Translating a Regular Grid over a Point Set*

Prosenjit Bose [§] Marc van Kreveld[¶] Anil Maheshwari[§]
Pat Morin[§] Jason Morrison[§]

Abstract

We consider the problem of translating a (finite or infinite) square grid G over a set S of n points in the plane in order to maximize some objective function. We say that a grid cell is k-occupied if it contains k or more points of S. The main set of problems we study have to do with translating an infinite grid so that the number of k-occupied cells is maximized or minimized. For these problems we obtain running times of the form $O(kn \text{ polylog}(n))$. We also consider the problem of translating a finite size grid, with m cells, in order to maximize the number of k-occupied cells. Here we obtain a running time of the form $O(knm \text{ polylog}(nm))$. In solving these problems, we design a data structure T that maintains in $O(\log n)$ time per operation, a function $f : \mathbb{R} \to \mathbb{R}$ under the following query and update operations where $[a, b)$ is a continuous interval in $\mathbb{R}$.

1. Insert(T, a, b, δ): Increase the value of $f(x)$ by δ for all $x \in [a, b)$.
2. Delete(T, a, b, δ): Decrease the value of $f(x)$ by δ for all $x \in [a, b)$.
3. Max-Cover(): Return $\max\{f(x) : x \in \mathbb{R}\}$.
4. Max-Cover-Witness(): Return a value x^* such that $f(x^*) = \max\{f(x) : x \in \mathbb{R}\}$.
5. Max-In(a, b): Returns $\max\{f(x) : x \in [a, b)\}$.
6. Max-Witness-In(a, b): Returns a value x^* such that $f(x^*) = \max\{f(x) : x \in [a, b)\}$.

as well as the Min counter-parts of these queries.

1 Introduction

In this paper we consider the problem of translating a (finite or infinite) square grid G over a set S of n points in the plane so that some objective function is

*This work was partly funded by the Natural Sciences and Engineering Research Council of Canada and the Dutch Organization for Scientific Research (N.W.O).

[§]School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa, Canada, K1S 5B6, {jit,maheshwari,morin,morrison}@scs.carleton.ca .

[¶]Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB, Utrecht, The Netherlands, marc@cs.uu.nl .

optimized. For a grid cell C of G we say that C is *k-occupied* if C contains at least k points of S. The *k-occupancy* of G is the number of cells of G that are k-occupied. A cell is simply *occupied* if it is 1-occupied and the *occupancy* denotes the 1-occupancy.

For infinite grids, we consider both minimization and maximization versions of the following problems where G is free to be translated by any vector and k is part of the input:

Problem 1. *Find a translation of G that optimizes the occupancy of G.*

Problem 2. *Find a translation of G that optimizes the k-occupancy of G.*

Problem 3. *Find a translation of G (if one exists) so that no cell of G is k-occupied.*

We also consider the following problems of optimizing k:

Problem 4. *Find the smallest integer k such that there exists a translation of G with k-occupancy 0.*

Problem 5. *Find the largest integer k such that there exists a translation of G with k-occupancy at least r.*

For a finite grid G with m cells and a fixed integer k we also consider the following problem:

Problem 6. *Find a translation of G that maximizes k-occupancy.*

This problem arises in several applications. For example, a problem from cartography is that of laying out maps into books, where one wants to avoid book pages with no features or with too many features. In the problem of gridding/interpolation one wants to derive a regular grid from irregular data. In order to minimize the interpolation error one would like to ensure that every grid cell contains at least 1 point to interpolate from. Finally, in the design of hash functions [2, 5], one wants to minimize the maximum number of points contained in any bucket.

The main tools we use to solve these problems are plane sweep [4] and a dynamic data structure for maintaining the maximum clique in an interval graph under insertions and deletions of intervals. Our approach is to reduce the grid translation problem to that of finding a maximally (or minimally) covered point in an arrangement of rectangles and then apply plane sweep and our data structure to find this point.

The previous work most closely related to our results is the work of Lee [10] on finding the maximum clique in a rectangle intersection graph. A rectangle intersection graph $G = (V, E)$ has vertices V, $|V| = n$, such that each vertex represents one of n iso-oriented rectangles. Every edge in E corresponds to a pair of rectangles that intersect and all such intersections are represented in E. Lee also uses plane sweep along with a data structure for maintaining the *size* of a maximum clique in an interval graph. The data structure is based on segment trees [3] and has $O(\log n)$ time for insertions, deletions and queries.

Problem	Running Time	Memory
Problem 1	$O(n \log n)$	$O(n)$
Problem 2	$O(kn \log n)$	$O(kn)$
Problem 3	$O(kn \log n)$	$O(kn)$
Problem 4	$O(kn \log n \log k)$	$O(kn)$
Problem 5	$O(kn \log n \log k)$	$O(kn)$
Problem 6	$O(kmn \log mn)$	$O(kn)$

Table 1: Summary of results.

As we were previously unaware of Lee's work, we developed a data structure based on prefix-sum trees that also has $O(\log n)$ running time for all operations. This data structure seems to be more flexible than the segment tree data structure. For example, it is fully dynamic, not requiring that the endpoints of the segments be known in advance. It can also report a value that is contained in the largest number of intervals. It is unclear whether the segment tree data structure can be extended to do either of these without increasing the running time. We conjecture that this data structure will find applications beyond those in this paper.

Several other researchers have considered grid placement problems in which rotations and resizing of the grid cells can take place [1, 2]. In these applications the objective function that is minimized is usually the maximum number of points that fall in any given cell. Because of the increased number of degrees of freedom, the running times of the algorithms are quite high (usually close to quadratic in n).

In comparison, we study a constrained version of the problem where only rigid translation is allowed. With this constraint we are able to consider variations with k-occupancy rather than just 1-occupancy. We obtain the low-order running times and memory requirements shown in Table 1. All results are valid for grid cells of arbitrary aspect ratio and scale. When we do use the idea of square unit size cells in our descriptions it is only for the sake of clarity and no results are affected by the change.

The remainder of this paper is organized as follows: In Section 2 we describe our new data structure for maintaining the maximum clique in an interval graph. In Section 3 we show how to use this data structure to solve Problems 1–6. In Section 4 we summarize and conclude with open problems.

2 The Data Structure

In this section we describe a data structure T that maintains a function $f : \mathbb{R} \to \mathbb{R}$ under the following query and update operations where $[a, b)$ is a continuous interval in $\mathbb{R}$.

1. Insert(T, a, b, δ): Increase the value of $f(x)$ by δ for all $x \in [a, b)$.

2. $\text{DELETE}(T, a, b, \delta)$: Decrease the value of $f(x)$ by δ for all $x \in [a, b)$.

3. $\text{MAX-COVER}()$: Return $\max\{f(x) : x \in \mathbb{R}\}$.

4. $\text{MAX-COVER-WITNESS}()$: Return a value x^* such that $f(x^*) = \max\{f(x) : x \in \mathbb{R}\}$.

5. $\text{MAX-IN}(a, b)$: Returns $\max\{f(x) : x \in [a, b)\}$.

6. $\text{MAX-WITNESS-IN}(a, b)$: Returns a value x^* such that $f(x^*) = \max\{f(x) : x \in [a, b)\}$.

We describe the data structure with respect to half-open intervals of the form $[a, b)$ solely for the purpose of simplicity of description. Once we have described the data structure, it is easy to see how to modify the data structure to handle open and closed intervals as well as infinite and semi-infinite intervals. The main reason we differentiate between MAX-COVER() and MAX-COVER-WITNESS() is that we can answer the former query in $O(1)$ time and the latter query in $O(\log n)$ time. Furthermore, a simple modification of this data structure answers the minimization queries MIN-COVER(), MIN-COVER-WITNESS(), $\text{MIN-IN}(a, b)$, and $\text{MIN-WITNESS-IN}(a, b)$ without any difficulties.

The data structure we use is a binary search tree T whose root is denoted by root. Each node v of T stores an interval. We denote by $\text{INT}(v)$ the interval stored at v, and by $\text{START}(v)$, respectively $\text{STOP}(v)$, the left, respectively right, endpoint of $\text{INT}(v)$. For an internal node v, the left, respectively right, child of v is denoted by $\text{LEFT}(v)$, respectively $\text{RIGHT}(v)$. The interval of an internal node v is always $\text{INT}(v) = \text{INT}(\text{LEFT}(v)) \cup \text{INT}(\text{RIGHT}(v))$. Initially, T contains one node root with $\text{INT}(\mathsf{root}) = [-\infty, +\infty)$.

In addition to this information, each node v of T also maintains two values, $\text{STAB}(v)$ and $\Delta(v)$. Let $K(v) = \max\{f(x) : x \in \text{INT}(v)\}$ so that $K(\mathsf{root}) = \text{MAX-COVER}()$. Intuitively, we want the value of $\text{STAB}(v)$ to be $K(v)$. Maintaining this property as an invariant would take linear time for each INSERT and DELETE operation. Instead, we maintain the following less strict invariant.

Invariant 1. *For every node v of T, $K(v) = \text{STAB}(v) + \sum_{u \in \mathsf{Path}(v)} \Delta(v)$, where $\mathsf{Path}(v)$ is the set of nodes (including v) on the path from v to the root of T.*

An implication of Invariant 1 is that an answer to MAX-COVER queries can be given in constant time since $K(\mathsf{root}) = \text{STAB}(\mathsf{root}) + \Delta(\mathsf{root})$. Furthermore, for a node v, if $\Delta(u) = 0$ for all nodes u on the path from v to root then $\text{STAB}(v) = K(v)$.

Our strategy for maintaining the STAB information is lazy because we postpone the update of any node values whenever possible. van Kreveld and Overmars [11] use a similar mechanism in the design of concatenable segment trees. When updating becomes necessary, the procedure that we use to update values at internal nodes of T is $\text{PUSH-DELTA}(v)$, defined as follows.

PUSH-DELTA(v)

$\text{STAB}(v) \leftarrow \text{STAB}(v) + \Delta(v)$
$\Delta(\text{LEFT}(v)) \leftarrow \Delta(\text{LEFT}(v)) + \Delta(v)$
$\Delta(\text{RIGHT}(v)) \leftarrow \Delta(\text{RIGHT}(v)) + \Delta(v)$
$\Delta(v) \leftarrow 0$

Note that, if Invariant 1 holds for node v and its two children before calling PUSH-DELTA(v), then it continues to hold after calling PUSH-DELTA(v).

2.1 Query Operations

Of the query operations, the easiest to implement is MAX-COVER() which returns the maximum $f(x)$ in T. This is trivially implemented by Invariant 1, in constant time since MAX-COVER() $= K(\mathsf{root}) = \text{STAB}(\mathsf{root}) + \Delta(\mathsf{root})$.

The procedure MAX-COVER-WITNESS() requires a little more care, as it must return a witness to the maximum.

MAX-COVER-WITNESS()

1: $v \leftarrow r$
2: **while** v is not a leaf **do**
3:    PUSH-DELTA(v)
4:    **if** $\text{STAB}(\text{LEFT}(v)) + \Delta(\text{LEFT}(v)) > \text{STAB}(\text{RIGHT}(v)) + \Delta(\text{RIGHT}(v))$ **then**
5:       $v \leftarrow \text{LEFT}(v)$
6:    **else**
7:       $v \leftarrow \text{RIGHT}(v)$
8:    **end if**
9: **end while**
10: **report** any $x^* \in \text{INT}(v)$

To see that this procedure is correct note that, before line 4 is executed, $\Delta(u) = 0$ for all $u \in \mathsf{Path}(v)$. Therefore, by Invariant 1, $K(u) = \text{STAB}(u) + \Delta(u)$ for each child u of v. During each iteration, v is reset to the child u of v that maximizes $K(u)$. It follows that the algorithm reports a point in $\text{INT}(v)$ where v is a leaf such that all values in $\text{INT}(v)$ maximize f. Since the only modifications to T are made by calls to PUSH-DELTA, which maintain Invariant 1, Invariant 1 is still true for all nodes after calling MAX-COVER-WITNESS().

Since the two queries MAX-IN(a, b) and MAX-WITNESS-IN(a, b) are similar to their counter-parts above, we simply give an overview and leave details to the reader. The query MAX-IN(a, b) proceeds as follows. Let v_a and v_b be the leaves of T containing a and b respectively. If they are the same leaf, simply report the value $K(v_a)$, otherwise compute v_s which is the lowest common ancestor of v_a and v_b in T. Now, consider the path $P(a)$ from v_s to v_a. Every time the path follows a left child at node v_i, compute $K(\text{RIGHT}(v_i))$. Similarly, for the path $P(b)$ from v_s to v_b, compute $K(\text{LEFT}(v_j))$ for every node v_j where a right child is followed. In order to compute each $K()$ value in $O(1)$ time, we need to ensure that a verification of the STAB and Δ value at these nodes is sufficient to compute the maximum over the interval represented by the node. To do this, we perform a Push-Delta at each node on the path from root to v_s, each node on $P(a)$ and each node on $P(b)$. This will ensure that the Δ value is zero for each of these nodes. This operation can be completed in $O(h)$ time where h is the height of the tree.

The maximum of these computed values is the maximum of f over the interval $[a, b)$. The reason is that the union of the intervals of the nodes verified covers precisely $[a, b)$. In order to report a witness, simply record which node gave rise to the maximum value over the interval $[a, b)$ and follow the same procedure as Max-Cover-Witness on this node. Both queries can be performed in $O(h)$ time where h is the height of the tree. In section 2.3 we show how to keep the tree balanced so that its height is $O(\log n)$.

2.2 Update Operations

Insertion and deletion operations are done using one general operation, Insert-Infinite-Interval(a, δ), where both a and δ are real numbers. Insert-Infinite-Interval(a, δ) has the effect of increasing $f(x)$ by δ for all $x \in [a, +\infty)$. It should be clear that Insert(a, b, δ) and Delete(a, b, δ) can each be implemented with two calls to Insert-Infinite-Interval (using values + or $-\delta$).

We implement Insert-Infinite-Interval by first creating a new leaf w with $\text{START}(w) = a$ if T does not already contain such a leaf. We then traverse $P(w)$ from root to w calling Push-Delta for each node we encounter. At the same time, we update the Δ values for nodes that lie to the right of $P(w)$. After doing this, Invariant 1 is maintained for all nodes except those on $P(w)$. A final traversal from w back up to root corrects the STAB and Δ values for all nodes of $P(w)$ thereby restoring Invariant 1. Finally the leaf w is checked to see if it needs to be present. If it does not then it is deleted.

These ideas are implemented by the following pseudo-code.

INSERT-INFINITE-INTERVAL(a, δ)

1: BASIC-INSERT(a)
2: $v \leftarrow r$
3: **while** v is not a leaf **do**
4: PUSH-DELTA(v)
5: **if** $a \geq$ START(RIGHT(v)) **then**
6: $v \leftarrow$ RIGHT(v)
7: **else**
8: $\Delta($RIGHT$(v)) \leftarrow \Delta($RIGHT$(v)) + \delta$
9: $v \leftarrow$ LEFT(v)
10: **end if**
11: **end while**
12: STAB$(v) \leftarrow$ STAB$(v) + \delta$
13: **repeat**
14: $v \leftarrow$ PARENT(v)
15: STAB$(v) \leftarrow \max\{$STAB(LEFT$(v)) + \Delta($LEFT$(v))$, STAB(RIGHT$(v)) + \Delta($RIGHT$(v))\}$
16: **until** $v = r$
17: CLEAN-UP(a)

We are now considering two different versions of the function f. To avoid confusion, we let f_{i} denote the function f before the operation and $f_{(\mathrm{i}+1)}$ denote the function f after the operation. Similarly, we refer to Invariant 1_{i} instead of Invariant 1 when $f = f_{\mathrm{i}}$ and to Invariant $1_{(\mathrm{i}+1)}$ when $f = f_{(\mathrm{i}+1)}$. Finally, $K_{\mathrm{i}}(v)$ and $K_{(\mathrm{i}+1)}(v)$ refer to the function $K(v)$ using $f = f_{\mathrm{i}}$ and $f = f_{(\mathrm{i}+1)}$, respectively.

The function BASIC-INSERT(a) creates a node w with START$(w) = a$ if such a node does not already exist and re-balances T. How this is accomplished is the topic of the next section. For now, we assume that after calling BASIC-INSERT(a), there exists a node v with START$(v) = a$ and Invariant 1_{i} holds for every node in T.

Claim 1. *Let w denote the value of v in Line 12. Then, by the time Line 12 is executed, Invariant $1_{(\mathrm{i}+1)}$ holds for all nodes not on $P(w)$.*

Proof. Suppose x is a node which has an ancestor u that is a node not on $P(w)$ and is a left child of a node on $P(w)$ (Fig. 1.a). Then Invariant 1_{i} and Invariant $1_{(\mathrm{i}+1)}$ are the same for x. The only changes to nodes in $P(x)$ are caused by calls to PUSH-DELTA, which maintains Invariant 1_{i}. Therefore Invariant $1_{(\mathrm{i}+1)}$ holds for x.

Suppose therefore that x has an ancestor u that is a node not on $P(w)$ but is the right child of a node on $P(w)$ (Fig. 1.b). Then $K_{(\mathrm{i}+1)}(x) = K_{\mathrm{i}}(x) + \delta$. If the Line 8 is omitted from the pseudo-code, then Invariant 1_{i} holds. Since Line 8 is included then Invariant $1_{(\mathrm{i}+1)}$ holds. □

Claim 2. *After Line 16, Invariant $1_{(\mathrm{i}+1)}$ holds for all nodes in T.*

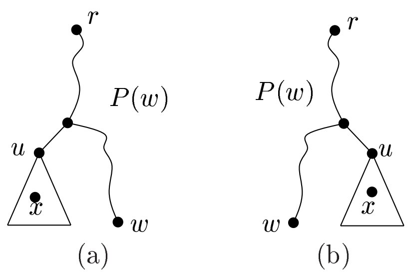

Figure 1: The two cases in the proof of Claim 1.

Proof. After Line 11, we only update STAB values of nodes in $P(w)$, thus we cannot invalidate Invariant $1_{(\mathrm{i+1})}$ for any node not on $P(w)$. Therefore, we only need to show that Invariant $1_{(\mathrm{i+1})}$ is maintained for nodes of $P(w)$.

To show that Invariant $1_{(\mathrm{i+1})}$ is maintained for a node $v \in P(w)$, we use induction on the distance between v and w to show that, after Line 15 is executed for v, Invariant $1_{(\mathrm{i+1})}$ holds for v. When $v = w$ and just before Line 12 is executed the value of $\mathrm{STAB}(v) + \Delta(v) = K_{\mathrm{i}}(v)$ due to the calls to PUSH-DELTA in Line 4l. But $K_{(\mathrm{i+1})}(v) = K_{\mathrm{i}}(v) + \delta$ and is therefore correctly updated in Line 12.

Next we consider a node v at distance greater than 0 from w. Note that every ancestor u of v (including v itself) has $\Delta(u) = 0$ (from Line 4), and Invariant $1_{(\mathrm{i+1})}$ holds for both children of v (by the inductive hypothesis and Claim 1). This implies that for each child x of v, $K_{(\mathrm{i+1})}(x) = \mathrm{STAB}(x) + \Delta(x)$. Since, by definition, $K_{(\mathrm{i+1})}(v)$ is the maximum of $K_{(\mathrm{i+1})}(\mathrm{LEFT}(x))$ and $K_{(\mathrm{i+1})}(\mathrm{RIGHT}(x))$, Invariant $1_{(\mathrm{i+1})}$ holds after the execution of Line 15. □

At this point, we see that INSERT-INFINITE-INTERVAL correctly updates the tree T to maintain Invariant 1. A simple examination of the code for INSERT-INFINITE-INTERVAL and MAX-COVER-WITNESS shows that their running time is $O(h)$ where h is the height of the tree. Two issues remain. We need to show that the number of nodes in the tree is related to the number of intervals currently defining the function f and that the height of the tree can be kept to $O(\log n)$.

We address the former first. Notice that INSERT-INFINITE-INTERVAL increases the number of nodes in the tree by one. Therefore, if we performed n insertions of the form $\mathrm{INSERT}(T, a, b, \delta)$ followed by n deletions $\mathrm{DELETE}(T, a, b, \delta)$, we would have a tree with $2n + 1$ nodes. However, all but one of these nodes would be redundant. In order to ensure that the number of nodes in the tree is related to the number of intervals currently defining the function, we use the procedure CLEAN-UP(a) to remove redundancy.

Essentially, CLEAN-UP(a) has the following form:

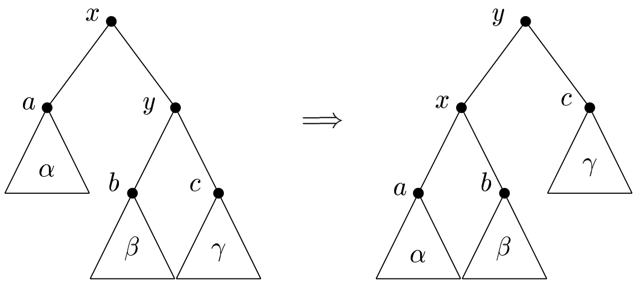

Figure 2: A left rotation

Clean-Up(a)

$v \leftarrow$ leaf representing interval $[a, b)$
$v' \leftarrow$ predecessor leaf of v representing interval $[c, a)$
if $K(v) = K(v')$ **then**
  Delete(v)
  Update v' to interval $[c, b)$
  Update all relevant nodes on the path from v' to root.
end if

After an interval (a, ∞) has been inserted with value δ it is key to note that the leaf interval v with start$(v) = a$ may have the same value $f(a)$ as its predecessor leaf. If that is true then the two intervals are merged into one single leaf by expanding the predecessor's interval and deleting the leaf. We prove in the next section that Delete(v) does not affect Invariant 1. None of the other operations affect Invariant 1.

2.3 Re-balancing

The details missing from the description of Insert-Infinite-Interval are the implementations of Basic-Insert(a) and Delete(v). Implementations of these functions are described in many algorithms texts (see Cormen *et al.*[6]) and thus details are not presented here.

Several methods exists for re-balancing binary search trees such that their height remains $O(\log n)$. Usually, these methods use *rotations* to re-balance the tree. For any of these re-balancing methods to be valid, it is necessary to show that the values stab, Δ, and int can be maintained during a rotation.

A left rotation is illustrated in Fig. 2. The following wrapper, placed around a left rotation, correctly updates the information stored at x and y. Right rotation is implemented symmetrically.

LEFT-ROTATE-WITH-UPDATE(x, y)

1: PUSH-DELTA(x)
2: PUSH-DELTA(y)
3: LEFT-ROTATE(x, y)
4: $\text{INT}(y) \leftarrow \text{INT}(x)$
5: $\text{INT}(x) \leftarrow \text{INT}(a) \cup \text{INT}(b)$
6: $\text{STAB}(x) \leftarrow \max\{\text{STAB}(a) + \Delta(a), \text{STAB}(b) + \Delta(b)\}$
7: $\text{STAB}(y) \leftarrow \max\{\text{STAB}(x), \text{STAB}(c) + \Delta(c)\}$

The following result is proven because the rotation can be implemented without affecting Invariant 1.

Theorem 1. *A set of n intervals can be stored in a data structure requiring $O(n)$ space and supporting the operations,* INSERT*,* DELETE*,* MAX-COVER-WITNESS*,* MAX-IN*(a, b), and* MAX-WITNESS-IN*(a, b) in $O(\log n)$ time per operation and* MAX-COVER *in $O(1)$ time per operation.*

3 Applications

In this section we show how the data structure from the previous section can be applied to solve a number of problems related to placing a regular square grid G over a set $S = \{p_1, \ldots, p_n\}$ of n points in the plane.

3.1 Infinite Grids

Throughout this section we assume that G is an infinite grid consisting of unit squares and that one of G's vertices is located at the origin. It is clear that all translations of G are equivalent up to additions of integers to the x and y components. Therefore we need only consider translations whose x and y components are in the range $[0, 1)$ so we represent translations as points in the square $[0, 1) \times [0, 1)$.

3.1.1 Optimizing 1-Occupancy

In this section, we consider the problem of optimizing the 1-occupancy, i.e., optimizing the number of grid cells occupied by at least 1 point of S. We focus on the maximization problem since the modifications required to solve the minimization problem are obvious.

Since we only consider translation of at most 1 unit in the x and y directions, each point $p_i \in S$ must occupy 1 of 4 cells of G after translation, call these C_i^1, C_i^2, C_i^3, and C_i^4. This partitions the set of possible translations into four rectangles R_i^1, R_i^2, R_i^3, and R_i^4 that contain the top-left, top-right, bottom-left, and bottom-right corners of $[0, 1) \times [0, 1)$, respectively (see Fig. 3). We call R_i^j an *occupation rectangle* for cell C_i^j.

For a cell C, let the *k-occupied set* of C be the set of translations of G for which C is k-occupied. It follows that the k-occupied set of C is the set of points contained in k or more occupation rectangles of C.

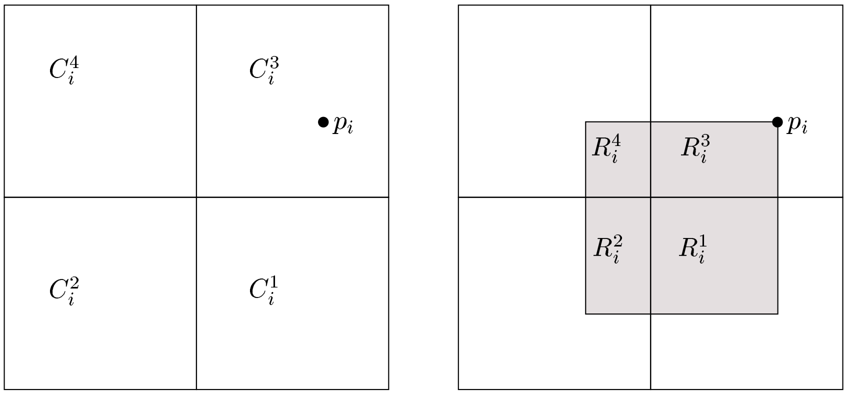

Figure 3: The point p_i will lie in one of four cells after the translation (left) which partitions the set of possible translations into four rectangles (right).

Lemma 1. *The* 1*-occupied set of a cell* C *can be computed in* $O(s \log s)$ *time, where* s *is the number of occupation rectangles of* C*.*

Proof. It is clear from the above discussion that the 1-occupied set of C is the union of the occupation rectangles of C. In general, the union of s rectangles has complexity $O(s^2)$, however occupation rectangles have the property that they are contained in the unit square, and each contains at least 1 corner of the unit square.

Let G_1 be the set of occupation rectangles that contain the top-left or top-right corner of the C and let G_2 be the set of occupation rectangles that contain the bottom-left or bottom-right corner of the C. Then the boundary of $\cup G_1$ is given by the lower envelope E_1 of the bottom edges of rectangles in G_1 and the boundary of $\cup G_2$ is given by the upper envelope E_2 of the top edges of rectangles in G_2. Since both of these are envelopes of at most s parallel line segments they each have at most $2s$ vertices and can be computed in $O(s \log s)$ time [9].

E_1 and E_2 are both x-monotone and the space above E_1 is $\cup G_1$ and the space above E_2 is $\cup G_2$. Any vertex of $\cup(G_1 \cup G_2)$ is either a vertex of G_1 or G_2 or an intersection point of E_1 and E_2. Since two monotone polygonal chains of length $O(s)$ intersect at most $O(s)$ times, the boundary of the 1-occupied set of C is of size $O(s)$ and can be obtained in an additional $O(s)$ time by merging E_1 and E_2. □

To determine a translation that maximizes 1-occupancy we proceed as follows: For each point $p_i \in S$ we compute the four occupation rectangles generated by p_i in $O(1)$ time using the floor function. Using an $O(n \log n)$ time sorting algorithm, we then gather the occupation rectangles of each cell C and compute the 1-occupied set for C in $O(s \log s)$ time, where s is the number of occupation rectangles of C. Since each point in S generates at most 4 occupation rectangles, this step takes $O(n \log n)$ time for all cells. Next, we partition the 1-occupied set for each cell C into rectangles, which also is achieved in $O(n \log n)$ time. We then overlay all of the 1-occupied sets of each cell C into one cell D. Notice that a point covered by the most rectangles in D is represents a translation that maximizes 1-occupancy.

Thus the problem of computing the translation that maximizes the number of occupied cells has been transformed into the problem of finding the point contained in the largest number of rectangles, i.e., finding the maximum clique in a rectangle intersection graph. This problem can be solved in $O(n \log n)$ time using plane sweep along with the data structure of Section 2. As the sweep line sweeps the set of rectangles, our data structure is used to maintain the maximum interval. After completion of the sweep, we can report a point contained in the largest number of rectangles. For details on the plane sweep technique, the reader is referred to [7].

Theorem 2. *Given n points $S = \{p_1, \ldots, p_n\}$ and an infinite grid G it is possible to report a translation of G which minimizes or maximizes the occupancy of G in $O(n \log n)$ time and $O(n)$ space.*

3.1.2 Optimizing k-Occupancy

Next we consider the problem of optimizing the k-occupancy. As before, the algorithm for maximizing k-occupancy is easily translated into an algorithm for minimizing k-occupancy. Therefore, we focus on the maximization problem.

We use the same general approach of computing, for each cell C, the k-occupied set of C. By partitioning these sets into rectangles and then applying the algorithm for maximum clique in a rectangle intersection graph we obtain our result. However, the difficulty comes from the fact that the k-occupied set of a cell has a richer combinatorial structure than its 1-occupied set.

To compute the k-occupied set of a cell C having s occupation rectangles, we partition the occupation rectangles of C into four sets S_1, S_2, S_3 and S_4, depending on which of the four corners of the unit square they contain (see Fig. 4). Rectangles that contain more than one corner of the unit square are assigned arbitrarily to one of their possible sets.

For each set S_i we then compute an arrangement A_i that partitions the unit square into $k+1$ regions $R_i^0, \ldots, R_i^{k-1}, R_i^{\geq k}$ where all points in R_i^x are contained in x occupation rectangles (see the top-right of Fig. 4). This is done by a vertical line sweep over the rectangles in S_i starting at the corner of the unit square contained in all elements of S_i. The sweep line status is maintained by a balanced-binary tree and we construct the arrangement by keeping track of the k-smallest elements in the tree. The only events occur when rectangles terminate, at which point an element is removed from the tree, causing at most k of the $k+1$-smallest elements to change rank. Each event can therefore be handled in $O(k + \log s)$ time and the resulting arrangement has complexity $O(ks)$.

Each A_i consists of k x-y monotone chains (sometimes calls staircases), where the directions of monotonicity are different for each. Because the chains are

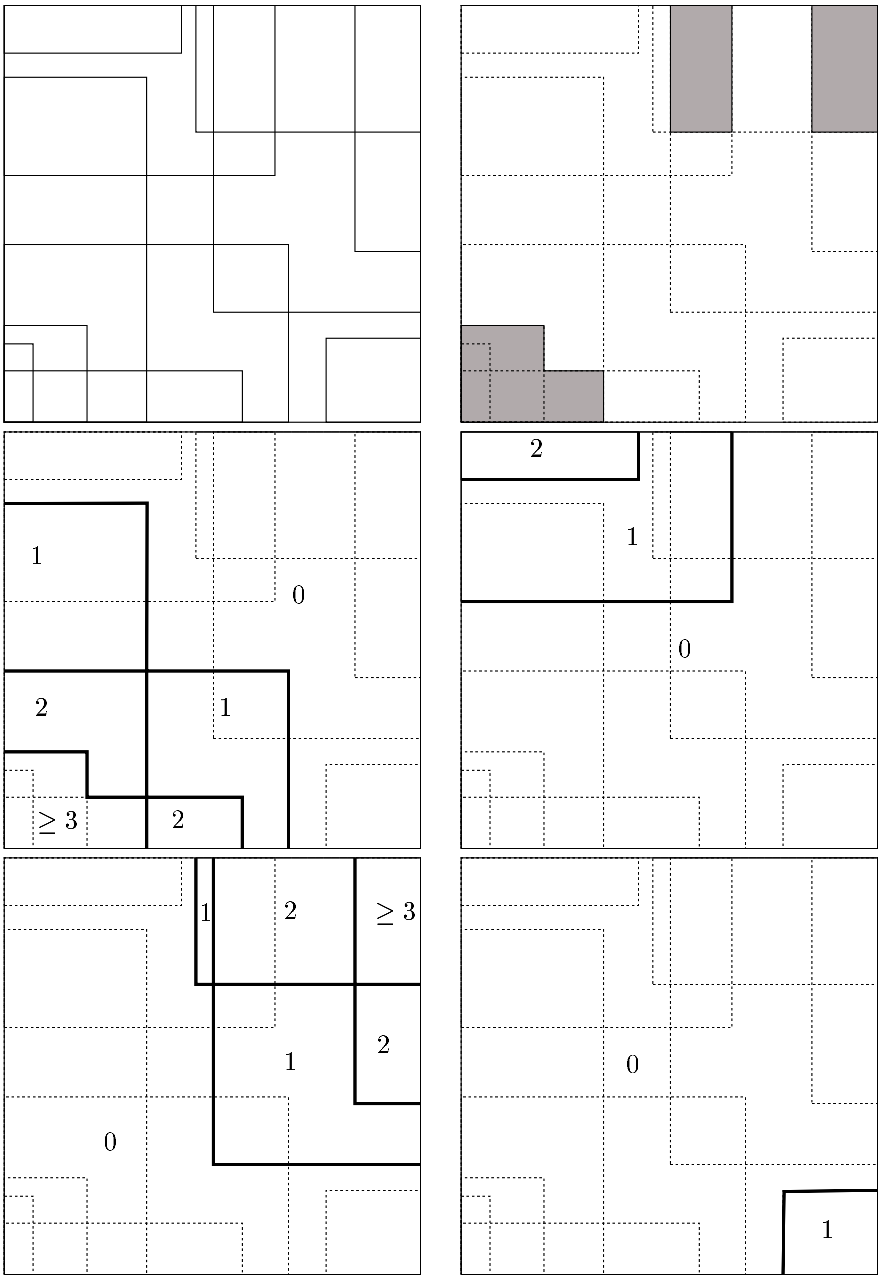

Figure 4: The original set of occupation rectangles for C, the k-occupied set of C and the four arrangements $A_1, \ldots, A_4$ for $k = 3$.

monotone, the intersection of any horizontal or vertical line segment with all chains consists of at most $4k$ points and line segments. Since the chains consists of a total of $O(s)$ line segments, this implies that the total number of intersections between all chains is $O(sk)$, and these intersections can be computed in $O(sk \log s)$ time using plane sweep [4]. This results in an arrangement A of orthogonal line segments that has $O(sk)$ faces.

For each face F of A, we can examine the faces of A_1, A_2, A_3 and A_4 that contain F. Since each face F' of A_i is labelled as being contained in $0, \ldots k-1$ or $\geq k$ occupation rectangles of S_i, we can determine whether the points in F are contained in k or more occupation rectangles of C. Thus we can determine whether for each face F of A, whether or not F is part of the k-occupied set. This step can be performed in $O(sk \log s)$ time using point location or, if we are a bit more careful, in $O(sk)$ time by doing a depth-first traversal of the faces of A.

Thus, for each cell C, we can find the k-occupied set of C in $O(sk)$ time, where s is the number of occupation rectangles of C. As in the previous section, once the the k-occupied sets have been computed for each cell, a plane sweep over all sets yields an optimal translation of G in an additional $O(kn \log n)$ time.

Theorem 3. *Given n points $S = \{p_1, \ldots, p_n\}$, an integer $1 \leq k \leq n$, and an infinite grid G it is possible to report a translation of G which minimizes or maximizes the k-occupancy of G in $O(kn \log n)$ time and $O(kn)$ space.*

3.1.3 Finding a Translation with k-Occupancy 0

We note that the problem of finding a translation with k-occupancy 0 is solvable using the algorithm of the previous section. The translation with the minimum k-occupancy is found and if that k-occupancy is 0 then the solution is found otherwise none exists.

Theorem 4. *Given n points $S = \{p_1, \ldots, p_n\}$, an integer $1 \leq k \leq n$, and an infinite grid G it is possible to decide if there exists a translation of G such that no cell of G is k-occupied and report such a translation if one exists in $O(kn \log n)$ time and $O(kn)$ space.*

3.1.4 Optimizing k

To solve Problems 4 and 5, we can perform a "doubling search" on the value of k. Initially, we guess k to be 1 and repeatedly double the value of k. For each value of k we use Theorem 3 or Theorem 4 to test if the value of k has become too large. If it has, we stop and perform binary search on the range $(k/2, k)$ to find the optimal value of k. The additional cost of this binary search is a $\log k$ factor in the running time.

Theorem 5. *Given n points $S = \{p_1, \ldots, p_n\}$ and an infinite grid G it is possible to report the smallest integer k for which there exists a translation with k-occupancy equal to 0 in $O(kn \log n \log k)$ time and $O(kn)$ space. Additionally,* *given an arbitrary non-negative integer r, it is possible to report the largest integer k for which there exists a translation with k-occupancy at least r in the same time and space.*

3.2 Placing a Finite Grid

Next we turn to the problem of placing an $m_r \times m_c$ grid G over the point set S so that the k-occupancy of G is optimized. This appears to be a very different problem, since now all translations are not equivalent up to integer additions to the x and y components. It should come as no surprise then that the grid size $m_r \times m_c = m$ becomes a factor in the running time.

For each point $p_i \in S$, we transform p_i into a unit square s_i whose bottom right corner is p_i. Thus, if we place a grid square with its top-left corner inside of s_i then that square will be occupied. Next, we compute the region R contained in k or more s_i's. This is achieved, as in Section 3.1.2, by overlaying an infinite grid and calculating the k-occupancy region for each occupied cell. The region R is simply the union of each of these disjoint k-occupancy regions. It follows that if we place a unit square with its top-left corner in R, then that square is k-occupied.

We then make m translated copies $R_1, \ldots, R_m$ of R, where R_i is R translated by the vector $(-i \bmod m_c, \lfloor i/m_c \rfloor)$. It follows that if we place the top-left corner of a $m_r \times m_c$ grid G at a point that is contained in x of the R_i then the number of k-occupied cells of G is x. Thus, we have reduced the problem of placing G to a problem of finding a point contained in the largest number of regions R_i. Since each R_i is a set of orthogonal polygons they can be decomposed into rectangles thereby reducing the problem to maximum-clique in a rectangle intersection graph.

In summary, we compute R in $O(kn \log n)$ time, partition R into rectangles and create $R_1, \ldots, R_m$. This results in a set of rectangles of size $O(mkn)$. We then solve the maximum-clique problem in the resulting set of rectangles in time $O(mkn \log mn)$. To reduce the space requirements we note that $R_1, \ldots, R_m$ need not be computed explicitly. Indeed, in the priority queue that implements the plane sweep of the maximum clique algorithm, each edge of R need appear only once at any given time. At all times during the sweep, an edge in the queue represents each of the m_c copies with the same y-coordinate. To initialize the queue the left and upper most copy of each edge is inserted into the queue. When an edge is swept over it generates the appropriate m_c simultaneous events, one for each grid column. After it has been swept the edge is translated downward and reinserted into the queue to represent the next set of m_c events. This translation is performed only m_r times for each edge thus providing the m copies of each edge.

Theorem 6. *Given n points $S = \{p_1, \ldots, p_n\}$, an integer $1 \le k \le n$, and a finite grid G with m cells it is possible to report a translation of G with maximal k-occupancy in $O(mkn \log mn)$ time and $O(kn)$ space.*

4 Conclusions

We have studied problems related to placing a regular square grid over a set of points in the plane in order to optimize some function of the k-occupancy. For small values of k most of our algorithms have near-linear running times.

To solve these problems we have introduced a new lazy data structure for maintaining a point in the maximum-clique of an interval graph under insertions and deletions of intervals. This data structure is fully dynamic as it doesn't require the interval endpoints to be known in advance.

The data structure is also rather general. Besides storing intervals, it could be used to, e.g., store piecewise polynomial functions and return a value x that maximizes or minimizes the sum of all the functions evaluated at x. One could imagine using such a data structure with plane-sweep to perform operations like maintaining the minimum in sums of functions, or combining it with the persistence paradigm [8] to get an efficient representation of sums of functions.

An obvious open problem is that of improving the running time of the algorithm for Problem 6. In particular, it would be nice to produce an algorithm which is output sensitive to the size of the k-occupied set.

Open Problem 1. *Find an output sensitive algorithm which computes the boundary of a k-occupied region.*

References

[1] P. K. Agarwal, B. K. Bhattacharya, and S. Sen. Output-sensitive algorithms for uniform partitions of points. In A. Aggarwal and C. P. Rangan, editors, *Proceedings of the 10th International Symposium on Algorithms and Computation, ISAAC*, volume 1741 of *Lecture Notes in Computer Science*, pages 403–414. Springer, December 1999.

[2] T. Asano and T. Tokuyama. Algorithms for projecting points to give the most uniform distribution with applications to hashing. *Algorithmica*, 9(6):572–590, June 1993.

[3] J. L. Bentley. Solutions to Klee's rectangle problems. Technical report, Carnegie-Mellon University, 1977.

[4] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28:643–647, September 1979.

[5] D. Comer and M. J. O'Donnell. Geometric problems with applications to hasing. *SIAM Journal on Computing*, 11(2):217–226, May 1982.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1990.

[7] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.

[8] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[9] J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Information Processing Letters*, 33(4):169–174, December 1989.

[10] D. T. Lee. Maximum clique problem of rectangle graphs. In F. P. Preparata, editor, *Advances in Computing Research*, pages 91–107. JAI Press, 1983.

[11] M. J. van Kreveld and M. H. Overmars. Concatenable segment trees (extended abstract). In B. Monien and R. Cori, editors, *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science*, volume 349 of *Lecture Notes in Computer Science*, pages 493–504. Springer, February 1989.