West, Up and Down respectively.

the UD-orientation, respectively, and in each branch, we assign one of the three orientations to the point p. The point p is marked as covered in the recursive calls to cover the rest of S. The points that fall into the same line with p are also marked as covered so that we do not consider these points again in the next recursive call. We keep track of how many assignments have been made and we do not assign more than k orientations (k rectilinear lines). That is, every recursive call reduces the value of k by one. Each branch of the tree stops when the upper bound has been reached or when every point is marked as covered. If the algorithm finds a leaf with k lines that cover every point in S, it answers YES. If all leaves result in no cover, then the algorithm answers NO. The algorithm is correct, because if there is a cover with k lines, we can follow the path in the tree to the leaf that agrees with the cover. We examine how each point in the node of the path is covered by the witness cover.

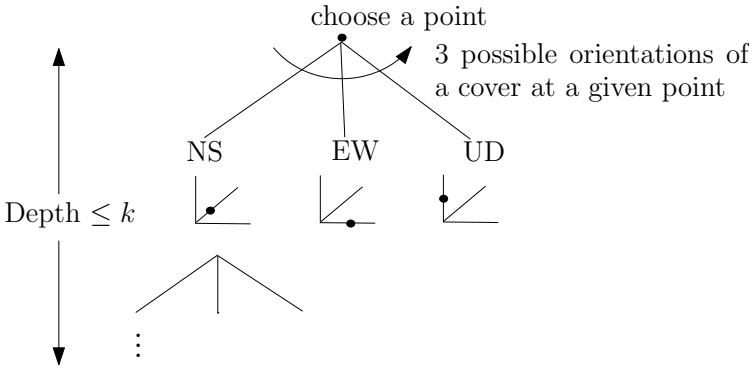


Figure 3.1: A search tree of the Rectilinear Line Cover problem in 3D with depth k.

Therefore, the depth of the tree is at most k. Note that the fan-out is a constant $d = 3$ and the depth of the tree is bounded by the parameter k, and not the input size n. The number of nodes in the tree is bounded by 3^k, thus we obtain an FPT-algorithm, because its time complexity is linear in the size of the input, although it is exponential in the parameter.

Note that when the problem is moved to d dimensions, the search tree will have to explore d candidate orientations at the given point p. Consider a set C of candidate orientations with the following members $\{x_1, x_2, x_3, \ldots, x_d\}$ where $x_i \in C$ is a straight line parallel to the i-th axis. The task is, again, to select a

point p for which a cover orientation x_i for $1 \leq i \leq d$ has not been determined, and then assign one of the d orientations to the point p and continue as in the three dimensional case. Recall that the Rectilinear Line Cover in two dimensions is polynomially solvable but in three dimensions (or higher) this problem is NP-complete [HM91].

Theorem 1. *The* Rectilinear Line Cover *in* $d \geq 3$ *dimensions can be solved in* $O(d^k n)$ *time.*

Proof. The work at the leaves and the nodes is linear in n and the number of nodes is $O(d^k)$. The total time complexity is then $O(d^k n)$. The problem is FPT, because it can be decided in running time $f(k) \cdot n^{O(1)}$. □

Note that the FPT-algorithms of Langerman and Morin [LM02, LM05] are generic enough for the Rectilinear Line Cover problem, but their complexities do not improve. Their first algorithm uses a bounded-search-tree and requires $O(k^{dk} n)$ time, while the second algorithm uses kernelization and requires $O(k^{d(k+1)} + n^{d+1})$ time. Our FPT-algorithm offers a better complexity than the FPT-algorithms of Langerman and Morin when specialized to the rectilinear case. Next, we will exploit the kernelization approach to solve the Rectilinear Line Cover problem and reduce the time complexity even further.

3.3.2 Using Kernelization

This approach reduces the problem to a smaller one using reduction rules. The rules are applied repeatedly until none of the rules applies. If the result is a problem of size no longer dependent on n, but only on k, then the problem is kernelizable, because the kernel can be solved by exhaustive search in time that depends only on the parameter. We now present our reduction rules for the Rectilinear Line Cover problem.

If k is large enough, we could use one rectilinear line for each point in S.

Reduction Rule 7. *If* $k \geq n$*, then the answer is YES.*

Consider the simplest case when $k = 1$, we would need to have all n points in S in a rectilinear line.

Reduction Rule 8. *If $k = 1$, then the answer is YES if and only if all points in S lie on only one rectilinear line, otherwise the answer is NO.*

If the input S has repeated points, then we can remove the points that are repeated.

Reduction Rule 9. *If S has repeated points, then S can be simplified to a set with no repetitions and the answer for the simplified set is an answer for the original input of the* Rectilinear Line Cover *problem.*

If a rectilinear line covers many points, it must be used in the cover (specifically, we need at least $k + 1$ points).

Reduction Rule 10. *If there is a rectilinear line with $k + 1$ or more co-linear points, place the line through them in the cover and remove them from further consideration. If there is a line with $k + 1$ or more co-linear points and the line is not rectilinear, then the answer is automatically NO.*

Similar to the rule for arbitrary lines [LM05], the above rule is correct, because if the rectilinear line through the $k + 1$ different points was not in the cover, then we would need more than k rectilinear lines to cover just these points. Thus, if the set accepts a cover with k lines, any rectilinear line with $k + 1$ or more points must be in the cover. If the line through the $k + 1$ points is not rectilinear, then we would also need more than k rectilinear lines to cover just these points. Since we do not accept the non-rectilinear line into the cover and we do not accept more than k rectilinear lines, the answer here is immediately NO.

The reduction rules above can be formulated into a kernel lemma. The result below leads to a quadratic size kernel.

Lemma 7. *If there is a subset S_0 of S, such that $|S_0| \geq k^2 + 1$, and the largest number of co-linear points of S_0 on a rectilinear line is at most k, then the answer is NO.*

Proof. Similar to the rule for arbitrary lines [GL06], if all rectilinear lines covered at most k points, any cover of S_0 with k lines would cover at most k^2 points. This means we cannot cover S_0, let alone S with k lines. □

Next, we let L_2 be the set of all rectilinear lines that cover at least two points in S; that is, a rectilinear line l is in L_2 if and only if there are two or more co-linear points of S in l. We let $cover(L_2)$ be all points in S covered by a line in L_2. With this notation, we now describe two more reduction rules.

Reduction Rule 11. *Let p be a point in $S \setminus cover(L_2)$, that is, $p \in S$, but $p \notin cover(L_2)$. Let $S' = S \setminus \{p\}$. Replace (S, k) by $(S', k-1)$.*

This reduction rule essentially says that if we can find a point that is never covered by a rectilinear line that covers two or more points, then we can reduce to a smaller problem that ignores this point and uses one less line.

Reduction Rule 11 is correct. Since p is the only point in a rectilinear line, any cover with $k-1$ lines of $S \setminus \{p\}$ can be made a cover of $S \cup \{p\}$ with one more rectilinear line. Also, a rectilinear cover C of S must use a rectilinear line l_p over p but $C \setminus \{p\}$ has $k-1$ rectilinear lines and covers $S \setminus \{p\}$.

Reduction Rule 12. *Let q be a point in $cover(L_2)$. Suppose that there is no other line in L_2 besides l_q that also covers q (that is, $\{l \in L_2 | l \text{ covers } q\} = \{l_q\}$). Let $S' = S \setminus cover(l_q)$. Replace (S, k) by $(S', k-1)$.*

This reduction rule is similar to the previous rule. That is, if we can find a point that can only be covered by one line and not by any other lines (even though this line is in L_2), then we can reduce to a smaller problem that ignores the points that lie on this line and reduce our budget by one.

Reduction Rule 12 is correct, because if a cover C did not use l_q, it must use the other rectilinear line $l_i \notin L_2$ to cover q. The line $l_i \in C$ can cover only the point q, thus $C \cup \{l_i\} \setminus \{l_q\}$ is a cover of the same cardinality that uses l_q.

Note that these two rules are similar to Reduction Rule 5 and Reduction Rule 6 in Chapter 2, where the more reduction rules we can apply, the smaller the kernel is. The use of these two rules is optional here. In theory, the rules do not reduce the size of our kernel. However, in practical circumstances, the incorporation of these additional rules can result in a kernel that has a size significantly fewer than k^2 points (or the size even goes down to zero) [ECHS09]. Therefore, the rules may be of interest for practical implementation and experimentation of expected-case performance.

We apply the above reduction rules in the preprocessing phase. When we apply these rules one after another until none of the rules applies, we have a problem kernel whose size is determined theoretically by Reduction Rule 10.

Lemma 8. *Any instance (S, k) of the of the* Rectilinear Line Cover *problem can be reduced to a problem kernel S' of size $O(k^2)$.*

Proof. Let S' be the set of points after which we cannot repeat the removal of points covered by a rectilinear line covering $k + 1$ or more points (Reduction Rule 10). If we repeated the removal more than k times, we know that it is a NO-instance. If we repeated no more than k times and it is a YES-instance, every line covers at most k points; thus, any cover with k lines would cover at most k^2 points. □

Now, we can invoke the bounded-search-tree approach in the previous section to solve the kernel. Recall that this algorithm runs in $O(d^k n)$ time. However, the exponential term now is no longer multiplied by n, because the input instance will be the reduced instance (the kernel). Therefore, we can improve the exponential term to $(d^k k^2)$ time and obtain a better algorithm. However, we will introduce an alternative approach to solve the kernel using the bounded-search-tree technique in conjunction with the application of Reduction Rule 12. Consider an alternative bounded-search-tree that focuses on the set of candidate lines $L_p \subset L_2$ that go through a given point $p \in S'$ where S' is the kernel and $|S'| \leq k^2$. Note that the number of axis-parallel lines that go through any point in $\mathbb{R}^d$ is at most d. Therefore, the branching of the search tree is bounded by the value of d. Our strategy is to choose a point $p \in S'$ that is covered by a set of axis-parallel lines $L_p \subset L_2$, and then we test each of the candidate lines in L_p. The number of recursive calls in each level of the tree is controlled by $|L_p| \leq d$. At each node of the tree, we also invoke Reduction Rule 12. If the rule applies, this will further reduce the size of the instance before we move to the next level of the tree. We illustrate the idea in Figure 3.2.

There are two main tasks to be performed at each node of the tree.

1. We pick a point p, and test the candidate lines that go through p. These are the lines l_1, l_2 and l_3 in Figure 3.2. If we pick l_1, we mark all the points that lie on l_1 including p as covered in a recursive call to cover the rest of $S' \setminus cover(l_1)$. We remove l_1 from L_2 and remove the covered points from $cover(L_2)$.

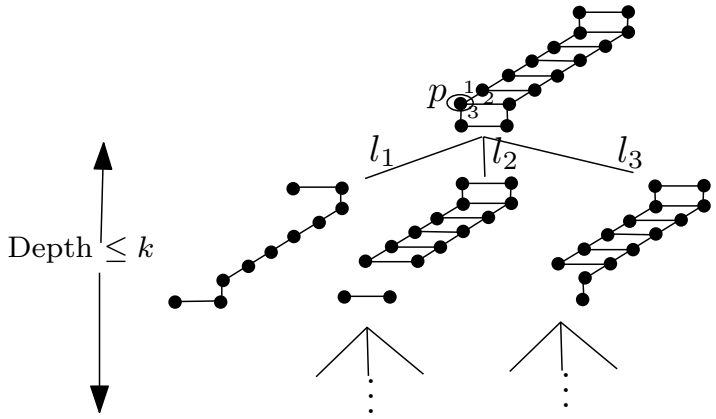


Figure 3.2: Example of the ladder instance ($n = 16, k = 4$). The bounded-search-tree is used in conjunction with Reduction Rule 12.

2. We check if Reduction Rule 12 can be applied to the instance $S' \setminus cover(l_i)$. The rule can be exhaustively applied no more than k times if the instance is a YES-instance. Although this technique appears to increase the cost to the algorithm, it can reduce the size of the instance and thus essentially, could reduce the depth of the tree. The best case scenario is shown in the left branch of Figure 3.2, where the rule removes all the remaining candidate lines and the tree stops at the first level of the tree. The worst case is when the rule cannot be applied at any level of the tree. However, this is not exactly true, because at the $(k-1)$-th level, there will be at most 2 lines left to cover. When one of the lines is taken, Reduction Rule 12 will automatically remove the remaining line and there is no need to proceed to the k-th level. Hence, effectively, the depth of the search tree is at most $k - 1$.

We repeat the two tasks above in each recursive call until there are no points left to cover or the tree has reached the upper bound that is the depth of $k - 1$. Here, if every point in S' is covered, we answer YES. Otherwise, we answer NO.

In addition, we can incorporate a greedy strategy, such as sorting heavy-lines (lines that cover many points) that go through p, to speed up the process. Alternatively, we may use a hashing scheme to avoid exploring the same candidate line in the search tree twice. These strategies may be of interest for practical implementation and experimentation of expected-case performance, but we do

not elaborate on the idea here. We now conclude our result as follows.

Lemma 9. *The* Rectilinear Line Cover *in $d \geq 3$ dimensions can be decided in $O(n^2d^2 + d^{k-1}k^2)$ time.*

Proof. Reduction Rule 7 can be performed in constant time. Reduction Rule 8 can be applied in $O(nd)$ time. Reduction Rule 9 can be checked in $O(n^2d)$ time. Reduction Rule 10 can be applied if using a naive approach in $O(n^2d^2)$ time by computing $O(dn)$ rectilinear lines and checking if these lines cover at least $k+1$ points. Hence, the preprocessing phase can be carried out in $O(n^2d^2)$ time. The depth of the new bounded-search-tree is at most $k-1$ and the fan-out is at most d. The number of nodes in the tree is $O(d^{k-1})$. The work at each node is to remove at most k lines each having at most k points and update which points are covered by which lines (that is, update a set L_2). Note that the set L_2 can be computed in the preprocessing phase that is already bounded by the term $O(n^2d^2)$. To update L_2, we require k^2 time to update what points are covered by which remaining lines. Therefore, the overall work of the bounded-search-tree is $O(d^{k-1}k^2)$ time and the total time complexity is $O(n^2d^2 + d^{k-1}k^2)$. □

3.4 Line Cover Restricted to a Finite Number of Orientations

We now extend our result by considering the Line Cover problem, where a set of k covering lines is restricted to the lines having a finite number of orientations, ϕ. This is the case where these k lines must have their orientations taken from a given finite set $R = \{r_1, r_2, \ldots, r_\phi\}$ where $|R| = \phi$ [Güt83, RW91].

We can construct the search tree with ϕ branches similar to the one in Figure 3.1 where each branch represents a cover orientation that is a straight line with a slope r_i, for $1 \leq i \leq \phi$. We then explore the possibilities of a cover at a given point $p \in S$, as we did before in the previous section. Since any point must lie on at least one of the ϕ orientations (slopes), we select a point p for which a cover orientation has not been determined, and assign one of the ϕ orientations to the point p. The point p together with the points that fall into the same cover with p are marked as covered in the recursive call. After k recursive calls, we have a leaf where we either have a cover or not. If all leaves do not lead to

a cover, this is a NO-instance. A leaf with a cover provides the certificate for a YES-instance. The depth of the tree is at most k. The number of nodes in the tree is bounded by ϕ^k. Thus, we obtain an FPT-algorithm, because its time complexity is polynomial in the size of the input n, although it is exponential in the parameter k. Theorem 2 below summarizes this result.

Theorem 2. *The* Line Cover *problem when restricted to the lines having a finite number $\phi \geq 2$ of orientations is FPT.*

3.5 NP-Completeness Results

We will first prove that the Rectilinear Minimum-Links Traveling Salesman Problem in three dimensions is NP-complete by a reduction from One-In-Three 3-SAT, a known NP-complete problem. It is easy to show that the Rectilinear Minimum-Links Traveling Salesman Problem belongs to NP. The verification algorithm checks whether a given set of $|S|$ points in $\mathbb{R}^3$ can constitute a tour with at most k links and every link in the tour is axis-parallel. This can be performed in polynomial time.

Now, we show that the One-In-Three 3-SAT problem can be reduced in polynomial time to the Rectilinear Minimum-Links Traveling Salesman Problem. The One-In-Three 3-SAT problem is to decide whether there exists a satisfying assignment to the variables so that each clause has exactly one true literal and, thus, exactly two false literals. In contrast, the 3-SAT problem only requires that every clause has at least one true literal. The One-In-Three 3-SAT problem remains NP-complete even when no clause contains a negated literal [GJ79, LO4]. Hence, to simplify our reduction, we will assume that no literals are negated in our problem.

Suppose that we are given an instance of One-In-Three 3-SAT with a set of m clauses $E_j = w'_j \vee w''_j \vee w'''_j$ for $j = 1, 2, \ldots, m$ where $\{w'_j, w''_j, w'''_j\} \subset \{u_1, \bar{u}_1, \ldots, u_{n'}, \bar{u}_{n'}\}$. From the instance $E_1 \wedge \ldots \wedge E_m$ of One-In-Three 3-SAT we construct an instance of the Rectilinear Minimum-Links Traveling Salesman Problem by gadget design. We represent each variable u_i by a set S_i of $4400m$ points, for $i = 1, 2, \ldots, n'$. We represent each clause E_j with three points p'_j, p''_j, p'''_j, for $j = 1, 2, \ldots, m$. We let $P = \{p'_1, p''_1, p'''_1, \ldots, p'_m, p''_m, p'''_m\}$

be the set of points representing all clauses. The instance of the Rectilinear Minimum-Links Traveling Salesman Problem will have a set $S = P \cup \bigcup_{i=1}^{n'} S_i$ where $|S| = 4400mn' + 3m$. The set S_i representing u_i will be made of a gadget. Such a gadget will be forced to use $22m$ different covering line-segments parallel to one of the axes in $\mathbb{R}^3$. Because of this, rather than describing the points in S_i, we will be describing these line-segments. The value $4400m$ is chosen so that there are many points in the covering line-segments that a rectilinear tour with the minimum number of bends will be forced to use each of these line-segments.

The plan for the proof is to show that there exists a satisfying assignment for $E_1 \wedge \ldots \wedge E_m$ if and only if a set of $|S|$ points in 3D can be covered optimally by a rectilinear tour that has k links, where k will be set to $34mn' + 2n' - 2$. The tour will be made up of two types of line-segments, the covering line-segments we alluded to before which cover $\bigcup_{i=1}^{n'} S_i$ and line-segments that do not cover points in $\bigcup_{i=1}^{n'} S_i$. The second type of line-segments joins the first type of line-segments together. We will design the gadget (and the set S_i) representing u_i so that there are two ways of covering the points in S_i. These two ways will be used to represent a variable being assigned the value "true" or "false". To present our 3D gadget we require some notation.

Definition 16. *The three possible types of line-segments parallel to an axis in* $\mathbb{R}^3$ *are called* **forms** *and are denoted by* $line(x, y, *), line(x, *, z)$ *and* $line(*, y, z)$.

According to the above definition, a point with a coordinate (x_0, y_0, z_0), therefore, can be covered only by $line(x_0, y_0, *)$, $line(x_0, *, z_0)$ or $line(*, y_0, z_0)$.

Definition 17. *We denote the three axis-parallel planes with a fixed* x*-value, a fixed* y*-value, and a fixed* z*-value as* $\Pi(x, *, *)$, $\Pi(*, y, *)$, *and* $\Pi(*, *, z)$, *respectively.*

Definition 18. *Given two line-segments of the form* $A = (x_1, y_1, *)$ *and* $B = (x_1, y_2, *)$ *that are in the plane* $\Pi(x_1, *, *)$ *we write* $A \sqcap B$ *when they are joined by one line-segment and two bends on the high* z*-values (the joining line-segment has the form* $(x_1, *, z_\sqcap)$ *and the* z*-value* $z_\sqcap$ *is larger than any* z*-value in* A *or* B*). We denote* $A \sqcup B$ *when they are joined by one line-segment* $(x_1, *, z_\sqcup)$ *and two bends, and the* z*-value* $z_\sqcup$ *is smaller than any* z*-value in* A *or* B *(see Figure 3.3). Similarly, given two line-segments of the form* $A = (x_1, *, z_1)$ *and* $B = (x_2, *, z_1)$

that are in the plane $\Pi(*, *, z_1)$*, we write* $A \sqcap B$ *when they are joined by one line-segment* $(*, y_\sqcap, z_1)$ *and two bends, and* $y_\sqcap$ *is larger than any* y*-value in* A *or* B*. While* $A \sqcup B$ *is when the line-segment* $(*, y_\sqcup, z_1)$ *joining them has* $y_\sqcup$ *smaller than any* y*-value in* A *or* B*. For the third option the definition is analogous. Finally,* $A \oplus B$ *is when the line-segments* A *and* B *are extended and joined with one bend.*

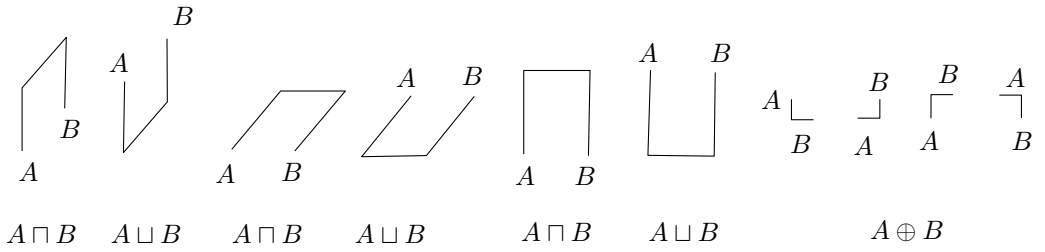
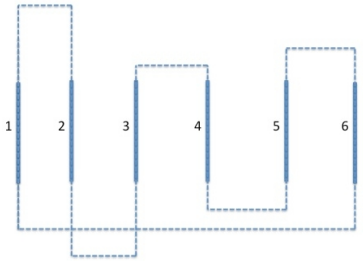


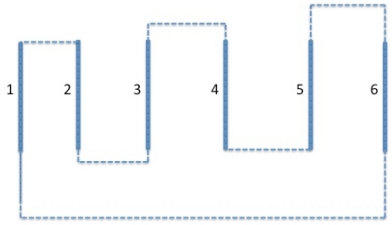
Figure 3.3: Definitions when joining two axis-parallel line-segments.

Definition 19. *If two rectilinear tours cover a set* $T = \{l_1, \ldots, l_t\}$ *of line-segments in the same order (or exactly in the reverse order) with the same number of bends, we say they are equivalent.*

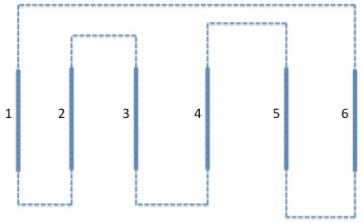
Figure 3.4 illustrates the notion that two parallel line-segments l_i and l_j can be covered with two bends in two ways: $l_i \sqcap l_j$ and $l_i \sqcup l_j$, and the line-segment in between is free to shift by extending the tour's line-segments that cover l_i and l_j. Figure 3.4(a) and Figure 3.4(b) are in the same equivalence class, because $l_i \sqcap l_j$ and $l_i \sqcup l_j$ alternate for the same line-segments and enable us to extend the tour by shifting the horizontal line-segments in the gap of $line_2$-$line_3$ and the gap of $line_4$-$line_5$ downwards. This may suggest this equivalence class as the "true" setting of the gadget. While Figure 3.4(c) and Figure 3.4(d) enable the extension downwards in the gap of $line_1$-$line_2$, the gap of $line_3$-$line_4$, and the gap of $line_5$-$line_6$ as the false setting. However, Figure 3.4(e) shows a different equivalence class that respects the same extensions downwards. Moreover, Figure 3.4(f) shows even one more equivalence class and this one mixes the possible extensions. Our gadgets will be placed in 3D because we can alternate the planes where parallel line-segments are placed and, in this way, restrict the equivalence classes of the rectilinear tours with minimum bends to two equivalence classes. We call the 3D gadgets, "true-false" gadgets.



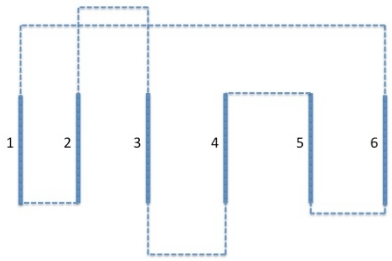
(a) A member of the equivalence class $line_1 \sqcap line_2$, $line_2 \sqcup line_3$, $line_3 \sqcap line_4$, $line_4 \sqcup line_5$, $line_5 \sqcap line_6$, $line_6 \sqcup line_1$.



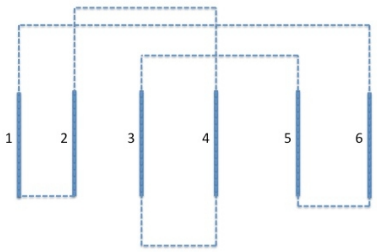
(b) Another member of the equivalence class $line_1 \sqcap line_2$, $line_2 \sqcup line_3$, $line_3 \sqcap line_4$, $line_4 \sqcup line_5$, $line_5 \sqcap line_6$, $line_6 \sqcup line_1$.



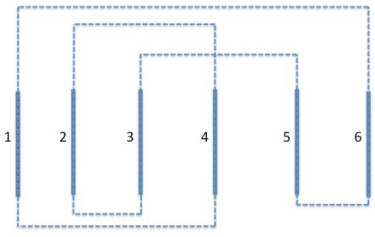
(c) A member of the equivalence class $line_1 \sqcup line_2$, $line_2 \sqcap line_3$, $line_3 \sqcup line_4$, $line_4 \sqcap line_5$, $line_5 \sqcup line_6$, $line_6 \sqcap line_1$.



(d) Another member of the equivalence class $line_1 \sqcup line_2$, $line_2 \sqcap line_3$, $line_3 \sqcup line_4$, $line_4 \sqcap line_5$, $line_5 \sqcup line_6$, $line_6 \sqcap line_1$.



(e) It is not a member of the other equivalence classes but it respects where extensions are possible.



(f) It is not a member of the other equivalence classes and it enables other extensions.

Figure 3.4: Different tours with the same number of bends cover six vertical line-segments in 2D.

3.5.1 A Building Block

First, we illustrate the true-false gadgets for a variable u_i when $m = 1$ (see Figure 3.5). We call the gadget when $m = 1$, a *building block*, because we will repeat the structure of this gadget m times for a variable participating in m clauses.

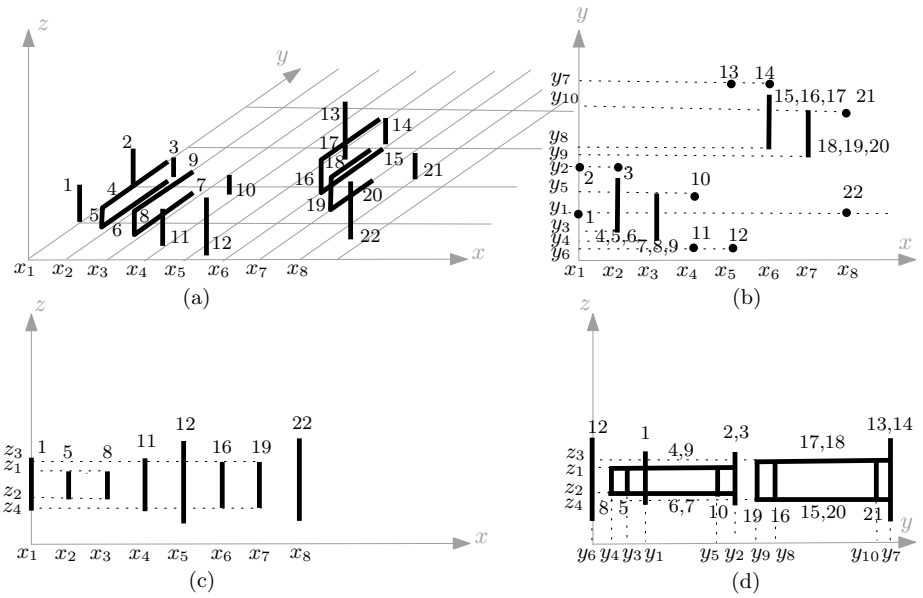


Figure 3.5: The structure of S_i (for $m = 1$) represents a variable u_i, (a) the building block in three dimensions, (b)–(d) the three different projections.

The building block of gadgets has the following properties:

1. All points lie within eight different x-values $\{x_1, \ldots, x_8\}$, ten different y-values $\{y_1, \ldots, y_{10}\}$ and four different z-values $\{z_1, \ldots, z_4\}$.

2. In each line-segment illustrated with a solid line in Figure 3.5, we place 200 points equally spaced along the segment. There are many points in the line-segments so that a path with the minimum number of bends will be forced to use these line-segments.

3. The maximum number of line-segments of the same form in the same axis-parallel plane is two.

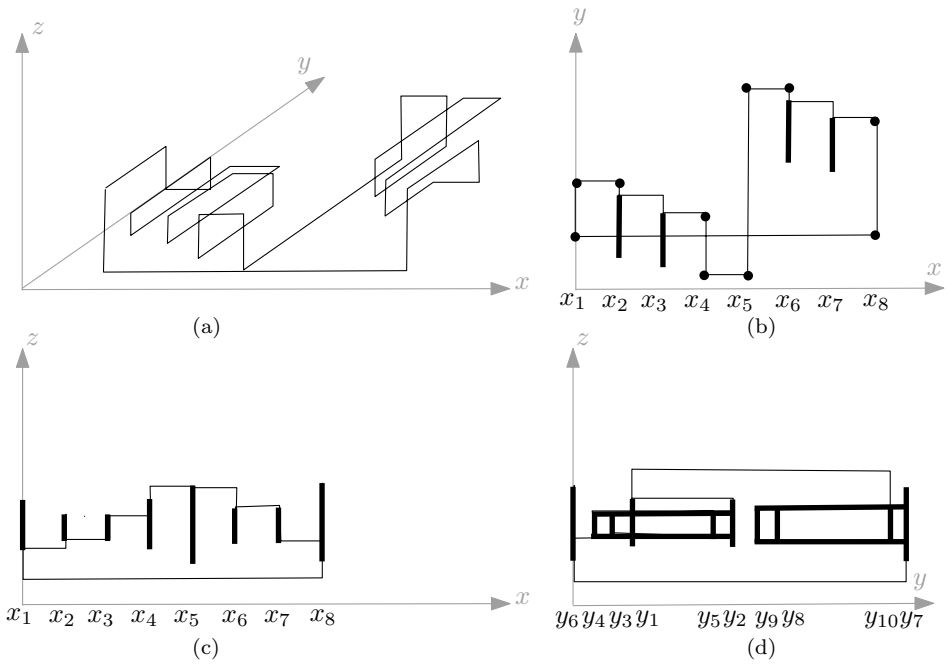


Figure 3.6: A tour for setting the gadget to true ($m = 1$) and its three projections.

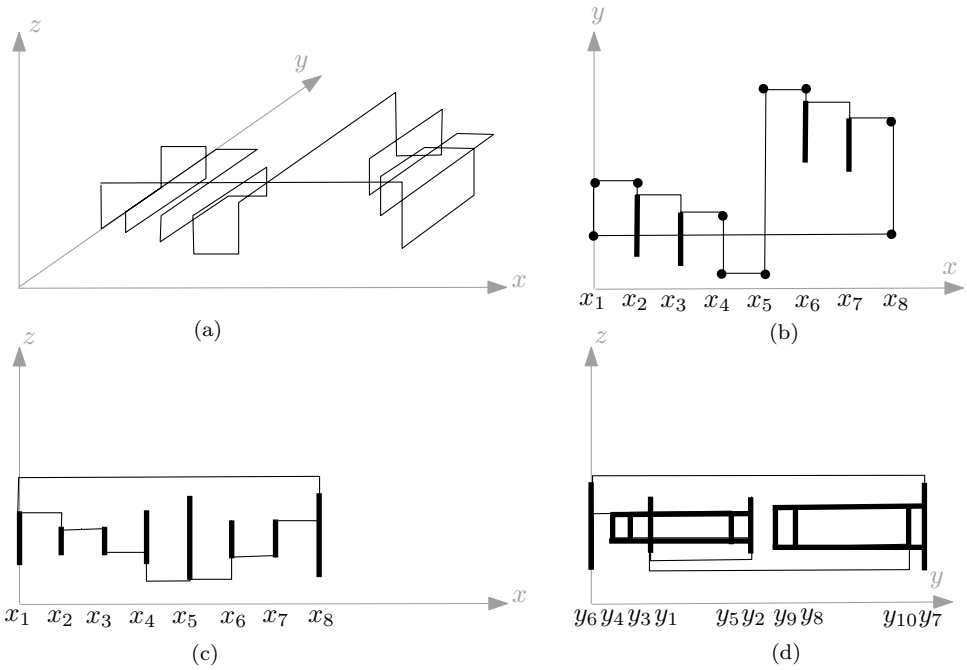


Figure 3.7: A tour for setting the gadget to false ($m = 1$) and its three projections.

4. The set B_j of 4400 points can be covered by 14 line-segments of the form $line(x, y, *)$ and 8 line-segments of the form $line(x, *, z)$ where

$$B_j \subseteq \bigcup_{s=1}^{22} cover(\text{line-segment}_s).$$

Table 3.1 shows line-segment$_s$ for $s = 1, 2, \ldots, 22$. Note that in Figure 3.5 these line-segments are labelled as $1, 2, \ldots, 22$.

Table 3.1: A building block B_j can be covered by 22 line-segments.

line-segment$_s$($s = 1, \ldots, 11$)	line-segment$_s$($s = 12, \ldots, 22$)
line-segment$_1$ = $line(x_1, y_1, *)$	line-segment$_{12}$ = $line(x_5, y_6, *)$
line-segment$_2$ = $line(x_1, y_2, *)$	line-segment$_{13}$ = $line(x_5, y_7, *)$
line-segment$_3$ = $line(x_2, y_2, *)$	line-segment$_{14}$ = $line(x_6, y_7, *)$
line-segment$_4$ = $line(x_2, *, z_1)$	line-segment$_{15}$ = $line(x_6, *, z_4)$
line-segment$_5$ = $line(x_2, y_3, *)$	line-segment$_{16}$ = $line(x_6, y_8, *)$
line-segment$_6$ = $line(x_2, *, z_2)$	line-segment$_{17}$ = $line(x_6, *, z_3)$
line-segment$_7$ = $line(x_3, *, z_2)$	line-segment$_{18}$ = $line(x_7, *, z_3)$
line-segment$_8$ = $line(x_3, y_4, *)$	line-segment$_{19}$ = $line(x_7, y_9, *)$
line-segment$_9$ = $line(x_3, *, z_1)$	line-segment$_{20}$ = $line(x_7, *, z_4)$
line-segment$_{10}$ = $line(x_4, y_5, *)$	line-segment$_{21}$ = $line(x_8, y_{10}, *)$
line-segment$_{11}$ = $line(x_4, y_6, *)$	line-segment$_{22}$ = $line(x_8, y_1, *)$

The building block of all gadgets for all variables will be placed so that:

1. The lengths of line-segment$_1$ and line-segment$_2$ are both longer than the lengths of line-segment$_3$, line-segment$_5$, line-segment$_8$ and line-segment$_{10}$.

2. The y-coordinate of line-segment$_{10}$ in B_j is smaller than the y-coordinate of line-segment$_2$.

3. The sequence of the line-segments from line-segment$_{12}$ to line-segment$_{22}$ follows the same pattern of the first 11 segments. However, we expand this part of the building block in the z-direction, equally on both sides, by 10% and we place them in increasing x and y values as shown in Figure 3.5(c). The length of every line-segment being expanded is essentially 1.2 times longer than the length of the previous 11 line-segments. This requirement will prevent any possible interference between lines of different forms.

4. Any two line-segments of the form $line(x, y, *)$ in the same axis-parallel plane in B_j do not coincide in such a plane with those line-segments of this or any other gadget.

5. Any two line-segments of the form $line(x, *, z)$ in the same axis-parallel plane in B_j do not coincide in such a plane with those line-segments of this or any other gadget.

We will show that there are exactly two equivalence classes of covering the building block in Figure 3.5 and both require at least 34 bends. To prove this claim, we require the following observations.

Observation 1. *All line-segments in the building block are in one of two forms, namely,* $line(x, y, *)$ *or* $line(x, *, z)$.

Observation 2. *In the building block, any two line-segments of the same form and in a common axis-parallel plane can be joined with two bends (and no fewer). That is:*

- $line(x_1, y_1, *)$ *and* $line(x_1, y_2, *)$ *with* $y_1 \neq y_2$ *or*
- $line(x_1, y_1, *)$ *and* $line(x_2, y_1, *)$ *with* $x_1 \neq x_2$ *or*
- $line(x_1, *, z_1)$ *and* $line(x_1, *, z_2)$ *with* $z_1 \neq z_2$ *or*
- $line(x_1, *, z_1)$ *and* $line(x_2, *, z_1)$ *with* $x_1 \neq x_2$ *or*

can be joined in a rectilinear path of two bends.

Observation 3. *In the building block, any two line-segments of mixed form on different axis-parallel planes, such that there is an axis-parallel plane that contains one line-segment and intersects the other line-segment in an interior point, can be joined with three bends (and no fewer). That is:*

- $line(x_1, y_1, *)$ *and* $line(x_2, *, z_2)$ *where* z_2 *is chosen such that* (x_1, y_1, z_2) *is an interior point in* $line(x_1, y_1, *)$, *can be joined with three bends.*
- $line(x_1, y_1, *)$ *and* $line(x_2, *, z_2)$ *where* y_1 *is chosen such that* (x_2, y_1, z_2) *is an interior point in* $line(x_2, *, z_2)$, *can be joined with three bends.*

- *line*$(x_1, y_1, *)$ *and line*$(x_2, *, z_2)$ *where the values* z_2 *and* y_1 *are chosen such that* (x_1, y_1, z_2) *is an interior point in line*$(x_1, y_1, *)$*, and* (x_2, y_1, z_2) *is an interior point in line*$(x_2, *, z_2)$*, can be joined with four bends.*

Observation 4. *In the building block, any two line-segments of mixed form on different axis-parallel planes, such that there is an axis-parallel plane that contains one line-segment and intersects the other line-segment in an exterior point, can be joined with two bends (and no fewer). That is, line*$(x_1, y_1, *)$ *and line*$(x_2, *, z_2)$ *where* z_2 *and* y_1 *are chosen, such that* (x_1, y_1, z_2) *is an exterior point in line*$(x_1, y_1, *)$*, and* (x_2, y_1, z_2) *is an exterior point in line*$(x_2, *, z_2)$*, can be joined with two bends.*

Observation 5. *In the building block, any two line-segments of mixed form on the same axis-parallel plane where a line of one line-segment intersects the other line-segment in an interior point, can be joined with three bends (and no fewer). That is,*

- *line*$(x_1, y_1, *)$ *and line*$(x_1, *, z_2)$ *where* z_2 *is chosen such that* (x_1, y_1, z_2) *is an interior point in line*$(x_1, y_1, *)$*, or*
- *line*$(x_1, y_1, *)$ *and line*$(x_1, *, z_2)$ *where* y_1 *is chosen such that* (x_1, y_1, z_2) *is an interior point in line*$(x_1, *, z_2)$*, or*
- *line*$(x_1, y_1, *)$ *and line*$(x_1, *, z_2)$ *where* (x_1, y_1, z_2) *is a point of line-segment intersection*

can be joined in a rectilinear path of three bends (and no fewer).

Observation 6. *In the building block, any two line-segments of mixed form on the same axis-parallel plane, where lines hosting these two segments intersects in an exterior point of the segments, can be joined with one bend. That is, line*$(x_1, y_1, *)$ *and line*$(x_1, *, z_2)$ *can be extended to meet at a point* (x_1, y_1, z_2) *that is exterior to line*$(x_1, y_1, *)$ *and line*$(x_1, *, z_2)$*.*

Lemma 10. *A building block as shown in Figure 3.5 can be covered by a rectilinear tour with* 34 *bends and no fewer in only two equivalent classes as Figure 3.6 and Figure 3.7.*

Proof. First, we prove 34 bends are necessary. We know from the previously described six observations that any pair of line-segments in the building block

can be joined with one bend, two bends, three bends or four bends. One-bend is the best we can do and four-bends is the worst case of connecting any pairs of line-segments. We will show that an optimal tour uses only one-bend type of connections and two-bends type of connections, but does not use the three-bends and four-bends type of connections. If we could use one bend for every pair of line-segments, we would have an optimal tour with 22 bends, because there are 22 line-segments in the building block. However, the best we can do is to use one-bend type of connections (Observation 6) at the 10 locations in the building block (see Figure 3.8).

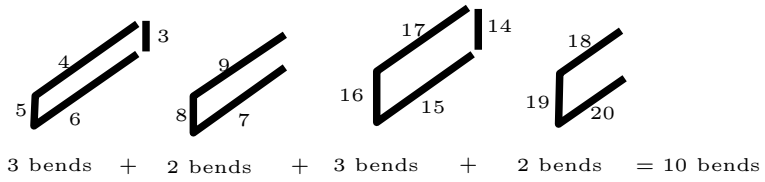


Figure 3.8: Ten bends that use a connection-type described by Observation 6.

Therefore, any optimal tour can have at most 10 bends of a connection-type described by Observation 6. The structure of the building block is then reduced to 12 components. The next best thing that we can do is to join any pair of these components with two bends. This would result in at least another 24 bends. Hence, an optimal tour covering the building block in Figure 3.5 requires 34 bends or more. Note that if a tour uses the three-bends and four-bends type of connections, we would definitely have more than 34 bends in the tour.

The fact that 34 bends are sufficient is shown by any of the tours in Figure 3.6 or Figure 3.7.

We now show these are the only two possible equivalence classes. According to Observation 2 and Observation 4, we have two options when joining any pair of line-segments with two bends. That is, we have an option to join them at the top or at the bottom. These two options give rise to exactly the two equivalent classes, as per Figure 3.6 and Figure 3.7. Disobeying this pattern would require a three-bends connection or a four-bends connection and, consequently, will result in more than 34 bends in the tour. □

3.5.2 A Complete Gadget

The structure of the 22 line-segments in a building block is repeated. Our aim is to generate a complete gadget that holds 22 covering line-segments per clause or $22m$ covering line-segments overall (see Figure 3.9).

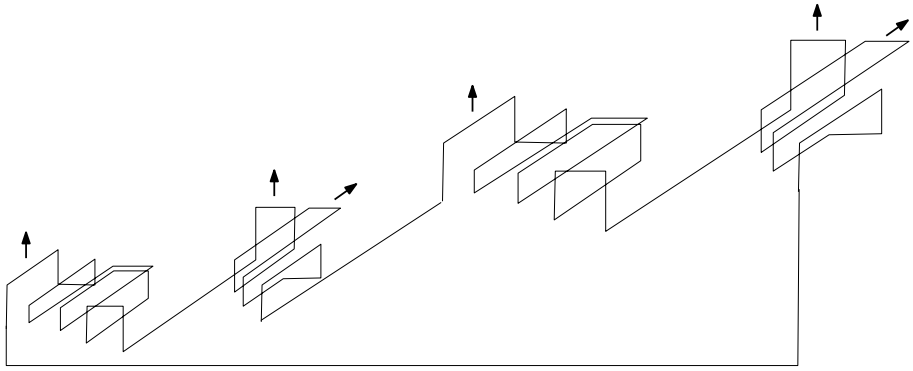


Figure 3.9: A complete gadget when a variable is set to true ($m = 2$).

A complete gadget has $S_i = \bigcup_{j=1}^{m} B_j$ for $j = 1, 2, \ldots, m$. The set S_i can be covered by axis-parallel line-segments that have a cyclic structure as follows:

$$\begin{aligned} S_i \;=\; & cover(\{line(x_1, y_1, *), line(x_1, y_2, *), line(x_2, y_2, *), line(x_2, *, z_1), \\ & line(x_2, y_{3,*}), line(x_2, *, z_2), line(x_3, *, z_2), line(x_3, y_{4,*}), \\ & line(x_3, *, z_1), line(x_4, y_5, *), line(x_4, y_6, *), line(x_5, y_6, *), \ldots, \\ & line(x_{8m}, y_{10m}, *), line(x_{8m}, y_1, *), line(x_1, y_1, *)\}). \end{aligned}$$

The last line-segment of B_j and the first line-segment of $B_{(j \bmod m)+1}$ are always on the same plane $\Pi(*, y, *)$. For every 11 line-segments in the gadget, we still expand the shape of the gadget in the z-direction, equally on both sides, by 10%. Therefore, a building block of the first clause is smaller than the block of the second clause, a building block of the second clause is smaller than the block of third clause, and so on.

Finally, we put n' gadgets that represents n' variables on different x, y, and z-coordinates, such that no two covering line-segments of different variables are on the same axis-parallel plane. We call two parallel line-segments of the form $line(x, *, z)$ and the line-segment $line(x, y, *)$ that connects them, a "C" shape. For example, line-segment$_4$, line-segment$_5$, and line-segment$_6$ in Figure 3.5 form a "C" shape. We enforce the following relationship between the "C" shape and

the line-segments of different building blocks of the form $line(x, y, *)$ in front of it: the z-coordinates of every line-segment in the "C" shape must be between the smallest and largest z-coordinates of the line-segment $line(x, y, *)$ that has larger y-coordinates than the "C" shape. To maintain this relationship across all gadgets, we make sure that the n' gadgets are different in size. That is, in a clause the gadget of a variable that participates as the first literal is slightly bigger than the one that participates as the second literal, and the gadget of a variable that participates as the second literal is slightly bigger than the one that participates as the third literal. Again, this requirement is to prevent any possible interference between lines of different gadgets.

The structures of a tour T_i for setting the gadget to "true" (see Figure 3.6) and a tour F_i for setting the gadget to "false" (see Figure 3.7) are described next.

$$
\begin{aligned}
T_i \;=\; & \{line(x_1, y_1, *) \sqcap line(x_1, y_2, *) \sqcup line(x_2, y_2, *) \oplus line(x_2, *, z_1) \oplus \\
& line(x_2, y_{3,*}) \oplus line(x_2, *, z_2) \sqcap line(x_3, *, z_2) \oplus line(x_3, y_{4,*}) \oplus \\
& line(x_3, *, z_1) \sqcap line(x_4, y_5, *) \sqcup line(x_4, y_6, *) \sqcap line(x_5, y_6, *), \ldots, \\
& line(x_{8m}, y_{10m}, *) \sqcap line(x_{8m}, y_1, *) \sqcup line(x_1, y_1, *)\} \text{ and} \\
F_i \;=\; & line(x_1, y_1, *) \sqcup line(x_1, y_2, *) \sqcap line(x_2, y_2, *) \oplus line(x_2, *, z_2) \oplus \\
& line(x_2, y_{3,*}) \oplus line(x_2, *, z_1) \sqcap line(x_3, *, z_1) \oplus line(x_3, y_{4,*}) \oplus \\
& line(x_3, *, z_2) \sqcup line(x_4, y_5, *) \sqcap line(x_4, y_6, *) \sqcup line(x_5, y_6, *), \ldots, \\
& line(x_{8m}, y_{10m}, *) \sqcup line(x_{8m}, y_1, *) \sqcap line(x_1, y_1, *)\}.
\end{aligned}
$$

If we assign "true" to the variable u_i, then we cover the set S_i by the tour of T_i. On the other hand, if we assign "false" to the variable u_i, then we cover the set S_i by the tour of F_i. In each of these two configurations, the rectilinear tour has $34m$ bends, which is the minimum number of bends any tour can have.

It is important to note that the arrows in Figure 3.9 indicate the paths in the true-gadgets that have the flexibility to be extended to cover points of clauses (discussed later in the next section). In the false-gadgets, all arrows will be pointing in the opposite direction.

3.5.3 Gadget Assignment

A complete gadget is built in such a way that it can be covered by a tour in the two ways that give the minimum number of bends covering the points in S_i. One of the ways represents the variable being assigned the value "true", and the other way represents the variable being assigned the value "false". The minimum number of bends for each complete gadget is set to $34m$ bends. That is, each gadget consists of $34m$ line-segments (that is, $22m$ covering line-segments plus the $12m$ joining line-segments that do not cover the points in $\bigcup_{i=1}^{n'} S_i$).

Lemma 11. *A complete gadget can be covered by a rectilinear tour with* $34m$ *bends and no fewer in the "true" configuration or the "false" configuration, where* m *is the number of clauses.*

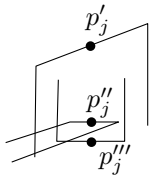
Proof. First we establish necessity. A complete gadget has $S_i = \bigcup_{j=1}^{m} B_j$ for $j = 1, 2, \ldots, m$. Since each building block is replicated in the same way, there are $10m$ locations where we can use one-bend type of connections. The structure of a complete gadget is then reduced to $12m$ components. If we could join any pair of these components with two bends, this would result in at least another $24m$ bends. Hence, an optimal tour covering a complete gadget requires $34m$ bends and no fewer.

To establish sufficiency consider Figure 3.9. This shows how the path pattern of Figure 3.6 extends for $m > 1$.

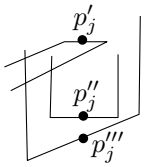
That these are the only two equivalence classes, is established as follows. According to Observation 2 and Observation 4, we have two options when joining two parallel line-segments A and B in the same axis-parallel plane with two bends. That is, we can join them as $A \sqcap B$ or $A \sqcup B$. These two options corresponds to the "true" configuration and the "false" configuration. A tour that deviates from these two configuration would require a three-bends or a four-bends connection somewhere, and consequently would result in more than $34m$ bends in the tour. □

For every clause, if it is a YES-instance, there are three patterns, true-false-false, false-true-false and false-false-true. This is equivalent to choosing exactly one of the three variables that participate in a clause, setting it to true and placing the other two as false. Formally, we represent each clause E_j by three points $p'_j = (a_j, b_j, c'_j)$, $p''_j = (a_j, b_j, c''_j)$ and $p'''_j = (a_j, b_j, c'''_j)$ in a line $(a_j, b_j, *)$

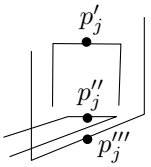
where $c'_j > c''_j > c'''_j$ and $j = 1, 2, \ldots, m$. The coordinates of all these points will be pairwise distinct. The gadget passing through the point p'_j is the one set to true. The gadget passing through the points p''_j and p'''_j will be the gadget set to false. Essentially, the points p'_j, p''_j and p'''_j are covered by the extendible paths (see Figure 3.9) of the three gadgets participating in a clause.



(a) u_1 = "true", u_2 = "false", u_3 = "false"



(b) u_1 = "false", u_2 = "true", u_3 = "false"



(c) u_1 = "false", u_2 = "false", u_3 = "true"

Figure 3.10: Three cases where the clause $E_j = u_1 \vee u_2 \vee u_3$ is satisfiable and p'_j, p''_j and p'''_j is covered.

The line-segments of these three gadgets are positioned with respect to the the line $(a_j, b_j, *)$ as follows.

1. The line $(a_j, b_j, *)$ is placed on the plane $\Pi(x_{8j-7}, *, *)$ between two line-segments $line(x_{8j-7}, y_{10j-9}, *)$ and $line(x_{8j-7}, y_{10j-8}, *)$ of a gadget that participates as the first literal. Here, we set $x_{8j-7} = a_j$, $y_{10j-9} < b_j < y_{10j-8}$, $c'_j \geq z_{\sqcap}$, and $c'''_j \leq z_{\sqcup}$.

2. The line $(a_j, b_j, *)$ is placed between the top plane $\Pi(*, *, z_{4j-1})$ and the bottom plane $\Pi(*, *, z_{4j})$. The plane $\Pi(*, *, z_{4j-1})$ has two line-segments, $line(x_{8j-2}, *, z_{4j-1})$ and $line(x_{8j-1}, *, z_{4j-1})$. The plane $\Pi(*, *, z_{4j})$ has two line-segments, $line(x_{8j-2}, *, z_{4j})$ and $line(x_{8j-1}, *, z_{4j})$. Here, we set $z_{4j-1} = c'_j$, $z_{4j} = c''_j$, , $x_{8j-2} < a_j < x_{8j-1}$ and $b_j \geq y_{\sqcap}$.

3. The line $(a_j, b_j, *)$ is placed on the plane $\Pi(*, y_{10j-3}, *)$ between two line-segments $line(x_{8j-3}, y_{10j-3}, *)$ and $line(x_{8j-2}, y_{10j-3}, *)$ of a gadget that participates as the third literal. Here, we set $y_{10j-3} = b_j$, $x_{8j-3} < a_j < x_{8j-2}$, $c'_j \geq z_{\sqcap}$, and $c''_j \leq z_{\sqcup}$.

If the clause E_j is satisfiable, then one path from the true-gadget must cover p'_j and two paths from the false-gadgets must cover p''_j and p'''_j (see Figure 3.10).

Lemma 12. *To maintain the minimum number of bends, a complete gadget must be covered only in the "true" configuration or the "false" configuration, but cannot be covered by both configurations.*

Proof. A tour that switches the setting of a gadget from "true" to "false" would require the three-bends or four-bends type of connections. This contradicts the optimal tour described in Lemma 10 and Lemma 11 in which the tour uses only the one-bend and two-bends type of connections. □

Lemma 13. *A tour, once it enters a complete gadget, maintains its equivalent classes (therefore, all have the same number of bends) as per Definition 19, and it only changes in extensions that cover points of clauses.*

Proof. Since all the gadgets are different in size and we place n' gadgets on different x, y, and z-coordinates, such that no two covering line-segments of the different gadgets are on the same axis-parallel plane, the same argument applies as per Lemma 11. If the tour leaves the gadget (not to cover points of clauses), then at least three additional bends are required and we would need more than $34m$ bends to cover the points in S_i. Note that a clause was mapped to three co-linear points. If a path that covers a gadget passes through more than one of these three points, it would be incurring on a line $(a_j, b_j, *)$ that is additional to the axis-parallel line-segments it is already covering. By Observation 1 to Observation 6, this implies at least one more bend. □

3.5.4 A Complete Tour

We have discussed a tour that covers S_i and there are n' of these tours, each having $34m$ bends. We now explain how to obtain a single tour that covers $\bigcup_{i=1}^{n'} S_i$. Two tours of S_i can be merged into one tour with the minimum number of additional bends. Merging any two tours of S_i requires a removal of no

more than one line-segment from each tour (four bends fewer) but we have to add two line-segments for connecting any two line-segments from the different axis-parallel planes (six bends more). Thus, in total, we require two additional bends for merging any two tours of S_i. We illustrate this merging in Figure 3.11, where we place two tours of S_i distant from each other for easy viewing. Note that the line-segments removed above cannot be the covering line-segments of S_i and cannot be the line-segments that are set to cover points of clauses. Therefore, we set the following two options. We either remove a line-segment joining $line(x_{8m}, y_{10m}, *)$ with $line(x_{8m}, y_1, *)$ in the tour, or remove a line-segment joining $line(x_{8m}, y_1, *)$ with $line(x_1, y_1, *)$. We repeat the merging for $n' - 1$ times to get a single tour covering $\bigcup_{i=1}^{n'} S_i$. The number of bends in a complete tour is, therefore, $34mn' + 2(n' - 1) = 34mn' + 2n' - 2$. The number of links in a complete tour is also $34mn' + 2n' - 2$.

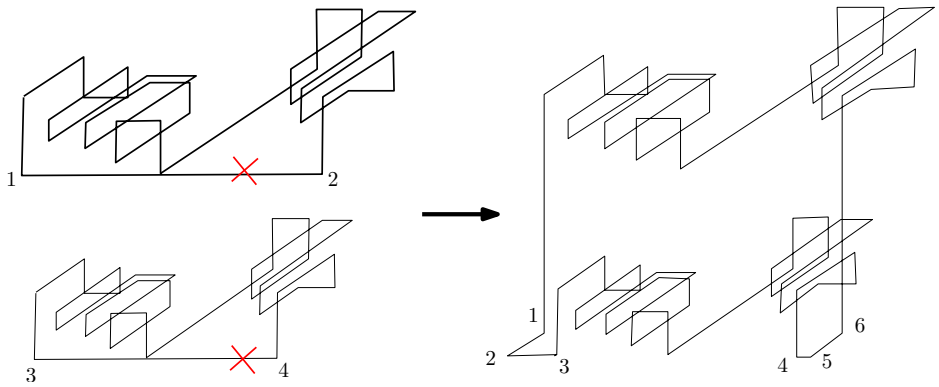


Figure 3.11: Merging any two tours of S_i requires two additional bends.

Lemma 14. *There exists a satisfying assignment for $E_1 \wedge \ldots \wedge E_m$ if and only if a set of $|S|$ points in 3D can be covered optimally by a rectilinear tour that has $34mn' + 2n' - 2$ links.*

Proof. If a clause is satisfiable, then the three points p'_j, p''_j and p'''_j that represent the clause are covered by three different line-segments. The point p'_j is covered by a line-segment arising from a true-gadget while the points p''_j and p'''_j are covered by two different line-segments arising from two different false-gadgets. Thus, there are no additional bends incurred to cover points in P. Then, the number of bends in the tour is the minimum number of bends to cover points in $\bigcup_{i=1}^{n'} S_i$, which is $34mn' + 2n' - 2$ bends or $34mn' + 2n' - 2$ links. On the other hand,

if a clause is not satisfiable (that is, a clause has one false literal or all equal literals), then at least one of the three points that represent the clause would not be covered in the tour, and to include these points in the tour we would need more than $34mn' + 2n' - 2$ links. Conversely, if a tour with $34mn' + 2n' - 2$ links covers $\bigcup_{i=1}^{n'} S_i$ and it also covers P that represents the points of the clauses $E_1, E_2 \ldots, E_m$, then $E_1 \wedge \ldots \wedge E_m$ is clearly satisfiable. □

As a consequence of Lemma 14, we obtain the following result.

Theorem 3. *The decision version of the* Rectilinear Minimum-Links Traveling Salesman Problem *in three dimensions is NP-complete.*

Similarly, we prove the NP-completeness of the Rectilinear Minimum-Links Spanning Path Problem by a reduction from the One-In-Three 3-SAT. We follow the same line of argument as above, except that we do not connect the first line-segment to the last line-segment to complete the tour.

Theorem 4. *The decision version of the* Rectilinear Minimum-Links Spanning Path Problem *in three dimensions is NP-complete.*

Since Theorem 4 is true in three dimensions, the problem is also NP-complete for $d > 3$ dimensions.

3.6 Rectilinear Spanning Path Problem

In this section, we prove that the problem associated with Theorem 4 belongs to the class FPT under the assumption that no two line-segments share the same line. By revisiting the NP-completeness proof above, we find that this restricted problem remains NP-complete. This is important, because it means that even with this restriction, the problem has no polynomial-time exact algorithm, unless P=NP. Now, we specify the problem in the format of parameterized complexity as follows. Given a set S of n points in $\mathbb{R}^d$, and a positive integer k, we are asked if there is a piecewise linear path through these n points in S having at most k line-segments (links) where every line-segment in the path is axis-parallel.

We follow the standard convention of the problem, that is, to restrict the turns in the path to 90° turns. We decide a problem instance for the rectilinear spanning path with at most k links in two phases. First, we compute the

candidate sets of k lines that cover all the n points in S. We assume that one line-segment in the spanning path covers all the points on the same line[2], thus, once we have the candidate set of k covering lines, we also have the candidate set of k line-segments. Second, we test if the candidate set can constitute a spanning path based on these k line-segments in the set.

We can compute the candidate sets of k lines that cover all the n points in S using the algorithm that solves the Rectilinear Line Cover in d dimensions. Recall that this can be done in $O(d^k n)$, using the bounded-search-tree technique. However, the bounded-search-tree by itself is not to establish the solution, but at the leaves, we check if the candidate lines from the tree can be completed into a spanning path with no more than k links. Each of the candidate set of lines in a leaf of the search tree in Figure 3.1 results in a candidate set of line-segments, because we can simply connect the extreme points in $cover(l)$ for each l in the k lines of the candidate set to get the candidate line-segments. Now, we can explore exhaustively if they conform to the required solution of spanning path. In the worst case, the k line-segments can be organized into $k!$ orders in a possible spanning path. For any given order, there are at most 2^k possible ways of connecting these k line-segments as a path because we can connect the current segment with the next segment in the sequence by two possible end-points. This is necessary, because the two segments could have different orientations, and we could save an extra line-segment if we connect the two segments at the correct end-point. This means a total of $(k!)(2^k)$ tests. In the simplest case, the extension of the two segments is enough to make a turn; therefore, no extra line-segment is required. In the worst case, it requires at most $d + 1$ additional line-segments in order to connect two consecutive line-segments. In the construction of the rectilinear spanning path, these additional line-segments can be added in $O(kd)$ time. Note that if the total number of line-segments in the final rectilinear spanning path exceeds k, we immediately answer NO.

Theorem 5. *The* Rectilinear k-Links Spanning Path Problem *in d dimensions is fixed-parameter tractable (FPT) under the assumption that no two line-segments share the same line.*

[2]This is a constraint of the problem. In the unrestricted case, several line-segments could cover points on the same line. We leave this case as an open problem.

Proof. The search tree for the first phase has at most $O(d^k)$ leaves and $O(d^{k-1})$ internal nodes. The work at each internal node can be performed in time linear in n. The dominant work is the computation at the leaves of the tree. Here, we perform the tests whether we have the spanning path that covers all the points in S and has at most k links. This results in time bounded by

$$\begin{aligned} O(d^k(k!)(2^k)(kd+n)) &= O((2d)^k\sqrt{2\pi k}(k/e)^k(kd+n)) \\ &= O((0.74dk)^k(kd+n)\sqrt{k}). \end{aligned}$$

The time complexity is exponential in the parameter but polynomial in the size of the input. Thus, the problem is FPT. □

3.7 Spanning Path Restricted to a Finite Number of Orientations

We now consider the Minimum-Links Spanning Path Problem where the set of k links is restricted to the lines having a finite number, ϕ, of orientations. Note that the statements that follow hold without specifying the dimensions, but in this case, the orientations are vectors based at the origin. That is, in this case the k line-segments must have their orientations taken from a given finite set $R = \{r_1, r_2, \ldots, r_\phi\}$ where $|R| = \phi$. This also implies that the possible angles between the turns at the linking points belong to a finite set. We decide a problem in two phases. First, we compute the candidate sets of k lines that cover all the n points in S. We assume that one line-segment in the spanning path covers all the points on the same line. Second, we test if the candidate set can constitute a spanning path based on these k lines that have a finite number of orientations defined in R.

The first phase is achieved using the algorithm in Section 3.3.1. Recall that this algorithm outputs a set of leaves, and for each leaf a set of k lines covering S that is restricted to the lines having orientations in R. The algorithm achieves this in $O(\phi^k n)$ time. The second phase is carried out in a way similar to that of the previous section. Each leaf is evaluated to decide if it can be made into a path. That is, an exhaustive search with the total of $(k!)(2^k)$ possible constructions of a path from the cover. In each test, $O(k\phi)$ additional lines may be added in the construction of a path. The total time complexity is $O((0.74\phi k)^k(k\phi+n)\sqrt{k})$.

Thus, we obtain an FPT-algorithm, because its time complexity is polynomial in the size of the input, although it is exponential in the parameter. This result is stated in Theorem 6.

Theorem 6. *The* Rectilinear k-Links Spanning Path Problem*, when restricted to the lines having a finite number $m \geq 2$ of orientations and no two line-segments share the same line, is FPT.*

Chapter 4

Rectilinear Hyperplane Cover Problem

We study the problem of covering a set of n points in d dimensions with k axis-parallel hyperplanes of $d-1$ dimensions. Since the hardness of this problem is unknown, we provide the NP-completeness proof of the problem by a reduction from 3-SAT. We then deliver an FPT-algorithm for this problem based on the bounded-search-tree technique. Moreover, if we extend our problem to the case where the hyperplanes of $d-1$ dimensions are restricted to a finite number of orientations, this variant of the problem remains FPT.

4.1 Introduction

The Rectilinear Hyperplane Cover problem is to cover a set of n points in $\mathbb{R}^d$ with the minimum number of axis-parallel hyperplanes of dimension $d-1$. For example, in 2D the problem is to cover points with lines, but in 3D the problem is to cover points with planes. The complexity of this problem is polynomial for 2D [HM91], but its complexity is unknown even for 3D [Hur08]. For related problems, the following result is known. The problem of covering points with arbitrary lines in 2D is NP-hard [MT82]. With this result, covering points with arbitrary hyperplanes is also NP-hard [MT82, LM05]. Langerman and Morin show that for any d dimensions, the problem of covering points with arbitrary hyperplanes also belongs to the class FPT. When objects used to cover points in d dimensions are axis-parallel, we found the following result. The problem of covering points with axis-parallel lines in 2D is polynomially solvable [HM91,

SW01]. However, this problem in 3D is NP-complete [HM91]. Our problem of covering points with axis-parallel hyperplanes is unknown, whether it is NP-complete or if it is polynomially solvable. Therefore, we will establish the NP-completeness proof to this problem. Based on parameterized complexity theory, we then provide an improved FPT-algorithm that uses the bounded-search-tree technique. Finally, we prove that when the hyperplanes of $d-1$ dimensions are restricted to a finite number of orientations, this variant of the problem is also FPT.

4.2 NP-Completeness Result

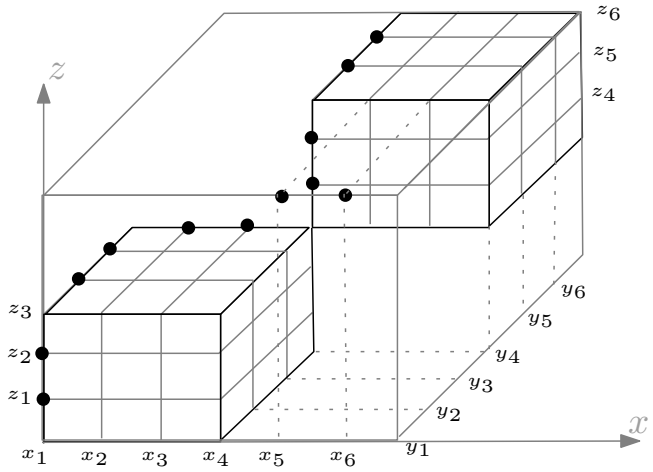
We prove that the problem of covering a set of $|S|$ points in 3D with k axis-parallel planes of 2D is NP-complete. The method of our proof is inspired by the NP-completeness proof of the Rectilinear Line Cover problem in 3D [HM91]. First, our problem is trivially in NP. The verification algorithm checks whether each point in the given set S is covered by at least one of the k planes and every plane is axis-parallel. This can be performed in polynomial time.

To prove NP-hardness, we present a reduction from 3-SAT (3-satisfiability), a classical NP-complete problem. Our reduction is by gadget construction. Assume that we are given an instance of 3-SAT with a set of m clauses $E_j = w'_j \vee w''_j \vee w'''_j$ for $j = 1, 2, \ldots, m$, where $\{w'_j, w''_j, w'''_j\} \subset \{u_1, \bar{u}_1, \ldots, u_{n'}, \bar{u}_{n'}\}$. The instance of 3-SAT has n' variables and we will represent each variable u_i for $i = 1, 2, \ldots, n'$ by a set S_i of $12m$ points. We represent each clause E_j with a point p_j. The corresponding instance of the Rectilinear Hyperplane Cover will have a set $S = \{p_1, p_2, \ldots, p_m\} \cup \bigcup_{i=1}^{n'} S_i$, where $|S| = 12mn' + m = n$.

The set S_i represents u_i by a gadget that will be covered optimally $\pi = 3m$ different axis-parallel planes. There are n' gadgets in total. Since the gadgets ensure that all occurrences of a variable across the formula have a consistent setting, we call these gadgets, "true-false" gadgets. First, we construct the true-false gadgets for each variable u_i (see Figure 4.1).

Each gadget is built from $2m$ axis-parallel cubes as a reference. The cubes are aligned one after the other with the bottom-left-front corner of the next cube

coinciding with the top-right-back corner of the previous one. In each of these $2m$ cubes there are six points (except on the last one, where there are four). Two points are in the edge that meets the front face and the left face. Two points are in the edge that meets the top face and the left face. And two points are in the edge that meets the top face and the back face. For the last cube. four are like the first four of all other cubes; however, the last two are in the edge that meets the top face of this cube and the front face of the first cube.



$$
\begin{aligned}
q_1 &= (x_1, y_1, z_1)\\
q_2 &= (x_1, y_1, z_2)\\
q_3 &= (x_1, y_2, z_3)\\
q_4 &= (x_1, y_3, z_3)\\
q_5 &= (x_2, y_4, z_3)\\
q_6 &= (x_3, y_4, z_3)\\
q_7 &= (x_4, y_4, z_4)\\
q_8 &= (x_4, y_4, z_5)\\
q_9 &= (x_4, y_5, z_6)\\
q_{10} &= (x_4, y_6, z_6)\\
q_{11} &= (x_5, y_1, z_6)\\
q_{12} &= (x_6, y_1, z_6)
\end{aligned}
$$

Figure 4.1: The structure of S_i that represents each variable u_i is guided by $2m$ cubes that are the building blocks of the gadget (illustration with $m = 1, \pi = 3$).

Therefore, for each gadget, the set S_i has a cyclic structure. The x-coordinate has a fixed value x_1 for the first four points, then a different value x_2 in the 5th point and another different value x_3 in the 6th point. Then, this pattern of four equal, and two distinct is repeated, with x_4 in the next four points, a new value x_5 in the 11th point and x_6 in the 12th point. The pattern of four equal values and two different is also repeated in the y-coordinate and the z-coordinate, but in the z-coordinate, the four equal values start from the 3rd to the 6th points, and for the y-coordinate from the 5th to the 8th points (in all coordinates, the pattern of four equal values, one different and one different wraps around modulo 12).

Thus, S_i appears as follows:

$$\begin{aligned} S_i &= \{(x_1, y_1, z_1), (x_1, y_1, z_2), (x_1, y_2, z_3), (x_1, y_3, z_3), \\ &\quad (x_2, y_4, z_3), (x_3, y_4, z_3), (x_4, y_4, z_4), \ldots, \\ &\quad (x_{2\pi-2}, y_{2\pi-1}, z_{2\pi}), (x_{2\pi-2}, y_{2\pi}, z_{2\pi}), \\ &\quad (x_{2\pi-1}, y_1, z_{i2\pi}), (x_{i2\pi}, y_1, z_{2\pi})\}. \end{aligned}$$

We denote the plane perpendicular to the x axis, that intersects x at x_0 and is parallel to the plane of the y and z axes by $\Pi(x_0, *, *)$. Similarly, we represent the plane perpendicular to the y axis, that intersects y at y_0 and is parallel to the plane of the x and z axes by $\Pi(*, y_0, *)$. The plane perpendicular to the z axis, that intersects z at z_0 and is parallel to the plane of the x and y axes is called $\Pi(*, *, z_0)$. Now, to cover the set S_i it requires π distinct axis-parallel planes. The gadget is built in such a way that there are exactly two sets of π planes that cover S_i (see Figure 4.2). These are

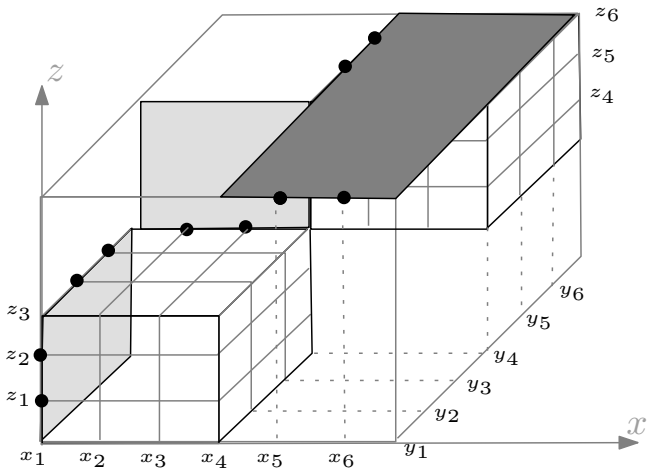
$$T_i = \{\Pi(x_1, *, *), \Pi(*, y_4, *), \Pi(*, *, z_6), \ldots, \Pi(*, y_{2\pi-2}, *), \Pi(*, *, z_{2\pi})\}$$

and

$$F_i = \{\Pi(*, *, z_3), \Pi(x_4, *, *), \Pi(*, y_7, *), \ldots, \Pi(x_{2\pi-2}, *, *), \Pi(*, y_1, *)\}.$$

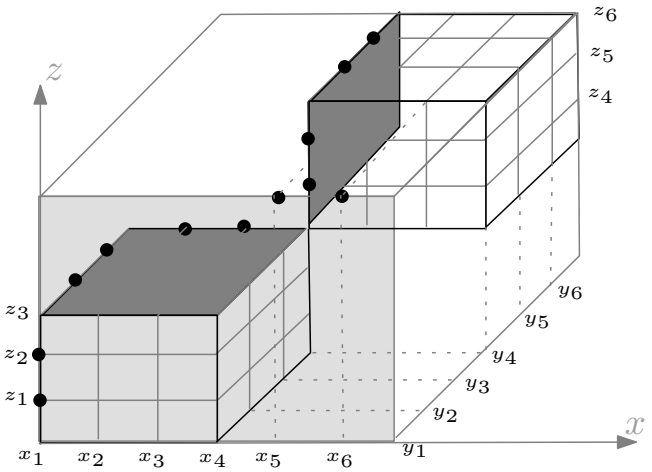
In each of these two configurations, each plane covers four points, which is the maximum any plane can cover. The two sets, T_i, F_i, each consists of π axis-parallel planes and each covers S_i, for $i = 1, 2, \ldots, n'$. If we assign "true" to the variable u_i, then we cover the set S_i by the planes of T_i. On the other hand, if we assign "false" to the variable u_i, then we cover the set S_i by the planes of F_i.

We represent each clause E_j by a single point $p_j = (a_j, b_j, c_j)$ for $j = 1, 2, \ldots, m$. The coordinates of all the points is S_i are all pairwise distinct from those of $S_{i'}$, for $i \neq i'$. The only constraints on the coordinates of the points in the set S_i are relative to the clauses where the variable represented in the gadget participates. The idea to set up the gadget to have planes that can stretch to the points of the clauses E_j where it participates. To enable at least one of the planes in the true-false gadgets to covers the point $p_j = (a_j, b_j, c_j)$ representing clause E_j where it participates, we proceed as follows. If $u_i = w'_j$ (it participates as the first literal and does so positively), then in S_i we set $x_{6j-5} = a_j$.



$$
\begin{aligned}
q_1 &= (x_1, y_1, z_1)\\
q_2 &= (x_1, y_1, z_2)\\
q_3 &= (x_1, y_2, z_3)\\
q_4 &= (x_1, y_3, z_3)\\
q_5 &= (x_2, y_4, z_3)\\
q_6 &= (x_3, y_4, z_3)\\
q_7 &= (x_4, y_4, z_4)\\
q_8 &= (x_4, y_4, z_5)\\
q_9 &= (x_4, y_5, z_6)\\
q_{10} &= (x_4, y_6, z_6)\\
q_{11} &= (x_5, y_1, z_6)\\
q_{12} &= (x_6, y_1, z_6)
\end{aligned}
$$

(a) Cover for "true".



$$
\begin{aligned}
q_3 &= (x_1, y_2, z_3)\\
q_4 &= (x_1, y_3, z_3)\\
q_5 &= (x_2, y_4, z_3)\\
q_6 &= (x_3, y_4, z_3)\\
q_7 &= (x_4, y_4, z_4)\\
q_8 &= (x_4, y_4, z_5)\\
q_9 &= (x_4, y_5, z_6)\\
q_{10} &= (x_4, y_6, z_6)\\
q_{11} &= (x_5, y_1, z_6)\\
q_{12} &= (x_6, y_1, z_6)\\
q_1 &= (x_1, y_1, z_1)\\
q_2 &= (x_1, y_1, z_2)
\end{aligned}
$$

(b) Cover for "false".

Figure 4.2: The optimal covers for the illustration of $m = 1$ from Figure 4.1.

This means a setting to "true" will make the $2j-1$ cube have its left face as a plane with value a_j and cover p_j. But if $\bar{u}_i = w'_j$ (it is the first literal in the clause, but participates negated), then in S_i we set $x_{6j-2} = a_j$. Now the plane is the right face of cube $2j-1$ (which is the left face of cube $2j$). If $u_i = w''_j$ (participates in the second literal), then in S_i we set $y_{6j-2} = b_j$. If $\bar{u}_i = w''_j$, then in S_i we set $y_{6j-5} = b_j$. If $u_i = w'''_j$ (the third literal), then in S_i we set $z_{6j} = c_j$. If $\bar{u}_i = w'''_j$, then in S_i we set $z_{6j-3} = c_j$ (see Figure 4.3).

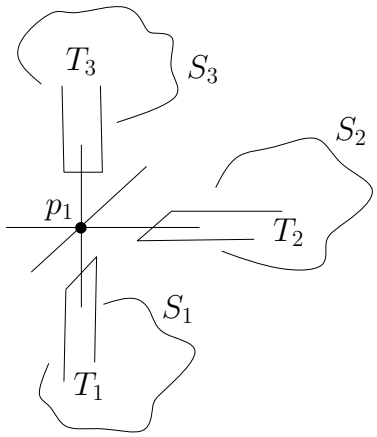


Figure 4.3: Schema where the clause $E_1 = u_1 \vee u_2 \vee u_3$ is satisfiable and p_1 is covered.

Lemma 15. *There exists a satisfying assignment for $E_1 \wedge \ldots \wedge E_m$ if and only if a set of $|S|$ points in 3D can be covered by $k = \pi n'$ axis-parallel 2D-planes.*

Proof. Suppose $E_1 \wedge \ldots \wedge E_m$ is satisfiable. By assigning planes to n' gadgets as per a satisfying assignment, we can cover $\bigcup_{i=1}^{n'} S_i$ with $3mn'$ axis-parallel planes. Moreover, these planes can cover the points in $\{p_1, p_2, \ldots, p_m\}$. That is, there are no additional planes needed for covering p_j. Therefore, if $E_1 \wedge \ldots \wedge E_m$ is satisfiable, then the set $\{p_1, p_2, \ldots, p_m\} \cup \bigcup_{i=1}^{n'} S_i$ can be covered by $\pi n'$ axis-parallel planes.

Conversely, if the $\pi n'$ axis-parallel planes cover $\bigcup_{i=1}^{n'} S_i$ and these planes also cover $p_1, p_2, \ldots, p_m$, then, first all of them need to cover the gadgets as consistent assignments for each gadget. But then, as the points p_j represents the clause $E_1, E_2 \ldots, E_m$ respectively, then $E_1 \wedge \ldots \wedge E_m$ is clearly satisfiable. □

As a consequence of Lemma 15, we obtain the following result.

Theorem 7. *The decision version of covering a set of points in 3D with axis-parallel planes of 2D is NP-complete.*

Since Theorem 7 is true in three dimensions, the problem is also NP-complete for $d > 3$ dimensions.

4.3 Rectilinear Hyperplane Cover

In this section, we will use the bounded-search-tree technique to show that the Rectilinear Hyperplane Cover belongs to the class FPT. In this problem, given n points in d dimensions and a positive integer k, we are asked whether it is possible to cover these n points with at most k axis-parallel hyperplanes of $d-1$ dimensions. We will show that this problem is FPT when parameterized by the number of dimensions, d, and the size of the solution, k. Using the bounded-search-tree technique, the proof goes as follows. First, consider the problem in 3D. For each point p in 3-dimensional space, there are three possibilities of being covered, namely, 1) a plane that is orthogonal to height, 2) a plane that is orthogonal to depth and, 3) a plane that is orthogonal to width. In addition to the point p, other points on the same plane with p are also marked as covered in the recursive call. The search tree has a branching factor of three and if we use k as the depth, that is, we assign no more than k different axis-parallel planes, then the tree has 3^k leaves. If all the leaves do not make a cover, then the answer is NO. Otherwise, we have a cover and the instance is YES.

In d dimensions, each hyperplane is orthogonal to one of the dimensions, thus the branching factor becomes d while the depth is still k. The running time of the algorithm is then $O(d^k n)$.

Theorem 8. *The problem of covering a set of n points in $d \geq 3$ dimensions with no more than k axis-parallel hyperplanes of $d-1$ dimensions can be solved in $O(d^k n)$ time.*

Note that the algorithms of Langerman and Morin [LM02, LM05] can be adapted to solve the Hyperplane Cover in both the general and the rectilinear case, but their complexity remains the same. Their bounded-search-tree algorithm that runs in $O(k^{dk} n)$ time is clearly FPT with respect to k and d. However, the kernelization algorithm that runs in $O(k^{d(k+1)} + n^{d+1})$ time is FPT for

the parameter k, but not for the parameter d. When the problem is specialized to the rectilinear case, our FPT-algorithm above offers a better complexity.

4.4 Hyperplane Cover Restricted to a Finite Number of Orientations

We now extend our result by considering the problem of covering a set of points in d dimensions with k hyperplanes of $d-1$ dimension, where a set of these k hyperplanes is restricted to the hyperplanes having a finite number of orientations, ϕ. In other words, these k hyperplanes must have their orientations taken from a given finite set $\mathcal{H}^{d-1} = \{\Pi_1, \Pi_2, \ldots, \Pi_\phi\}$. We can construct the search tree with ϕ branches similar to the early discussion in Section 4.3 and the depth is still k. Thus, we obtain another FPT-algorithm, because the time complexity of our algorithm is polynomial in the size of the input n, although it is exponential in the parameter k.

Theorem 9. *The problem of covering a set of n points in $d \geq 3$ dimensions with no more than k hyperplanes of $d-1$ dimensions, such that these hyperplanes have a finite number $\phi \geq 2$ of orientations, can be solved in $O(\phi^k n)$ time.*

Chapter 5

The k-Bends Traveling Salesman Problem

This chapter discusses the k-Bends Traveling Salesman Problem. In this NP-complete problem, the inputs are n points in the plane and a positive integer k; we are asked whether we can travel in straight lines through these n points with at most k bends. There are a number of applications where minimizing the number of bends in the tour is desirable, because bends are considered very costly. We prove that this problem is FPT. The proof is based on the kernelization approach using reduction rules. We show that although the kernel size depends only on the parameter, a bounded-search-tree is necessary to ensure correctness when a line is exclusively used by a line-segment.

5.1 Introduction

The Minimum-Bends Traveling Salesman Problem seeks a tour through a set of n points in 2D, consisting of of straight lines, so that the number of bends in the tour is minimized. The n points in the tour are visited exactly once. The bends are the turns between the pair of line-segments. There are applications where fewer bends are preferred, such as surveying an area by aircraft, traffic routing, and space travel [Wag06, Chapter 2].

A tour with the minimum number of bends gives rise to a line cover. A line cover is a set of lines that cover the points in S. A simple observation is that a tour with k bends is a cover with k-line segments and, therefore, a cover by lines.

The main idea of our algorithms is to consider the line cover of the given points, because the size of the minimum line cover provides a bound for the number of bends of any traveling salesman tour. It should be noted that the Minimum-Bends Traveling Salesman Problem aims to minimize only the number of bends and not the distance; therefore, crossings are allowed. Both, general and rectilinear, versions of the Minimum-Bends Traveling Salesman Problem are studied in the literature. In the general version, the lines could be in any configuration, whereas in the rectilinear version, the line-segments are either horizontal or vertical. We look at the related work of this problem in details in the next section. Here, we find that, although an approximation algorithm does exist, there has been little progress in exact algorithms for the problem.

We reformulate the Minimum-Bends Traveling Salesman Problem as a parameterized problem and explore the solution based on parameterized complexity theory [DF99, FG06, Nie06]. Given n points in the plane and a positive integer k, we are asked if we can travel in straight lines through these n points in the plane with at most k bends. To the best of our knowledge, whether or not this problem has a fixed-parameter algorithm has not previously been established. We will prove that the general version of the k-Bends Traveling Salesman Problem is fixed-parameter tractable using kernelization with reduction rules in combination with the bounded-search-tree technique.

5.2 Related Work

We divide the results related to the Minimum-Bends Traveling Salesman Problem into three classes. First, we review the results concerning path problems that are related to minimum-bends tours. Then, we summarize the results about the tour problems and, finally, we discuss the results about problems required to cover regions. Covering regions is similar, but a region is covered instead of a set of points.

Tour problems aim to find a tour where each point lies along some line-segments of the tour; the challenge is to get back to the start minimizing some costs. Typically, these problems do not have obstacles. On the other hand, path problems try to connect two points with a path and, typically, the challenge is to avoid obstacles that block the direct path. In covering region problems, one

will have a target region to cover, and this region may or may not have holes. In the computational geometry literature, there are, again, two versions of the above problem, namely, the general version (non-rectilinear) and the rectilinear version.

5.2.1 Path Problems Related to Minimum-Bends Tours

Here, we only consider path problems that are related to minimum-bends tours. Given two distinct points in the plane, the problem is usually to find a path between these two points subject to a certain optimization function, such as the number of bends (turns) and/or the sum of the length of line-segments that make up the path. Some researchers focused on finding a collision-free shortest path inside a polygon [MPA92], while others considered a set of obstacles in the plane [LYW94, LYW96, DSW05, WDS09].

Tracing back the minimization of bends, we discover that, in 1992, Mitchell et al. [MPA92] studied the SHORTEST k-LINK PATH problem between two points inside a simple polygon P. The path is not restricted to being a rectilinear path. For minimizing the length of the k-link path, they gave two approximation algorithms, one for computing an (approximately shortest) k-link path inside a polygon with holes and a second for the case without holes. They also solved the problem of minimizing the total turn (the sum of all angles of all turns along the k-link path). For the latter problem, they produced an exact polynomial-time algorithm for a polygon with holes.

For the rectilinear case, Lee et al. [LYW94] presented algorithms to find rectilinear paths among obstacles in a two-layer interconnection model (in VLSI design). They consider four variants of rectilinear path problems; these are the SHORTEST PATH, the MINIMUM-BEND PATH, the SHORTEST MINIMUM-BEND PATH and the MINIMUM-BEND SHORTEST PATH. All algorithms are polynomial requiring $\Theta(e \log e)$ time, where e is the number of obstacle edges. A later survey [LYW96] looks at problems arising in motion planning for robotics and wire routing in VLSI. It shows that problems requiring a collision-free path between two distinguished points among rectilinear obstacles received significant attention.

In 2004, while considering the problem of covering a set of points with a minimum number of turns in rectilinear domain (2D and 3D), Collins [Col04]

proved new upper and lower bounds on the number of line-segments. The 3D case prompted Wagner et al. [WDS09] to solve the rectilinear version of the Minimum-Bends Path Problem in 3D. Their algorithm finds a path between two different points in 3D that has a minimum number of turns in $O(n^{5/2} \log n)$ time, where n is the number of corners among all obstacles. The Minimum-Bends Path Problem is also known as the Minimum Link Path Problem or the Minimum Link-Distance Problem [Wag06] (the number of line-segments in a path is sometimes referred to as the *link-distance* or the *link-length*).

5.2.2 Minimum-Bends Tour Problems

Given a set of points in the plane, the Minimum-Bends Traveling Salesman Problem (sometimes referred to as the Minimum Link Tour Problem) finds a tour through these points, consisting of the least number of straight lines. Stein and Wagner [SW01] studied the Minimum-Bends Traveling Salesman Problem. For the general version, they provided an $O(\log(\min(z, c)))$-approximation algorithm where z is the maximum number of co-linear points and c is the minimum number of lines used to cover all points. For its rectilinear version, they described a 2-approximation algorithm that runs in $O(n^{1.5})$ time.

5.2.3 Covering Region Problems

The class of region-covering problems is slightly more general because points are generalized to regions. These problems are usually known as "lawn-mowing" and "milling" problems introduced by Arkin et al. [AFM00]. Given a region in the plane, and given the shape of a "cutter", the problems ask for the shortest tour for the cutter, such that the cutter covers every point within the region as it travels along the tour. The difference between the lawn-mowing problem and the milling problem is that in the milling problem one do not allow the cutter to exit the region that it must cover. The milling problem has applications in the area of automatic tool path generation for NC (Numerically Controlled) pocket machining. The lawn-mowing problem arises in optical inspections, spray painting, and optimal search planning. Originally, only constant-factor approximation algorithms for finding shortest tours for the lawn-mowing and the milling problems were given [AFM00], because it was also proved that both problems

are NP-hard. Later, Arkin et al.[ABD+05] prove the NP-completeness of the minimum-turn milling problem and provide approximation algorithms for finding optimal covering tours where a cost that depends mainly on the number of turns is minimized.

Fellows et al. [FGK+08] studied the discrete milling problem in graphs. The problem asks whether a graph $G = (V, E)$ contains a k-lawnmower walk where a k-lawnmower walk is a walk of cost at most k that covers (mills) all the vertices of G. Fellows et al. prove that this problem is FPT, if the problem is parameterized by (k, t, d) where k is the number of turns, t is the tree-width of the input graph, and d is its maximum degree.

5.2.4 Our Contribution

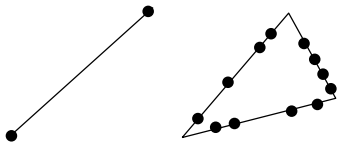
From the review above, we see that there has been little progress in exact algorithms that minimize the number of bends in tours covering points. The number of bends was considered in the path problems where polynomial algorithms are possible in some cases. Although Stein et al. considered the Minimum-Bends Traveling Salesman Problem both in the general and rectilinear versions or tour problems, their algorithms are approximation algorithms that cannot guarantee optimal solutions.

In this chapter, we prove that some variants of the tour problems are fixed-parameter tractable (belong to the class FPT). As such, they can be solved exactly and in polynomial time in the size of the input, although exponential time may be required as a function of the parameter. We will commence our discussion with the Minimum-Bends Traveling Salesman Problem in 2D. We reformulate this problem as a decision problem and we refer to it as the k-Bends Traveling Salesman Problem. We then show that the k-Bends Traveling Salesman Problem is FPT.

5.3 Line-Segments with Arbitrary Orientation

Problems with $k \leq 2$ are considered NO-instances. A bend of 180° is considered as two bends (see Figure 5.1). This is the same as the turn of a 360° angle. Thus, traversing the same line-segment back and forward is four bends. Each

vertex of a polygon is a bend and $k = 3$ is the smallest value with a YES-instance (for example, when S is covered by a triangle). Note that this is an important distinction of tours. Thus, we will assume that $k > 2$ and S has no duplicates (it is a set).



a) 4 bends b) 3 bends

Figure 5.1: The number of bends in a tour, a) a straight line-segment ($k = 4$), b) a triangle ($k = 3$).

Lemma 16. *There is always a minimum-bends tour without* 180° *bends.*

Proof. Let r be a 180°-bend in a tour T and b be the bend before r. Assume b is covered twice by T (else, reverse the order in T). Let p be the point in T before b. The tour that replaces the section "from p to b to r" by "from p to r" has one less bend. □

We now proceed with the kernelization approach by presenting structural lemmas and reduction rules.

Lemma 17. *If the set of points in S can be covered with $k > 2$ lines, then there is a tour with at most $2k$ bends.*

Proof. Let L be a set of lines that cover S. Construct a tour by linking each pair of lines in L by an extra line-segment in between (if they are not parallel, we can joining them at their intersection, thus saving a bend). In the worst case, the number of line-segments is at most $2|L|$ (because crossings are allowed). In a tour, one line-segment contributes to one bend. Thus, the number of bends is $2|L|$. □

Lemma 18. *If there exists a tour with at most k bends through a set S of n points in the plane, then these points can be covered by no more than k line-segments.*

Proof. Any tour with k bends is a cover with no more than k line-segments. □

This lemma implies the following result.

Reduction Rule 13. *If the minimum number of line-segments to cover a set S of n points in the plane is greater than k, then the instance (S, k) of the* k-Bends Traveling Salesman Problem *is a NO-instance.*

We refer to a set of k lines that cover the points in S as a k-cover. If (S, k) is a YES-instance of the k-Bends Traveling Salesman Problem, then the tour induces a k-cover. In fact, we can discover lines[1] that host line-segments of any tour.

Lemma 19. *Let (S, k) be a YES-instance of the* k-Bends Traveling Salesman Problem. *Let l be a line through $k + 1$ or more co-linear points. Then, the line l must host a line-segment of any tour T with k or fewer bends.*

Proof. If a line through $k+1$ different points was not part of the k-cover induced by the tour T, then $S' = S \setminus cover(l)$ (where $cover(l) = S \cap l$) has more than k points. The set S' would need more than k line-segments in T to be covered by the tour. This contradicts T has k or fewer bends. □

Note the distinction between a line (an unbounded infinite set of points with zero-width) and a line-segment that is the bounded portion of a line that constitutes the shortest path between the extremes of the segment [PS85, p. 18]. Line covers are made with lines, while tours are made with line-segments and bends at the extremes of the segments. However, finding a tour with the minimum number of bends is more complicated than finding a line cover for a set of points. The main reason is that a tour with the minimum number of bends may use a line-segment that does not cover all the points on the same line. See Figure 5.2 for an example. Although it is not difficult to compute the candidate lines, finding the segments of the tour needs more attention, because the same line may appear as separate line-segments in the optimal tour. Moreover, there could be many of these line-segments. Therefore, we will consider the following two cases. That is, we investigate tours where we allow several line-segments in the tour to be on the same line, then we look at the problem where one line-segment in the tour covers all the points on the same line.

[1] Lines contain line-segments.

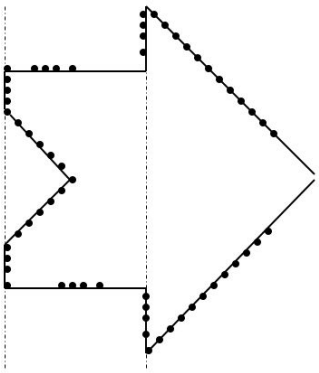


Figure 5.2: A tour that has two lines that are not just one line-segment in the tour with minimum bends.

5.3.1 Tours where Several Line-Segments Cover Points on the Same Line

First, we describe how to compute the k-cover. Consider a preprocessing of an input instance that consists of repeatedly finding $k+1$ or more co-linear points [LM05]. This process can be repeated until:

- $k+1$ lines, each covering $k+1$ or more different points are found, or
- no more lines covering $k+1$ points are found.

In the first case, we have $k+1$ disjoint subsets of points each co-linear and each with $k+1$ or more points. When this happens, we halt indicating a NO-instance. By Lemma 19, even if each of the $k+1$ lines hosts only one line-segment in the tour, we would still have more than k line-segments. In the second case, once we discover that we cannot find a line covering $k+1$ points and not exceeding k repetitions, we have a problem kernel[2].

Lemma 20. *Any given instance (S, k) of the k-*Bends Traveling Salesman Problem *can be reduced to a problem kernel S' of size at most k^2.*

[2]Although this preprocessing step has been used before (in more general form [LM05]), we suggest using dual space transformation to find $k+1$ or more co-linear points. Also, in the main algorithm we need to show that a tour with at most k bends can be constructed from the k-cover in polynomial time in the size of the input if the kernel is small (a tour is a cover with an additional connectivity constraint on the chosen subsets and therefore any results about k-line covering can not easily be translated into results for k-bend tours).

Proof. Let S' be the set of points after which we cannot repeat the removal of points covered by a line covering $k+1$ or more points. Recall that if we repeated the removal more than k times, we know it is a NO-instance. If we repeated no more than k times and it is a YES-instance, a witness tour T (a tour with k or fewer bends) would have matched the hosting lines with the lines removed. Also T is a k-cover of S'. Thus, the lines in T cover no more than k points. This means $|S'| \leq k^2$. □

Finding a line with more than $k+1$ points (or determining that one does not exist) can be achieved by placing all points in dual space and finding a vertex in the dual of degree $k+1$ or more (or determining that this does not exist). In fact, the dual can be constructed once and placed in an arrangement that can be searched by a plane sweep and updated by removing the dual-lines that intersect at a vertex of degree equal or greater than $k+1$. Asymptotically, this can have even faster time-complexity than with the algorithm of Guibas et al. [GOR91, GOR96]; however, our point is that the running time of finding each line covering $k+1$ or more co-linear points is bounded by $O((n+k)\log n+n^2)) = O(n^2)$, resulting in $O(kn^2)$ total time for k repetitions. Reduction Rule 13 is invoked if we exceed k repetitions. In total, the kernelization process requires $O(kn^2)$ time.

If kernelization resulted in a large kernel (greater than k^2), we answer as a NO-instance of the k-Bends Traveling Salesman Problem in constant time (Lemma 20). However, we still have to decide the instance of the k-Bends Traveling Salesman Problem if the kernel is small. Unfortunately, we cannot formulate Lemma 19 as a reduction rule in the usual sense, because we do not know how many line-segments of the tour are going to be hosted by a line over $k+1$ points or more. So, we cannot provide a value of k for exhaustive search of the reduced instance of the tour. Therefore, to complete the argument that the k-Bends Traveling Salesman Problem is fixed-parameter tractable, we need to show that a tour with at most k bends can be constructed from the k-cover in polynomial time in the size of the input when the kernel is small (no more than k^2 points). For this, we require the following result.

Lemma 21. *If a tour T has the minimum number of bends, then every line segment in T covers at least two points.*

Proof. If a line-segment in the tour covers no points, it can be replaced by a parallel line-segment that covers at least one point. Consider a tour with a line-segment l that covers only one point $p \in S$. Then, simply spinning l (rotating on p as an axle and sliding the bends of l until another point is met), we find an equivalent tour with the same number of bends where now l covers at least two points. □

We now propose the following algorithm. Let L_{k+1} be the set of all lines that cover at least $k+1$ points in S. We know $|L_{k+1}| \leq k$ and all these lines have segments that are part of the tour (if it is a YES-instance) so, we will later use these lines to construct the tour. Consider the kernel that has at most k^2 points. We put the two extremal points of each line of L_{k+1} into the kernel, thus our reduced instance S' has at most $k^2 + 2k$ points. Let R be all the lines through two or more points of S'. The size of R is less than the number of pairs of points in S'. Thus, $|R| = O(k^4)$. Now, we search exhaustively all tours over the lines in $L_{k+1} \cup R$. For example, a naive algorithm for this part tests all subsets of size k of the intersections of the lines in $L_{k+1} \cup R$ to decide whether we can link these k candidate-intersections into a tour with k or fewer bends and all the n points are covered by the line-segments that make up the tour. Note that the k candidate-intersections of the lines essentially host the bends in the tour. Since $|L_{k+1} \cup R| = O(k^4)$, the number of intersections is $O(k^8)$, the number of subsets of size k is bounded by $\binom{k^8}{k} = O(k^{8k}/k!)$. Testing all permutations of the k candidate-intersections hosting the bends in the tour results in a time bounded by $O((k!)(n)(k^{8k}/k!)) = O(k^{8k}n)$. Therefore, we can decide the k-Bends Traveling Salesman Problem in $O(kn^2 + k^{8k}n)$ time.

Theorem 10. *The k-Bends Traveling Salesman Problem is FPT and can be solved in $O(kn^2 + k^{8k}n)$ time.*

5.3.2 One Line-Segment Covers All the Points on the Same Line

In this subsection, we show that the bounded-search-tree technique does apply to a special case of the k-Bends Traveling Salesman Problem. Namely, consider this problem but restricted to the case where the tour must be such that each line-segment in the tour covers all the points on the same line. Thus, in this

case, a cover using a line is the same as a cover using a line-segment. By revisiting the proof by Arkin et al. [AMP03], we will show that this restricted problem remains NP-complete. This is important, because it means that even with this restriction the problem has no polynomial-time exact algorithm, unless P=NP. The reduction is from the Line Cover problem. Consider an instance (S,k) of Line Cover (we are asked if the set S of n points can be covered with k lines). We will produce an instance of the k-Bends Traveling Salesman Problem where the tour with minimum bends has line-segments that correspond uniquely to lines. The idea is to surround the points in S with k parallel wedges, each wedge is made of two lines (refer to Figure 5.3).

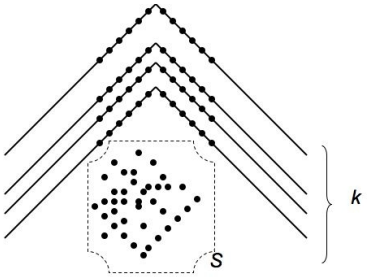


Figure 5.3: Construction to reduce in polynomial time the Line Cover problem to the k-Bends Traveling Salesman Problem where one line-segment covers all the points on the same line.

By pre-computing all possible slopes of any line covering two or more points in S, we can make sure that none of the lines in the wedges are parallel to lines that cover two or more points in S. We also place more than $|S|$ points in the apex of each wedge to force the tours to travel each wedge and make one bend at its apex. Call these additional points S'. The instance for k-Bends Traveling Salesman Problem has an input set $S \cup S'$ and a number of bends is set to $3k$. It is easy to see that if the instance of the Line Cover is a YES-instance, then the k lines can be used to make a tour by traveling a covering line until it meets a leg of a wedge, moving along the wedge to its apex, then to the other leg of the wedge, and exiting with another covering line. The tour has $3k$ bends because there is the bend at the apex and one bend at each leg of each wedge. More importantly, for our purposes, note that every point on a line is always covered by just one line-segment. Conversely, a tour with $3k$ bends would need to use the wedges, and therefore, would need to cover S with k lines. However, we now show that this version of the problem is FPT.

We apply the preprocessing of the previous section repeatedly to find a line through $k+1$ or more co-linear points. When we can no longer find such a line, we have a reduced instance of size at most k^2. In this setting of the problem, we know that once we find $k+1$ or more co-linear points, the entirety of these points must be in a line-segment of the tour. This enables us to be precise about how to reduce the number of bends in the reduced instance of the tour when such a line is removed. However, to complete the algorithm, there would be several cases and this time we can use the bounded-search-tree technique to ensure all possibilities of linking the $k+1$ or more points into a tour. We discuss this in the next paragraph.

Recall that L_{k+1} is a set of all lines that cover at least $k+1$ points. Let l_i be a line in the set L_{k+1}. The minimum number of bends required to cover the tour when $l_i \in L_{k+1}$ was removed is denoted by k'. Although Lemma 19 at the moment tells us that we can remove the points covered by l_i from consideration in the reduced instance, it does not tell us what value for k' should be considered in the reduced instance. That is, from the instance $(S,\ k)$ of the k-Bends Traveling Salesman Problem, we move to an instance of the form $(S \setminus \mathit{cover}(l_i), k')$. Interestingly, the parameter k' that is the number of bends in the tour of the new (reduced) instance can appear in four different ways depending on how l_i is connected to the reduced instance tour to build a tour for the original instance. Figure 5.4 shows the four possibilities for reducing the number of bends in the tour. Each reduction of the instance (S, k) is required to meet some conditions in order to ensure correctness. We define four conditions that the tour of the reduced instance may have to meet. For the correctness of the algorithm, we will recursively test all of the following claims.

Claim 1. *If the reduced instance can be covered with $k' = k-3$ bends, then the original instance can be covered with k bends (see Figure 5.4a).*

Proof. Let t be a tour of $S \setminus \mathit{cover}(l_i)$ with $k' = k-3$ or fewer bends and with line-segments in a bijection with lines. A tour of S can be constructed as follows. Let p be the next point in t after a bend b. Construct a tour of S by replacing a line-segment s (a line-segment from b to p) by a path of three line-segments as follows. A segment from b to the first point in l_i, a segment along l_i covering all these points and then a segment from the last point in l_i to p. This tour adds two bends in l_i and one at p, thus the new tour has k bends.

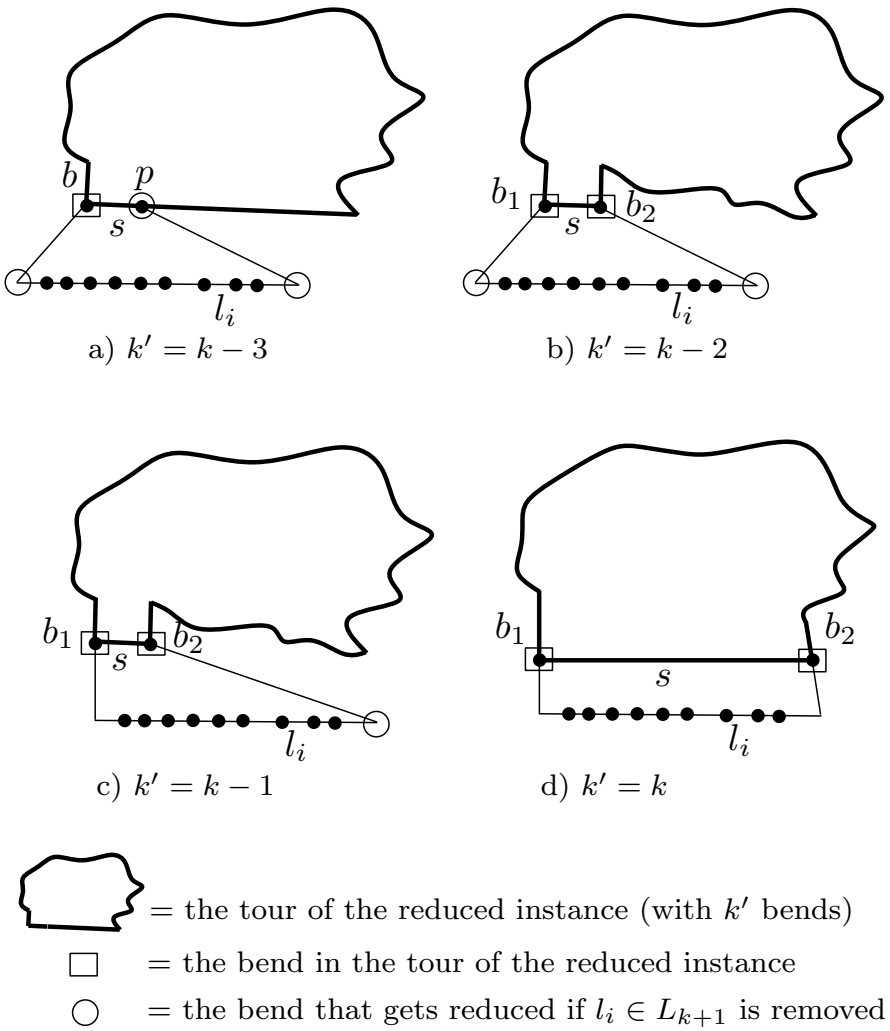


Figure 5.4: The minimum number of bends (k') after removing a line covering at least $k + 1$ points.

The line for l_i is a new line, because all points in S covered by l_i were removed in the reduced instance. The line-segments b to the first point in l_i and the last point in l_i to p can be shifted along l_i away from points in l_i to ensure these are new lines. □

Claim 2. *If the reduced instance can be covered with $k' = k - 2$ bends and the tour has a line-segment with no points (except maybe at the extremes) and no two line-segments share a line, then the original instance can be covered with k bends (see Figure 5.4b).*

Proof. Let b_1 and b_2 be the extremes of a line-segment s in a tour t of the reduced instance with $k-2$ bends and there are no points in s (except maybe at b_1 or at b_2). Construct a tour for the original instance from t but replacing s with a path that goes from b_1 to the first point in l_i, then traveling l_i to cover all points of l_i and then a segment from the last point of l_i to b_2. This new tour adds onlytwo bends at l_i so it has k bends. The line l_i is new and the other two segments can be shifted away from points in l_i to ensure they are also in new lines. □

Claim 3. *If the reduced instance can be covered with $k' = k - 1$ bends of a tour t, and there is one bend b_1 where the line-segment s' at b_1 can be extended to meet l_i without partitioning the points covered by l_i, then the original instance can be covered with k bends (see Figure 5.4c).*

Proof. Let s be the segment after s' in t. Similar to Claim 1 and Claim 2 above, the path $s's$ in t is replaced to construct a tour of the original instance S. First, s' is extended to meet l_i, then travel along l_i covering all the points and second a new segment is made from the last point of l_i to the extreme of s. This adds one bend. The extension of s' is the same line, the line l_i is new and the last segment can be shifted on l_i to make sure that it is hosted by a new line. □

Claim 4. *If the reduced instance can be covered with $k' = k$ bends, and there are two bends b_1 and b_2 with a line-segment s between them, and the previous line-segment s' and the next line-segment s'' can both be extended to meet l_i without partitioning the points, but the points in cover(l_i) lie in between the intersection, then the original instance can be covered with k bends (see Figure 5.4d).*

Proof. Again, the segment s in t is replaced by the extensions of the segment before it, the traveling of l_i and the extension of the segment after s. This adds

no bends, because b_1 and b_2 are moved to sit at l_i and the new line-segment along l_i is in a new line. □

k-Bends($S, bound, depth$)

Input: A set S of n points and a positive integer k
Output: A list of tours covering S with k or fewer bends where no two line-segments are hosted by the same line; the list is empty if there is no such tour.

```
1  if bound ≤ 2 ∨ depth > k
2     then return ∅
3  if S can be covered by a triangle
4     then return {triangle}
5  if there is no line with k + 1 points or more
6     then    (*Solving the instance of at most k^2 points*)
7          return all tours of (S, k) by exhaustive search
8     else    (*Checking 4 claims recursively*)
9             Let l_i be a line that covers at least k + 1 points in S
10            R ← ∅;
11            T ← k-Bends(S \ cover(l_i), bound − 3, depth + 1)
12            for t ∈T ∧ CheckClaim(t, Claim 1)
13                do R ← R ∪ BuildTours(t, Claim 1)
14            T ← k-Bends(S \ cover(l_i), bound − 2, depth + 1)
15            for t ∈T ∧ CheckClaim(t, Claim 2)
16                do R ← R ∪ BuildTours(t, Claim 2)
17            T ← k-Bends(S \ cover(l_i), bound − 1, depth + 1)
18            for t ∈T ∧ CheckClaim(t, Claim 3)
19                do R ← R ∪ BuildTours(t, Claim 3)
20            T ← k-Bends(S \ cover(l_i), bound, depth + 1)
21            for t ∈T ∧ CheckClaim(t, Claim 4)
22                do R ← R ∪ BuildTours(t, Claim 4)
23            return R
```

Figure 5.5: Recursive FPT-algorithm with bounded depth.

Figure 5.5 shows this corresponds to the bounded-search-tree technique, because the recursion tree is bounded. If there is no line with $k+1$ points or more, we solve the instance (S, k) exhaustively. Otherwise, we remove a line $l_i \in L_{k+1}$ from the tour and recursively explore the four possibilities of reducing the number of bends in the tour when such a line is removed. The parameters are passed by value and the returned result is a sequence of all tours with k or fewer bends that have one line-segment covering all the points on the same line (modulus

the equivalence of Lemma 21). To simplify the pseudo-code, we assume we have a Boolean function CheckClaim(t,C) that, given a tour t, returns `true` if the tour t satisfies Claim C (and provides also in a parameter by reference the list of bends where the claim is satisfied). We also assume that we have a function BuildTours(t,C) that builds the corresponding list of tours for the original instances from a solution tour of the reduced instances that satisfy Claim C. Note that from a solution of the reduced instance, there are at most k ways of constructing a solution of the original instance by any of the constructions of the four claims above, because all of these constructions replace at most one line-segment in the tour and the reduced instance must have a tour with less than k line-segments.

The k-Bends function above is called as k-Bends$(S, k, 0)$, and note that k is maintained as a global value while the third parameter accounts for the depth in the recursion tree (ensuring bounded depth of k). We analyze the asymptotic running time of the k-Bends function in Figure 5.5. First, checking if *bound* ≤ 2 or if S can be covered by a triangle can be performed trivially in linear time. However, the running time of checking four claims recursively is not that obvious. We show that this task can be carried out in polynomial time in n (although exponential in k). The fan-out of the tree is four. Hence, the worst-case number of leaves in the recursion tree is $O(4^k)$ and the number of internal nodes in the tree is $O(4^{k-1})$. The work at each internal node is polynomial in n. At the leaves, there is no line with $k+1$ points or more; we solve the problem kernel exhaustively by testing all subsets of size k of $\binom{k^2}{2}$ lines. That is, $\binom{k^4}{k} = O(k^{4k}/k!)$ subsets to be tested. When we consider all the leaves, the work becomes $O((4^k)(k^{4k}/k!)(k!)(k^2)) = O(4^k k^{4k+2})$ time. Note that the list passed back by the recursive calls has a size bounded by a function of k (and independent of n), the work in the nodes by CheckClaim and BuildTours depends only on k and not on n. A careful analysis produces the total running time of $O(kn^2 + 4^k k^{4k+2})$.

Theorem 11. *Under the constraint that one line-segment in a tour covers all the points on the same line, the k-Bends Traveling Salesman Problem is FPT and can be solved in $O(kn^2 + 4^k k^{4k+2})$ time.*

We note that the complexity of the fixed-parameter algorithm here is smaller than when we allow line-segments to share a line.

5.4 Chapter Summary

We established that the two hard problems in this chapter belong to the class FPT. We remark that our algorithms remain far from practical since the complexity even for small k grows very rapidly. See Table 5.1 for an illustration.

Table 5.1: The value $kn^2 + k^{8k}n$ for various values of n and k.

	$n = 50$	$n = 100$	$n = 150$
$k = 2$	3.28×10^6	6.57×10^6	9.88×10^6
$k = 3$	1.41×10^{13}	2.82×10^{13}	4.24×10^{13}
$k = 5$	4.55×10^{29}	9.09×10^{29}	1.36×10^{30}
$k = 7$	1.06×10^{49}	2.12×10^{49}	3.17×10^{49}
$k = 10$	5×10^{81}	1×10^{82}	1.5×10^{82}

However, as it is usually the case with parameterized complexity, establishing a problem is in the class FPT leads to improvements that frequently result in practical methods. This also immediately suggest an open problem, namely, to improve the complexity of the algorithms.

Chapter 6

The Rectilinear k-Bends Traveling Salesman Problem

Given n points in the plane and a positive integer k, the Rectilinear k-Bends Traveling Salesman Problem asks if there is a piecewise linear tour through the n points in S with at most k bends, where every line-segment in the path is either horizontal or vertical. This problem is motivated by the applications in VLSI design where the number of bends in a path affects the resistance and hence the accuracy of the expected timing and the voltage in the chips. We prove that this problem belongs to the class FPT. We introduce two types of constraints derived from the distinction between line-segments and lines. Note that a rectilinear tour with k bends is a cover with k rectilinear line-segments and, therefore, a cover by rectilinear lines. We derive FPT-algorithms with different techniques and improved time complexity for these cases.

6.1 Introduction

In the previous chapter, we studied the general version of the Minimum-Bends Traveling Salesman Problem that seeks a tour through a set of n points in 2D, consisting of straight lines, so that the number of bends in the tour is minimized. This version of the problem is NP-complete [AMP03]. When restricted to only horizontal and vertical line-segments, we call this problem the Rectilinear Minimum-Links Traveling Salesman Problem. It remains open if this version of the problem is NP-complete, or is polynomially solvable [Wag06], although the Rectilinear Minimum-Links Traveling Salesman Prob-

LEM has closely related problems such as the MINIMUM-TURN MILLING and the THIN ORTHOGONAL MILLING that are NP-complete [ABD+05]. Originally posed as a demanding problem by Klamkin [Kla54] the MINIMUM-BENDS TRAVELING SALESMAN PROBLEM of a $n \times n$ grid was shown to require at least $2n-2$ segments (for $n \geq 3$) [Kla55] before considered solved because $2n-2$ are also necessary [Sel55].

Stein and Wagner [SW01] approximately solved the rectilinear version of the MINIMUM-BENDS TRAVELING SALESMAN PROBLEM. They gave a 2-approximation algorithm that runs in $O(n^{1.5})$ time. However, no polynomial-time exact algorithm is known for this rectilinear tour problem, despite the motivating applications in VLSI. We reformulate the RECTILINEAR MINIMUM-LINKS TRAVELING SALESMAN PROBLEM as a parameterized problem in 2D and we call it the RECTILINEAR k-BENDS TRAVELING SALESMAN PROBLEM. From the parameterized complexity perspective, we show that the problem belongs to the class FPT by kernelization.

The approximation algorithms [SW01] use algorithms for covering points with lines and then stitch together the lines to constitute a tour. In those approximation algorithms, the solutions seem to require that no two line-segments of the tour share the hosting line. The requirement that line-segments of the tour are hosted exclusively by a line motivates a different variant of the problem. Another variant also emerges if we require the same line-segment orientation[1] to cover all points on the same line. We provide FPT-algorithms with improved complexity for these two variants of the rectilinear tour problem. Our algorithms for these variants are based on bounded-search-tree techniques. Results are summarized in Table 6.1.

6.2 Types of Rectilinear Tours

Given a set S of n points in the plane, and a positive integer k, we are asked if there is a piecewise linear tour (which may self-intersect) through the n points in S with at most k bends where every line-segment in the path is either horizontal

[1] *Orientation* will usually refer to the slope of a line with respect to the x-axis; when applied to a line segment, it refers to the orientation of the line hosting it. We will use *direction* for the direction of travel of a line-segment in a tour; for example, a vertical line-segment can be travelled North-South or on the opposite direction, South-North.

or vertical (the tour must return to its starting point). An instance of the Rectilinear k-Bends Traveling Salesman Problem is encoded as the pair (S, k), and we call the solution a *rectilinear tour*. Note that in this rectilinear version of the problem, the standard convention is to restrict the tour to 90° turns. A 180° turn is considered two 90° turns with a zero-length line-segment in between. If $n \geq 3$, it is always possible to transform a tour with a 180° turns into a tour with only proper 90° turns and line-segments of a positive length. With these conventions, every 90° turn consists of one horizontal line-segment and one vertical line-segment, both of positive length. Therefore, we will assume $n \geq 3$; we also accept that there are no tours with an odd number of bends and that the required number, k, of bends is even. A rectilinear tour must have at least four bends.

Lemma 22. *If there exists a rectilinear tour with at most k bends, the number of horizontal line-segments is at most $k/2$ and the number of vertical line-segments is at most $k/2$.*

Proof. If there exists a tour with at most k bends, there are at most k line-segments. In a rectilinear tour, the number of horizontal line-segments is equal to the number of vertical line-segments. Hence, there cannot be more than $k/2$ horizontal line-segments and no more than $k/2$ vertical line-segments. □

We distinguish three types of rectilinear tours derived from the distinction between line-segment and line. Figure 6.1 illustrates this. In the first case, we require that if l is the line containing a line-segment s of the tour (that is, $l \cap s = s$), then the line-segment s covers all the points in $S \cap l$. In the second case, if a point p is on a line l used by the tour, then there must be a segment of the tour with the same orientation as l covering p. The third type does not have any of the above constraints. For an illustration, consider the set of points in Figure 6.1 (a). Each vertical cluster or horizontal cluster of points is sufficiently numerous to force being covered by at least one line-segment of the tour with minimum bends. Without any constraint, the two tours with eight bends in Figure 6.1 (b) are optimal; however, in both, there are two vertical line-segments that share a common line (in the second one, the two segments $\overline{2,3}$ and $\overline{6,7}$ are not drawn exactly co-linear, so the reader can appreciate that the tour self-intersects).

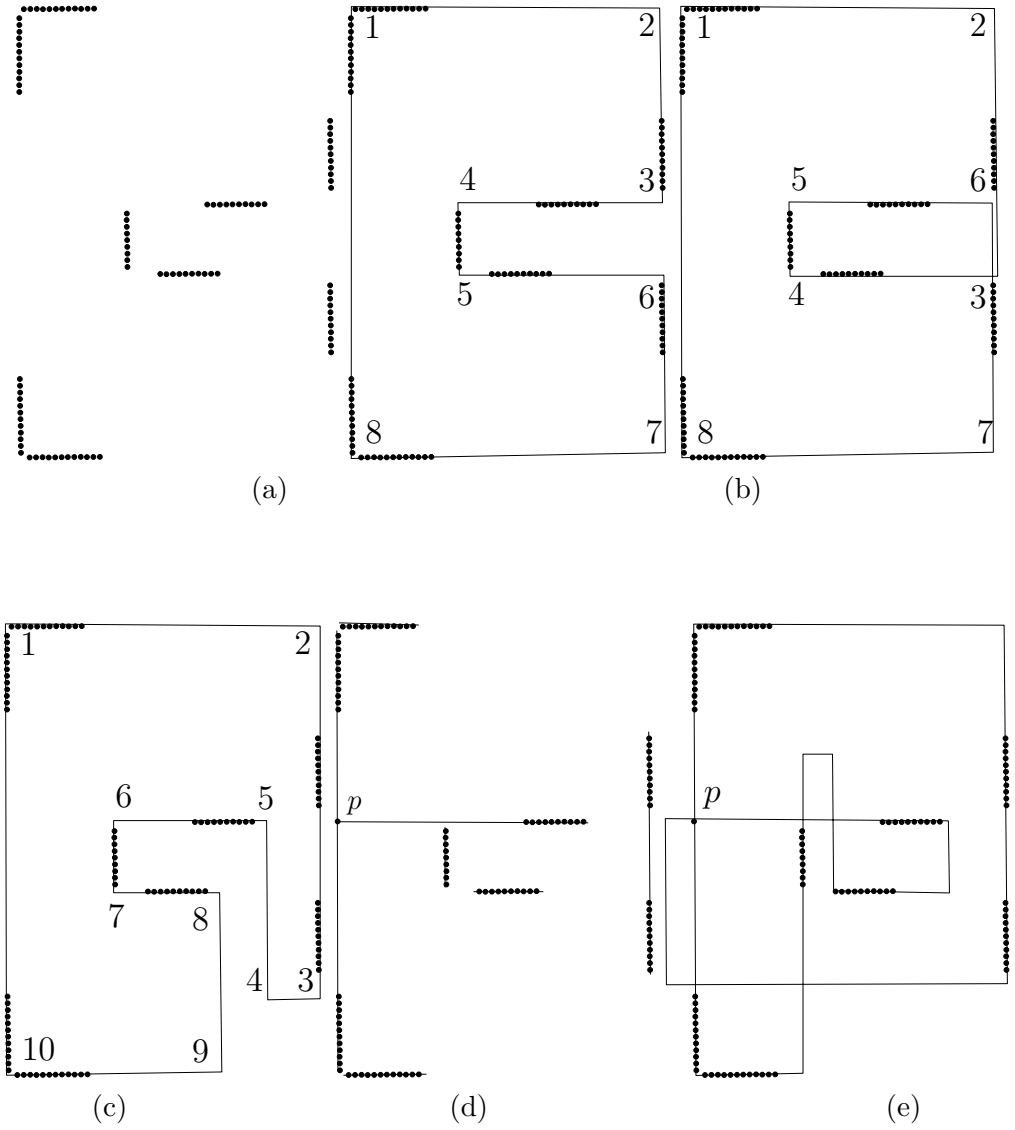


Figure 6.1: Three types of the Rectilinear k-Bends Traveling Salesman Problem emerge as we consider the legal line-segments that are part of the tour.

The first constraint will require that all the points under these two line-segments be covered by only one line segment. In Figure 6.1 (c), we see that this constraint forces the tour to travel over the large line-segment $\overline{2,3}$ and the minimal tour has now 10 bends. The second constraint can be illustrated if we add a point p to S, as in Figure 6.1 (d). This new point lies on the lines used by two types of line-segments, one horizontal and one vertical and both of these types of line-segments must cover the point (that is, the point p cannot be only covered by the vertical line segment, because it belongs to a line where there is a horizontal line-segment of the tour). Note that this constraint is satisfied by points that are located at a bend. That is, if a bend is placed at a data point q, this constraint is automatically satisfied for q, because the horizontal line-segment at q plus all other horizontal line-segments on the same horizontal line will cover q (and symmetrically for the vertical line-segment). In Figure 6.1 (d) all the line-segments drawn must be contained in a line-segment of a minimum bends tour satisfying the second constraint (which now has 12 turns; see Figure 6.1 (e)).

We will show that the problem in general (without any constraints) is FPT. The first variant requires that one line-segment covers all the points on the same line while the second variant requires the same orientation be represented by a line-segment that covers the points. We will prove that the first variant and the second variant are also FPT but the time complexity of the FPT-algorithms here is smaller than in the general setting.

6.3 Tours without Constraints

We start our discussion with the following series of reduction rules. They have the rectilinear variant to some of the rules in Chapter 5.

Reduction Rule 14. *If $k \geq 4$ and all points in S lie on only one rectilinear line, then the instance (S, k) of the* Rectilinear k-Bends Traveling Salesman Problem *is a YES-instance.*

The next rule is derived from Lemma 22, which states that there cannot be more than $k/2$ horizontal line-segments and no more than $k/2$ vertical line-segments in a rectilinear tour.

Reduction Rule 15. *If the minimum number of line-segments needed to cover a set S of n points in the plane is greater than k, then the instance (S, k) of the* Rectilinear k-Bends Traveling Salesman Problem *is a NO-instance.*

We refer to a set of k rectilinear lines that cover the points in S as a k-cover. If (S, k) is a YES-instance of the Rectilinear k-Bends Traveling Salesman Problem, then the tour induces a k-cover. In fact, we can discover lines that host line-segments of any tour.

Lemma 23. *Let (S, k) be a YES-instance of the* Rectilinear k-Bends Traveling Salesman Problem. *Let l be a rectilinear line through $1 + k/2$ or more co-linear points. Then the line l must host a line-segment of any tour T with k or fewer bends.*

Proof. Without loss of generality, assume l is a vertical line. In contradiction to the lemma, assume there is no vertical segment on l for a tour T that covers with k bends. Then, the $1 + k/2$ points in $S \cap l$ would be covered by horizontal lines in T. According to Lemma 22, this contradicts T has k or fewer bends. □

Note that the rectilinear line through S' in the proof above may be represented by the separate line-segments of a witness tour. We now describe how to compute a k-cover if one exists. Consider a preprocessing of an input instance that consists of repeatedly finding $1 + k/2$ or more co-linear points and on a rectilinear line (that is, they are on a vertical or horizontal line). This process can be repeated until:

- $k + 1$ rectilinear lines, each covering $1 + k/2$ or more different points are found, or
- no more rectilinear lines covering $1 + k/2$ points are found.

In the first case, we have $k + 1$ disjoint subsets of points each co-linear and each with $1 + k/2$ or more points. When this happens, we halt indicating a NO-instance. By Lemma 23, even if each of the $k + 1$ rectilinear lines hosts only one line-segment in the tour, we would still have more than k line-segments. In the second case, once we discover that we cannot find a line covering $1 + k/2$ points and not exceeded k repetitions, we have a problem kernel.

Lemma 24. *Any instance (S, k) of the* Rectilinear k-Bends Traveling Salesman Problem *can be reduced to a problem kernel S' of size at most $k^2/2$.*

Proof. Let S' be the set of points after which we cannot repeat the removal of points covered by a rectilinear line covering $1 + k/2$ or more points. Recall that if we repeated the removal more than k times, we know that it is a NO-instance. If we repeated no more than k times and it is a YES-instance, a witness tour T would have matched hosting lines with the lines removed. Also T is a k-cover of S'. Thus, the lines in T are rectilinear and each covers no more than $k/2$ points. This means $|S'| \leq k^2/2$. □

From the above lemma, if we have a kernel of size larger than $k^2/2$, then it is a NO-instance. Algorithmically, we can either determine that we have a NO-instance in polynomial time in k and in n, or we have a problem kernel where we still have to determine if it is a YES or NO-instance. What follows resolves this issue. With the next lemma we prove that each best tour is always equivalent to a tour with the same number of bends but where line-segments on the same line are not disjoint (as the two tours with 8 bends in Fig. 6.1 (b)).

Lemma 25. *Every optimal rectilinear tour T that has two disjoint line-segments hosted by the same line can be converted into a tour T' with the same number of bends and where the line segments have no gap.*

Proof. Consider the case where the two disjoint line segments, s_1 and s_2, are traversed by T in the same direction. Figure 6.2 (a) and (b) shows the transformation of the two line segments s_1 and s_2 in the same hosting line l_p into a new tour by enlarging both s_1 and s_2 and flipping the direction of two bends. Clearly, there are no more bends and although the direction of the path between these two bends is reversed, we still have a well formed tour. Note that Figure 6.2 (a) deals with the case when the bends share a label[2] while (b) is the case the two bends share no label. Once this case is clear, the case where two disjoint line segments s_1 and s_2 are traveled by T in opposite directions is also clear, although now four bends are involved. Figure 6.2 (c) illustrates this. □

[2]We say the turns have a common label if the turns are NE at p and NW at q (with N in common) or the turns are SE at p and SW at q (with S in common) and, symmetrically, the turns have a common label if the turns are SE at p and NE at q (with E in common) or the turns are SW at p and NW at q (with W in common) where N, S, W, and E stand for North, South, West and East, respectively.

Moreover, the transformation in the proof above always increases the length of the tour. So, if we apply it again, it will not undo the work done by its previous application. Thus, we can apply it repeatedly, until there are never two disjoint line-segments in an optimal tour. In particular, we can assume optimal tours have no gap between co-linear line-segments, as in Figure 6.2 (a).

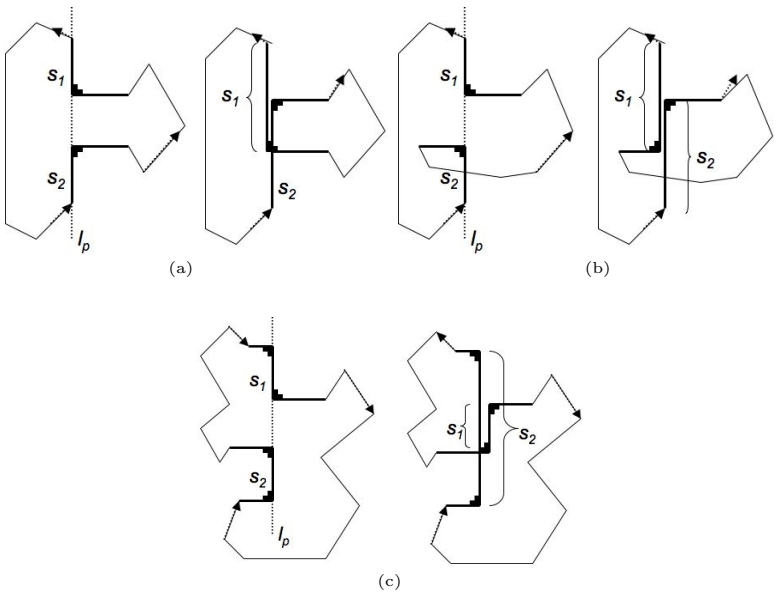


Figure 6.2: The tour T is changed, preserving the number of bends. Thick lines correspond to the segments of the tour, while thin lines indicate the Jordan curve of the tour somewhere in 2D.

Let L_k be the set of rectilinear lines found by kernelization having at least $1 + k/2$ co-linear points. We know $|L_k| \leq k$ and all these lines have segments that are part of the tour. Given a vertical line $l \in L_k$, $cover(l) = S \cap l$. we let h_{max} be the horizontal line through the point $p_{max} \in cover(l)$ with the largest y coordinate, while h_{min} is the horizontal line through the point $p_{min} \in cover(l)$ with the smallest y coordinate. The line $h_{(max-i)}$ is the horizontal line through the i-th point in $cover(l)$ below p_{max} while $h_{(min+i)}$ is the horizontal line through the i-th point in $cover(l)$ above p_{min}, for $1 \leq i \leq (k/2-1)$. We expand the set L_k as follows. For every vertical line $l \in L_k$, we add k horizontal lines $h_{max}, h_{(max-1)}, \ldots, h_{(max-k/2+1)}$ and $h_{min}, h_{(min+1)}, \ldots, h_{(min+k/2-1)}$ to L_k. Symmetrically, for every horizontal line $l \in L_k$, we add k vertical lines $v_{max}, v_{(max-1)}, \ldots, v_{(max-k/2+1)}$ and $v_{min}, v_{(min+1)}, \ldots, v_{(min+k/2-1)}$ to L_k, where

v_{max} passes through $p_{max} \in cover(l)$ with the largest x-coordinate and v_{min} passes through the point $p_{min} \in cover(l)$ with the smallest x-coordinate. The lines $v_{(max-i)}$ and $v_{(min+i)}$ are defined in a similar way to that of $h_{(max-i)}$ and $h_{(min+i)}$. Note that, if l covers less than k different points, we add a line that is orthogonal to l on every point in $cover(l)$. We call L_k with all these additional lines the set L'_k and $|L'_k| \leq k^2$.

Let H be all the horizontal lines through a point in the kernel S' and let V be all the vertical lines through a point in S'. Thus, $|H| \leq k^2/2$ and $|V| \leq k^2/2$. Now, we add to L'_k all the lines in V and all the lines in H. The new set $R = L'_k \cup H \cup V$ has quadratic size in k. Moreover, the set I of all intersections of two lines in R has also polynomial size in k (that is, $O(k^4)$). We will argue that a rectilinear tour of k bends exists for any given instance if and only if a tour with k bends exists with bends placed in the set I of intersections.

Lemma 26. *If the instance has a rectilinear tour T with k or fewer bends, then a tour can be built with k or fewer bends and all the bends are at I, the set of all intersections of lines in R where $R = L'_k \cup H \cup V$.*

Proof. We will show that for every YES-instance, we can transform the witness tour T to a tour T' with the same number of bends, where every line-segment in T' is hosted by a line in R. From this, it follows that the set I hosts the possible positions for the bends in T'.

Let p be any point in S. If this is a YES-instance, then there is a witness tour that has a line-segment l_p covering the point p. Moreover, because of Lemma 25, we can assume that if $s_1, \ldots, s_i$ are line-segments hosted by a rectilinear line l_p, there are no gaps; that is, $\cup_{j=1}^{i} s_j$ is not disjoint. If the point p was from the kernel, the case is proved because the line-segment is hosted on $H \subseteq R$ or on $V \subseteq R$. Otherwise, the point p was from a line discovered in kernelization. If the point p is covered by some s_j in the same orientation as the line discovered by kernelization, then again the case is proved. If the point p is covered in the k-bends witness tour by a segment orthogonal to the line discovered in kernelization, then the case becomes delicate. Assume p is on a vertical line l_p discovered by the kernelization, but in the witness tour T', p is covered in a horizontal line-segment over a line h_p. If h_p was discovered by kernelization, the case is proved, and the same if it was a line in H. Therefore, line h_p is either above or below $\cup_{j=1}^{i} s_j$, because we are in a case where p is not covered by $\cup_{j=1}^{i} s_j$

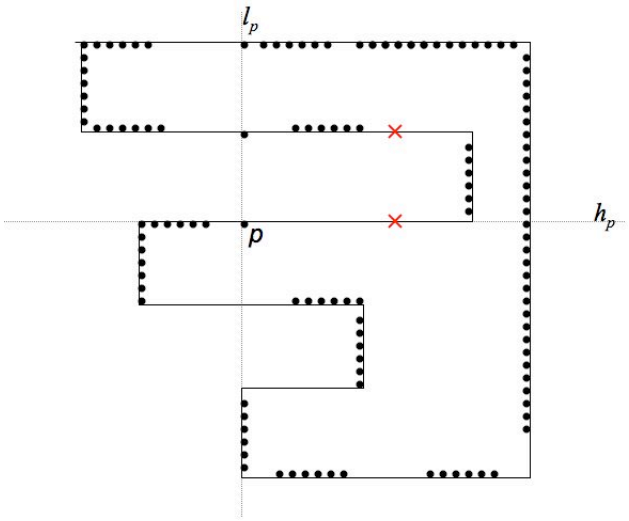


Figure 6.3: The point p is covered by a horizontal line that is not found in the set H or the set L_k.

(see Figure 6.3, for example). Moreover, there cannot be more than $k/2$ points in the same situation as p. Otherwise, the witness structure would have more than $k/2$ horizontal line-segments (Lemma 22) at those positions. Therefore, the rank of p from either end of l_p must be no more than $k/2$. That is, p is in one of the lines $h_{max}, h_{(max-1)}, \ldots, h_{(max-k/2+1)}$ or $h_{min}, h_{(min+1)}, \ldots, h_{(min+k/2-1)}$ that we have in L'_k. Since $h_p \subseteq L'_k$, we have $h_p \subseteq R$. □

Theorem 12. *The* Rectilinear k-Bends Traveling Salesman Problem *is FPT and can be solved in* $O(kn^2 + k^{4k}n)$ *time.*

Proof. The algorithm computes R and searches exhaustively all tours over the lines in R. For example, a naive algorithm tests all subsets of size k of I to decide whether or not these k candidate-intersections can be completed into a tour with at most k bends and all the n points lie on the line-segments that make up the tour. Since the number $|I|$ of intersections by lines in R is $O(k^4)$, the number of subsets of size k is bounded by $\binom{k^4}{k} = O(k^{4k}/k!)$. Testing all permutations to cover all tours result in a time bounded by $O((k!)(n)(k^{4k}/k!)) = O(k^{4k}n)$. Kernelization can be performed in $O(kn^2)$ time. Thus, we can decide the Rectilinear k-Bends Traveling Salesman Problem in $O(kn^2+k^{4k}n)$ time. □

6.4 Tours with Constraints

Now, we decide the problem for the rectilinear tour with at most k bends for the two variants introduced earlier. In fact, for both variants, in the first phase we compute several k-covers (candidates set L of k lines that cover all the n points in S). However, instead of kernelization, to identify the hosting lines, the technique here will be a bounded-search-tree. In the second phase, we check if each candidate k-cover can constitute a tour based on such a k-cover. To find these k-covers, we consider a search tree as illustrated in Figure 6.4.

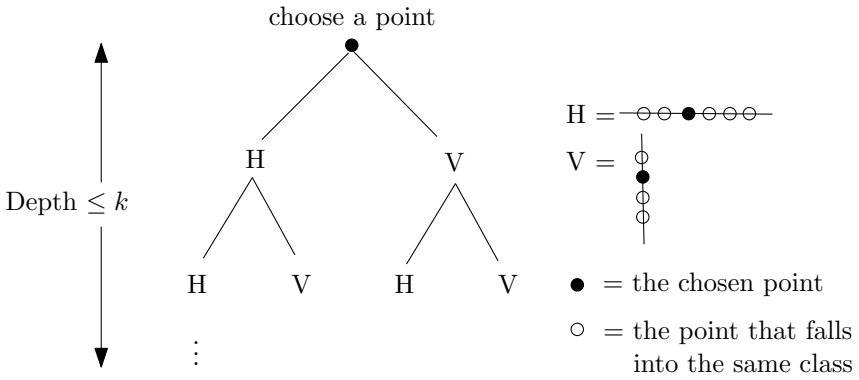


Figure 6.4: A search tree for computing candidate sets of k lines.

In every node of the tree, the set L is a partial cover of S and, in some leaves, it will be a k-cover. The root of the tree is initialized with the set L assigned the empty set. In each internal node of the tree, we choose a point $p \in S \setminus cover(L)$ and we explore the two possibilities of enlarging L by analyzing the situation at p. Note that if the given instance were a YES-instance, every point in S is covered by a horizontal line (H-line) or a vertical line (V-line) (or both, if a bend or crossing is placed at the point). The point p is marked as covered with an H-line or V-line, depending on the branch of the tree. Also, the chosen line is added to L. The points that fall into the same line with p are also marked as covered, so that we do not consider these points again in the next recursive call.

We keep track of how many vertical and horizontal assignments have been made and we emphasize that we do not assign more than $k/2$ horizontal lines and also no more than $k/2$ vertical lines. Each branch of the tree stops when the upper bound is reached or when every point is marked as covered. At the

leaves of the tree, we have at most k lines that cover the set of n points, or exactly k lines that do not cover; in this case, we know the candidate set of lines cannot be completed into a covering tour. Therefore, the depth of the tree is at most k. This matches the pattern of a bounded-search-tree. The depth of the tree is bounded by the parameter and the fan-out is a constant (in this case the constant is 2).

Let L be the set of lines that cover the set S at a leaf of the tree where $|L| \leq k$. We only consider tours where every line-segment covers at least one point, because for every tour T that covers S, there is an equivalent tour T' where every line-segment of the tour covers at least one point[3]. If T is a tour, we let $lines(T)$ be the set of lines used by the line-segments in T. Each line in $lines(T)$ covers at least one point and $lines(T)$ is a cover with k or fewer orthogonal lines. These observations allow us to state the following lemma.

Lemma 27. *If the instance has a tour T with k or fewer bends, then there is a leaf of the tree at the end of phase one where the cover made by the lines in L is consistent with the cover by $lines(T)$ (that is, $L \subseteq lines(T)$).*

Proof. The algorithm of the first phase picks up a point p at each node. This point must be covered by $lines(T)$ with a horizontal or a vertical line. Therefore, there is a child that agrees with $lines(T)$ for this node. Traveling the path from the root of the tree down the child that agrees with $lines(T)$, we reach a leaf where each line in L must be in $lines(T)$. □

In the second phase, we investigate if those leaves of the first phase that cover the n points can result in a tour with the constraints of the variants.

6.4.1 Tours where One Line-Segment Covers All the Points on the Same Line

Here, we require that every line-segment s of a tour T must cover all the points in S on the line that includes s. In this special case of the Rectilinear k-Bends Traveling Salesman Problem, each candidate set L of lines at a leaf of the tree (recall Figure 6.4) results in a candidate set of line-segments, because

[3]If a line-segment in the tour covers no points, it can be translated in parallel until it is placed over one point.

we can simply connect the extreme points in $cover(l)$ for each line $l \in L$ to get the candidate line-segments. We can explore exhaustively if these candidate line-segments result in a tour. In the worst case, we have k line-segments from the first phase. These k line-segments are organized into $k!$ orders in a possible tour. In each of these orders, we can connect the line-segments (in the order chosen) one by one to form a tour. There are at most four ways to connect the line-segment l_i to the consecutive line-segment $l_{(i \bmod k)+1}$ for $i \in \{1, 2, \ldots, k\}$. Let a and b be the extreme points of l_i, while c and d are the extreme points of $l_{(i \bmod k)+1}$. We can connect these two line-segments as ac, ad, bc or bd (see Figure 6.5).

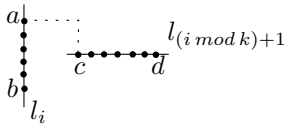


Figure 6.5: One of four possible ways of joining two consecutive lines.

This means a total of $(k!)(4^k)$ tests. In some cases, when we connect two line-segments together, the extension of these two line-segments may be enough to make a turn and, therefore, no additional line-segment is required. In some cases, it requires two additional line-segments, as shown in Figure 6.5. These extra line-segments cover no points, but they can be added in constant time when constructing the tour. Note that if the total number of line-segments in the final tour exceeds k, we simply answer NO (Lemma 22).

We now analyze the total running time of our algorithm. It is obvious that the search tree for the first phase in Figure 6.4 has at most $O(2^k)$ leaves and $O(2^{k-1})$ internal nodes. However, not every branch in the tree has to be explored. We explore only those branches that have an equal number of H-lines and V-lines. This is equivalent to choosing (among the k levels) a subset of size $k/2$ where to place the H-lines (or V-lines). Another way to recognize this is that each path from the root to a leaf in Figure 6.4 is a word of length at most k with exactly $k/2$ symbols H and $k/2$ symbols V. Based on this analysis ,we conclude that the number of different paths from the root to a leaf (and therefore, leaves) is given by

$$\left(\begin{array}{c} k \\ k/2 \end{array} \right) = \frac{k!}{(k/2)!(k/2)!} = O(2^k/\sqrt{k})$$

using Stirling's approximation. The work at each internal node is to choose a point, record the line and mark the associated points that are covered by that line. The dominant work is the computation at the leaves of the tree. Here we perform the tests to cover all tours that require time bounded by

$$\begin{aligned} O(\frac{2^k}{\sqrt{k}}(k!)(4^k)n) &= O(\frac{8^k}{\sqrt{k}}\sqrt{2\pi k}(k/e)^k n) \\ &= O((2.95k)^k n). \end{aligned}$$

The time complexity is exponential in the parameter, but linear in the size of the input. This gives the following result.

Theorem 13. *The* Rectilinear k-Bends Traveling Salesman Problem*, where one line-segment covers all the points on the same line, is FPT and can be solved in $O((2.95k)^k n)$ time.*

6.4.2 Tours where the Same Line-Segment Orientation Covers Points on the Same Line

In this special case, a point that lies on a line hosting a horizontal-segment of the tour must be covered by a horizontal line-segment of the tour (possibly another horizontal-line segment, and possibly also a vertical line-segment). The trick is that it cannot be covered by only a vertical line-segment. The symmetric condition holds for points in a line hosting a vertical line-segment. We call this the *no distinct type of line-segment condition.*

In the first phase, a leaf that may hold a YES-instance has a candidate set L of no more than k lines and $L \subseteq lines(T)$. We expand this set of candidate lines as follows. For every vertical line $l \in L$ we add two horizontal lines h_{max} and h_{min}. The line h_{max} is the horizontal line through the point $p \in cover(l)$ with the largest y coordinate, while h_{min} is the horizontal line through the point $p \in cover(l)$ with the smallest y coordinate. Symmetrically, for every horizontal line $l \in L$, we add two lines v_{max} and v_{min} where v_{max} passes through $p \in l$ with the largest x-coordinate and v_{min} passes through the point $p \in l$ with the smallest x-coordinate. Note that L with all these additional lines has size linear in k. In what follows, we call the set L at a leaf with these additional lines, the set C of lines. Our aim is the next result.

Lemma 28. *If the instance has a tour T with k or fewer bends (and meeting the no distinct type of line-segment condition), then there is a leaf of the tree at the end of phase one where a tour can be built with k or fewer bends and all the bends are at intersections of the lines in C.*

The proof shows that if we have a YES-instance, we can transform the witness tour T to a tour T' with the same numbers of bends, where every line-segment covers at least one point and $lines(T') \subseteq C$. From this, it follows that the intersections of all lines in C host the possible positions for the bends in T'.

The argument shows that every time we have a line-segment $\overline{pq}$ in a tour with its hosting line in $lines(T) \setminus C$, we can find a covering tour T' with the same bends and leading to the same leaf in phase one, the line-segment $\overline{pq}$ is not used and more line-segments in T' have their hosting lines in C. Consider a line-segment $\overline{pq}$ in a tour T that is a witness that the leaf is a YES-instance, but the line l hosting $\overline{pq}$ in the tour is such that $l \notin C$ (that is, $l \in lines(T) \setminus C$). Without loss of generality, assume $\overline{pq}$ is horizontal (if $\overline{pq}$ were vertical, we rotate S and the entire discussion by 90°). Also, we can assume that $\overline{pq}$ covers at least one point in S and T has minimal number of bends. Let l_1 be the line-segment in the tour before $\overline{pq}$ and l_2 the line-segment in the tour after $\overline{pq}$.

Claim 5. *For all $p' \in S \cap \overline{pq} \setminus \{p, q\}$, there is a vertical line $l \in L$ (and thus $l \in C$) such that $p' \in cover(l)$.*

Proof. Let $p' \in S \cap \overline{pq} \setminus \{p, q\}$. Then p' is covered by a vertical line in L, because L covers S, and if p' was covered by a horizontal line, then the line hosting $\overline{pq}$ would be in L and $L \subseteq C$. This contradicts that the line hosting $\overline{pq}$ is not in C. If $\emptyset = S \cap \overline{pq} \setminus \{p, q\}$ the claim is vacuously true. □

The points in $S \cap \overline{pq} \setminus \{p, q\}$ are covered in T by vertical line-segments (if any point p' was covered only by $\overline{pq}$ and not a vertical segment in T through the vertical line at p', then T' would not satisfy the no distinct type of line-segment condition).

We will now distinguish two cases (refer to Figure 6.6). In the first case, the tour T makes a U-return shape reversing direction, while in the second case, the tour makes a zig-zag shape and continues in the same direction. The bends at p and q of the line-segment $\overline{pq}$ make a U-return if they have one common label.

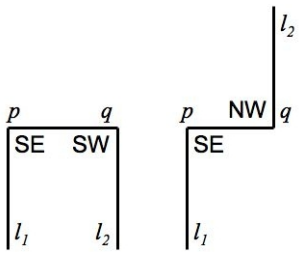


Figure 6.6: The line $l \notin L$ hosts $\overline{pq}$ from T.

The bends at p and q make a zig-zag of the line-segment $\overline{pq}$ if they have no common label. In this case the turns are SE at p and NW at q (with no letter label in common) or NE at p and SW at q (with no letter label in common). Without loss of generality, note also that a horizontal reflection along $\overline{pq}$ can be made, so the other subcases can be ignored and we can assume the cases are as the two drawings of Figure 6.6.

Case 1: In this case, we obtain the corresponding equivalent tour by shifting $\overline{pq}$ vertically. This is possible, because all points in $S \cap \overline{pq} \setminus \{p, q\}$ are already covered by other vertical lines of T. In fact, if there is any point in $S \cap \{(x, y) | p_x \leq x \leq q_x \wedge y \geq p_y\}$[4], by shifting $\overline{pq}$ vertically (up) we can (without increasing the number of bends) achieve an overlap with h_{max} for some vertical line in L. In fact, the set $S \cap \{(x, y) | p_x \leq x \leq q_x \wedge y \geq p_y\}$ is not empty because $\overline{pq}$ has at least one point covered by a vertical line in L (L is a cover and we assumed the horizontal line hosting $\overline{pq}$ is not in L). It is important to note that our tour T' may self-intersect, but that is not a constraint for the problem.

Case 2: This setting has the following subcases. First, we show that if $l_2 \notin L$, we can also change to a tour T' where now the set $lines(T') \setminus C$ is smaller. Second, the symmetric argument shows that we can do this also if $l_1 \notin L$. Finally, the case left is when both $l_1, l_2 \in L$.

Subcase 2.a: If $l_2 \notin L$, then $q \notin S$. Because the line-segment $\overline{qq'}$ hosted by l_2 must cover one point $q' \in S$ and L is a cover, the point q' is covered by a horizontal line $l_3 \in L$. Figure 6.7 (a) shows that the tour cannot have a SE turn

[4]Here p_x is the x-coordinate of point x, thus $S \cap \{(x, y) | p_x \leq x \leq q_x \wedge y \geq p_y\}$ is all points in S above or on $\overline{pq}$.

at q', because then we can make a tour with two fewer bends, thus contradicting the minimality of the witness tour (neither $\overline{qq'}$ nor $\overline{pq}$ are needed).

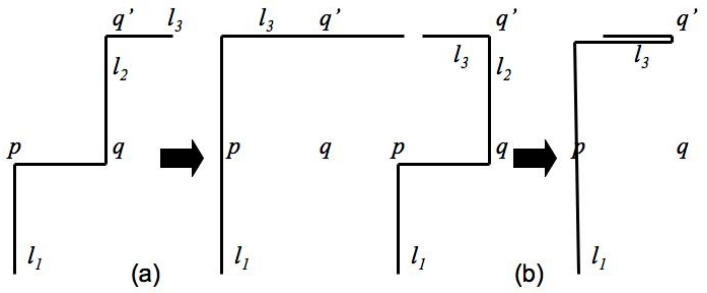


Figure 6.7: The subcases if $l_2 \notin L$ can be converted and eliminate l_2.

Thus, the turn at q' must be a SW turn. Figure 6.7 (b) shows that a tour with a 180°-bend has equivalent coverage and l_2 and the line hosting $\overline{pq}$ are not needed. This makes $lines(T') \setminus C$ smaller by two lines.

Subcase 2.b: If $l_1 \notin L$, then $p \notin S$. The arguments is analogous to the previous case.

Subcase 2.c: Now we consider the case where both $l_1, l_2 \in L$. In this case, we make a cut-and-join operation to obtain a new tour that now appears as in Case 1 (that is, we have a U-turn and not a zigzag). Note that in this case there must be points in S covered by l_2 below or including q. Otherwise, the line h_{min} from l_2 can be made to coincide with $\overline{pq}$ and the case is proved. Similarly, there must be points in S covered by l_1 above p, otherwise $\overline{pq}$ can be made to coincide with h_{max} for l_1. Note also that there must be points in S above q covered by l_2, otherwise shifting $\overline{pq}$ can be made to coincide with h_{max} for l_2. Also, by an analogous argument, there must be points in S below p covered by l_1. The tour T must use a vertical line-segment to cover the points in l_2 below q,[5] because the tour complies with the no distinct type of line-segment condition. Assume that the tour T is traveled in the direction l_1, then $\overline{pq}$, then l_2, and let $\overline{p'q'}$ be a segment of this tour under q hosted by l_2, with $p'_y > q'_y$. In the chosen traversal of the tour T, $\overline{p'q'}$ may be traveled from p' to q' or from q' to p' (refer to Figure 6.8).

[5]An analogous argument happens if the tour T uses a vertical line-segment to cover the points above p covered by l_1.

If T travels from p' to q', we build a new tour that travels from q to p' and continues along T but in reverse, this ensures we will meet l_2 and then q again, but we now continue until q'. From here, we continue along T in the same direction, which ensures we reach p. This new tour now makes a U-turn at $\overline{pq}$ and has the same set L as T and the same number of bends. For the case that T travels from q' to p', the new tour travels from $\overline{pq}$ to q', then, along T in reverse order, which guarantees arriving at q from l_2. We continue until p' and then in reverse order in T until we reach p again. Once again, this conversion reduces this case to Case 1.

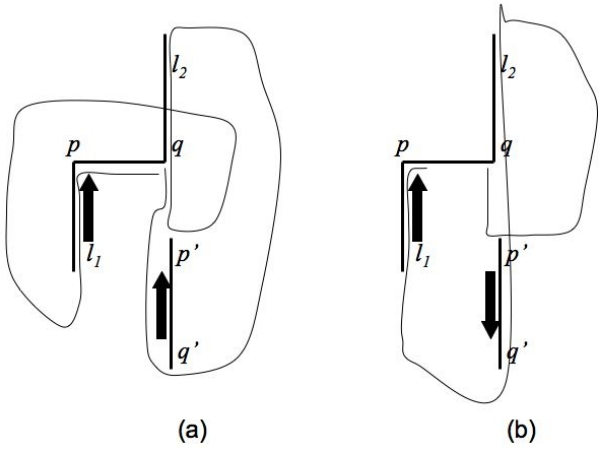


Figure 6.8: The tour T is changed so there is no zig-zag.

We can now carry out the following procedure at each leaf. The number of intersections between any pair of lines in C is bounded by $O(k^2)$. We enumerate all words with repetitions of length k over the alphabet of intersections. We require repetitions, because we need to consider tours with 180° bends. A word like this encodes where the bends of a tour will be. For each of these words, we generate each of the possible placements given by the four types of 90° bends (NE, NW, SE, SW) at each intersection. A placement of bends can be tested (in polynomial time) if it can, in fact, be completed into a tour and whether such tour covers S. The running time is bounded by

$$\begin{aligned} O(\frac{2^k}{\sqrt{k}}\frac{(2k^2)^k}{k!}(4^k)kn) &= O(\frac{16^k k^{2k}}{\sqrt{k}\sqrt{2\pi k}(k/e)^k}kn) \\ &= O((43.5k)^k n). \end{aligned}$$

This has clearly exponential complexity in k, but the overall algorithm has polynomial complexity in n and demonstrates the following result.

Theorem 14. *Under the constraint that a tour satisfies the no distinct type of line-segment condition, the* Rectilinear k-Bends Traveling Salesman Problem *is FPT and can be solved in* $O((43.5k)^k n)$ *time.*

6.5 Chapter Summary

We have presented three FPT-algorithms for different variants of the Rectilinear k-Bends Traveling Salesman Problem. We summarize these results in Table 6.1.

Table 6.1: FPT-algorithms for finding a rectilinear tour with at most k bends.

Rectilinear Tour and Its Conditions	Time Complexities
general case (without constraints)	$O(kn^2 + k^{4k}n)$
same line-segment orientation cover points on the same line	$O((43.5k)^k n)$
one line-segment covers all the points on the same line	$O((2.95k)^k n)$

It is apparent that the complexity of the algorithms is slightly better as more constraints are placed on the solution. Also, as we argued with an example, a solution for the general case may not be a solution for either of the constrained cases, and a solution of the constrained cases may require more bends although it is a solution for the general case. Therefore, the fact that one variant is FPT does not imply the other variant is FPT. The kernelization approach required additional analysis of the witness structure and could be considered to follow the ideas of M. Fellows on extremal structure. The bounded-search-tree technique enabled us to obtain a significant improvement for computing candidate sets of k lines that cover all the n points (this is carried out in $O(2^k/\sqrt{k})$ time).

While we have discussed the decision-version of the problem, it is not hard to adapt to an FPT-algorithm for the optimization version. For example, we can use an optimization-version FPT-algorithm to find a k'-cover where k' is the number of covering lines and $O(k')$ is a bound for k where k is the minimum number of bends. In fact, it has been shown [SW01] that $k \leq 2k' + 2$. That is,

we can approximate the minimum number of bends in FPT-time with respect to the sought minimum (or use a polynomial-time approximation algorithm [SW01] to obtain the approximate minimum). Then, the decision problem can be used to find the exact value of the minimum in FPT-time (using a binary search, for example).

We note that to further improve the time complexity of the algorithms we may need further algorithmic engineering. However, even the vertex cover was first found to be in FPT by a totally impractical argument using the Minor Theorem [FL92] and steady progress in the parameterized complexity field has now delivered practical implementations. Our aim for this chapter is to establish that the problems we presented are in the class FPT.

Chapter 7

Applications

This chapter discusses the covering problems from application's point of view. We provide a case study of Cyclone Nargis in Burma where the mission was to air-drop relief supplies to the disaster area. We give the simulation result and suggest a potential research that arises from our work.

7.1 Introduction

One of the challenges facing mathematics researchers is showing that there is a place for their results in the real world. We have already mentioned some applications of our results in the introduction of each geometric problem. The intention in this chapter is to demonstrate further examples of those applications.

We choose the Line Cover algorithms to demonstrate practical examples, because we have implemented and evaluated these algorithms in Chapter 2. We consider a scenario where the input data comes from a real-world application. Therefore, we provide a case study of Cyclone Nargis and a simulation of air-dropping the relief supplies in the cyclone-affected area.

On May 2, 2008, Cyclone Nargis made landfall in Burma (officially known as Myanmar). The cyclone caused the worst natural disaster in the history of Burma. About 2.4 million people were affected by Nargis, including 75% of the people in the Ayeyarwady Division (also known as the Irrawaddy Delta), which includes the townships of Bogale, Labutta, Ngaputaw, Dedaye, Pyapon, Kyaiklat and Mawlamyinegyun, and about 680,000 people in severely affected

areas of Yangon [Cen08]. The Labutta Township alone was reported to have 80,000 dead, with dozens of the 63 villages surrounding the town being wiped out [The08a]. After the cyclone had moved ashore in the Ayeyarwady Division, it passed near the city of Yangon (Rangoon) and gradually weakened as it turned to the northeast toward the border of Burma and Thailand.

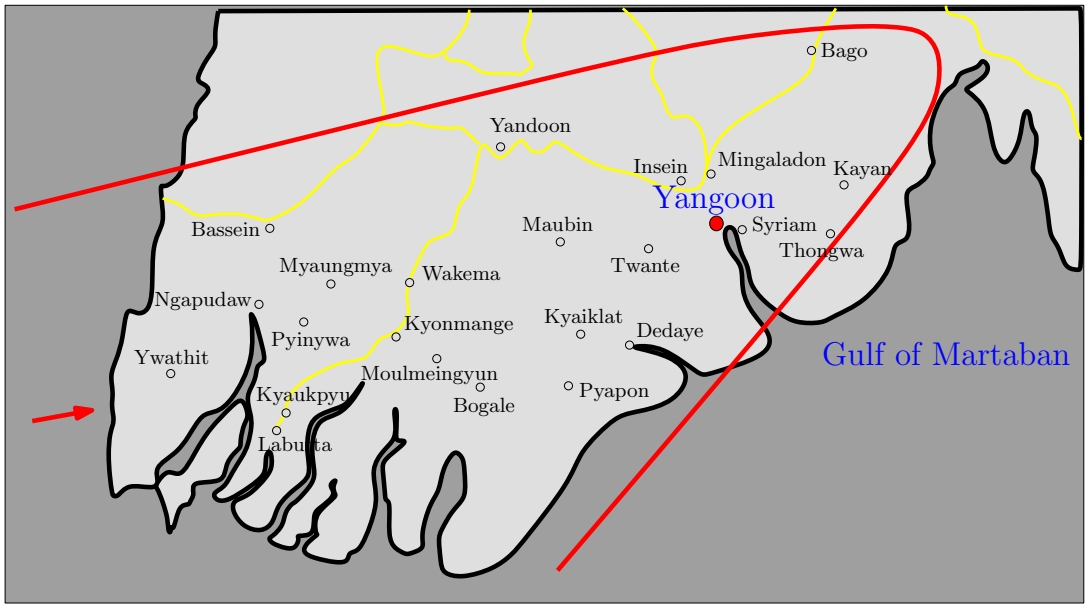


Figure 7.1: Disaster area in Ayeyarwady Division and Yangon Division (Google-Map view [Goo09]).

The damage was estimated at over US$10 billion [The08c]. At least 134,000 people were killed or left missing, and more than two million people were living in poor conditions, most of them without shelter, enough food, drinking water or medical care [The08c]. Relief efforts were hampered for political reasons. The European Union nations issued a warning that the junta could be committing a crime against humanity by blocking aid intended for the survivors. India, the country that maintains close relations with Burma, sent two Naval ships carrying essential relief material supplies [The08b]. They were the first ships to arrive in Burma. Two Indian Air Force (IAF) planes also carried relief supplies under the *Operation Sahayata* [The08b].

7.2 Cyclone Nargis: A Case Study

Suppose that the Operation Sahayata previously mentioned must air-drop the relief supplies using the two IAF's planes. Under this scenario, what are the routes for the planes to operate, so that all the towns in the worst-affected area are covered? In flight, the planes should also minimize the number of routes and the routes should be operated in a straight-line structure. That is, we are required to cover all the towns with the minimum number of lines.

The main objective of this scenario is to save fuel. Nevertheless, it also shrinks the rescuing time since the planes can fly at much faster speed when traveling in a straight line. Out of respect for the victims of the disaster, we emphasize that this scenario is a simulated exercise only. In real circumstances, one would consider saving lives as the number one priority.

We simulate the above scenario on 24 Townships in Burma, namely, Ywathit, Ngapudaw, Bassein, Labutta, Kyaukpyu, Pyinywa, Myaungmya, Kyonmange, Wakema, Moulmeingyun, Bogale, Yandoon, Maubin, Kyaiklat, Dedaye, Pyapon, Twante, Insein, Mingaladon, Yangon, Syriam, Bago, Kayan, and Thongwa (see Figure 7.1). The coordinates of these townships are based on Google-Map data [Goo09]. We use the algorithm Binary Search in Chapter 2 to find the minimum line cover. According to our simulation result, we have eight covering lines. Therefore, we suggest the following routes for the pilots.

Route 1: Ywathit, Ngapudaw, Yandoon

Route 2: Bassein, Myaungmya, Kyonmange

Route 3: Pyinywa, Wakema, Insein

Route 4: Bogale, Maubin

Route 5: Kyaukpyu, Syriam, Kayan

Route 6: Labutta, Moulmeingyun, Yangon

Route 7: Kyaiklat, Twante, Mingaladon, Bago

Route 8: Pyapon, Dedaye, Thongwa

Since there are two planes in the scenario, the first plane may take Route 1, Route 2, Route 3 and Route 4 while the second plane may take the remaining ones. For the graphical view of these routes, we refer to Figure 7.2.

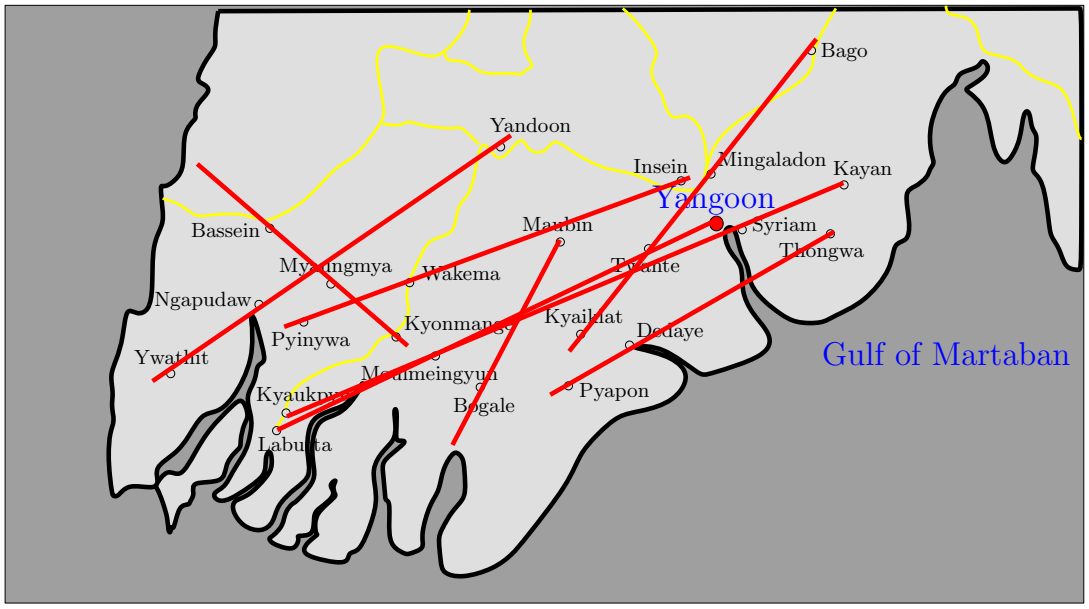


Figure 7.2: Simulation result exhibits eight covering lines for $n = 24$ (Google-Map view [Goo09]).

7.3 Discussion

In the real operation, the wind speed, the distance and other factors would be taken into consideration. Nevertheless, the information obtained from the above simulation provides a quick overview of the situation and assists in the decision-making that may not follow this information entirely. Our Line Cover algorithm accepts the input points as the x and y-coordinates; therefore, the accuracy of the longitude and latitude of the points also influences the outcome. In our case, we approximate the coordinates of the towns at the centre of the town. We preprocess the input points, such that the points that deviate slightly from a line, are placed on a line. The simulation above allows the distance of at most 1 km where points can be deviated from a line. This distance can be varied, depending on our applications.

There are several potential research points that arise from our work. We suggest some of this research. First, we can look at the closely-related problem of covering points with strips of a given width. For example, the strips could be a long rectangular shape with 500 metres wide and the problem is changed to cover the towns using the minimum number of strips with a specified width. Second, we may also take the distance into consideration. That is, we minimize both the distance and the number of strips. Third, a covering path problem is also interesting. We might travel from point A to point B and cover a set of points in between. Finally, if k objects (for instance, planes, helicopters) are given, then what would be the k covering paths that minimize the total distance and the number of strips? If the objective is to save lives, one might consider minimizing the average time for an individual to receive aid, or minimizing the maximum time to receive aid, one might consider the population distribution, zig-zag flight paths and so on. We leave all of these questions as open problems.

7.4 Other Applications

While considering the case study of air-dropping relief supplies in the cyclone-affected areas, we find that there are other applications worth considering. For example, the routes for a boat to visit oil pumping stations in the ocean, a helicopter to water-bomb a bushfire, a robot to collect golf balls etc.

In particular, there is one application of a covering problem worth mentioning. This research [SCCN05] applies the path-covering problem of a mobile robot to the real-world aircraft inspection that usually costs the airline industry a great deal of money each year. Most of the commercial aircraft applications use multiple rivets to hold together multi-layer skins; therefore, the fatigue stress causes, fatigue cracks that can be induced in the vicinity of the rivet holes, these cracks should be detected as early as possible. Given a single robot and a set of n rivets in a 2D space, an algorithm [SCCN05] is developed to generate the required path that must satisfy the following, 1) the distance between any two consecutive rivets on the path should be less than a threshold distance, 2) the number of turns should be minimized and, 3) the overall distance should be minimized. This algorithm first identifies the minimum set of line segments partitioning all the rivets by solving the Set-Partition-Problem and then connects the line segments to minimize the overall distance. Therefore, in the first stage of solving

this problem, our LINE COVER algorithms can be used to compute the minimum set of line segments that cover all the rivets. However, we need to modify our algorithms slightly to satisfy the first constraint of the problem.

The bottom line is that our research has opened up new directions for researchers interested in the covering problems from the application's point of view.

Chapter 8

On the m-Convex-Polygons TSP

This chapter is intended to be an exploratory peek at how we can solve the special case of the Euclidean Traveling Salesman Problem. The idea is to apply some geometrical properties of the input objects to confine the explosion of the exponential term. We provide preliminary ideas and early results for the problem. Given a set of m disjointed convex polygons in 2D, even as degenerate as a single point, or a line segment (that is, 2 points), the problem is to find a tour that visits, at least once, all the vertices of the polygons and is as short as possible, but does not travel through the interior of the polygons. We refer to this problem as the m-Convex-Polygons TSP. It should be emphasized that this chapter is future research work.

8.1 Introduction

Papadimitriou [Pap77] proved that the Euclidean Traveling Salesman Problem is NP-complete. The m-Convex-Polygons TSP is a generalization of the Euclidean Traveling Salesman Problem and therefore NP-complete. There are applications where the setting of our problem is useful; for example, the situations where a car cannot travel through a lake; a helicopter is not flying through high rise buildings; a truck is not allowed to go inside a big city; or when avoiding an unwanted area, such as a desert or a volcanic area. This setting is also useful in VLSI design, because we occasionally have a certain area blocked by a rectangle or a convex polygon.

Abrahamson et al. [ASW05] studied the special case of the Euclidean Traveling Salesman Problem between two nested convex obstacles. They showed that tours that visit some vertices of the obstacle more than once results in shorter tours than the tours that visit each vertex exactly once. Therefore, we allow the tour in our problem to visit the vertices of the polygons more than once. We then extend this result to a simpler case where the tour visits the vertices of the polygons exactly once. However, the convex polygons that we study are not nested.

Sometimes, a problem has overlapping subproblems, that is, the same subproblems are used to solve many different larger problems. A dynamic programming algorithm solves every subproblem once and saves its answer in a look-up table to avoid the work of repeatedly recomputing the answer for the subproblem. The computation of the binomial coefficients is a very simple example of this kind [Nie06, p. 124]. Pascal's formula [GKP89, p. 158] states that,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all} \quad n \geq 1 \quad \text{and} \quad 1 \leq k \leq n.$$

Knowing that $\binom{i}{i} = \binom{i}{0} = 1$ for all $1 \leq i \leq n$, we can compute $\binom{2}{1} = \binom{1}{0} + \binom{1}{1} = 2$ by this $\binom{3}{1}$ and $\binom{3}{2}$, and so on.

Our algorithm uses the dynamic programming technique. We generalize the idea of Deĭneko et al. [DHOW04] by increasing the size of an array. Deĭneko et al. generalized Held and Karp's algorithm [HK62], and gave two FPT-algorithms for a special case of the Euclidean Traveling Salesman Problem where cities are points in the plane. The total number of input points is n and the number of inner points, that is, the points in the interior of the convex hull, is taken as a parameter k. Hence, the number of outer points, that is, the points on the boundary of the convex hull, is $n - k$. Their first algorithm runs in $O(k!kn)$ time. The second algorithm runs in $O(2^k k^2 n)$ time.

In this chapter, based on dynamic programming, we present an exact algorithm that runs in $O(2^{m(1.6d+2\log\alpha+2)}m^2)$ time where α is the maximum number of points on any of the m convex polygons. We believe that this result will bring us one step closer to finding the FPT-algorithms. The first parameterized problem we started to explore is the problem where we have one convex hull with many points on the convex position, but there are k convex-polygonal-obstacles

in the interior of the convex hull. We show that this problem is FPT. However, this solution seems restricted. Therefore, the future research is to improve on this result and also to focus more on problems that have practical applications.

8.2 Related Work

Held and Karp in 1962 solved exactly the Traveling Salesman Problem in time $O(n^2 2^n)$ [HK62] using the technique of dynamic programming. Although this is exponential, it is still the best-known guarantee on the running time of a general solution method for the problem and has survived over 40 years of challenges [ABCC06]. Bellman [Bel62] in the same year independently discovered the same result that also runs in $O(n^2 2^n)$ time. Deĭneko et al. [DHOW04] generalized Held and Karp's algorithm, and gave an FPT-algorithm for a special case of the TSP. This problem is restricted to the two-dimensional Euclidean plane where cities are points in the plane. The number of inner points (points in the interior of the convex hull) is taken as a parameter k, and the number of outer points (points on the boundary of the convex hull) is $n - k$, where n is the total number of input points. The algorithm runs in $O(2^k k^2 n)$ time.

A convex hull of a set S is the smallest convex set containing S [Ede87, dBvKOS00]. Intuitively, we can think of each point in S as being a nail sticking out from a board. The convex hull is the shape formed by a tight rubber band that surrounds all the nails. There are several methods to find the convex hull on n points. Graham's scan computes the convex hull in $O(n \log n)$ time while Jarvis's march computes it in $O(nh)$ time, where h is the number of points on the convex hull [CLRS01, Lev07]. Using geometrical properties of a convex hull not only helps to simplify the problem, but also confines the explosion of the exponential term of algorithms. For example, the process of peeling a planar point set, known as a computation of *convex layers*, is central in the study of robust estimators in statistics [Cha85, DP04]. There are several algorithms for computing convex layers efficiently. The algorithm of Chazelle [Cha85] finds convex layers of a set of n planar points in $O(n \log n)$ time, whereas Datta and Pal's algorithm [DP04] finds them in $O(n)$ time, using a multi-layered neural network model with $O(n)$ processors.

Using geometric objects, such as convex hull, points and lines, Deĭneko et al. also solved [DvDR94, DW96] several more variants of the Euclidean Traveling Salesman Problem, such as using convex hull and lines for solving the special case of the problem. This is a special case of the Euclidean Traveling Salesman Problem where $n - m$ of the points lie on the convex hull and m of the points lie on k line-segments in the interior of the hull. Deĭneko et al. solved this problem with the constraint that k lines are parallel line-segments or almost parallel line-segments.

García [GT97] studied the Traveling Salesman Problem of two nested convex polygons. The problem where m points are on one convex polygon M, and n points are on another convex polygon N, inside M, is polynomially solved in $O(m^3n^3)$ time. García also gave the number of possible orders of points on the second convex hull N (with n points) of a simple polygon, this is $O(5^n)$ [GT95].

Abrahamson et al. [ASW05] studied the two nested convex obstacles TSP. This problem is similar to the problem of García [GT97]. However, the points are the vertices of a convex polygon M and a convex obstacle N located completely in the interior of M. Thus, a tour can travel through the interior of M but not of N. By transforming the problem to $O(m)$ shortest-paths computations in an appropriately defined digraph, they gave an $O(m^2 \log m + mn)$ time, where m and n represent the number of vertices in M and N, respectively. Their approach is a generalization of the algorithm [DvDR94] that solved the convex-hull-and-line traveling salesman problem.

8.3 Fundamental Lemmas

In this section, we prove the fundamental lemmas for the m-Convex-Polygons TSP. We also define the properties of backtracking in the tours that visit the vertices of the obstacle more than once. We first define the problem formally as follows:

Instance:	A set of m convex polygons in 2D.
Question:	Find a tour that visits at least once all the vertices of the polygons and is as short as possible, but does not travel through the interior of the polygons.

These convex polygons are disjoint (even as degenerate as a single point, or a line segment, that is, two points). They are not nested and they are not overlapped. Since we cannot not travel through the interior of the polygons, we will sometimes refer to these polygons as convex-polygonal-obstacles.

We define S to be a set of n points in 2D, which are the points in the convex position of the convex-polygonal-obstacles combined. Let us use the notation $H_i(S)$, the i-th convex polygon of the set S for $i \in \{1, \ldots, m\}$ where $H_1(S)$ is the left-most convex polygon or the bottom one if there is a tie. Then $H(S) = H_1(S) \cup H_2(S) \cup \ldots \cup H_m(S)$, and we have $H_m(S)$ the right-most convex polygon or the top one if there is a tie.

Now, we can label the points in S clockwise around the first convex polygon; this would be $v_1^1, v_2^1, \ldots, v_{h_1}^1$ where $|H_1| = h_1$. Note that the superscript refers to the index of the polygon, and the subscript is the sequence number in that polygon. Then, we can go to the next polygon and, also in clockwise order, we label the points $v_1^2, v_2^2, \ldots, v_{h_2}^2$ where $|H_2| = h_2$ and so on until the last convex polygon is labelled in clockwise order $v_1^m, v_2^m, \ldots, v_{h_m}^m$ where $|H_m| = h_m$.

Proposition 1. *In any triangle ABC, any strictly monotonic path with respect to the segment AB from A to B and inside ABC has a length shorter than $|AC| + |CB|$ [ASW05].*

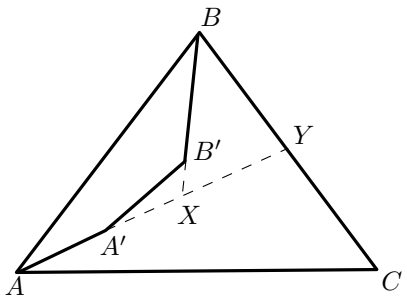
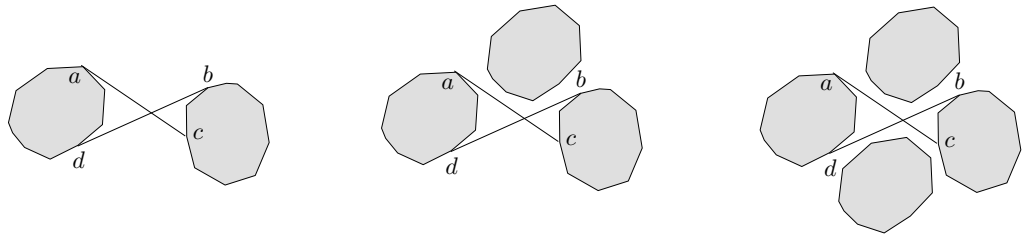


Figure 8.1: A strictly monotonic path between A and B in the triangle ABC.

The proof [ASW05] is based on the triangle inequality. For example, in Figure 8.1, we have $|AC| + |CB| > |AY| + |YB| > |AX| + |XB| > |AA'| + |A'B'| + |B'B|$.

Lemma 29. *A shortest tour T has intersections only on the vertices of $H_i(S)$.*



Case 1 Case 2 Case 3

Figure 8.2: Cases where intersections occur in the region outside $H_i(S)$.

Proof. The region inside $H_i(S)$ is prohibited, because it is an obstacle. Assume by contradiction that the shortest tour T has an intersection in the region outside $H_i(S)$. Consider the quadrilateral $abcd$ in Figure 8.2. The intersection in T occurs in three cases, as follows, 1) the sides of the quadrilateral $abcd$ are not pierced, that is, ab and cd are lines of sight, 2) one side of the quadrilateral $abcd$ is pierced and, 3) two sides of the quadrilateral $abcd$ are pierced. In each case, we can show that T is not the shortest tour, because it is always possible to obtain a shorter tour, T'. In Case 1, T' is obtained by replacing the intersecting line-segments with ab and cd. In Case 2 and Case 3, we can travel along the polygonal edges of the obstacle (Proposition 1) that is a strictly monotonic path between a and b, and c and d, to obtain the shorter tour T'. In these three cases, we can ignore a subcase where $a = d$ and $b = c$, that is, the two line-segments ac and bd are overlapped, because the tour that uses these two line-segments is not an optimal tour. Let a' be a point in T before a, such that the tour visits a' then a then b then back to a. By Proposition 1, $|a'a| + |ab| + |ba| > |a'b| + |ba|$. Therefore, a tour T' that uses the segments $a'b$ and ba results in a shorter tour than T that uses the segments ab and ba. □

Consider a shortest tour $\{v_1^1, v_2^1, v_3^1, v_4^1, v_3^1, v_1^2, v_4^2, v_1^2, v_2^2, v_3^2, v_1^1\}$ in Figure 8.3. Note that this tour contains two retracting paths. We will refer to such duplicate traversals as backtracking. In particular, we will refer to $\{v_4^2, v_1^2\}$ as *Type-A backtracking* and $\{v_4^1, v_3^1\}$ as *Type-B backtracking*.

Definition 20. *Type-A backtracking is when a tour travels to a point when a backtracking begins, it then travels in an opposite direction with respect to the convex polygon's cyclic order, but when it retraces, it follows the convex polygon's cyclic order, whereas Type-B backtracking follows the convex polygon's cyclic*

order but retraces in an opposite direction with respect to the convex polygon's cyclic order.

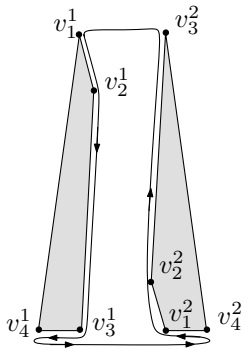


Figure 8.3: A shortest tour with two portions of backtracking.

Each convex polygon can have several portions as backtracking that is Type-A or Type-B or both types as shown in Figure 8.4.

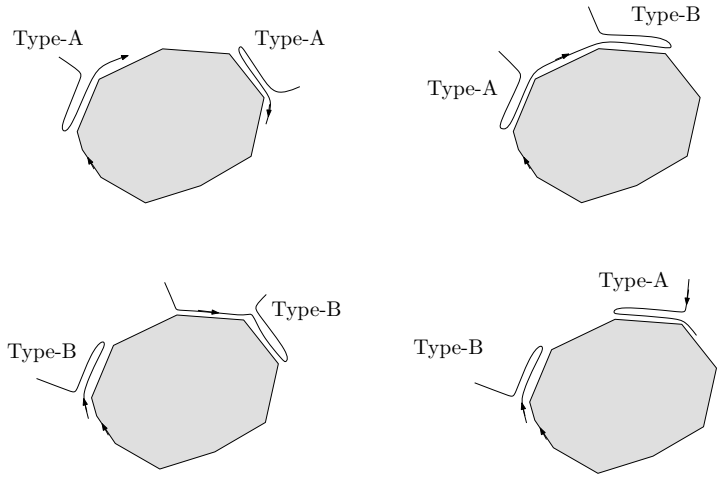


Figure 8.4: Convex polygons can have Type-A backtracking, Type-B backtracking or both.

Lemma 30. *Other than backtracking, a shortest tour visits the vertices of $H_i(S)$ in cyclic order.*

Proof. Suppose a shortest tour T contains a clockwise portion from point a to point b on $H_i(S)$ and an anti-clockwise portion from point d to point c. One of these portions creates an intersection when connecting to the final shortest tour, because there can only be a single cyclic orientation in the final tour. This

intersection contradicts Lemma 29. To avoid a violation of Lemma 29, one of the portions may retrace along the polygonal edges, but this also contradicts the assumption, because backtracking is not taken into consideration here. □

Lemma 31. *The number of possible orders of points on any convex-polygonal-obstacle with α points is 2α if backtracking is not permitted.*

Proof. There are α possible starting points for the cyclic order and two possible directions, clockwise and anti-clockwise. □

Lemma 32. *The number of possible orders of points on any convex-polygonal-obstacle with α points is $(2\alpha)(3^\alpha)$ when backtracking in the tour is permitted.*

Proof. Let p_j be a point on a convex-polygonal-obstacle where $1 \leq j \leq \alpha$. Consider a search tree with three branches; p_j is not part of the backtracking, p_j is part of the Type-A backtracking, and p_j is part of the Type-B backtracking. The depth of the tree is α, because we mark the α points with one of three possibilities, let us call them "$-$", "A", "B" for short. Hence, the search tree has 3^α leaves. We note that the tree itself is not to establish the orders of α points, but at the leaves of the tree, we obtain the backtracking patterns, because we know which types of backtracking each of the α points follows. The task at the leaves is then to generate all possible orders of α points from the patterns obtained from the tree . We now describe how we generate these orders. As an example, consider a convex polygon with five points. Label these points, Point 1, Point 2, Point 3, Point 4 and Point 5. Suppose that a search tree examines these points in order from Point 1 to Point 5. Let us look at the following branches of the tree (from the root of the tree down to the child) with the patterns "$AABB-$", "$BBAA-$", "$AA-AA$", and "$BB-BB$", for example. Since at the leaves of the tree, the backtracking patterns of all the points are assigned, we can generate the orders, according to the definitions of backtrackings Type-A and Type-B, as follows:

$$AABB- \implies \underline{2\,1}\,1\,2\,3\,4\,\underline{4\,3}\,5$$

$$BBAA- \implies 1\,2\,\underline{2\,1}\,\underline{4\,3}\,3\,4\,5$$

$$AA-AA \implies \underline{2\,1}\,1\,2\,3\,\underline{5\,4}\,4\,5$$

$$BB-BB \implies 1\,2\,\underline{2\,1}\,3\,4\,5\,\underline{5\,4}$$

Note that the numbers that are underlined above indicate the backtracking. We can remove the duplicate points in the above orders if the points are adjacent. Since there are α possible starting points in any of these orders and two possible directions, clockwise and anti-clockwise, we generate 2α orders in each leaf. Therefore, the number of possible orders of points on any convex-polygonal-obstacle with α points is $(2\alpha)(3^{\alpha})$. □

8.4 The Algorithm for the m Convex-Polygons TSP

Our goal is to find the FPT-algorithms for this problem or to show that the problem is very unlikely to be in FPT by proving the $W[1]$-hardness. If, indeed, the problem is not in the class FPT, then our plan is to look at the special case of the problem and explore other possibilities of the FPT-algorithms for that special case. In this section, we start the design of our algorithm for the m-Convex-Polygons TSP using dynamic programming. Note that our first algorithm presented here will not be in FPT. However, we will describe how this algorithm works so that the reader can understand the motivation behind our second algorithm (in the next section).

Our algorithm generalizes the algorithm of Deĭneko et al. [DHOW04] by expanding the array with more dimensions in dynamic programming and testing all possible orders of points on m-convex-polygonal-obstacles. We know from Lemma 32 that if we first construct a search tree, we obtain the number of possible orders of points on each convex polygon. For example, there are at most $(2h_1)(3^{h_1})$ orders on $H_1(S)$, and $(2h_2)(3^{h_2})$ orders on $H_2(S)$ etc. When taking the m polygons into the account, the total of these orders is $(2h_1)(3^{h_1}) \times \ldots \times (2h_m)(3^{h_m})$. The idea is to fix the order of each convex polygon and solve a shortest tour problem by dynamic programming. Then, repeat the process for all combinations of these orders and pick the solution that is optimal. We refer to a fixed-order on $H_1(S)$ as π_1, a fixed-order on $H_2(S)$, π_2, a fixed-order on $H_3(S)$, π_3, and so on until a fixed-order on $H_m(S)$, π_m.

Let us consider an $(m+1)$-dimensional array C, where the first m entries indicate how much we have advanced in the given order of each convex polygon

and the last entry indicates the polygon in which we wish to finish. Our problem is to find the cheapest way (the smallest C) of visiting each point in $H(S)$ at least once and then returning to the starting point. The array $C[s_1, s_2, \ldots, s_{m-1}, s_m, i]$ is the value of the shortest path that starts at $p_1^{\pi_1}$ and makes progress in the first convex polygon covering exactly the points that appear in a fixed-order π_1 until a stop index s_1. It then makes progress in the second convex polygon covering points in a fixed-order π_2 till s_2 (the path may go back and forth between the convex polygons, but it will cover exactly the points till s_2 of this second convex polygon) and so on until the i-th convex polygon. Note that the notation $p_{s_i}^{\pi_i}$ has the superscript that refers to a fixed-order π_i of the i-th convex polygon and the subscript is the index of points in π_i.

Formally, the $(m+1)$-dimensional array $C[s_1, s_2, \ldots, s_m, i]$, where $1 \leq s_1 \leq 2h_1$ and $0 \leq s_j \leq 2h_j$ for $j \in \{2, \ldots, m\}$ has the following indices:
The first index s_1 represents the index of points in π_1 of $H_1(S)$.
The second index s_2 represents the index of points in π_2 of $H_2(S)$.
The m-th index s_m represents the index of points in π_m of $H_m(S)$.
The $(m+1)$-th index i represents the i-th convex polygon.

The value $C[s_1, s_2, \ldots, s_m, i]$ represents the cost of visiting a set of points, that is, the length of a shortest path that satisfies the following conditions.

1. It starts at $p_1^{\pi_1}$ on $H_1(S)$.
2. It finishes at the i-th convex polygon.
3. It respects the fixed-orders $\pi_1, \pi_2, \ldots, \pi_m$.
4. It visits exactly the points in

$$\{p_1^{\pi_1}, \ldots, p_{s_1}^{\pi_1}\} \cup \{p_1^{\pi_2}, \ldots, p_{s_2}^{\pi_2}\} \cup \ldots \cup \{p_1^{\pi_m}, \ldots, p_{s_m}^{\pi_m}\}.$$

We set $\{p_1^{\pi_i}, \ldots, p_{s_i}^{\pi_i}\} = \emptyset$ if $s_i = 0$.

We assume that the visibility graph of n vertices are given. Therefore, we will not consider the cost of pre-computing this graph, although this can be simply done in $O(n^2 \log n)$ time [dBvKOS00]. We define $d(p_s, p_{s+1})$ as the distance from points $p_s \in S$ to $p_{s+1} \in S$. We set $d(p_s, p_{s+1}) = \infty$, if p_{s+1} is not visible from p_s.

The cost of a shortest tour is then computed as

$$\begin{aligned}\min\{\quad & C[f_1, f_2, \ldots, f_m, 1] + d(p^{\pi_1}_{f_1-1}, p^{\pi_1}_1),\\ & C[f_1, f_2, \ldots, f_m, 2] + d(p^{\pi_2}_{f_2}, p^{\pi_1}_1),\\ & \ldots,\\ & C[f_1, f_2, \ldots, f_m, m] + d(p^{\pi_m}_{f_m}, p^{\pi_1}_1)\}\end{aligned}$$

where f_i is the finish-index (the largest index) in the fixed-order π_i for $i \in \{1, \ldots, m\}$. We note that $f_i \leq 2h_i \leq 2\alpha$, where α is the maximum number of points on any convex polygon. The 2α comes from the fact that any generated order (Lemma 32) could have points repeated twice. Computing a shortest tour is, therefore, only a matter of filling up the array $C[s_1, s_2, \ldots, s_m, i]$ from the initial values and climbing all the way to the largest values of $s_1, s_2, \ldots, s_m$ and i. To establish this recurrence, we set the boundary cases as follows:

$C[1, 0, \ldots, 0, 1] = 0$.
$C[1, s_2, \ldots, s_m, 1] = \infty$ for $1 \leq s_j \leq f_j$ and $j \in \{2, \ldots, m\}$.
$C[s_1, s_2, \ldots, s_m, 2] = \infty$ if $s_2 = 0$.
$C[s_1, s_2, \ldots, s_m, 3] = \infty$ if $s_3 = 0$.
$C[s_1, s_2, \ldots, s_m, i] = \infty$ if $s_i = 0$.
$C[s_1, s_2, \ldots, s_m, m] = \infty$ if $s_m = 0$.

Consider three convex polygons as an example and suppose we want to visit the points of $\{p^{\pi_1}_1, \ldots, p^{\pi_1}_{s_1}\} \cup \{p^{\pi_2}_1, \ldots, p^{\pi_2}_{s_2}\} \cup \{p^{\pi_3}_1, \ldots, p^{\pi_3}_{s_3}\}$ and arrive at $p^{\pi_3}_{s_3}$ such that the chosen orders π_1, π_2 and π_3 are respected. There are three possible ways to get to this state.

1. We visit $\{p^{\pi_1}_1, \ldots, p^{\pi_1}_{s_1}\} \cup \{p^{\pi_2}_1, \ldots, p^{\pi_2}_{s_2}\} \cup \{p^{\pi_3}_1, \ldots, p^{\pi_3}_{s_3-1}\}$ to arrive at $p^{\pi_1}_{s_1}$ then move to $p^{\pi_3}_{s_3}$.

2. We visit $\{p^{\pi_1}_1, \ldots, p^{\pi_1}_{s_1}\} \cup \{p^{\pi_2}_1, \ldots, p^{\pi_2}_{s_2}\} \cup \{p^{\pi_3}_1, \ldots, p^{\pi_3}_{s_3-1}\}$ to arrive at $p^{\pi_2}_{s_2}$ then move to $p^{\pi_3}_{s_3}$.

3. We visit $\{p^{\pi_1}_1, \ldots, p^{\pi_1}_{s_1}\} \cup \{p^{\pi_2}_1, \ldots, p^{\pi_2}_{s_2}\} \cup \{p^{\pi_3}_1, \ldots, p^{\pi_3}_{s_3-1}\}$ to arrive at $p^{\pi_3}_{s_3-1}$ then move to $p^{\pi_3}_{s_3}$.

The same idea above applies when we have m convex polygons, that is, we have m possible states in which we could finish. The last index of the array controls

these finishing positions from $i = 1$ until $i = m$. The main part of the recurrence is, therefore:

$$\begin{aligned} C[s_1, s_2, \ldots, s_m, 1] = \min\{ \quad & C[s_1 - 1, s_2, \ldots, s_m, 1] + d(p^{\pi_1}_{s_1-1}, p^{\pi_1}_{s_1}), \\ & C[s_1 - 1, s_2, \ldots, s_m, 2] + d(p^{\pi_2}_{s_2}, p^{\pi_1}_{s_1}), \\ & \ldots, \\ & C[s_1 - 1, s_2, \ldots, s_m, m] + d(p^{\pi_m}_{s_m}, p^{\pi_1}_{s_1}) \end{aligned}$$

$$\begin{aligned} C[s_1, s_2, \ldots, s_m, 2] = \min\{ \quad & C[s_1, s_2 - 1, \ldots, s_m, 1] + d(p^{\pi_1}_{s_1}, p^{\pi_2}_{s_2}), \\ & C[s_1, s_2 - 1, \ldots, s_m, 2] + d(p^{\pi_2}_{s_2-1}, p^{\pi_2}_{s_2}), \\ & \ldots, \\ & C[s_1, s_2 - 1, \ldots, s_m, m] + d(p^{\pi_m}_{s_m}, p^{\pi_2}_{s_2}) \end{aligned}$$

When finishing at the m-th convex polygon, we have,

$$\begin{aligned} C[s_1, s_2, \ldots, s_m, m] = \min\{ \quad & C[s_1, s_2, \ldots, s_m - 1, 1] + d(p^{\pi_1}_{s_1}, p^{\pi_m}_{s_m}), \\ & C[s_1, s_2, \ldots, s_m - 1, 2] + d(p^{\pi_2}_{s_2}, p^{\pi_m}_{s_m}), \\ & \ldots, \\ & C[s_1, s_2, \ldots, s_m - 1, m] + d(p^{\pi_m}_{s_m-1}, p^{\pi_m}_{s_m}). \end{aligned}$$

Therefore, the general case of finishing at the i-th convex polygon is,

$$\begin{aligned} C[s_1, s_2, \ldots, s_m, i] = \min\{ \quad & C[s_1, s_2, \ldots, s_i - 1, \ldots, s_m, 1] + d(p^{\pi_1}_{s_1}, p^{\pi_i}_{s_i}), \\ & C[s_1, s_2, \ldots, s_i - 1, \ldots, s_m, 2] + d(p^{\pi_2}_{s_2}, p^{\pi_i}_{s_i}), \\ & \ldots, \\ & C[s_1, s_2, \ldots, s_i - 1, \ldots, s_m, i] + d(p^{\pi_i}_{s_i-1}, p^{\pi_i}_{s_i}) \\ & \ldots, \\ & C[s_1, s_2, \ldots, s_i - 1, \ldots, s_m, m] + d(p^{\pi_m}_{s_m}, p^{\pi_i}_{s_i}) \end{aligned}$$

where $1 \leq s_1 \leq f_1$ and $0 \leq s_j \leq f_j$ for $j \in \{2, \ldots, m\}$. Note that the distance from $p^{\pi_i}_{s_i}$ to $p^{\pi_i}_{s_i}$ (the distance to itself) is considered zero and the distance from $p^{\pi_i}_0$ to any other point is considered ∞.

Consider Figure 8.5 that shows how the array C is filled up from the initial conditions. Taking the input from Figure 8.3 as an illustration, we assume

that $\pi_1 = v_1^1, v_2^1, v_3^1, v_4^1, v_3^1$ and $\pi_2 = v_1^2, v_4^2, v_1^2, v_2^2, v_3^2$. First we fill in the initial conditions with zero and ∞, then we compute,

$$C[1,1,2] = \min\{C[1,0,1] + d(p_1^{\pi_1}, p_1^{\pi_2}), C[1,0,2] + d(p_0^{\pi_2}, p_1^{\pi_2})\}$$

and $C[1,2,2], C[1,3,2]$ until $C[1,5,2]$. Then we compute $C[2,0,1], C[2,1,1]$ until $C[2,5,1]$ and so on. Generally, computing $C[s_1, s_2, i+1]$ requires looking up the values of $C[s_1, s_2-1, i]$ and $C[s_1, s_2-1, i+1]$, while computing $C[s_1, s_2, i]$ requires looking up the values of $C[s_1-1, s_2, i]$ and $C[s_1-1, s_2, i+1]$. So, we fill up each row from left to right, top to bottom and repeat in this fashion (see Figure 8.5 for an illustration).

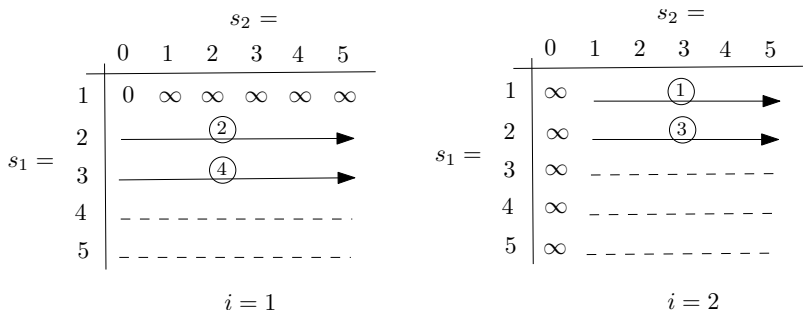


Figure 8.5: Filling up the array $C[s_1, s_2, i]$ for all possible values of s_1, s_2 and i of two convex polygons.

The size of the array C is $O((2\alpha)^m \times m)$. Recall that α is the maximum number of points on any convex polygon and, because of the backtracking, any generated order could have points repeated twice. The computation of each entry in the array requires looking up at most m other entries, so we have $O((2\alpha)^m \times m \times m)$. However, this is only for one fixed-order of each convex polygon. We still have to iterate the same process for all possible orders of $\pi_1, \ldots, \pi_m$. The number for all combinations of these orders is $(2\alpha \times 3^\alpha)^m$. The total running time of the algorithm is thus,

$$\begin{aligned} O((2\alpha)^m m^2 (2\alpha)^m (3^\alpha)^m) &= O(3^{\alpha m} 4^m \alpha^{2m} m^2) \\ &= O(2^{m(1.6\alpha + 2\log\alpha + 2)} m^2). \end{aligned}$$

We state the above result in the following theorem.

Theorem 15. *The m-Convex-Polygons TSP where the tour visits at least once all the vertices of the polygons can be solved in $O(2^{m(1.6\alpha+2\log\alpha+2)}m^2)$ time where α is the maximum number of points on any of the m convex-polygonal-obstacles.*

Consider the special case when backtracking is not permitted. The number of possible orders of points on any convex-polygonal-obstacle in this case is 2α, which is less than what is presented in Lemma 32. The running time of the algorithm in this case is only

$$\begin{aligned} O(\alpha^m m^2 (2\alpha)^m) &= O(2^m \alpha^{2m} m^2) \\ &= O(2^{m(2\log\alpha+1)} m^2). \end{aligned}$$

Thus, we have the following theorem.

Theorem 16. *The m-Convex-Polygons TSP where the tour visits exactly once all the vertices of the polygons can be solved in $O(2^{m(2\log\alpha+1)}m^2)$ time, where α is the maximum number of points on any of the m convex-polygonal-obstacles.*

8.5 Explore the Possibility of FPT-Algorithms

Let us consider the parameterized version of the m-Convex-Polygons TSP above. We have seen that there are two parameters that appear to be small with respect to the size of the input n, namely α and m where α is the maximum number of points on any of the m convex-polygonal-obstacles. Therefore, the immediate question here is whether the problem can be parameterized by α and m. Is there an FPT-algorithm in such a case?

The answer here is NO. Although we can choose the parameter m to be a small number of convex polygons, we cannot separate the parameters m and α from n. If we keep both parameters small, then our instance is also small. In fact, we believe that the problem above is in $W[1]$-hard.

Next, we will consider a variant of the problem where we have one convex hull with many points on the convex position (that is, many points are on the hull) but there are k convex-polygonal-obstacles in the interior of the convex hull (see Figure 8.6). This setting expands the idea of Deĭneko et al. [DHOW04]. In this version of the problem, we will show that we can easily obtain an FPT-algorithm.

Instance: A convex hull and a set of k convex-polygons inside, each with at most α points.
Parameter: k and α.
Question: Find the tour that visits at least once all the vertices of the polygons (the convex hull and the obstacles) and is as short as possible, but does not travel through the interior of the k polygons.

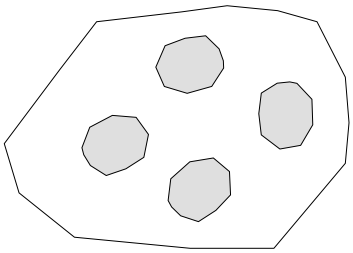


Figure 8.6: A convex hull and a set of k convex-polygons inside.

Note that the k convex-polygons are inside the convex hull and are disjoint. Also throughout this chapter, we will assume that α is small. Let us use the same notation as before. However, we also add $H_0(S)$, the convex hull that encloses all the k obstacles. Then $H(S) = H_0(S) \cup H_1(S) \ldots \cup H_m(S)$. We label the points in $H_0(S)$ clockwise around the convex hull, and let these be $v_1^0, v_2^0, \ldots, v_{h_0}^1$ where $|H_0| = h_0$. It is important to note that, in this problem, we assume there are many points in the convex position of $H_0(S)$. That is, $h_0 \geq (n - k\alpha)$, where n is the total number of vertices of all convex polygons and $n >> k\alpha$.

Lemma 33. *A shortest tour visits the vertices of $H_0(S)$ exactly once and in a cyclic order.*

Proof. We refer to the proof of Abrahamson et al. [ASW05] where P, an outer convex polygon, is used instead of $H_0(S)$. □

Let us consider a $(k+2)$-dimensional array C where the first $k+1$ entries indicate how much we have advanced in the given order of each polygon and the last entry indicates the polygon in which we finish. The array $C[s_0, s_1, \ldots, s_{k-1}, s_k, i]$ is the value of the shortest path similar to the previous section where one additional index s_0 represents the stop-index of points in $H_0(S)$ and $1 \leq s_0 \leq h_0$.

According to Lemma 33, we can fix a cyclic order on $H_0(S)$ in a clockwise direction starting at v_1^0, we call this order, π_0. The value $C[s_0, s_1, \ldots, s_k, i]$ must satisfy the following conditions.

1. It starts at $p_1^{\pi_0}$ on $H_0(S)$.
2. It finishes at the i-th convex polygon.
3. It respects a cyclic order π_0 and the fixed-orders $\pi_1, \pi_2, \ldots, \pi_k$.
4. It visits exactly the points in

$$\{p_1^{\pi_0}, \ldots, p_{s_0}^{\pi_0}\} \cup \{p_1^{\pi_1}, \ldots, p_{s_1}^{\pi_1}\} \cup \{p_1^{\pi_2}, \ldots, p_{s_2}^{\pi_2}\} \cup \ldots \cup \{p_1^{\pi_k}, \ldots, p_{s_k}^{\pi_k}\}.$$

   We set $\{p_1^{\pi_i}, \ldots, p_{s_i}^{\pi_i}\} = \emptyset$ if $s_i = 0$.

The cost of a shortest tour is then computed as

$$\begin{aligned} \min\{ \quad & C[h_0, f_1, \ldots, f_k, 0] + d(p_{h_0-1}^{\pi_0}, p_1^{\pi_0}), \\ & C[h_0, f_1, \ldots, f_k, 1] + d(p_{f_1}^{\pi_1}, p_1^{\pi_0}), \\ & \ldots, \\ & C[h_0, f_1, \ldots, f_k, k] + d(p_{f_k}^{\pi_k}, p_1^{\pi_0})\}. \end{aligned}$$

We do not explain the recurrences here, because they are carried out in a way similar to that of the previous section.

The size of the array C is $O(n \times (2\alpha)^k \times (k+1))$. The computation of each entry in the array requires looking up at most $(k+1)$ other entries, so we have $O(n \times (2\alpha)^k \times (k+1)^2)$ (note that $n >> k\alpha$). However, this is only for one fixed-order of each convex-polygonal-obstacle. We still have to iterate the same process for all possible orders of $\pi_1, \ldots, \pi_k$. The number for all combinations of these orders are $(2\alpha \times 3^\alpha)^k$. Since we start at v_1^0 and finish at v_1^0, we do not need to consider any other orders with respect to the outer convex hull. The total running time of the algorithm is thus,

$$\begin{aligned} O(n(2\alpha)^k(k+1)^2(2\alpha)^k(3^\alpha)^k) &= O(2^{2k}3^{\alpha k}\alpha^{2k}k^2 n) \\ &= O(2^{k(1.6\alpha+2\log\alpha+2)}k^2 n). \end{aligned}$$

We state the above result in the following theorem.

Theorem 17. *A shortest tour visiting at least once all the vertices of a convex hull that has k convex polygons inside can be solved in $O(2^{k(1.6\alpha+2\log\alpha+2)}k^2n)$ time, where α is the maximum number of points on any of the k convex-polygonal-obstacles.*

Consider the special case when backtracking is not permitted. The running time of the algorithm in this case is reduced to

$$\begin{aligned} O(n\alpha^k(k+1)^2(2\alpha)^k) &= O(2^k\alpha^{2k}k^2n) \\ &= O(2^{k(2\log\alpha+1)}k^2n). \end{aligned}$$

We summarize the above result in the following theorem.

Theorem 18. *A shortest tour visiting exactly once all the vertices of a convex hull that has k convex polygons inside can be solved in $O(2^{k(2\log\alpha+1)}k^2n)$ time, where α is the maximum number of points on any of the k convex-polygonal-obstacles.*

8.6 Chapter Summary

To improve on our early results, we will consider solving the remaining problems:

- Prove the conjecture that the m-Convex-Polygons TSP is $W[1]$-hard
- Solve the variant of the problem where we do not need to visit every vertex on the m polygons, but at least one vertex of each polygon must be visited. We believe that this version of the problem has more practical applications. For example, in VLSI design a certain area may be blocked by a rectangle or a convex polygon; however, we may not need to visit every vertex of that polygon.

Chapter 9

Conclusions

Our research provides an alternative approach for computer scientists and mathematicians to cope with hard geometric problems by considering a small parameter k of the given problem and aiming for algorithms, which are exponential only in k, and not in the input size. We refer to the algorithms designed this way as FPT-algorithms.

The research requires a good knowledge of several fields in computer science, namely, the classical computational complexity theory, the parameterized complexity theory, the computational geometry and the Traveling Salesman Problem. Chapter 1 provides the basic concepts of these fields and serves as an introduction to the rest of the thesis.

Later in the thesis, we introduced the following eight hard geometric problems:

1. The Line Cover problem (LC),
2. The Rectilinear Line Cover problem in higher dimensions (RLC),
3. The Line Cover problem restricted to a finite number of orientations (OLC),
4. The Rectilinear k-Links Spanning Path Problem in higher dimensions (RLSPP),
5. The k-Links Spanning Path Problem restricted to a finite number of orientations (OLSPP),

6. The Rectilinear Hyperplane Cover Problem (RHC),
7. The k-Bends Traveling Salesman Problem (KBTSP) and
8. The Rectilinear k-Bends Traveling Salesman Problem (RKB-TSP).

Note that in addition to the eight problems above, we actually extended some of the results to cover several other variants of the problems. The abbreviations in the parentheses are used in Table 9.1, where we summarize all of the fixed-parameter tractable algorithms of the eight problems and their variants.

Our fixed-parameter algorithms use several algorithmic design approaches. The kernelization approach based on a reduction of the problem instance to a kernel of size bounded by some function of the parameter k is a simple, but effective, FPT designing technique. Other techniques, such as the bounded-search-tree and dynamic programming, have also proved themselves to be quite useful in coping with the hardness of NP-hard problems. In particular, the bounded-search-tree technique, where the idea is to limit the depth of the tree for the recursive calls of the corresponding algorithm to a small parameter value, is not hard to understand, but very well applicable to many of the hard geometric problems. When combining the two techniques together, the kernelization and the bounded-search-tree as we did on several occasions, the results are more pronounced. Note that, in some applications, it might be of interest not only to determine one optimal solution, but also to enumerate all of them. In this thesis, we did not consider the enumeration problem. Some reduction rules can be used for enumeration, while some reduction rules may need further structural engineering. How do we design reduction rules that also work in the enumeration problem? We leave this question as an open problem.

In this thesis, we produced eighteen FPT-algorithms for the above problems including their variants. We summarize these results in Table 9.1. We also summarize open problems mentioned in the thesis in Table 9.2. While we have discussed the decision version of the problems, it is not difficult to adapt to FPT-algorithms for the optimization version. We have demonstrated this fact in Chapter 2, where we provided three optimization-version algorithms for the minimum line covering problem. Instead of covering points in the plane with k

Table 9.1: FPT-Algorithms for solving hard geometric problems.

No.	Algorithms	Time Complexities
	LC (decision version)	
1	cascading a rule on itself and sorting the points in kernel	$O(n \log k + k^{2k+2})$
2	prioritizing points in heavy lines	$O(n \log k + k^{2k+2})$
3	cascading with one additional rule	$O(n \log k + k^4(k/2.22)^{2k})$
	LC (optimization version)	
4	searching with increments of one	$O(nk \log k + k^5(k/2.22)^{2k})$
5	guessing a lower bound	$O(n^3 + k^5(k/2.22)^{2k})$
6	using a binary search	$O(n \log^2 k + k^4 \log k(k/1.11)^{4k})$
	RLC	
7	using a bounded-search-tree	$O(d^k n)$
8	using kernelization	$O(n^2 d^2 + d^{k-1} k^2)$
	OLC	
9	using a bounded-search-tree	$O(\phi^k n)$
	RLSPP	
10	using a bounded-search-tree	$O((0.74dk)^k(kd + n)\sqrt{k})$
	OLSPP	
11	using a bounded-search-tree	$O((0.74\phi k)^k(k\phi + n)\sqrt{k})$
	RHC	
12	using a bounded-search-tree	$O(d^k n)$
13	having finite number of orientations	$O(\phi^k n)$
	KBTSP	
14	several line-segments cover points on the same line	$O(kn^2 + k^{8k} n)$
15	one line-segment covers all the points on the same line	$O(kn^2 + 4^k k^{4k+2})$
	RKBTSP	
16	without constraints	$O(kn^2 + k^{4k} n)$
17	same line-segment orientation cover points on the same line	$O((43.5k)^k n)$
18	one line-segment covers all the points on the same line	$O((2.95k)^k n)$

Table 9.2: The summary of open problems chapter by chapter.

Chapter No.	Open Problems
Chapter 2	The enumeration problem of the LINE COVER problem.
Chapter 3	The unrestricted case of the RECTILINEAR k-LINKS SPANNING PATH PROBLEM where several line-segments could cover points on the same line.
Chapter 4	Improving the time complexity of the algorithms for the HYPERPLANE COVER problem.
Chapter 5	Improving the time complexity of the algorithms for the k-BENDS TRAVELING SALESMAN PROBLEM.
Chapter 6	The NP-completeness proof of the RECTILINEAR MINIMUM-LINKS TRAVELING SALESMAN PROBLEM. Improving the time complexity of the algorithms for the RECTILINEAR k-BENDS TRAVELING SALESMAN PROBLEM.
Chapter 7	The problem of covering points with strips. The problem of k covering paths. The problem of minimizing the average time (or the maximum time) for an individual to receive aid in the disaster area.
Chapter 8	The $W[1]$-hardness proof of the m-Convex-Polygons TSP. The problem of visiting every vertex on the m polygons where at least one vertex of each polygon must be visited.

lines, the optimization-version algorithms find a cover for all the points with the minimum number of lines possible.

We also discussed briefly that we can use these optimization-version FPT-algorithms to find the approximate answers for other problems, for example, the minimum-bends tour problem, because the minimum number of covering lines is a bound for the minimum number of bends in the tour. We did not discuss in great detail how we can systematically obtain the optimization-version algorithms from the decision-version FPT-algorithms. In some cases, obtaining practical implementation may require significant algorithmic engineering effort. In the next paragraph, we give some suggestions to what we call "the systematic design of optimization-version algorithms".

Assuming that the decision-version FPT-algorithm of that problem is given, we suggest the following techniques to obtain the optimization-version algorithm running also in $f(k) \cdot n^{O(1)}$ time.

1. Increments: This is the simplest technique. We can make increments of one or smarter techniques, such as square and double etc. We have options for setting the initial value for k where k is the minimum or the maximum value of the solution produced by the algorithm. Typically, we start with $k' = 0$ and continue until we have $k' = k$.

2. Decrements: Similar to the previous technique. However, the initial value for k must be carefully chosen, such that it depends on the parameter and not on the input size.

3. Finding the lower bound: For some problems, this may already have been known in the literature. Otherwise, there are still several techniques for approximating it. For example, we may use reduction rules to place some parts of the instance in the optimal solution. Thus, if we can apply some useful reduction rules prior to calling the decision-version algorithm, then we can approximate the optimal value and avoid having to start searching from zero. We can also consider a reduction of the problem. For example, a rectilinear tour with minimum bends is made up of line-segments and is therefore a cover by lines. Finding the lines gives the approximate answer for the minimum bends. Once we find the lower bound of the problem, we can do the first technique, that is, the increments.

4. Finding the upper bound: For example, for a minimization problem we can double the search index k' until this gives the YES-instance, then use $2k'$ as the upper bound to search for the optimal value between k' and $2k'$. Similarly, for a maximization problem we can double the search index k' until this gives the NO-instance.

5. Binary Search: This technique searches for the desired value between the lower bound and the upper bound in logarithmic time. We demonstrated the usefulness of this technique in Chapter 2.

6. Using polynomial-time approximation algorithms: Suppose that for some problem, a polynomial-time approximation algorithm is available in the literature. A minimization problem having the approximate solution suggests the use of decrements. On the other hand, for a maximization problem having the approximate solution then we can use increments.

7. Approaching from NO-instances: For most of the problems, it is easier and faster to decide the NO-instances. This may even be achievable through a reduction rule. Therefore, another technique is to focus on inventing a good reduction rule or a polynomial-time algorithm to quickly decide the NO-instances.

8. Knowing the nature of the parameter: If the parameter of the given problem cannot be an odd number, this reduces half of the solution space. For example, in rectilinear minimum-bends tours, the number of bends in the tour is always even.

9. Evaluation of the current solution: We can improve our next-guess based on the current solution. For example, while having so many points left uncovered, we might increase the number of lines used to cover the points in proportion to this number.

10. Mixtures of the above techniques: We can choose a number of the techniques that are suitable for the problem.

Our final remark to the reader is that, before attacking a problem with fixed-parameter tractability, it is useful first to verify the hardness of that problem[1].

[1]Note that one can also try FPT approaches on problems in the complexity class P. In fact, that suggests the possibility of comparing performance of implementations of existing polynomial time algorithms with one suggested by FPT approaches.

Often the hardness is known and given in the literature of classical computational complexity theory in terms of NP-hard or NP-complete. Therefore, we can merely refer to these results. However, for certain problems, this is not the case and thus we prove the hardness ourselves. In this thesis, we gave the NP-completeness proofs for the decision version of the following problems:

1. The Rectilinear Minimum-Bends Traveling Salesman Problem in 3D,
2. The Rectilinear Minimum-Links Spanning Path Problem in 3D and
3. Covering Points in 3D with Axis-Parallel Planes of 2D.

Therefore, we confidently conclude that our research will not only advance the field of the parameterized complexity theory, but also the fields of the classical computational complexity theory, the computational geometry and the Traveling Salesman Problem.

Bibliography

[ABCC06] D.L. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, NJ, USA, 2006.

[ABD+05] E.M. Arkin, M.A. Bender, E.D. Demaine, S.P. Fekete, J.S.B. Mitchell, and S. Sethia. Optimal covering tours with turn costs. *SIAM Journal of Computing*, 35(3):531–566, 2005.

[AFM00] E.M. Arkin, S.P. Fekete, and J.S.B. Mitchell. Approximation algorithms for lawn mowing and milling. *Computational Geometry*, 17(1-2):25–50, 2000.

[Aga90] P.K. Agarwal. Partitioning arrangements of lines II: Applications. *Discrete and Computational Geometry*, 5:533–573, 1990.

[AMP03] E.M. Arkin, J.S.B. Mitchell, and C.D. Piatko. Minimum-link watchman tours. *Information Processing Letters*, 86(4):203–207, 2003.

[AMS92] E.M. Arkin, J.S.B. Mitchell, and S. Suri. Optimal link path queries in a simple polygon. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 269–279. SIAM, Philadelphia, USA, 1992.

[Aro96] S. Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 2–12. IEEE, USA, 1996.

[ASW05] J. Abrahamson, A. Shokoufandeh, and P. Winter. Euclidean TSP between two nested convex obstacles. *Information Processing Letters*, 95(2):370–375, 2005.

[BBD+07] S. Bereg, P. Bose, A. Dumitrescu, F. Hurtado, and P Valtr. Axis-aligned traversals of points with a minimum number of turns. In *Proceedings of the 23rd Annual Symposium on Computational Geometry, (SOCG'07)*, pages 46–55, New York, USA, June 2007. ACM.

[BBD+09] S. Bereg, P. Bose, A. Dumitrescu, F. Hurtado, and P Valtr. Traversing a set of points with a minimum number of turns. *Discrete and Computational Geometry*, 41:513–532, 2009.

[Bel62] R. Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63, 1962.

[BG93] J.F. Buss and J. Goldsmith. Nondeterminism within P. *SIAM Journal of Computing*, 22(3):560–572, June 1993.

[BH98] V. Brattka and P. Hertling. Feasible real random access machines. *Journal of Complexity*, 14(4):490–526, 1998.

[BK06] P. Berman and M. Karpinski. 8/7-approximation algorithm for (1, 2)-TSP. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 641–648, New York, USA, 2006. ACM.

[Cen08] Center for Excellence in DMHA. Cyclone Nargis Update, May 24, 2008. http://www.coe-dmha.org/Myanmar/Cyc05242008.htm. Retrieved on: 06 August, 2009.

[CGA07] CGAL Editorial Board. CGAL User and Reference Manual, 3.3 edition, 2007. http://www.cgal.org/Manual/3.3/doc_html /cgal_manual/packages.html.

[Cha85] B. Chazelle. On the convex layers of a planar set. *IEEE Transactions on Information Theory*, 31(4):509–517, 1985.

[Chr76] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 338, Graduate School

of Industrial Administration, Carnegie-Mellon University, Pittsburgh, USA, 1976.

[CKJ01] J. Chen, I.A. Kanj, and W. Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.

[CKX06] J. Chen, I.A. Kanj, and G. Xia. Improved parameterized upper bounds for vertex cover. In R. Kralovic and P. Urzyczyn, editors, *Proceedings of the 31st Mathematical Foundations of Computer Science*, volume 4162 of *Lecture Notes in Computer Science*, pages 238–249. Springer, Berlin, August 28–September 1, 2006.

[CLRS01] T.H Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Massachusetts, 2nd edition, 2001.

[Col04] M.J. Collins. Covering a set of points with a minimum number of turns. *International Journal Computational Geometry and Applications*, 14(1-2):105–114, 2004.

[dBvKNO92] M. de Berg, M.J. van Kreveld, B.J. Nilsson, and M.H. Overmars. Shortest path queries in rectilinear worlds. *International Journal Computational Geometry Applications*, 2(3):287–309, 1992.

[dBvKOS00] M. de Berg, M.J. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, 2nd edition, 2000.

[DF95a] R.G. Downey and M.R. Fellows. Fixed-parameter tractability and completeness I: basic results. *SIAM Journal of Computing*, 24(4):873–921, August 1995.

[DF95b] R.G. Downey and M.R. Fellows. Fixed-parameter tractability and completeness II: on completeness for $W[1]$. *Theoretical Computer Science*, 141(1-2):109–131, April 1995.

[DF99] R.G. Downey and M.R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, New York, 1999.

[DHOW04] V.G. Deĭneko, M. Hoffmann, Y. Okamoto, and G.J. Woeginger. The traveling salesman problem with few inner points. In K.Y. Chwa and J.I. Munro, editors, *Proceedings of the 10th COCOON*, volume 3106 of *Lecture Notes in Computer Science*, pages 268–277. Springer, Berlin, August 17–20, 2004.

[DP04] A. Datta and S. Pal. Computing convex-layers by a multi-layer self-organizing neural network. In N.R. Pal, N. Kasabov, R.K. Mudi, S. Pal, and S.K. Parui, editors, *Proceedings of the 11th International Conference on Neural Information Processing (ICONIP'04)*, volume 3316 of *Lecture Notes in Computer Science*, pages 647–652. Springer, Berlin, November 22–25, 2004.

[DSW05] R.L.S. Drysdale, C. Stein, and D.P. Wagner. An $O(n^{5/2} \log n)$ algorithm for the rectilinear minimum link-distance problem. In *Proceedings of the 17th Canadian Conference on Computational Geometry*, pages 97–100. University of Windsor, Ontario, Canada, 2005.

[DvDR94] V.G. Deĭneko, R. van Dal, and G. Rote. The convex-hull-and-line traveling salesman problem: A solvable case. *Information Processing Letters*, 51(3):141–148, 1994.

[DW96] V.G. Deĭneko and G.J. Woeginger. The convex-hull-and-k-line traveling salesman problem. *Information Processing Letters*, 59(6):295–301, 1996.

[ECHS09] V. Estivill-Castro, A. Heednacram, and F. Suraweera. Reduction rules deliver efficient FPT-algorithms for covering points with lines. *ACM Journal of Experimental Algorithmics*, 14:1.7–1.26, November 2009.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1987.

[ESS93] H. Edelsbrunner, R. Seidel, and M. Sharir. On the zone theorem for hyperplane arrangements. *SIAM Journal of Computing*, 22(2):418–429, 1993.

[FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, Berlin, 2006.

[FGK+08] M.R. Fellows, P. Giannopoulos, C. Knauer, C. Paul, F. Rosamond, S. Whitesides, and N. Yu. On the parameterized complexity of the discrete milling problem with turn costs, 2008. manuscript.

[FL92] M.R. Fellows and M.A. Langston. On well-partial-order theory and its application to combinatorial problems of VLSI design. *SIAM Journal on Discrete Mathematics*, 5(1):117–126, 1992.

[FMRS00] M.R. Fellows, C. McCartin, F.A. Rosamond, and U. Stege. Coordinatized kernels and catalytic reductions: An improved FPT algorithm for max leaf spanning tree and other problems. In S. Kapoor and S. Prasad, editors, *Proceedings of the 20th Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *Lecture Notes in Computer Science*, pages 240–251. Springer, Berlin, December 13–15, 2000.

[For09] L. Fortnow. The status of the P versus NP problem. *Communications of the ACM*, 52(9):78–86, 2009.

[FW96] E. Fink and D. Wood. Fundamentals of restricted-orientation convexity. *Information Sciences*, 92(1-4):175–196, 1996.

[FW04] E. Fink and D. Wood. *Restricted-Orientation Convexity*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, New York, 2004.

[GJ79] M.R. Garey and G.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, USA, 1979.

[GKP89] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Publishing Company, USA, 1989.

[GKW08] P. Giannopoulos, C. Knauer, and S. Whitesides. Parameterized complexity of geometric problems. *Computer Journal*, 51(3):372–384, 2008.

[GL06] M. Grantson and C. Levcopoulos. Covering a set of points with a minimum number of lines. In T. Calamoneri, I. Finocchi, and G.F. Italiano, editors, *Proceedings of the 6th Italian Conference on Algorithms and Complexity (CIAC'06)*, volume 3998 of *Lecture Notes in Computer Science*, pages 6–17. Springer, Berlin, 2006.

[Goo09] Google Inc. Google Maps, 2009. http://maps.google.com/. Retrieved on: 06 August, 2009.

[GOR91] L.J. Guibas, M.H. Overmars, and J. Robert. The exact fitting problem for points. In *Proceedings of the 3rd Canadian Conference on Computational Geometry*, pages 171–174. Simon Fraser University, Vancouver, British Columbia, August 6–10, 1991.

[GOR96] L.J. Guibas, M.H. Overmars, and J. Robert. The exact fitting problem in higher dimensions. *Computational Geometry: Theory and Applications*, 6:215–230, 1996.

[GP07] G. Gutin and A.P. Punnen. *The Traveling Salesman Problem and Its Variations*. Springer, New York, 2007.

[GT95] A. García and J. Tejel. The order of points on the second convex hull of a simple polygon. *Discrete and Computational Geometry*, 14(2):185–205, 1995.

[GT97] A. García and J. Tejel. The two-convex-polygons TSP: A solvable case. *TOP*, 5(1):105–126, 1997.

[Güt83] H.R. Güting. Stabbing c-oriented polygons. *Information Processing Letters*, 16:35–40, 1983.

[Gut09] S. Gutner. Polynomial kernels and faster algorithms for the dominating set problem on graphs with an excluded minor. In *Proceedings of the 4th International Workshop on Parameterized and Exact Computation (IWPEC'09)*, September 7–11, 2009.

[HÖ7] F. Hüffner. *Algorithms and Experiments for Parameterized Approaches to Hard Graph Problems*. PhD thesis, Institut für Informatik, Friedrich-Schiller-Universität Jena, Germany, October 2007.

[HK62] M. Held and R.M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, March 1962.

[HM91] R. Hassin and N. Megiddo. Approximation algorithms for hitting objects with straight lines. *Discrete Applied Mathematics*, 30(1):29–42, 1991.

[HNW08] F. Hüffner, R. Niedermeier, and S. Wernicke. Techniques for practical fixed-parameter algorithms. *Computer Journal*, 51(1):7–25, 2008.

[Hur08] F. Hurtado, 2008. Private communication.

[KAR00] V.S.A. Kumar, S. Arya, and H. Ramesh. Hardness of set cover with intersection 1. In U. Montanari, J.D.P. Rolim, and E. Welzl, editors, *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *Lecture Notes in Computer Science*, pages 624–635. Springer, Berlin, 2000.

[KB06] I. Kara and T. Bektas. Integer linear programming formulations of multiple salesman problems and its variations. *European Journal of Operational Research*, 174(3):1449–1458, November 2006.

[Kla54] M.S. Klamkin. Problem 1123: Polygonal path covering a square lattice. *The American Mathematical Monthly*, 61(6):423, June-July 1954.

[Kla55] M.S. Klamkin. E 1123. *The American Mathematical Monthly*, 62(2):124, February 1955.

[KMM97] S. Kapoor, S.N. Maheshwari, and J.S.B. Mitchell. An efficient algorithm for Euclidean shortest paths among polygonal obstacles in the plane. *Discrete and Computational Geometry*, 18(4):377–383, 1997.

[KMSV98] S. Khanna, R. Motwani, M. Sudan, and U. Vazirani. On syntactic versus computational views of approximability. *SIAM Journal of Computing*, 28(1):164–191, 1998.

[LCY90] D.T. Lee, T.H. Chen, and C.D. Yang. Shortest rectilinear paths among weighted obstacles. In *Proceedings of the 6th ACM Symposium on Computational Geometry (SCG'90)*, pages 301–310, New York, USA, 1990. ACM.

[Lev07] A. Levitin. *Introduction to The Design and Analysis of Algorithms*. Addison-Wesley Publishing Company, USA, 2nd edition, 2007.

[LM01] S. Langerman and P. Morin. Cover points with lines. In *Proceedings of the 11th Fall Workshop on Computational Geometry*. Polytechnic University, Brooklyn, New York, USA, 2001.

[LM02] S. Langerman and P. Morin. Cover things with things. In R.H. Möhring and R. Raman, editors, *Proceedings of the 10th European Symposium on Algorithms*, volume 2641 of *Lecture Notes in Computer Science*, pages 662–673. Springer, Berlin, 2002.

[LM05] S. Langerman and P. Morin. Cover things with things. *Discrete and Computational Geometry*, 33(4):717–729, 2005.

[LYW94] D.T. Lee, C.D. Yang, and C.K. Wong. On bends and distances of paths among obstacles in two-layer interconnection model. *IEEE Transactions on Computers*, 43(6):711–724, 1994.

[LYW96] D.T. Lee, C.D. Yang, and C.K. Wong. Rectilinear paths among rectilinear obstacles. *Discrete Applied Mathematics*, 70(3):185–215, 1996.

[Mar05] D. Marx. Efficient approximation schemes for geometric problems? In G.H. Brodal and S. Leonardi, editors, *Proceedings of the 13th European Symposium on Algorithms*, volume 3669 of *Lecture Notes in Computer Science*, pages 448–459. Springer, Berlin, October 3–6, 2005.

[Mar08] D. Marx. Parameterized complexity and approximation algorithms. *Computer Journal*, 51(1):60–78, 2008.

[Mat90] J. Matoušek. Cutting hyperplane arrangements. In *Proceedings of the 6th Annual Symposium on Computational Geometry*, pages 1–9, New York, USA, 1990. ACM.

[McC03] C. McCartin. *Contributions to Parameterized Complexity*. PhD thesis, Victoria University of Wellington, New Zealand, September 2003.

[Meh84a] K. Mehlhorn. *Data Structure and Algorithm 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer, Berlin, 1984.

[Meh84b] K. Mehlhorn. *Data Structure and Algorithm 2: Graph algorithms and NP-completeness*. EATCS Monographs on Theoretical Computer Science. Springer, Berlin, 1984.

[Mit93] J.S.B. Mitchell. Shortest paths among obstacles in the plane. In *Proceedings of the 9th ACM Symposium on Computational Geometry (SCG'93)*, pages 308–317, New York, USA, 1993. ACM.

[MPA92] J.S.B. Mitchell, C.D. Piatko, and E.M. Arkin. Computing a shortest k-link path in a polygon. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science (FOCS'92)*, pages 573–582. IEEE, USA, 1992.

[MS00] S. Mishra and K. Sikdar. On the hardness of approximating some NP-optimization problems related to minimum linear ordering problem. In J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors, *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, volume 1872 of *Lecture Notes in Computer Science*, pages 186–199. Springer, Berlin, August 17–19, 2000.

[MT82] N. Megiddo and A. Tamir. On the complexity of locating linear facilities in the plane. *Operations Research Letters*, 1(5):194–197, November 1982.

[Nie04] R. Niedermeier. Ubiquitous parameterization - Invitation to fixed-parameter algorithms. In J. Fiala, V. Koubek, and J. Kratochvíl, editors, *Proceedings of the 29th Mathematical Foundations of Computer Science*, volume 3153 of *Lecture Notes in Computer Science*, pages 84–103. Springer, Berlin, August 2004.

[Nie06] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and its Applications 31. Oxford University Press, New York, 2006.

[Pap77] C.H. Papadimitriou. The Euclidean traveling salesman problem is NP-complete. *Theoretical Computer Science*, 4:237–244, 1977.

[Pri05] E. Prieto. *Systematic Kernelization in FPT Algorithm Design*. PhD thesis, The University of Newcastle, Callaghan, NSW, Australia, March 2005.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry An Introduction*. Texts and Monographs in Computer Science. Springer, New York, 1985.

[Ros09] F. Rosamond, Editor. Parameterized Complexity Newsletter, September 2009. http://fpt.wikidot.com.

[Rot92] G. Rote. The N-line traveling salesman problem. *Networks*, 22:91–108, 1992.

[RW88] G. Rawlins and D. Wood. Computational geometry with restricted orientations. In *Proceedings of the 13th IFIP Conference on System Modelling and Optimization*, volume 113 of *Lecture Notes in Computer Science*, pages 375–384. Springer, Berlin, 1988.

[RW91] G. Rawlins and D. Wood. Restricted-oriented convex sets. *Information Sciences*, 54:263–281, 1991.

[Sau07] S. Saurabh. *Exact Algorithms for Optimization and Parameterized Versions of some Graph Theoretic Problems*. PhD thesis, The Institute of Mathematical Sciences, Homi Bhabha National Institute Mumbai, India, August 2007.

[SCCN05] W. Sheng, H. Chen, H. Chen, and Xi N. Optimal planning of a mobile sensor for aircraft rivet inspection. In *IEEE International Conference on Robotics and Automation (ICRA'05)*, pages 3181–3186. IEEE, USA, April 18–22, 2005.

[Sel55] J.L. Selfridge. E 1123. *The American Mathematical Monthly*, 62(6):443, June-July 1955.

[SH73] J.A. Svestka and V.E. Huckfeldt. Computational experience with an M-salesman travelling salesman algorithm. *Management Science*, 19(7):790–799, March 1973.

[Slo01] C. Sloper. Parameterized complexity and the method of test sets. Master's thesis, The University of Bergen, Norway, 2001.

[Slo05] C. Sloper. *Techniques in Parameterized Algorithm Design*. PhD thesis, The University of Bergen, Norway, 2005.

[SMT99] S. Somhom, A. Modares, and Enkawa T. Competition-based neural network for the multiple travelling salesman problem with minimax objective. *Computers & Operations Research*, 26(4):395–407, April 1999.

[SW01] C. Stein and D.P. Wagner. Approximation algorithms for the minimum bends traveling salesman problem. In K. Aardal and B. Gerards, editors, *Proceedings of the 8th International IPCO Conference*, volume 2081 of *Lecture Notes in Computer Science*, pages 406–422. Springer, Berlin, June 13–15, 2001.

[The08a] The Australian. 80,000 dead in one Burma province, May 08, 2008. http://www.theaustralian.news.com.au/story/0,25197,23664740-12377,00.html. Retrieved on: 06 August, 2009.

[The08b] The Hindu. Naval ships discharge supplies in Yangon, May 09, 2008. http://www.hindu.com/2008/05/09/stories/2008050955341300.htm. Retrieved on: 06 August, 2009.

[The08c] TheStar.Com, Toronto Star. Asian bloc to handle Burma aid, May 19, 2008. http://www.thestar.com/article/427381. Retrieved on: 06 August, 2009.

[Wag06] D.P. Wagner. *Path Planning Algorithms under the Link-Distance Metric*. PhD thesis, Dartmouth College, Hanover, NH, USA, February 2006.

[WDS09] D.P. Wagner, R.L.S. Drysdale, and C. Stein. An $O(n^{5/2} \log n)$ algorithm for the rectilinear minimum link-distance problem in three dimensions. *Computational Geometry: Theory and Applications*, 42(5):376–387, 2009.

[Wei00] K. Weihrauch. *Computable analysis: An introduction.* Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin, 2000.

[Woe03] G.J. Woeginger. Exact algorithms for NP-hard problems: A survey. In M. Michael Jünger, G. Reinelt, and G. Rinaldi, editors, *Proceedings of the 5th International Workshop 2001, Combinatorial Optimization - Eureka! You shrink!*, volume 2570 of *Lecture Notes in Computer Science*, pages 185–207. Springer, Berlin, March 5–9, 2003.