The Grid Placement Problem*

P. Bose A. Maheshwari P. Morin J. Morrison

School of Computer Science, Carleton University
1125 Colonel By Dr., Ottawa, Ontario, CANADA, K1S 5B6
email: {jit,maheshwa,morin,morrison}@scs.carleton.ca

Abstract

We consider the problem of placing a regular grid over a set of points in order to minimize (or maximize) the number of grid cells not containing any points. We give an $O(n \log n)$ time and $O(n)$ space algorithm for this problem. As part of this algorithm we develop a new data structure for solving a problem on intervals that is interesting in its own right and may have other applications.

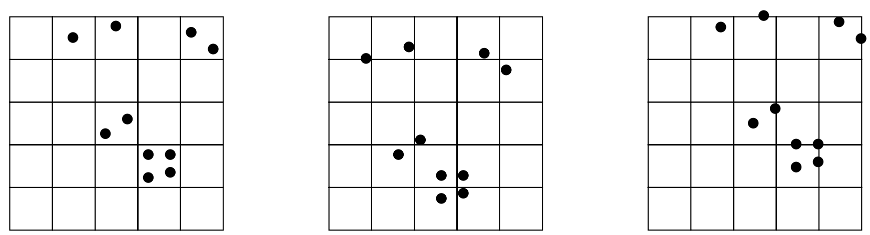
1 Introduction

The grid placement problem relates the positioning of a regular rectangular grid over a pointset. The focus of the problem is to place the grid and then count the number of grid cells containing at least one point. The problem has two variations, in the first variation the task is to calculate the grid position which minimizes the number of empty cells. The second variation calculates the position which maximizes the number of empty cells. Figure 1 shows an example solution for both problems.

This grid placement problem arises in *kriging* where one attempts to derive a regular grid of data points from an irregular set of sample points. In this setting, one wants to minimize the number of grid cells not containing any sample point.

Another example application of this problem is in the layout of maps into books. In this case each grid cell corresponds to a page in the book and each point represents map features. In this case, by minimizing the number of empty cells, we decrease the average number of features per page, making the maps easier to label and reducing the number of uninteresting pages.

*This research was partly supported the Natural Sciences and Engineering Research Council of Canada and by the Government of Ontario.



(a) (b) (c)

Figure 1: A grid placement (a) with the minimum number of empty cells, (b) with the maximum number of empty cells and (c) in standard position.

In this paper, we give an $O(n \log n)$ time algorithm for solving the grid placement problem for a set of n points and a grid of arbitrary size. As part of this algorithm we develop a new data structure for solving a problem on intervals that is interesting in its own right and may have other applications.

In the following sections we give a discussion of the solution to the minimization problem and leave the trivial extension to the maximal problem to the reader. Our discussion formally defines the problem and describes and proves correctness of a basic algorithm. This basic algorithm is then refined into our final solution and analyzed.

2 Problem Definition and Analysis

We start by defining the set of n points in the problem to be the points $P = \{p_1, p_2, \ldots, p_n\}$ embedded in the plane. The location of a point p_i in the plane is represented as (p_{ix}, p_{iy}). A grid is defined as a packed tiling of regular rectangles called cells. The sides for each of the cells are assumed to be parallel to the x and y axes of the plane and have dimensions (c_x, c_y). A specific instance of a grid, G is defined by its size and its position. The size of G is defined by the number of cell columns (c) and rows (r). The *position* of the grid (G_x, G_y) is the location of the cell corner which has minimal x and y co-ordinates(i.e., the lower left corner). A grid position is allowable if and only if every point of P is contained in some cell of G.

There are a variety of allowable grid positions relative to the pointset P. In standard position (G_{sx}, G_{sy}), as shown in Figure 1.c, the uppermost and rightmost points in P are coincident with the uppermost and rightmost boundaries of G, respectively. We finish the problem definition by assuming that in the standard position the leftmost column and bottommost row of cells do not contain any points. Thus the problem is to find an allowable grid position which minimizes the number of cells that do not contain points from P.

By examining the problem we note that there are a limited number of allowable grid positions that need to be considered. The exact space of allowable grid positions is defined in Theorem 1.

Theorem 1 *The only allowable grid positions that need to be considered are the "kernel" positions defined by the bottom, left cell in standard position.*

Proof: The proof is divided into two parts. First that all of the kernel positions are allowable and second that all other allowable positions are equivalent to some kernel position. Given that the standard position is allowable, that the bottommost row is empty and that the leftmost column is also empty, it is clear that the kernel is allowable.

It remains to be proven that all other allowable positions are equivalent to some kernel position. Let A be the set of allowable positions. By the definition of the standard position any translation of the grid down and/or to the left will result in at least one point p_i being outside of the grid. Thus all elements $a = (a_x, a_y)$ of A must be expressible by non-negative translation from the standard grid position. That is, $a = (G_{sx} + h_x, G_{sy} + h_y)$ where $\{h_x, h_y\} >= 0$. Given that Lemma 1 is also true, then the theorem follows trivially. $\square$

Lemma 1 *Given two distinct allowable positions $a = (a_x, a_y)$ and $b = (a_x + h_x, a_y + h_y)$ where $h_x = ic_x$, $h_y = jc_y$ and $\{i, j\} \in \mathcal{N}$, the number of non-empty cells is identical for both positions.*

Proof: Since both positions are allowable the grid at position a and b must contain all points. This further implies that the grid located at a must intersect the grid located at b and that only intersecting cells can contain points. Also, since i and j are integers then any cell can only intersect at most one other cell. Thus, the sets of cells which overlap form a one-to-one correspondence. This establishes a bijection between non-empty cells at a and non-empty cells at b. Therefore there are the same number of non-empty cells at both locations. $\square$

3 Basic Algorithm

The algorithm itself breaks into two main stages. In the first stage, we calculate the regions of the kernel for which each cell is empty. In the

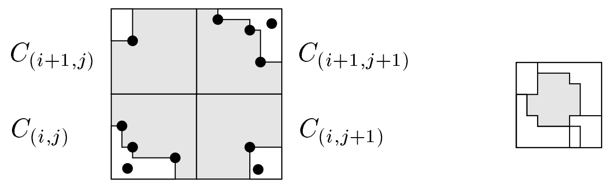


Figure 2: Using maximal elements to compute the empty region for $C_{(i,j)}$.

second stage, we overlay these empty regions and calculate a position which has the minimum number of regions.

For the cell $C_{(i,j)}$[1] we define $P(C_{i,j})$ to be the subset of P contained in $C_{(i,j)}$ at standard position. Since only the kernel positions are considered, then $C_{(i,j)}$ can only contain elements from $P(C_{(i,j)})$, $P(C_{(i+1,j)})$, $P(C_{(i,j+1)})$ and $P(C_{(i+1,j+1)})$. For $C_{(i,j)}$ to be empty at a point a, the cell must not include any element of any of these sets.

To understand the empty region for $C_{(i,j)}$ first consider only the set $P(C_{(i,j)})$. A point $X = (X_x, X_y)$ is dominated by the set $P(C_{(i,j)})$ if and only if there exists an element p_i of $P(C_{(i,j)})$ such that $p_{ix} > X_x$ and $p_{iy} > X_y$. The cell $C_{(i,j)}$ does not contain an element of $P(C_{(i,j)})$ if and only if the lower left corner of $C_{(i,j)}$ is not dominated by $P(C_{(i,j)})$. Thus the maximal elements [3, 2] define the region in which the cell does not contain any points of $P(C_{(i,j)})$. By symmetry the same can be done for $P(C_{(i+1,j)})$, $P(C_{(i,j+1)})$ and $P(C_{(i+1,j+1)})$ and the intersection of these four regions define the empty region for $C_{(i,j)}$. See Figure 2 for an example.

The intersection of these four regions is done in a particular manner to avoid added complexity in the second stage. First, the appropriate dominance hull is calculated for each of the four cells[2]. Then the intersection of the regions for $P(C_{(i,j)})$ and $P(C_{(i,j+1)})$ is calculated followed by the intersection for the remaining two regions. The result of these intersections are two orthogonal convex polygons.

Each of these polygons is stored as a list corresponding to the events of a vertical sweep from the top to bottom of the kernel. At each event in the vertical sweep, the line interval resulting from the intersection of the sweep line and the polygon are stored. The vertical position of the event and the corresponding width of the interval is stored with each of the intervals. It is

[1]Each cell $C_{(i,j)}$ is labeled by it's row and column with $C_{(1,1)}$ in the bottom left and $C_{(r,c)}$ in the upper right.

[2]If one of the pointsets to be calculated does not exist due to grid boundary then the pointset can be considered empty.

assumed that the list of intervals is sorted by its vertical position. The two polygons are then intersected by merging the two lists to create a new list of intervals and their insertion and deletion heights.

Suppose $|P(C_{(i,j)}) \cup P(C_{(i+1,j)}) \cup P(C_{(i,j+1)}) \cup P(C_{(i+1,j+1)})| = m$. The maximal elements can be computed in $O(m \log m)$ time [3, 2], and the intersection of the four empty regions can also be done in $O(m)$ time as described above. Furthermore, any point in P is used in the computation of the empty regions for at most 4 grid cells. Therefore, the overall running time thus far is $O(n \log n)$.

In the second stage of the algorithm the empty regions for all of the cells have to be overlayed and a position with the minimum number of empty regions found. This could be done by computing an arrangement of rectillinear polygons, but this would take $\Omega(n^2)$ time and space. Instead, we perform a vertical plane sweep [1, 4] of the kernel from top to bottom. There are two types of events during the sweep, the addition of an interval at it's maximum height and the deletion of an interval at it's minimum height. At each point in time we would like to keep track of a point that is contained in the least number of intervals. The minimum such value, over all points in time, corresponds to a point in the kernel that is contained in the least number of empty regions.

Since the problem has reduced to insertion and deletion of intervals in a sweep of the plane, then the second stage of the algorithm yields a solution by solving the following problem. Given that a set of one dimensional intervals I are inserted and deleted at times $\{t_1, t_2, t_3, \ldots, t_s\}$, find the time t and the location x where the minimum number of intervals overlap.

The algorithm will maintain a tree structure T which will answer queries about how many intervals are overlapping at a given x-coordinate and what the minimum overlapping intervals are. A complete binary search tree is created with a leaf for every x-coordinate in P plus the maximal x-coordinates for the kernel. The tree is assumed to have enough information to perform searches on x-coordinates ($right$, $left$, $parent(p)$, key, $\ldots$). Every vertex v in the tree will hold values $stab[v]$ and $ptr[v]$.

In a leaf vertex u, $stab[u]$ will contain the number of intervals that overlap at its corresponding x-coordinate. At an internal vertex $stab[v] = \min(stab[u])$ such that u is any leaf of the sub-tree rooted at v. Each vertex v also contains a pointer $ptr[v]$ to its child v' that has the minimum $stab$ value. In all future discussions, we will use the notation v' to denote the child of v with minimum $stab$ value. For all leaves $ptr[v] = v$.

With this structure an interval is inserted by incrementing $stab[u]$ at each leaf u whose key is contained by the interval (all intervals are closed). Then all ancestors v of the updated leaves are updated in a bottom up fashion. Deletion of an interval is done in an identical fashion by decrementing the necessary leaves.

For the purpose of our algorithm and its proof each interval update is further broken into two updates. An interval $[x_1, x_2]$ is equivalent to $[x_1, \infty) - (x_2, \infty)$. Therefore the insertion of $[x_1, x_2]$ is broken into inserting $[x_1, \infty)$ and deleting $[x_3, \infty)$ where x_3 is the minimal x-coordinate greater than x_2. Deletion of an interval is done similarly. Thus all of the interval updates which define the empty regions must be converted into two updates.

The algorithm is run by first sorting all of the interval updates by y-coordinate. Two null events are also added to occur at the maximal y-coordinates of the kernel. Any events with identical y-coordinate are then grouped together. The answer to the problem is then found by processing each group of events in order. After each group, excepting the last group, compare the known minimum value with the value at the root of the tree. If it is less than the current minimum then make it the new minimum. Also follow the chain of vertices connected by *ptr* values from $root[T]$ to vertex u. Record $key[u]$ as the x coordinate and the y-coordinate of that event group as the y-coordinate of the minimum.

4 Refined Algorithm

The main problem with the Basic Algorithm is the complexity of each interval update. For any given update $[x, \infty)$ let the vertex chain $V(x) = \{v_k, \dots, v_1, v_0\}$ be the vertices along the path from $root[T] = v_k$ to the leaf v_0 whose $key[v] = x$. Then consider updating the tree T with $[x, \infty)$ and a value val ($val = 1$ for insertion, $val = -1$ for deletion). The basic algorithm requires every vertex in any right sub-tree of the path $V(x)$ be updated. The refined algorithm reduces the cost of updating by postponing the updating for any such sub-tree until absolutely necessary. This is achieved by performing an update on the root of a sub-tree and setting a mark bit for that root. This mark bit signifies that the node has been updated and its descendents have not.

By refining the data structure, a different algorithm is required to maintain T during an interval update $[x, \infty), val$. The refined $Update\_Interval$ algorithm is broken conceptually into two phases. In the first phase the path $V(x)$ is followed from root to leaf performing necessary updates to T (see lines 1-17 of Algorithm 1). These necessary updates involve unmarking all vertices on $V(x)$ and marking the root off all right sub-trees of $V(x)$. The algorithm then updates the leaf node v_0 by adding val to $stab[v_0]$.

The second phase of the algorithm then traverses the vertices $V(x) = v_i$ from leaf to root updating $ptr[v_i]$ appropriately and updating $stab[v_i]$ to be the value stored in the child of v_i that has the minimal $stab$ value (see lines 19-25 of Algorithm 1).

Algorithm 1 $Update\_Interval(x, val)$ – Adds $[x, \infty)$ of value val to tree T

1: $v \leftarrow root[T]$
2: $d \leftarrow 0$
3: **while** $leaf[v] = FALSE$ **do**
4: **if** $mark[v] = TRUE$ **then**
5:   $Unmark(v)$ (see Algorithm 2)
6: **end if**
7: $d \leftarrow d + 1$
8: **if** $key[v] < x$ **then**
9:   $v \leftarrow right[v]$
10: **else** {interval begins in the left sub-tree of v}
11:   **if** $leaf[right[v]] = FALSE$ **then**
12:    $mark[right[v]] \leftarrow TRUE$
13:   **end if**
14:   $v \leftarrow left[v]$
15: **end if**
16: **end while**
17: $stab[v] \leftarrow stab[v] + val$
18: **while** $d \neq 0$ **do**
19: **if** $left[p[v]] = v$ **then**
20:   $stab[right[p[v]]] \leftarrow stab[right[p[v]]] + val$
21: **end if**
22: $v \leftarrow p[v]$
23: $d \leftarrow d - 1$
24: $stab[v] \leftarrow stab[v']$
25: $ptr[v] \leftarrow v'$
26: **end while**

We define the following variables:

$$\Delta_v = \begin{cases} stab[v], & \text{if } leaf[v] = TRUE; \\ stab[v] - stab[v'], & \text{if } mark[v] = TRUE; \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Algorithm 2 $Unmark(v)$

Require: $leaf[v] = FALSE$

$\delta \leftarrow stab[v] - stab[v']$
$stab[right[v]] \leftarrow stab[right[v]] + \delta$
$stab[left[v]] \leftarrow stab[left[v]] + \delta$
if $leaf[right[v]] = FALSE$ **then**
  $mark[right[v]] \leftarrow TRUE$
  $mark[left[v]] \leftarrow TRUE$
end if
$mark[v] \leftarrow FALSE$

$$STAB_i[x] = \text{Num. of intervals overlapping } x \text{ after the } i^{th} \text{ interval update.} \tag{2}$$

Property 1 *Given the path of vertices $V(x) = \{v_k, \ldots, v_1, v_0\}$ from $root[T] = v_k$ to the leaf v_0 whose $key[v_0] = x$ then after the i^{th} run of Algorithm 1:*

$$\sum_{j=0}^{k} \Delta_{v_j} = STAB_i[x]. \tag{3}$$

Lemma 2 *Given T such that Property 1 holds, then the property will still hold after performing procedure $Unmark(v)$ (see Algorithm 2) on any marked vertex v.*

Proof of Lemma 2:

Since $Unmark(v)$ only affects the $stab$ values of $v, right[v]$ and $left[v]$ then only $\Delta_v, \Delta_{right[v]}$ and $\Delta_{left[v]}$ are affected. Since v is a marked vertex before $Unmark(v)$ is run and an unmarked vertex after then any sum involving Δ_v must decrease by Δ_v. However, $Unmark(v)$ also increases the values of $\Delta_{right[v]}$ and $\Delta_{left[v]}$ by the same amount and each sum must contain one of these two values. □

Proof of Property 1: by induction on the number of interval updates(i).

Base Case (After initialization): Before the first update there are no intervals in T. Thus if T is initialized with $mark[v] = FALSE$, $stab[v] = 0$ and $ptr[v] = left[v]$ then Property 1 is trivially true.

Induction Hypothesis: Assume that it is true for the i^{th} interval update. $(i+1)^{st}$ **Interval Update:** We prove that Property 1 is true for the $(i+1)^{st}$ update in three cases, for vertices v with $key[v] = x$, $key[v] < x$ and $key[v] > x$.

The first case considers the leaf v_0 representing the leftmost portion of the interval being updated (i.e., $key[v_0] = x$). After line 16 all vertices in the path $V(x)$ are unmarked. It follows from the definition of Δ_v that $\Delta_v = 0$, $\forall v \in V(x)$ except v_0 before the $(i+1)^{st}$ update. By Lemma 2 the expression $\sum_{j=0}^{k} \Delta_{v_j} = STAB_i(x)$ must be true before line 17. Combining these two equations with Equation 1 yields that $\Delta_{v_0} = STAB_i(x) = stab[v_0]$. Thus, after line 17 $stab[v_0] = STAB_i(x) + val = STAB_{i+1}(x)$, which proves the first case.

The second case considers, without loss of generality any leaf u_0 with key value less than the leftmost portion of the interval being updated (i.e., $key[u_0] = w < x$). Again, it follows from the definition of Δ_v that $\Delta_v = 0$, $\forall v \in V(x)$ except v_0 before the $(i+1)^{st}$ update. By Lemma 2 the expression $\sum_{j=0}^{k} \Delta_{v_j} = STAB_i(x)$ must be true before line 17. Let the lowest common ancestor $lca(u_0, v_0) = u_h$. By definition of $lca()$, all vertices $\{u_h, u_{h+1}, \ldots, u_k\} \in V(x)$. Thus $\sum_{j=0}^{h-1} \Delta_{u_j} = STAB_i(w)$. Since vertices $\{u_{h-1}, u_{h-2}, \ldots, u_0\}$ are unaffected by the $(i+1)^{st}$ update, then after line 26: $\sum_{j=0}^{h-1} \Delta_{u_j} = \sum_{j=0}^{k} \Delta_{u_j} = STAB_i(w)$. Since $STAB_i(w) = STAB_{i+1}(w)$ then the second case is proven.

The third case follows the same logic as the second case with only one difference. On lines 19 and 20 the value of $stab[u_{h-1}]$ is incremented by val. Thus after line 26: $\sum_{j=0}^{h-1} \Delta_{u_j} = \sum_{j=0}^{k} \Delta_{u_j} = STAB_i(w) + val$. Since $STAB_{i+1}(w) = val + STAB_i(w)$ then the third case is proven. □

Lemma 3 *After the i^{th} interval update, the pointer $ptr[v]$ at an arbitrary vertex v equals its child vertex v' with the minimum stab value.*

Proof: by induction on the number of interval updates(i).

Base Case (after initialization): Before the first update there are no intervals in T. Thus if T is initialized with $mark[v] = FALSE$, $stab[v] = 0$ and $ptr[v] = left[v]$ then Property 1 is trivially true.
Induction Hypothesis: Assume that it is true for the i^{th} interval update.
Proof of $(i+1)^{st}$ update: During the $(i+1)^{st}$ update the only vertices whose $stab$ values change are the vertices on the path $V(x)$ or their children. Since the property holds after the i^{th} update the path vertices are the only vertices whose ptr values may need adjustment. Since these values are

adjusted on line 25 then the property must be true for the $(i+1)^{st}$ update. $\square$

Consider the path $v_k = root[T], v_{k-1} = ptr[v_k], \ldots, v_0 = ptr[v_1]$ and let $x = key[v_0]$. By Property 1 after the i^{th} insertion $STAB_i(x) = \sum_{j=0}^{k} \Delta_{v_j}$. Furthermore, by the definition of Δ_v and Property 3 of v_i, $\Delta_{v_i} = v_i - v_{i-1}$. Therefore,

$$\begin{aligned} STAB_i(x) &= \sum_{j=0}^{k} \Delta_{v_j} && (4) \\ \Delta_{v_i} &= stab[v_i] - stab[v_{i-1}] && (5) \end{aligned}$$

By using Equations 4 and 5 together:

$$\begin{aligned} STAB_i(x) &= stab[v_0] + \sum_{j=0}^{k} stab[v_i] - stab[v_{i-1}] && (6) \\ &= stab[v_k] && (7) \end{aligned}$$

Theorem 2 *Given the definition of v_k stated above, $stab[v_k]$ contains the appropriate minimal value after the i^{th} interval update.*

Proof: directly

Consider any leaf vertex u_0 such that $key[u_0] = w$. By rearranging Property 1 and we arrive at Equation 10

$$\begin{aligned} STAB_i(w) &= \sum_{j=0}^{k} \Delta_{u_j} && (8) \\ &= stab[u_0] + \sum_{j=1}^{k} stab[u_j] - stab[u'_j] && (9) \\ &= stab[u_k] + \sum_{j=0}^{k-1} stab[u_j] - stab[u'_{j+1}] && (10) \end{aligned}$$

By the definition of u' the value $stab[u_j] - stab[u'_{j+1}]$ is minimized when $u_j = u'_{j+1}$. By Property 3 $ptr[u_{j+1}] = u'_{j+1}$ and therefore $STAB_i(x)$ is minimal by its definition. The proof is concluded because $stab[v_k] = STAB_i(x)$ by Equation 7. $\square$

It is clear that the running time of *Update_Interval* is $O(\log n)$. Furthermore, 2 shows how to find a point overlapped by the least number of intervals in $O(\log n)$ time. Since these are exactly the operations required to solve the grid placement problem, we obtain our main result.

Theorem 3 *The grid placement problem can be solved in $O(n \log n)$ time and $O(n)$ space.*

Our data structure for the interval problem is extremely simple and can be implemented as an array, thereby eliminating the need for pointers from parents to children. Indeed, the total amount of memory required at each node is 2 $\log n$ bit words (for the *key* and *stab* values) and 2 bits (for the *ptr* and *mark* values).

5 Conclusions

We have studied the grid placement problem and reduced it to the following data structuring problem: Maintain a set of intervals with endpoints drawn from a set of $O(n)$ points under insertions and deletions so that at any point in time a point included in the least number of intervals can be reported. We have developped a lazy data structure for this problem that requires $O(n)$ space and operates in $O(\log n)$ time per operation, resulting in an $O(n \log n)$ time algorithm for the grid placement problem.

References

[1] J. L. Bentley. Algorithms for Klee's rectangle problems. Canegie-Mellon University, Department of Computer Science, unpublished notes, 1977.

[2] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *JACM*, 22(4):469–476, 1975.

[3] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[4] J. van Leeuwen and D. Wood. The measure problem for rectangular ranges in d-space. *J. Algorithms*, 2:282–300, 1981.