

Reduction Rules Deliver Efficient FPT-Algorithms for Covering Points with Lines

Vladimir Estivill-Castro
Apichat Heednacram
and
Francis Suraweera
Griffith University, Australia

We present efficient algorithms to solve the LINE COVER problem exactly. In this NP-complete problem, the inputs are n points in the plane and a positive integer k , and we are asked to answer 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 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 algorithms to solve the corresponding optimization problem exactly. We experimentally evaluated ten variants of the algorithms to determine the impact and 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 previously known algorithms.

Categories and Subject Descriptors: F.2.2 [Theory of Computation]: Nonnumerical Algorithms and Problems—*Geometrical Problems and Computations*

General Terms: Algorithms, Design, Experimentation

Additional Key Words and Phrases: Covering points with lines, line cover problem, parameterized complexity, practical FPT-algorithm

1. INTRODUCTION

The LINE COVER problem consists of covering a set S of n points in the plane with the minimum number of line segments possible. It has been known to be NP-hard [Megiddo and Tamir 1982] for over 20 years (in fact, APX-hard [Kumar et al. 2000]) and therefore, considered to be intractable. Nevertheless, this problem emerges in direct connection to variations of TRAVELING SALESMAN PROBLEM (TSP), one of the most studied problems in algorithms, operations research, optimization, and computational complexity [Applegate et al. 2006]. The N -LINE TRAVELING SALESMAN PROBLEM consists of covering points with lines, and

Author's address: School of Information and Communication Technology, Griffith University, Brisbane, QLD, 4111, Australia.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20xx ACM 0000-0000/20xx/0000-0001 \$5.00

Rote [1992] designed an algorithm to solve this version approximately. Deĭneko et al. [1996] also investigated how the convex-hull and k lines are used to solve the TSP. There are also 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 since moving objects pick up speed when traveling in a straight line. The LINE COVER problem also appears in the MINIMUM BENDS TRAVELING SALESMAN PROBLEM [Stein and Wagner 2001], 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 the movement of heavy machinery [Lee et al. 1994; 1996; Arkin et al. 2000; 2005; Drysdale et al. 2005].

These applications demand practical implementations. Parameterized complexity offers alternatives to those problems regarded as intractable from the perspective of classical complexity theory [Downey and Fellows 1999; Niedermeier 2006; Flum and Grohe 2006]. Giannopoulos et al. [2008] survey the recent success of parameterized complexity when applied to geometric problems. The decision version of LINE COVER was shown to be fixed-parameter tractable by Langerman and Morin [2001; 2002; 2005] who also provided two FPT-algorithms.¹ 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 one as KERNELIZE, since 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. [1996] to obtain $O(nk + k^{2(k+1)})$ time complexity. Later, Grantson and Levcopoulos [2006] used both of these algorithms to obtain theoretical improvements and solve the optimization version 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, 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.

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, reduction rules) leads to algorithms that can handle even large instances. The main contribution is the fast implementation of the known and 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 deliver smaller kernels than the theoretical guarantee. We show this for the decision and the optimization versions of the problem.

¹An FPT-algorithm has polynomial worst-case complexity on the size of the input, although maybe exponential time complexity on an integer parameter. Formal definitions appear in [Downey and Fellows 1999, p.8] or [Niedermeier 2006, p.23]. A problem that has an FPT-algorithm is said to belong to the class FPT.

2. SIMPLE REDUCTION RULES

A very powerful aspect of the theory of parameterized complexity is the alternative characterization of the class FPT; namely, a decision problem is FPT if and only if it is kernelizable [Downey and Fellows 1999; Niedermeier 2006].

Definition 2.1 Kernelization. [Niedermeier 2006, p. 55] Let \mathcal{P} be a parameterized problem with inputs (S, k) , where S is the problem instance and k is the parameter. Kernelization means to replace instance (S, k) by a reduced instance (S', k') such that $k' \leq f(k)$, $|S'| \leq g(k)$ where f and g are arbitrary functions depending only on k and $(S, k) \in \mathcal{P}$ if and only if $(S', k') \in \mathcal{P}$. The reduction from (S, k) to (S', k') must be computable in polynomial time.

The reduced instance (S', k') defined above is called a *kernel* [Downey and Fellows 1999, p. 39]. The function $g(k)$ is the size of the kernel. Kernelization can be achieved through 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 technique [Hüffner et al. 2008]. We start by illustrating how the reduction-rules approach provides a simple proof that the decision problem is fixed-parameter tractable.

REDUCTION RULE 1. *If $k \geq \lceil n/2 \rceil$, then the answer is yes [Grantson and Levcopoulos 2006].*

REDUCTION RULE 2. *If $k = 1$, then the answer is yes if and only if all points in S are co-linear [Grantson and Levcopoulos 2006].*

REDUCTION RULE 3. *Remove duplicated points in S [Langerman and Morin 2005].*

While BST-DIM-SET-COVER [Langerman and Morin 2005] can handle duplicated points in S , Grantson et al. [2006] 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 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 [Langerman and Morin 2005].*

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. Grantson and Levcopoulos [2006] use this rule to decide NO-instances as follows.

LEMMA 2.2. *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 LINE COVER can be reduced to a problem kernel of size at most k^2 . At the point 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 3 or more points in S . If L is a set of lines, we let

$\text{cover}(L)$ be all points in S covered by some line in L . With this notation, we can describe the next reduction rule which can be taken as a structural lemma as well.

REDUCTION RULE 5. *Let $p_1 \neq p_2$ be two points in $S \setminus \text{cover}(L_3)$. Let $S' = S \setminus \{p_1, p_2\}$. Transform (S, k) into $(S', k - 1)$.*

PROOF. Consider an instance where $\{p_1, p_2\} \subset S \setminus \text{cover}(L_3)$. 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 $S \setminus \text{cover}\{\overline{p_1, p_2}\}$ 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 Fig. 1 for an illustration. \square

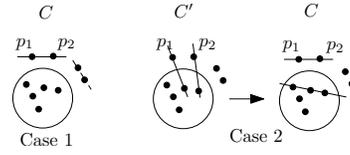


Fig. 1. Two cases for the proof of Reduction Rule 5.

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

REDUCTION RULE 6. *Let $p_1 \neq p_2$ be two points in $\text{cover}(L_3)$. Suppose that no other line in L_3 besides $\overline{p_1, p_2}$ covers p_1 or p_2 (i.e. $L_3 \cap \{l \mid l \text{ covers } p_1\} = L_3 \cap \{l \mid l \text{ covers } p_2\} = \{\overline{p_1, p_2}\}$). Let $S' = S \setminus \text{cover}\{\overline{p_1, p_2}\}$. Transform (S, k) into $(S', k - 1)$.*

PROOF. 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 $\overline{p_1, p_2}$. \square

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 algorithms where the decision about which rules are involved is different, and we test our claim experimentally.

3. ALGORITHMS FOR THE DECISION PROBLEM

We will focus on the decision problem first. Namely, 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 another until none of the rules apply. 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 reduction rules, we will invoke the kernel lemma (Lemma 2.2). We may need to invoke a bounded-search-tree approach to solve the kernel, but we will also suggest how to heuristically set the scene for the exhaustive search. Although the kernel may be difficult to solve, it could have properties where other heuristics work well.

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 power of the cascading effects of reduction rules is crucial to the algorithmic engineering of FPT-algorithms [Niedermeier 2006].

The preprocessing algorithm below resolves the instance (S, k) or returns a kernelized instance (S', k') where $S' \subseteq S$ and $k' \leq k$. In it, all rules except our last one 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 2.2), 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)

```

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

Our ordering of the rules has been the result of experimental evaluation of the trade-off of cost of application of the rule with respect to overall running time of the algorithm. For example, we recommend that Reduction Rule 4 operate 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, since this is a constant-time check before the more costly construction of L_3 .

3.1 Details of REDUCTIONRULE_KPLUS

The Boolean function REDUCTIONRULE_KPLUS checks if Reduction Rule 4 is applicable. Grantson et al. [2006] present a function (they call LINES) to find lines covering more than k points in the plane. This LINES function [2006, p. 9] uses Guibas et al.'s algorithm [1996] 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 three subroutines depending on the values of $k + 1$. If $k \leq 3$, the first

subroutine transforms $k + 1$ points into $k + 1$ lines in the dual space² and uses the topological line sweep algorithm to find all vertices incident to $k + 1$ lines. The second subroutine is invoked when $(k + 1)^3 \leq n^2$, otherwise the third subroutine is used. However, the latter two subroutines also call the Matušek [1990] algorithm or the Agrawal [1990] algorithm to find incidences between sets of points and sets of lines.³ 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 arrangement⁴ 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 dual space is the same as finding lines through at least $k + 1$ points in primal space but has 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 [2006], there are several distinctive points.

- (1) Our function REDUCTIONRULE_KPLUS cascades the same rule 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 function finds settings where Lemma 2.2 applies where 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 transformation [de Berg et al. 2000, p. 169]. Guibas et al.’s algorithm [1996] 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 [2006] implicitly requires a test for the k lines found to cover S (and a test that S is empty). While the test for emptiness of S is presented in the last two lines of the original pseudo-code [Grantson and Levkopoulos 2006], 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 $S \setminus \text{cover}\{l_1, \dots, l_s\}$ is computed explicitly and the exit points of the loop are clearer.

²The dual-space mapping sends 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 has the property that two lines l_p and l_q in dual space intersect at a point p_l , who in primal space is the line l through the points p and q that are images of the two lines [de Berg et al. 2000, p. 169].

³Note that in the late 80s till 1990 there were several independent and incremental improvements to the incidence problem by both Matušek and Agrawal with significant use of machinery from computational geometry.

⁴Given a set $L = \{l_1, \dots, l_n\}$ of n lines in the plane, the *arrangement of lines* $\mathcal{A}(L)$ is the subdivision of the plane into vertices, edges, and faces induced by L [de Berg et al. 2000, p. 172]

In the pseudo-code for our function REDUCTIONRULE_KPLUS the arrangement of dual-lines is denoted by \mathcal{A} . This function scans the original set S and examines one point at a time. As an invariant of this iteration, it maintains a set of lines $L_k = \{l_1, l_2, \dots, l_s\}$ 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 $\text{cover}(L_k) \cap S' = \emptyset$ (that is, 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 2.2 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.

```

REDUCTIONRULE_KPLUS( $S, k, S', k'$ )
1   $S' \leftarrow \emptyset; L_k \leftarrow \emptyset; \text{arrangement } \mathcal{A} \leftarrow \emptyset$       (*initialization*)
2  while ( $S \neq \emptyset$ )
3      do choose  $p \in S; S \leftarrow S \setminus \{p\}$ 
4          if  $p$  not in  $\text{cover}(L_k)$ 
5              then  $S' \leftarrow S' \cup \{p\}$ 
6                  Transform  $p$  into a line  $\text{dual}(p)$  in dual space
7                   $\mathcal{A} \leftarrow \mathcal{A}.\text{insert}(\text{dual}(p));$ 
8                  while there is vertex  $v$  in  $\mathcal{A}$  incidents to more
9                  than  $k - |L_k|$  lines and  $|L_k| \leq k$ 
10                     do  $L_k \leftarrow L_k \cup \{\text{dual}(v)\}; k' \leftarrow k - |L_k|$ 
11                         for each  $p' \in S' \cap \text{cover}(\text{dual}(v))$ 
12                             do  $\mathcal{A} \leftarrow \mathcal{A}.\text{delete}(\text{dual}(p'));$ 
13                              $S' \leftarrow S' \setminus \text{cover}(\text{dual}(v))$ 
14                     if ( $|S'| > ((k - |L_k|)^2)$       (*Lemma 2.2*)
15                         then return false
16 return true      (*return  $S'$  and  $k'$  by reference*)

```

When the next point $p \in S$ is examined, it is removed from S . If $p \in \text{cover}(L_k = \{l_1, l_2, \dots, l_s\})$, then nothing needs to be done to maintain the invariants. However, when $p \notin \text{cover}(L_k)$, then $S' \leftarrow S' \cup \{p\}$. This addition of p to S' may now trigger the reduction rule in S' . So 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. So, in this inner-loop, we enlarge L_k and set $S' \leftarrow S' \setminus \text{cover}(L_k)$ 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 2.2 (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 2.2 for the function to return **false**.

3.2 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 now 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( $S', k'$ ) [Langerman and Morin 2005]
```

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 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 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).

3.2.1 Running Time Analysis. Reduction Rule 1 and Reduction Rule 2 can be performed in linear time. The dominant work in the preprocessing algorithm is the work performed by REDUCTIONRULE_KPLUS. For each point chosen from S , we 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 [Edelsbrunner et al. 1993], 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) [CGAL Editorial Board 2007]. 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} . Due to Lemma 2.2, 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. We can construct the dual-lines arrangement in $O(k^4)$ time. Sweeping for all vertices incident to more than $k - |L_k|$ lines can be done in $O(k^4)$ time. Thus, k executions of the inner-loop require $O(k^5)$ time. The total running time of the REDUCTIONRULE_KPLUS function is $O(n \log k + k^5)$.

Our next rule, Reduction Rule 5 requires the construction of L_3 . Since the size of S' is at most k^2 , 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 2 lines. We construct the arrangement in $O(k^4)$ time and build L_3

in $O(k^5)$ time (there are at most $O(k^4)$ lines in L_3 and each line covers at most k points). Hence, Reduction Rule 5 can be carried out in $O(k^5)$ time.

We repeatedly apply all the above rules at most $k - 1$ times since 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 $O(n \log k + k^6)$ 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 done together while we construct 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 $O(n \log k + k^{2k+2})$.

3.3 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 now follows.

ALGORITHM CASCADE FURTHER(S, k)

- 1 (S', k') \leftarrow THE PREPROCESSING ALGORITHM(S, k)
- 2 Construct L_3 , a set of all lines through 3 or more points in S' .
- 3 **while** there exist $\{p_1, p_2\} \in \text{cover}(L_3)$ 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** $S' \leftarrow S' \setminus \text{cover}\{\overline{p_1, p_2}\}$; $k' \leftarrow k' - 1$;
- 5 **for** $p \in S'$
- 6 **do** $\text{Weight}(p) \leftarrow \max_{l_i \in L_3 \cap p \in l_i} \{|\{q \in S' \mid q \in l_i\}| + \frac{1}{|L_3|}\}$
- 7 Sort S' in descending Weight order
- 8 **return** BST-DIM-SET-COVER(S', k') [Langerman and Morin 2005]

3.3.1 Running Time Analysis. The additional work here is to apply Reduction Rule 6. The set L_3 can be built in $O(k^5)$ time while also storing the information of 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. Hence, Reduction Rule 6 can be carried out in $O(k^5)$ 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 $O(k^{2k+2})$.

Therefore, the total running time for CASCADE FURTHER is $O(n \log k + k^{2k+2})$.

3.4 Prioritize Points in Heavy Lines

The variant we introduce now solves the kernel differently. There would be no need to sort the points, but to prioritize over lines around points. We solve the problem

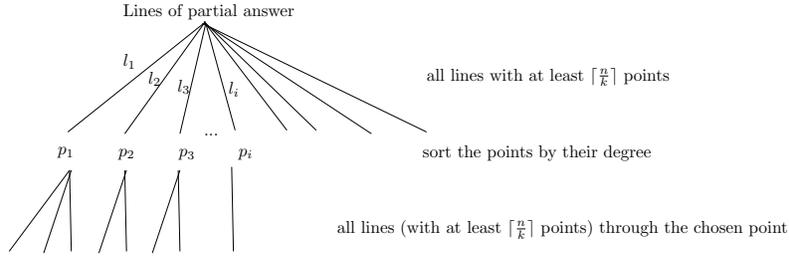
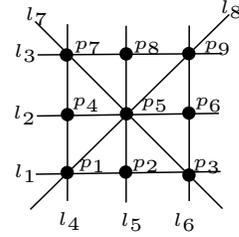


Fig. 3. A search tree in each recursive call.

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 $\text{cover}(L_{\lceil n/k \rceil})$ be all points in S covered by a line in $L_{\lceil n/k \rceil}$. Grantson et al. [2006] 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 Fig. 2.

In this example, l_1, l_2, \dots, l_8 are all candidate lines since 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 belong 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 average number of points results in potentially a quadratic number of recursive calls.

Fig. 2. Instance (S, k) where $|S| = 9$ and $k = 3$.

Our strategy is to revert 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 $\text{cover}(L_{\lceil n/k \rceil})$. 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 recursive calls from the perspective of p . Grantson et al. showed that there are at most $3k/2$ candidate lines passing through a given point in the plane [Grantson and Levcopoulos 2006]. We use this observation to create a new bounded-depth search tree, which has a virtual structure as shown in Fig. 3. 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 $L_{\lceil n/k \rceil}$, i.e., 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 ( $S', k'$ )  $\leftarrow$  THE PREPROCESSING ALGORITHM( $S, k$ )
2 Construct  $L_3$ , a set of all lines through 3 or more points in  $S'$ .
3 while there exist  $\{p_1, p_2\} \in \text{cover}(L_3)$  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  $S' \leftarrow S' \setminus \text{cover}\{\overline{p_1, p_2}\}$ ;  $k' \leftarrow k' - 1$ ;
5 return AROUNDTHEPOINTS ( $S', k'$ )

```

AROUNDTHEPOINTS(S', k')

```

1  $n' \leftarrow |S'|$ ;  $T \leftarrow \emptyset$ 
2 if  $n' \leq 2k'$ 
3   then return true
4 if  $k' > 0$ 
5   then  $L_{\lceil n'/k' \rceil} \leftarrow$  the set of all lines covering at least  $\lceil n'/k' \rceil$  points
6      $C \leftarrow \text{cover}(L_{\lceil n'/k' \rceil})$ 
7     Sort  $C$  in descending order of degree where
8      $\text{degree}(p) \stackrel{\text{def}}{=} |\{l \in L_{\lceil n'/k' \rceil} | p \in l\}|$ 
9     for each  $p \in C$  in descending degree
10    do
11      for each  $l \in L_{\lceil n'/k' \rceil}$ ,  $p \in l$ , and  $l \notin T$ 
12        do
13           $T \leftarrow T \cup \{l\}$  (*lines that were tested before*)
14          if AROUNDTHEPOINTS( $S' \setminus \text{cover}(l), k' - 1$ )
15            then return true
16 return false

```

3.4.1 *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 consider the AROUNDTHEPOINTS function which solves the reduced instance (S', k') . The search tree in Fig. 3 has a depth of at most k . 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 candidate lines covering at least $\lceil n'/k' \rceil$ points which is at most $3k^2/2$. In fact, the analysis of the size of (number of nodes in) the tree emulates the analysis of Grantson et al. [2006] since we apply a hashing scheme to avoid testing candidate lines that are already tested (the variable T in the pseudo-code). Thus, we can consider $3k^2/2$ nodes in the first level of recursion. In the next recursive call, we will have at most $(3/2)^2[k(k-1)]^2$. After k recursive calls, we will have at most $(3/2)^k(k!)^2$ nodes which is simplified to at most $(k/2.22)^2 k$ nodes (using also Stirling's approximation [Grantson and Levcopoulos 2006]).

The workload at each node consists of the time required to compute $L_{\lceil n'/k' \rceil}$, 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 done in $O(k^4)$ time. Sorting the points in $\text{cover}(L_{\lceil n'/k' \rceil})$ requires $O(k^2 \log k)$ time since

there are at most k^2 points in $\text{cover}(L_{\lceil n'/k' \rceil})$. Finally, building a list of candidate lines around each point in S' takes $O(k^3)$ time because $|S'| \leq k^2$ 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 or $O(k^4(k/2.22)^{2k})$ time for all the nodes. Thus, the running time of PRIORITIZE POINTS IN HEAVY LINES is $O(n \log k + k^4(k/2.22)^{2k})$.

4. ALGORITHMS FOR THE OPTIMIZATION PROBLEM

4.1 Search with increments of one

We now present 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 one.

ALGORITHM ONE INCREMENTS(S)

```

1   $k_0 \leftarrow 1$ ; success  $\leftarrow$  false
2  while (not success)
3      do
4          success  $\leftarrow$  call PRIORITIZE POINTS IN HEAVY LINES( $S, k_0$ )
5          if (not success) then  $k_0 \leftarrow k_0 + 1$ 
6  return ( $k \leftarrow k_0$ )

```

4.1.1 *Running Time Analysis.* 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 in $O(nk \log k + k^5(k/2.22)^{2k})$. But, this is the worst case analysis. In practice we expect only a constant number of calls where a kernel is resolved since many of the calls that fail are determined by a reduction rule, and not by analyzing the kernel.

4.2 Guessing a lower bound

We explore an alternative way to search 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 one. 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^* , and it can be set as the number of lines that Reduction Rule 6 can be applied upon.

4.2.1 *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, Reduction Rule 6 can be carried out in $O(n^3)$ 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 (i.e. $k_0 \leftarrow k_0 + 1$ in the pseudo-code). We saw earlier that 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 $O(n^3 + k^5(k/2.22)^{2k})$.

ALGORITHM TAKE A GUESS(S)

```

1   $k_0 \leftarrow 1; k^* \leftarrow -1; \text{success} \leftarrow \text{false}$ 
2  while (not success)
3      do
4          if  $k_0 > 5$  and  $k^* = -1$           (*value of  $k^*$  is computed once*)
5              then  $k^* \leftarrow 0$ 
6              construct  $L_3$ , a set of all lines with 3 points or more
7              while there exist  $\{p_1, p_2\} \in \text{cover}(L_3)$  such that no other
              line in  $L_3$  besides  $\overline{p_1, p_2}$  covers  $p_1$  or  $p_2$ 
8                  do  $k^* \leftarrow k^* + 1$     (*Reduction Rule 6 applies*)
9                  if  $k^* > k_0$  then  $k_0 \leftarrow k^*$ 
10 success  $\leftarrow$  call PRIORITIZE POINTS IN HEAVY LINES( $S, k_0$ )
11 if (not success) then  $k_0 \leftarrow k_0 + 1$ 
12 return ( $k \leftarrow k_0$ )

```

4.3 Binary search

An alternative way to search for the value of k is to use binary 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 . Then we search in the range with $k_0 + 1$ as the lower bound and $2k_0$ as the upper bound.

4.3.1 *Running Time Analysis.* The running time consists of the time for finding the upper bound ($2k_0$) and the time for binary search.

The first loop stops when $2k_0$ results in a successful call to PRIORITIZE POINTS IN HEAVY LINES. The number of calls to this subroutine, is $1 + \log 2k_0$ (note that $k \leq 2k_0 < 2k$). The running time is thus $(1 + \log 2k_0)(n \log 2k_0 + (2k_0)^4 (2k_0/2.22)^{4k_0}) = O(n \log^2 k + k^4 \log k (k/1.11)^{4k})$. The binary search is executed $\log k_0$ times. The running time here is $O(n \log^2 k + k^4 \log k (k/1.11)^{4k})$. Therefore, the overall time complexity of BINARY SEARCH is $O(n \log^2 k + k^4 \log k (k/1.11)^{4k})$.

5. 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 I summarizes the time complexities of the ten algorithms. Algorithm CASCADE-AND-KERNELIZE is the same as Algorithm KERNELIZE 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 than k points, we also look for a line covering one point shorter and repeat this step until no line is found. Note that in the original KERNELIZE [Langerman and Morin 2005], the value of k remains constant. We believe that this strategy might slightly improve the performance of KERNELIZE. EXACTLINECOVER is described in Grantson and Levkopoulos's paper [2006]. However, in the actual implementation of this algorithm, we replace the calls to Guibas et al.'s algorithm [1996] with the calls to the simple version of Guibas described earlier. Since these are the worst-case complexities they include some constant factors that are non-trivial in any implementation and also they may exhibit very different performances in a variety of instances. Therefore, we conducted an experimental

Table I. Summary of the time complexities of ten algorithms.

Algorithms	Time Complexity	Problem
CASCADE-AND-SORT (Sec. 3.2)	$O(n \log k + k^{2k+2})$	Decision
CASCADE FURTHER (Sec. 3.3)	$O(n \log k + k^{2k+2})$	Decision
PRIORITIZE POINTS IN HEAVY LINES (Sec. 3.4)	$O(n \log k + k^4(k/2.22)^{2k})$	Decision
BST-DIM-SET-COVER*	$O(k^{2k}n)$	Decision
KERNELIZE*	$O(n^3 + k^{2k+2})$	Decision
CASCADE-AND-KERNELIZE*	$O(n^3 + k^{2k+2})$	Decision
ONE INCREMENTS (Sec. 4.1)	$O(nk \log k + k^5(k/2.22)^{2k})$	Optimization
TAKE A GUESS (Sec. 4.2)	$O(n^3 + k^5(k/2.22)^{2k})$	Optimization
BINARY SEARCH (Sec. 4.3)	$O(n \log^2 k + k^4 \log k(k/1.11)^{4k})$	Optimization
EXACTLINECOVER†	$O(n \log k + k^{2k+2})$	Optimization

* [Langerman and Morin 2005]
† [Grantson and Levcopoulos 2006]

evaluation. We implemented the ten algorithms⁵ using the GNU C++ Compiler version 4.1.2 and the CGAL’s library [CGAL Editorial Board 2007]. All experiments are performed on a computer with an Intel Pentium(R) 4 processor, at 2.40 GHz with 512 MB of RAM. We consider this machine as a standard PC. We read the input points from a file and we shuffle once (with all permutations being equally likely) the input points 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) [CGAL Editorial Board 2007] and not real time.

5.1 Performance on Random Instances

5.1.1 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 firstly generating k random points $\{s_1, \dots, s_k\}$ 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 belong 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 average running times after executions of these files. We tested four combinations for the values of k and n and the results are given in Table II and Table III. The data from Table II is also presented graphically in Fig. 4 as an illustration. Note that all results regarding observed averages in this paper are reported with 95% confidence intervals.⁶

⁵The source code and the data of the problem instances are available upon request.

⁶A 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. In this case,

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

Algorithm	k n	Running Time		
		6		
		30	50	5000
CASCADE-AND-SORT		$0.38 \pm 0.03s$	$0.41 \pm 0.05s$	$1.36 \pm 0.07s$
CASCADE FURTHER		$0.40 \pm 0.04s$	$0.41 \pm 0.04s$	$1.33 \pm 0.08s$
PRIORITIZE POINTS IN HEAVY LINES		$0.41 \pm 0.04s$	$0.44 \pm 0.05s$	$1.34 \pm 0.07s$
BST-DIM-SET-COVER		$223 \pm 90s$	$525 \pm 126s$	$\approx 15hr$
KERNELIZE		$2.71 \pm 1.58s$	$0.17 \pm 0.09s$	$6.81 \pm 0.76s$
CASCADE-AND-KERNELIZE		$0.12 \pm 0.06s$	$0.07 \pm 0.03s$	$6.59 \pm 0.61s$

Algorithm	k n	Running Time		
		7		
		30	50	5000
CASCADE-AND-SORT		$0.98 \pm 0.14s$	$0.70 \pm 0.05s$	$1.71 \pm 0.12s$
CASCADE FURTHER		$1.17 \pm 0.19s$	$0.72 \pm 0.05s$	$1.64 \pm 0.10s$
PRIORITIZE POINTS IN HEAVY LINES		$1.18 \pm 0.20s$	$0.74 \pm 0.07s$	$1.72 \pm 0.13s$
BST-DIM-SET-COVER		$\approx 3hr$	$\approx 11hr$	$\gg 24hr$
KERNELIZE		$\approx 3hr$	$2.19 \pm 1.99s$	$7.72 \pm 0.51s$
CASCADE-AND-KERNELIZE		$\approx 3hr$	$0.21 \pm 0.11s$	$8.69 \pm 1.05s$

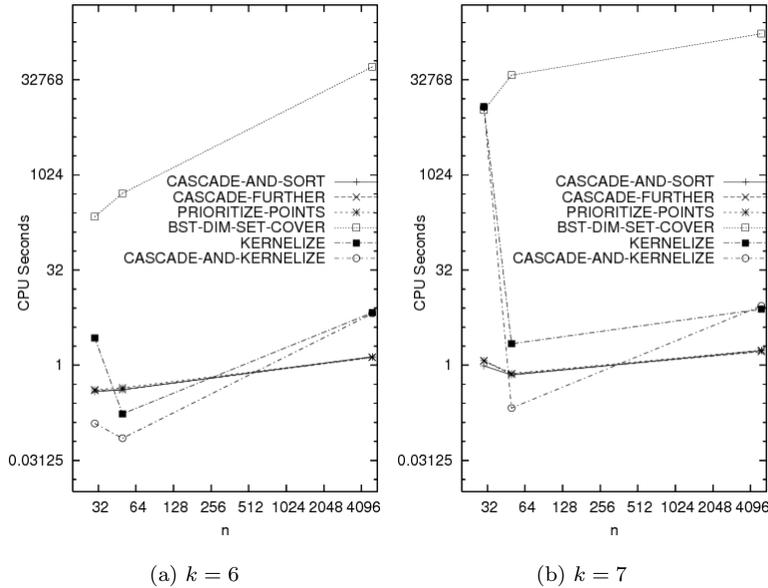


Fig. 4. Running time versus the instance size for random YES-instances.

5.1.2 *Discussion.* One rapidly discovers the pattern. Simply, the implementation of our algorithms requires CPU time of the order of seconds, while theoretical algorithms without our simple reduction rules require CPU time of 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 points are assigned to lines by the generation process with equal probability, so the expected number of

it indicates that the expected running time of the algorithm on an instance drawn as described has 95% probability of falling inside the interval. If the confidence intervals are disjoint, we have statistical significance that one algorithm's CPU time is smaller than the other.

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

Algorithm	k n	Running Time		
		5		
		30	50	5000
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
Algorithm	k n	Running Time		
		6		
		30	50	5000
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

Table IV. 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).

Algorithm	k n	Kernel's Size		Running Time (min)	
		30	5	30	5
		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

points on any given line is greater than k . The cascading is also effective and lines that have 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). Naturally, our three algorithms perform slightly poorer 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 in order to ensure the exhaustive application of the reduction rules in the kernelization phase.

When $n < k^2$ and k is sufficiently large, our three algorithms solve random YES-instances relatively fast. 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 very differently, depending on the values of (n/k) . If the average number of points on any given line is a lot 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 (reducing k' further). This illustrates again the power of more reduction rules, even if they seem simple and not producing a theoretically smaller kernel. Similar results are obtained in the case of NO-instances except that CASCADE-AND-SORT does not perform as well as the other two algorithms of ours. This is not surprising because

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

Algorithm	k n	Running Time			
		6		7	
		30	50	30	50
ONE INCREMENTS		$0.9 \pm 0.07s$	$0.8 \pm 0.04s$	$2.5 \pm 0.39s$	$1.7 \pm 0.10s$
TAKE A GUESS		$1.3 \pm 0.06s$	$2.0 \pm 0.04s$	$2.7 \pm 0.40s$	$2.4 \pm 0.09s$
BINARY SEARCH		$2.1 \pm 0.24s$	$1.6 \pm 0.10s$	$3.6 \pm 0.46s$	$2.3 \pm 0.18s$
EXACTLINECOVER		$2.4 \pm 0.42s$	$5.1 \pm 0.18s$	$\approx 7hr$	$6.4 \pm 0.29s$

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

Algorithm	k n	Running Time (min)		
		30	10	5
		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

CASCADE-AND-SORT invokes BST-DIM-SET-COVER after the kernelization phase failed to detect the NO-instances.

The above results show that implementation of 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 IV). The random instances are generated with the same procedure as before. The value of k used for instance generation is the same as the parameter k of the inputs. The result shows that our algorithms solve the instances in 5 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 subsection we generate structured instances and create the situations that disallow some of these reduction rules to work in order to see how our algorithms perform (we discuss this later).

5.1.3 Algorithms for the optimization problem. Naturally, we use the instances generated in this random model above to also evaluate algorithms for the optimization version of the problem. We organized the experiments as before. 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.

Results for the optimization problem are shown in Table V. Again, our implementations are several orders of magnitude more CPU-time efficient than previous algorithms, which are not practical for values like $k = 7$ and $n = 30$. Exploring the limits of how far we can take our implementations, we also considered larger values of k and n . Table VI shows the corresponding running times.

5.1.4 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 Algorithm BINARY SEARCH actually

pays off when k is big (see Table VI). It reduces the running time of Algorithm ONE INCREMENTS by 42% when $k = 30, n = 900$. Nevertheless, Algorithm ONE INCREMENTS still runs reasonably fast for k up to 30, and n up to 100,000 or higher. 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, 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 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 above type of instances since 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, along with instances where the rules do apply efficiently. We call these *structured instances* because they are generated under a certain prototype.

5.2 Performance on Structured Instances

5.2.1 Algorithms for the decision problem. We generate the instances along eight profiles. Fig. 5 illustrates the profiles of the 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 Fig. 5. As a representative of each profile, we typically generate one input file since 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 deviated 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 which lie on k horizontal lines (Fig. 5(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 $n \gg k^2$. Typically we will test our algorithms in at least 3 files of 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 algorithms using this profile.
- (2) The Elongated-Grid (Fig. 5(b)) profile is similar to the Small- k instance-profile and also has $n \gg k^2$. 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 kernelization stops because the input size is already smaller than the square of the parameter. (Fig. 5(c)).
- (4) The Grid profile has data points in each vertex of a square grid with k horizontal and k vertical lines; thus $n = k^2$ (Fig. 5(d)). We test this profile when k is 5, 6 and 7 (n is 25, 36 and 49 respectively). This is usually what we obtain as a

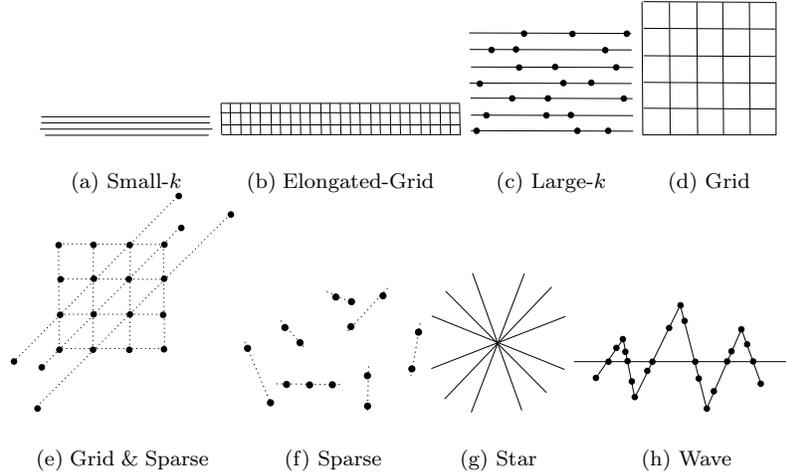


Fig. 5. Profiles for structured instances.

quadratic kernel, and 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 (Fig. 5(e)) is a case where besides 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 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 (Fig. 5(f)). The input size n is chosen such that $2k < n < 3k$.
- (7) The Star profile (Fig. 5(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 so no point can be covered by several candidate lines as it can in the grid.
- (8) The Wave profile (Fig. 5(h)) tests the case where one line which 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. For YES-instances, we make sure that all generated instances above are covered by a minimum number of lines that is equal to the value k used as input for the algorithm. For NO-instances, we reduce the value of k in the corresponding YES-instances by one. Results appear in Table VII and Table VIII. 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 IX). We did tests on the grid-type instances, but, the results are not obvious

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

Algorithm	k n	Input Profile						
		Small- k			Large- k	Elongated-Grid		
		4 100	4 1,000	4 10,000	7 21	4 100	4 1,000	4 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
Algorithm	k n	Input Profile (cont.)						
		Grid			Grid & Sparse	Sparse	Star	Wave
		5 25	6 36	7 49	6 22	7 15	6 36	6 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 VIII. Average running times for the decision problem (10 shuffles on the same input file for a NO-instance).

Algorithm	k n	Input Profile						
		Small- k			Large- k	Elongated-Grid		
		3 100	3 1,000	3 10,000	6 21	3 100	3 1,000	3 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
Algorithm	k n	Input Profile (cont.)						
		Grid			Grid & Sparse	Sparse	Star	Wave
		4 25	5 36	6 49	5 22	6 15	5 36	5 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

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

Algorithm	Running Time	
	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

for us to comment on the three algorithms. Therefore, we select a new structured instance called Incomplete-Grid (see Fig. 6(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. We also justify the earlier claim that PRIORITIZE POINTS IN HEAVY LINES performs better

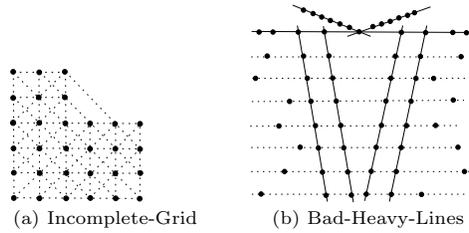


Fig. 6. Two special structured-instances for testing the features of our algorithms.

Table X. Average running times of Algorithm PRIORITIZE POINTS IN HEAVY LINES with and without sorting feature (10 shuffles on the Bad-Heavy-Lines instance).

Algorithm	Running Time	
	YES-instance ($k = 10, n = 63$)	NO-instance ($k = 9, n = 63$)
PRIORITIZE POINTS IN HEAVY LINES		
with sorting	$22.5 \pm 3s$	$145.8 \pm 0.6s$
without sorting	$77.4 \pm 22s$	$145.2 \pm 0.6s$

Table XI. 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

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 of Fig. 6(b) where in this instance we set the scene such that 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 the better chance of cutting out branches of the search tree, hence PRIORITIZE POINTS IN HEAVY LINES runs even faster. This result appears in Table X. 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. 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 around half an hour on a standard PC. Table XI 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.

5.2.2 *Discussion.* First let us discuss the case of YES-instances. Our three algorithms can handle larger values of k and n than the known algorithms. Our algorithms achieve the best result 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 of our three algorithms is also less than those of known algorithms. 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). Algorithm CASCADE FURTHER that has one more reduction rule than CASCADE-AND-SORT solves the Wave instance-type

Table XII. Average running times for the optimization problem (10 shuffles on the same input file).

Algorithm	k n	Input Profile						
		Small- k			Large- k	Elongated-Grid		
		4 100	4 1,000	4 10,000	7 21	4 100	4 1,000	4 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

Algorithm	k n	Input Profile (cont.)						
		Grid			Grid & Sparse	Sparse	Star	Wave
		5 25	6 36	7 49	6 22	7 15	6 36	6 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

200 times faster than CASCADE-AND-SORT. This reveals the power of our Reduction Rule 6. Algorithm PRIORITIZE POINTS IN HEAVY LINES requires less than a second to decide the NO-instances.

Let us look at the difference in performance among our three algorithms. The Incomplete-Grid instance-type in Fig. 6(a) represents the situation when the sorting strategy does not work really well, because here 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 IX). Moreover, there is a large variance in the performance of CASCADE-AND-SORT and CASCADE FURTHER while the variance is very small in Algorithm PRIORITIZE POINTS IN HEAVY LINES. 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. Algorithm PRIORITIZE POINTS IN HEAVY LINES also incorporates another strategy, that 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 X confirms that this strategy indeed improves the performance of this algorithm for 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 $k = 6$.

5.2.3 *Algorithms for the optimization problem.* We have tested the decision version of our algorithms with profiles such as those of Fig. 5. When we look at the optimization version with the same profiles, we get the results as shown in Table XII.

5.2.4 *Discussion.* Our three algorithms outperform EXACTLINECOVER in all cases. Algorithm EXACTLINECOVER is no longer practical when k is 7 as it solves the instances Sparse, Grid and Large- k in 2 hours, 4 hours and 6 hours respectively; our three algorithms solve them in less than 10 seconds.

5.3 Implementation Issues

In our experiments, when we read the input points, we also removed the duplicated points. This step was performed in all algorithms to ensure that all algorithms were working under the same assumptions. This step was not reflected at all 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.

Duality mapping is not defined for vertical lines. Therefore, implementation of our algorithm and Grantson and Levcopoulos's algorithm (as well as any algorithms that use Guibas et al. [1996]) need attention to this aspect. The solution is to rotate the scene so that there are no vertical lines [de Berg et al. 2000]. We did not include this work in our timings.

6. CONCLUSIONS

This paper presented efficient FPT-algorithms for the problem of covering a set S of n points in the plane with k lines. We also gave algorithms that solve the optimization version of the problem. We implemented all of our algorithms and four other algorithms found in the literature. Note that these algorithms provide exact solutions. In this paper 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 benefit in implementation although theoretically do not produce a smaller kernel. We solved the kernel using a bounded-search-tree technique with a new heuristic that looks at 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 improved significantly the running time in practice. Based on our experimental evidence we suggest the following. Algorithm ONE INCREMENTS is ideal for applications with small k value (less than 20). Algorithm BINARY SEARCH should be used when k is large.

REFERENCES

- AGARWAL, P. 1990. Partitioning arrangements of lines II: applications. *Discrete and Computational Geometry* 5, 533–573.
- APPLEGATE, D., BIXBY, R., CHVÁTAL, V., AND COOK, W. 2006. *The Traveling Salesman Problem*. Princeton University Press, Princeton, NJ, USA.
- ARKIN, E., BENDER, M., DEMAINE, E., FEKETE, S., MITCHELL, J., AND SETHIA, S. 2005. Optimal covering tours with turn costs. *SIAM J. Comput.* 35(3), 531–566.
- ARKIN, E., FEKETE, S., AND MITCHELL, J. 2000. Approximation algorithms for lawn mowing and milling. *Comput. Geom.* 17(1-2), 25–50.
- CGAL EDITORIAL BOARD. 2007. CGAL User and Reference Manual, 3.3 edition. http://www.cgal.org/Manual/3.3/doc.html/cgal_manual/packages.html.
- DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. 2000. *Computational Geometry: Algorithms and Applications*, 2nd ed. Springer, Berlin.
- DEŇNEKO, V. AND WOEGINGER, G. 1996. The convex-hull-and- k -line traveling salesman problem. *Inf. Process. Lett.* 59(6), 295–301.
- DOWNEY, R. AND FELLOWS, M. 1999. *Parameterized Complexity*. Monographs in Computer Science. Springer, New York.
- DRYSDALE, R., STEIN, C., AND WAGNER, D. 2005. An $O(n^{5/2} \log n)$ algorithm for the rectilinear minimum link-distance problem. In *17th Canadian Conference on Computational Geometry*. University of Windsor, Ontario, Canada, 97–100.
- EDELSBRUNNER, H., SEIDEL, R., AND SHARIR, M. 1993. On the zone theorem for hyperplane arrangements. *SIAM J. Comput.* 22(2), 418–429.

- FLUM, J. AND GROHE, M. 2006. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, Berlin.
- GIANOPOULOS, P., KNAUER, C., AND WHITESIDES, S. 2008. Parameterized complexity of geometric problems. *Comput. J.* 51, 3, 372–384.
- GRANTSON, M. AND LEVCOPOULOS, C. 2006. Covering a set of points with a minimum number of lines. In *Algorithms and Complexity, 6th Italian Conference, CIAC*. LNCS, vol. 3998. Springer, Berlin, 6–17.
- GUIBAS, L., OVERMARS, M., AND ROBERT, J. 1991. The exact fitting problem for points. In *3rd Canadian Conference on Computational Geometry*. Simon Fraser University, Vancouver, British Columbia, 171–174.
- GUIBAS, L., OVERMARS, M., AND ROBERT, J. 1996. The exact fitting problem in higher dimensions. *Computational Geometry: Theory and Applications* 6, 215–230.
- HÜFFNER, F., NIEDERMEIER, R., AND WERNICKE, S. 2008. Techniques for practical fixed-parameter algorithms. *Comput. J.* 51(1), 7–25.
- KUMAR, V., ARYA, S., AND RAMESH, H. 2000. Hardness of set cover with intersection 1. In *27th International Colloquium on Automata, Languages and Programming, ICALP*. LNCS, vol. 1853. Springer, Berlin, 624–635.
- LANGERMAN, S. AND MORIN, P. 2001. Cover points with lines. In *11th Fall Workshop on Computational Geometry*. Polytechnic University, Brooklyn, NY, USA.
- LANGERMAN, S. AND MORIN, P. 2002. Cover things with things. In *10th European Symposium on Algorithms*. LNCS, vol. 2641. Springer, Berlin, 662–673.
- LANGERMAN, S. AND MORIN, P. 2005. Cover things with things. *Discrete and Computational Geometry* 33(4), 717–729.
- LEE, D., YANG, C., AND WONG, C. 1994. On bends and distances of paths among obstacles in two-layer interconnection model. *IEEE Trans. Comput.* 43(6), 711–724.
- LEE, D., YANG, C., AND WONG, C. 1996. Rectilinear paths among rectilinear obstacles. *Discrete Applied Mathematics* 70(3), 185–215.
- MATOUŠEK, J. 1990. Cutting hyperplane arrangements. In *6th Annual Symposium on Computational Geometry*. ACM, New York, NY, USA, 1–9.
- MEGIDDO, N. AND TAMIR, A. 1982. On the complexity of locating linear facilities in the plane. *Operations Research Letters* 1(5), 194–197.
- NIEDERMEIER, R. 2006. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and its Applications 31. Oxford University Press, New York.
- ROTE, G. 1992. The N-line traveling salesman problem. *Networks* 22, 91–108.
- STEIN, C. AND WAGNER, D. 2001. Approximation algorithms for the minimum bends traveling salesman problem. In *8th International IPCO Conference*. LNCS, vol. 2081. Springer, Berlin, 406–422.

Received Month Year; revised Month Year; accepted Month Year