Connecting Grid Points to the Boundary of the Grid by Means of Non-intersecting Line Segments†

Leonidas Palios
The Geometry Center
University of Minnesota
Minneapolis, MN 55454

Abstract: We consider the problem of finding the maximum number of nodes in a grid (from a given set of such nodes) that can be connected to the boundary of the grid by means of non-intersecting line segments parallel to the grid axes. The work is motivated from the VLSI/WSI array processor technology, and in particular, the single-track switch model for configurable array processors ([4]), where finding whether the given nodes can *all* be connected to the boundary as described corresponds to determining whether it is feasible to reconfigure (and subsequently use) a faulty array processor.

The problem was first investigated by Bruck and Roychowdhury in [2], who described an algorithm that finds the maximum number of compatible connections of n given nodes in the grid in $O(n^3)$ time while requiring $O(n^2)$ space. We improve their result by exhibiting an $O(n^2 \log n)$ time and $O(n^2)$ space algorithm. The improvement relies on (i) establishing and using recursive definitions to compute some useful quantities faster than in [2], and (ii) introducing and using a new type of priority search trees to locate the optimal solution among a number of candidates in time logarithmic in their number.

Finally, we extend the algorithm to handle the additional constraint that connections in opposite directions in adjacent rows (columns resp.) intersect at most one common column (row resp.), thus disallowing *near-misses* (see [4]). This is the first algorithm to resolve this case; like the general algorithm, it runs in $O(n^2 \log n)$ time, and requires $O(n^2)$ space.

---

† This work has been supported by grants from the National Science Foundation (NSF/DMS-8920161), the Department of Energy (DOE/DE-FG02-92ER25137), Minnesota Technology, Inc., and the University of Minnesota.

1. Introduction.

The work presented in this paper is motivated from the VLSI/WSI array processor technology. An *array processor* is a synchronous parallel computer with a number of *processing elements* that operate in parallel in lockstep fashion (see [3], and [4], [9]). Unfortunately, due to faults during the manufacturing process, it is often the case that array processors contain faulty processing elements. The design can be made fault-tolerant, however, by incorporating spare processing elements in the array. Then, if each faulty processing element can be substituted by a spare one, the array processor can still be used. The process of finding the appropriate substitutes and establishing the necessary connections is called *reconfiguration*.

One of the models for reconfigurable array processors is the *single-track switch* model, described by Kung, Jean, and Chang ([4]): the array processor consists of a two-dimensional array of processing elements with double-row-column spare ones placed around them, and the reconfiguration process involves substituting a processing element for the one next to it along a straight-line *compensation path* that connects a faulty element to a spare one located in the same row or column. Under the additional restriction that compensation paths must not intersect, the problem of determining whether a given array processor is reconfigurable, that is, whether all the faulty processors can be compatibly substituted by spare ones, can be stated as follows:

> **P1:** Given a number of nodes located at vertices of (a bounded portion of) the grid, determine whether each of these nodes can be connected to the boundary of the grid by means of a single line segment parallel to one of the coordinate axes of the grid, so that no two such line segments intersect.

An instance of this problem where all the nodes can indeed be connected to the grid boundary as described above (along with the corresponding line segments) is shown in Figure 1. However, one can easily produce cases where this is not possible. For instance, consider a node p flanked with other nodes both in its row and column; clearly, each of the four possible line segments that would connect p to the boundary of the grid goes through another node and it would thus intersect its corresponding line segment. Another simple case is shown in Figure 2, where not both nodes p and q can be connected to the boundary by means of non-intersecting vertical or horizontal line segments.

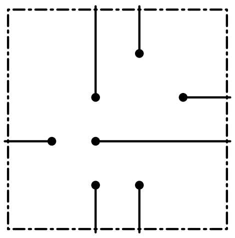
Figure 1

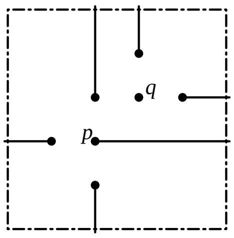

Figure 2

Problem P1 was first addressed by Kung, Jean, and Chang, who formulated it as a maximum independent set problem ([4]). Later, Roychowdhury and Bruck gave an $O(n^2)$ time algorithm, where n is the number of the given nodes in the grid ([8]). A year later, Birk and Lotspiech improved the above result, describing an $O(n \log n)$ time algorithm ([1]).

In this paper, we consider the *maximization* version of the above problem, namely:

> **P2:** Given a number of nodes in the grid, compute the *maximum* number of them that can be connected to the boundary of the grid by means of non-intersecting vertical or horizontal line segments.

An algorithm for this problem was presented by Bruck and Roychowdhury in [2]; it distinguishes two kinds of connection patterns depending on whether there is a *vertical partitioning* or not. In either case, the grid is partitioned into a constant number of rectilinear polygons, whose corresponding optimal connection patterns are independent of each other and are easier to compute as connections are restricted to three, two, or even just one direction. The combination of the optimal solutions for these polygons gives the optimal solution for the configuration. Taking the best among the optimal solutions of all the configurations yields the optimal solution for the entire grid. If the number of nodes is n, the algorithm runs in $O(n^3)$ time and requires $O(n^2)$ space.

In this paper, we improve the above result by describing an $O(n^2 \log n)$ time and $O(n^2)$ space algorithm for problem P2. The general approach is the same as that in [2]; our contribution involves

(i) establishing recursive definitions for some of the needed optimal partial solutions, which allow us to compute them faster than computing them independently, and

(ii) introducing and using data structures that enable us to locate the optimal solution among several candidates in time logarithmic in their number. It is important to note that the additional data structures do not increase the space complexity in the asymptotic sense.

Finally, it is interesting to note that the corresponding problem under Manhattan routing is NP-complete ([7]).

We also consider the variant of problem P2, where we forbid *near-misses* (see [4]), that is, we have the additional constraint that connections in opposite directions in adjacent rows (columns resp.) intersect at most one common column (row resp.). Both [4] and [1] discuss the case where near-misses are disallowed, the latter reference describing an $O(n \log n)$-time algorithm for the corresponding variant of problem P1. In this paper, we resolve the maximization version of the problem by extending our algorithm for the general case to avoid near-misses; like the general algorithm, it runs in $O(n^2 \log n)$ time, and requires $O(n^2)$ space.

The paper is structured as follows. In Section 2, we recall some of the terminology pertaining to the problem in question and present our notation. The algorithm is outlined in Section 3. Section 4 deals with the problem of computing the optimal solution among all the vertically partitioned connection patterns, which we achieve in $O(n^2)$ time. The description of the algorithm is completed in Section 5, where we show how we can compute the optimal solution among all the connection patterns that are *not* vertically partitioned in $O(n^2 \log n)$ time. Section 6 deals with the case where near-misses are disallowed, discussing how the algorithm for the general case needs to be modified; interestingly, the changes do not affect the asymptotic performance of the general algorithm. Finally, Section 7 concludes the paper with a summary of results, some final remarks, and open questions.

2. Terminology and Notation.

We begin by mentioning that we assume, without loss of generality, that the given n nodes are all contained in the rectangle with vertices at the grid points $(1,1)$, $(max\_row,1)$, (max_row,max_col), and $(1,max\_col)$, where

$$max\_row \;\leq\; n \qquad\qquad \text{and} \qquad\qquad max\_col \;\leq\; n. \tag{1}$$

This configuration can be achieved by ignoring the rows and columns of the grid that do not contain any nodes, and by renumbering the remaining ones; this normalization process can be performed in $O\big(\min\{n+\Delta_r+\Delta_c,\ n\log n\}\big)$ time, where Δ_r and Δ_c are the length and width of the smallest rectangle enclosing all n nodes in the grid initially.

Since connections to the grid boundary are restricted to be parallel to the grid axes, a node can be connected to the boundary by means of a line segment along one of four possible directions, namely, *up*, *down*, *left*, or *right*. Not all four of the directions may be allowed, however, especially if other nodes are located in the same row or column. Let R be a simple rectilinear polygon formed by a connected set of grid squares, which has been associated with one or more of the above four directions. Then, by a *connection pattern* in R, we refer to a set of non-intersecting line segments, each corresponding to a different node in R which it connects to the boundary of the grid along one of the associated directions. A connection pattern in R that maximizes the number of such line segments is called *optimal*, and the corresponding maximum number (of line segments) is referred to as the *optimal solution* in R. It should be obvious that there may be more than one optimal connection pattern, but there is only one optimal solution. A connection pattern is characterized as *vertically partitioned* if there is a column c $(1 \leq c < max\_col)$ such that no connection (line segment) intersects the *interior* of the vertical strip bounded by the columns c and $c+1$. See also [2]. From the same reference, we borrow the notion of *quadrants* of a node; the four quadrants of node p, denoted by A, B, C, and D counterclockwise around p, are depicted in Figure 3.

Next, we define the notion of a *near-miss*: a horizontal (vertical resp.) near-miss is formed by a pair of connections in adjacent rows (columns resp.) which are intersected by at least two common columns (rows resp.) of the grid. Figure 4 depicts a horizontal near-miss formed by the connections of p and q; note that neither the connections of p and r, nor those of q and s form a near-miss.

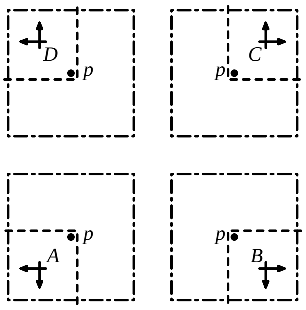

Figure 3

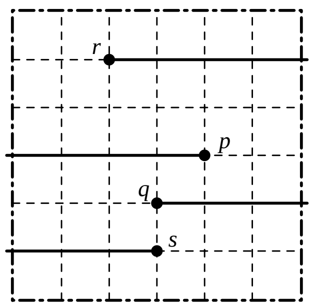

Figure 4

Often, in the following, we qualify optimal solutions in a portion in the grid with the associated allowed directions. For example, an "optimal *left-down* solution" would involve line segments connecting nodes to the left and bottom boundary sides of the grid only. In figures, allowed directions are indicated by arrows; for instance, an arrow pointing left indicates that nodes are allowed to be connected to the left side of the grid boundary. Finally, we use the notation $R(r_1, r_2; c_1, c_2)$ (where $r_1 \le r_2$ and $c_1 \le c_2$) to denote the rectangular portion of the grid enclosed by and including rows r_1 and r_2, and columns c_1 and c_2.

3. The Algorithm.

As observed in [2] (Lemma 11), the optimal connection pattern for the entire grid either has a vertical partitioning or is of one of the two symmetric configurations depicted in Figure 5 (the left and right connections shown are meant to indicate a pair of left and right connections in the optimal connection pattern whose horizontal spans overlap, and which have minimum separation; this justifies the assigned directions along which connections are allowed in the indicated grid partition). Hence, the maximum number of nodes in the grid (from the given set) that can be connected to the grid boundary by means of non-intersecting line segments is the maximum of the outcomes of the following two steps:

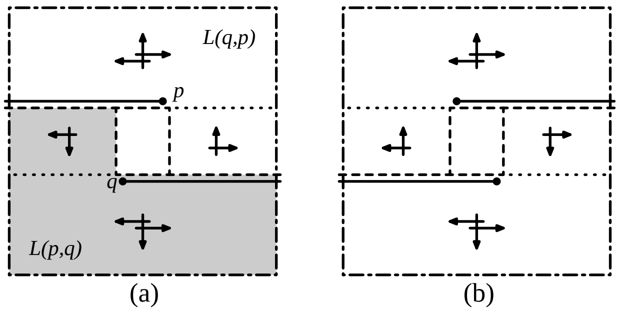

Figure 5

Step 1: compute the optimal solution over all vertically partitioned connection patterns;

Step 2: compute the optimal solution over all connection patterns that are not vertically partitioned (configurations shown in Figure 5).

Clearly, the total time required is dominated by the time that Steps 1 and 2 take. The methods to deal with these two steps are discussed in Sections 4 and 5 respectively.

4. Vertically Partitioned Connection Patterns.

According to its definition, a vertically partitioned connection pattern is characterized by a column c ($1 \le c < max\_col$) such that the nodes in the rectangle $R(1, max\_row; 1, c)$ are connected up, down, or left, while the nodes in the rectangle $R(1, max\_row; c+1, max\_col)$ are connected up, down, or right (Figure 6). Clearly, the solutions in the two rectangles are independent of each other. Therefore, the optimal solution corresponding to a vertically partitioned connection pattern in the grid is

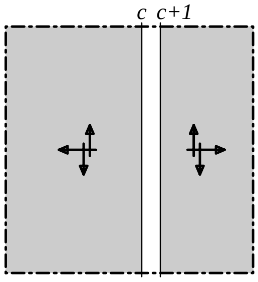

Figure 6

$$\max_{1 \le c < max\_col} \{ \quad \text{optimal } \textit{up-down-left} \text{ solution in } R(1, max\_row; 1, c) \\ + \quad \text{optimal } \textit{up-down-right} \text{ solution in } R(1, max\_row; c+1, max\_col) \quad \}.$$

In light of the normalization conditions (1), this implies that the optimal solution can be found in $O(n)$ time, provided that we have precomputed (and tabulated in $O(n)$ space) the above mentioned optimal solutions in the rectangles $R(1, max\_row; 1, k)$ $(1 \le k < max\_col)$ and $R(1, max\_row; k', max\_col)$ $(1 < k' \le max\_col)$.

In [2], Bruck and Roychowdhury present a method to compute all the above partial optimal solutions in $O(n^2)$ time, provided that the optimal solutions in each of the four quadrants of each node have been precomputed. Their approach to carry out this preprocessing step involves computing each of these solutions independently in $O(n^{1.5} \log n)$ time per node, for a total of $O(n^{2.5} \log n)$ time. However, the optimal solutions in corresponding quadrants of two nodes are not completely independent, and we can use this fact to our advantage. Indeed, thanks to recursive definitions that we establish, we are able to compute the optimal left-down solution in the quadrant A of all the nodes in $O(n^2)$ total time (Section 4.1); extending the method to deal with solutions in the three remaining quadrants is trivial. Since there are only four such quadrants, the total time to precompute all these optimal solutions is still $O(n^2)$.

Moreover, the idea of using recursive definitions can be also applied for the computation of the optimal solutions in $R(1, max\_row; 1, k)$ and $R(1, max\_row; k', max\_col)$. In Section 4.2, we show how we can compute the optimal up-down-left solution in $R(1, max\_row; 1, k)$ for all k such that $1 \le k < max\_col$ in $O(n)$ time, after the optimal solutions in the quadrants of all the nodes have been precomputed; computing the optimal up-down-right solution in $R(1, max\_row; k', max\_col)$ $(1 < k' \le max\_col)$ in $O(n)$ time is almost identical, except that we process the grid columns from right to left.

The above discussion and the results in Sections 4.1 and 4.2 yield:

Lemma 4.1. *Given n nodes in the grid, the maximum number of them that can be connected to the grid boundary by means of non-intersecting line segments that form a vertically partitioned connection pattern can be computed in $O(n^2)$ time and $O(n)$ space.*

4.1. Computing the optimal left-down solution in the quadrant A of all the nodes.

Although seemingly counterintuitive, it turns out that computing the optimal left-down solution in the quadrant A of *every grid vertex* takes less time than computing the corresponding solution for each node independently. Let $a(i, j)$ $(0 \le i \le max\_row,\ 0 \le j \le max\_col)$ denote the maximum number of nodes in the rectangle $R(0, i; 0, j)$ that can be connected to the left or bottom grid boundaries by means of non-intersecting line segments; clearly, the optimal left-down solution in the quadrant A of a node located in row r and column c is precisely the value $a(r, c)$ (see Figure 3).

Let us try to find a recursive definition for $a(i,j)$. First, since no nodes are found in either the 0-th row or the 0-th column,

$$\begin{aligned} a(i,0) &= 0, \qquad \forall\, i:\ 0 \le i \le max\_row; \\ a(0,j) &= 0, \qquad \forall\, j:\ 0 \le j \le max\_col. \end{aligned}$$

For the $a(i,j)$s, where both i and j are positive, we distinguish the following two cases:

1. *No node in the i-th row of $R(0,i;0,j)$, if any, is connected down:* Then, the optimal connection pattern in $R(0,i;0,j)$ consists of an optimal connection pattern in $R(0,i-1;0,j)$ plus a connection to the left from the leftmost node among the nodes in the i-th row of $R(0,i;0,j)$, if any such nodes exist. In other words, the optimal solution is

$$opt_1 \;=\; a(i-1,j) \;+ \begin{cases} 1, & \text{if } \texttt{leftmost}[i] \le j; \\ 0, & \text{otherwise,} \end{cases}$$

   where `leftmost`$[k]$ is equal to the column on which the leftmost node in row k is located.

2. *No node in the j-th column of $R(0,i;0,j)$, if any, is connected left:* Then, the optimal connection pattern in $R(0,i;0,j)$ consists of an optimal connection pattern in $R(0,i;0,j-1)$ plus a connection to the bottom boundary side of the grid from the lowest node among the nodes in the j-th column of $R(0,i;0,j)$, if any such nodes exist. In other words, the optimal solution is

$$opt_2 \;=\; a(i,j-1) \;+ \begin{cases} 1, & \text{if } \texttt{lowest}[j] \le i; \\ 0, & \text{otherwise,} \end{cases}$$

   where `lowest`$[k]$ is equal to the row on which the lowest node in column k is located.

It is not difficult to see that these two cases cover all possible connection patterns. Connection patterns that do not fall in either case 1 or 2 must have a connection down from a node in the i-th row *and* a connection left from a node in the j-th column of $R(0,i;0,j)$; such a connection pattern, however, contains at least a pair of intersecting connections, and is thus disallowed. Therefore, the optimal left-down solution in $R(0,i;0,j)$ is the maximum of the quantities opt_1 and opt_2, that is,

$$a(i,j) \;=\; \max\{\ opt_1,\ opt_2\ \}.$$

Provided that the arrays `leftmost[ ]` and `lowest[ ]` are available (they can be computed in $O(n)$ time), and the values of $a(i-1,j)$ and $a(i,j-1)$ are already known, the computation of $a(i,j)$ takes constant time. This implies that, if the $a(i,j)$s are computed by increasing row and in each row by increasing column, the entire computation will require $O(n^2)$ total time. Moreover, the total space required is only $O(n)$; notice that since $a(i,j)$ depends on $a(i-1,j)$ and $a(i,j-1)$, and the computation proceeds by increasing row, we need only maintain the values of $a(i,j)$ in the current row and the row below it. In fact, $max\_col + 1 \le n+1$ entries suffice, as illustrated by the use of the array `t[ ]` in the piece of C code included in Figure 7 that accomplishes the entire computation. The array has at most $n+1$ entries, and immediately before $a(r,c)$ is computed, its contents are:

$$\texttt{t}[k] \;=\; \begin{cases} a(r,k), & \text{if } 0 \le k < c; \\ a(r-1,k), & \text{if } c \le k \le max\_col. \end{cases}$$

```
/*  the array  node_ptr[0..n-1]  contains pointers to the nodes on the grid  */
/*  the array has been sorted by increasing row of the corresponding nodes   */
/*                  and by increasing column for the nodes in the same row   */
for (i = 0; i <= max_col; i++)   /* initialize table t[ ] to 0 */
    t[i] = 0;
for (i = k = 0; k < n && ++i <= max_row; )   /* rows;  k indexes node_ptr[ ] */
    for (j = 0; ++j <= max_col; ) {          /* columns */
        val = t[j] + (leftmost[i] <= j);
        if (val < (temp = t[j-1] + (lowest[j] <= i)))
            val = temp;
        t[j] = val;

        /*  if there exists a node at (i,j), update field "a" in its record  */
        if (node_ptr[k]->row == i && node_ptr[k]->column == j) {
            node_ptr[k]->a = t[j];
            if (++k >= n)      /* increment index of node_ptr[ ] */
                break;
        }
    }
```

Figure 7: Computing the optimal left-down solution in the quadrant A of all the nodes.

4.2. Computing the optimal *up-down-left* solution in $R(1, max\_row; 1, k)$ for all k such that $1 \leq k < max\_col$.

First, we consider the case where $k = 1$, i.e., we are trying to compute the optimal solution for the nodes in the first column, where their associated line segments can be connected up, down, or left only. Then, all these nodes can be connected left without conflict, and this obviously forms an optimal connection pattern. Therefore,

$$\text{optimal up-down-left solution in } R(1, max\_row; 1, 1) \;=\; (\text{number of nodes in column 1}).$$

In case $k > 1$, the optimal connection pattern may be of one of the following two configurations:

1. *None of the nodes in the k-th column is connected left:* This implies that the nodes in the k-th column, if any, can be connected up or down only, and hence their connections do not interfere with the connection pattern in the columns 1 through $k-1$. Therefore, the optimal solution sought in $R(1, max\_row; 1, k)$ equals

$$\text{optimal up-down-left solution in } R(1, max\_row; 1, k-1) + \begin{cases} 0, & \text{if } \exists \text{ no nodes in column } k, \\ 1, & \text{if } \exists\ 1 \text{ node in column } k, \\ 2, & \text{if } \exists \geq 2 \text{ nodes in column } k. \end{cases}$$

   The first term accounts for the optimal solution involving the nodes in the columns 1 through $k-1$, while the second term accounts for the optimal solution in the k-th column, where the optimal connection pattern consists of the top and bottom nodes (if any) being connected up and down respectively.

2. *At least one of the nodes in the k-th column is connected left:* Figure 8 shows the configuration of the connection pattern if a node p in the k-th column is connected left. Then, in light of the definitions of the quadrants associated with a node, the corresponding optimal solution is $p.a + p.d - 1$; $p.a$ and $p.d$ are the optimal solutions in the quadrants A and D of p respectively (note that if p can be connected left, there exists an optimal connection pattern in quadrant A of p such that p is connected left and the number of nodes connected to the grid boundary is $p.a$), while the term -1 takes care of the fact that the contribution of p has been counted in both $p.a$ and $p.d$. As a result, the optimal solution in this case is

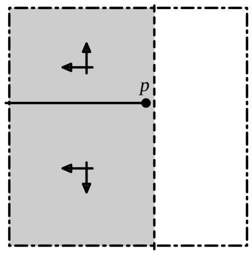

Figure 8

$$\max_{\text{node } p} \{p.a + p.d - 1\}$$

over all nodes p in the k-th column that can be connected left.

The above discussion implies that, provided that the optimal up-down-left solution in $R(1, max\_row; 1, k-1)$ and the optimal solutions in the quadrants A and D of each node are known, the optimal up-down-left solution in $R(1, max\_row; 1, k)$ can be computed in $O(n_k + 1)$ time, where n_k denotes the number of nodes in the k-th column. In other words, ignoring the time to precompute the optimal solutions in the quadrants of each node, the entire computation takes $O(n)$ time, if we sweep the grid from left to right computing the optimal up-down-left solutions in $R(1, max\_row; 1, k)$ by increasing k (we take into account normalization conditions (1)).

5. Connection Patterns that are not Vertically Partitioned.

In addition to the improvement presented in Section 4.1, the key ingredient that allows us to compute the optimal solution that is not vertically partitioned in $O(n^2 \log n)$ time (instead of $O(n^3)$ as described in [2]) is the use of a new type of priority search trees, which we introduce in Section 5.2; these trees enable us to compute the optimal solution (for a part of the grid) among a number of candidates in time logarithmic in their number. Moreover, the total space they take is $O(n^2)$; thus, the space complexity of the algorithm does not increase in the asymptotic sense. (In fact, if we reuse the space that an obsolete such tree is taking, $O(n)$ space suffices.)

As mentioned earlier, the two configurations of the optimal connection pattern that are not vertically partitioned are shown in Figure 5. Let us concentrate on the configuration (a); the configuration (b) is an easy extension. It is immediate from the figure that the solutions in the two L-shaped polygons defined by the line segments associated with p and q are independent; so, if we denote the lower (upper) one as $L(p, q)$ ($L(q, p)$ resp.), the optimal solution among all configuration (a) connection patterns is

$$\max_{p,q} \{ \text{ optimal solution in } L(p, q) \ + \ \text{optimal solution in } L(q, p) \ \}, \tag{2}$$

where the optimal solutions satisfy the restrictions indicated in Figure 5(a), and the maximum is computed over all pairs of nodes p and q such that they can be connected left and right respectively, and p is above and to the right of q. Therefore, assuming that the optimal solutions in $L(p,q)$ and $L(q,p)$ are available for all such pairs of nodes p and q, the optimal solution in the entire grid can be computed in time linear in the total number of such pairs; hence, in $O(n^2)$ time.

Section 5.1 outlines the algorithm to compute the optimal solution in $L(p,q)$ (the optimal solution in $L(q,p)$ can be found similarly). Before that, however, we present the data structures that we will need.

`r_left`[ ][ ]: for each node t that can be connected **right**, we maintain an array `r_left`[t][ ]; if t is located at the vertex (r,c) of the grid, the contents of `r_left`[t][ ] are

$$\texttt{r\_left}[t][k] = \begin{cases} 0, & \text{if } 1 \le k \le r \\ \text{optimal left solution in } R(r+1,k;1,c-1), & \text{if } r+1 \le k \le n \end{cases}$$

`r_right`[ ][ ]: for each node t that can be connected **right**, we maintain an array `r_right`[t][ ]; if t is located at the vertex (r,c) of the grid, the contents of `r_right`[t][ ] are

$$\texttt{r\_right}[t][k] = \begin{cases} 0, & \text{if } 1 \le k \le r \\ \text{optimal right solution in } R(r+1,k;c+1,n), & \text{if } r+1 \le k \le n \end{cases}$$

`d_down`[ ][ ]: for each node t that can be connected **down**, we maintain an array `d_down`[t][ ]; if t is located at the vertex (r,c) of the grid, the contents of `d_down`[t][ ] are

$$\texttt{d\_down}[t][k] = \begin{cases} 0, & \text{if } 1 \le k \le c \\ \text{optimal down solution in } R(1,r;c+1,k), & \text{if } c+1 \le k \le n \end{cases}$$

The entries in each of these arrays can be computed in $O(n)$ time per node, so that the total time and space required is $O(n^2)$. (Note that, in order to compute the optimal solution in $L(q,p)$, we will need arrays `l_right`[ ][ ], `l_left`[ ][ ], and `u_up`[ ][ ] defined similarly.)

5.1. Outline of the algorithm to compute the optimal solution in $L(p,q)$.

In very general terms, the method to compute the optimal solution in $L(p,q)$ (for a given pair of nodes p and q as shown in Figure 5(a), i.e., $p.row > q.row$, $p.column \ge q.column$, and p and q can be connected left and right respectively) proceeds as follows (see also [2]):

1. For each pair of nodes u and v as shown in Figure 9 (i.e., $u.row > v.row$, $u.column < v.column$, and u and v can be connected down and right respectively), we compute the optimal solution in the L-shaped portion of the grid, denoted as $M(u,v)$, under the indicated direction restrictions. In order to do that, we consider all possible instances of the configuration shown in Figure 10, where the node t is the leftmost node in $M(u,v)$ connected right; for each such case, the corresponding optimal solution in $M(u,v)$ is given by

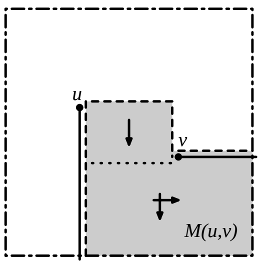

Figure 9

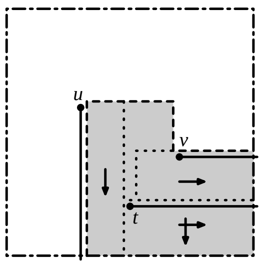

Figure 10

$$\texttt{d\_down}[u][t.column-1] \;+\; \texttt{r\_right}[t][v.row] \;+\; t.b,$$

where $t.b$ is the optimal down-right solution in the quadrant B of t. Note that t can be v or any node in the rectangle $R(1, v.row-1; u.column+1, v.column-1)$ that can be connected right. The optimal solution in $M(u,v)$ for each such pair u and v is stored in an array (of $O(n^2)$ size) for quick referencing in the next step.

2. The optimal solution in $L(p,q)$ is of one of the two configurations shown in Figure 11. We need therefore to compute the corresponding optimal solutions; the largest one yields the optimal solution in $L(p,q)$. In particular, the optimal solution (in $L(p,q)$) that corresponds to the configuration (a) is

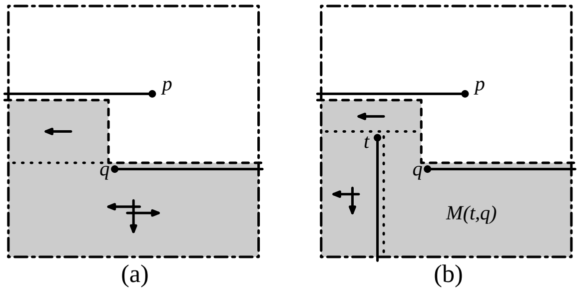

Figure 11

$$\text{optimal down-left-right solution in } R(1, q.row; 1, max\_col) \; + \; \texttt{r\_left}[q][p.row - 1], \qquad (3)$$

where the optimal down-left-right solutions in $R(1, i; 1, max\_col)$ for all i $(1 \leq i < max\_row)$ can be precomputed in $O(n)$ time (plus $O(n^2)$ preprocessing time) in a fashion similar to that described in Section 4.2. In the configuration (b), the node t is meant to be the highest node connected down in the optimal connection pattern in $L(p,q)$, and the corresponding optimal solution is

$$t.a \; + \; \big(\texttt{r\_left}[q][p.row - 1] - \texttt{r\_left}[q][t.row]\big) \; + \; \text{optimal solution in } M(t,q). \qquad (4)$$

Note that t may be any node in the rectangle $R(q.row + 1, p.row - 1; 1, q.column - 1)$ that can be connected down.

Each of the above steps can be accomplished in $O(n^3)$ time ([2]). We improve on that by using appropriate priority search trees, which we describe in the next section.

5.2. The Data Structure: A new type of Priority Search Trees

Suppose that we are given an ordered set S of m objects (in our case, nodes in the grid, whose order is defined as a function of their coordinates). Additionally, the elements of S have been assigned priorities. Our objective is to perform the following operations on the elements of S fast:

MaxUpTo(p):	find the maximum among the priorities of all the elements of S preceding p and including p if it belongs to S.
SubUpTo(k, p):	subtract k from the priority of all the elements of S preceding p.
Remove(p):	remove the element p from the set S.

It is interesting to observe that the description of the elements of S (and to a lesser degree, the desired operations) suggests that a priority search tree, that is, a balanced binary search tree with an additional heap ordering, may be the right choice for the data structure to represent the set S. Unfortunately, the standard priority search described in [5] (see also [6]) stores elements at the internal nodes of the tree, which causes the **SubUpTo( )** operation to take $\Omega(\log^2 m)$ time in the worst case: imagine the case where the first $\log m/100$ elements of S have the largest priorities so that they are located at the top $\log m/100$ levels of the tree (one per level), and assume that we reduce their priorities by such an amount that they all have to be moved to the bottom $\log m/100$ levels; filling each one of the $\Omega(\log m)$ vacant slots can be viewed as moving the vacant slot all the way down the tree, thus incurring an $\Omega(\log m)$ cost.

We therefore use a different type of priority search trees. In particular, we build a balanced binary search tree (a red-black tree for instance) whose leaves from left to right correspond to the elements of S in order. (Although such a tree requires almost twice the space that a priority search tree like the one described in [5] does, it helps us perform the **SubUpTo( )** operation in $O(\log m)$ time.) Moreover, we need to establish a heap ordering of priorities in the tree. Ideally, we would just have a single field, say *max*, in every tree node that stores the maximum over the priorities of all the leaves of the subtree rooted at that node, or equivalently, for an internal tree node t

$$t.max \;=\; \max\{\; c_l.max,\; c_r.max \;\}, \tag{5}$$

where c_l and c_r are the two children of t. However, in order to be able to perform the **SubUpTo( )** operation in logarithmic time, we associate each tree node with two fields, *debt* and *max*, the idea being that the field *debt* stores the amount by which the priorities of all the leaves descending from that node have to be reduced. Thanks to this field, subtracting k from the priorities of the first m elements of S reduces to finding the (at most $2\ \log m$) maximal subtrees whose sets of leaves partition the set of the m leftmost leaves of our priority search tree, and adding k to the *debt* fields of their roots. In the beginning, all the *debt* fields are initialized to 0; the *max* fields of the leaves are initialized to the priorities of the associated elements of S, while the *max* fields of the internal nodes are initialized according to equality (5). Clearly, building and initializing the tree takes time linear in the size of S.

The introduction of the *debt* fields has two implications: first, the heap ordering condition becomes

$$t.max \;=\; \max\{\; c_l.max - c_l.debt,\; c_r.max - c_r.debt \;\} \tag{6}$$

(note that initializing the *max* fields according to equality (5) agrees with equality (6), since all the *debt* fields are initialized to 0); second, the current priority of an element of S associated with a leaf t equals $t.max$ minus the sum of the *debt* fields of all the tree nodes on the path from the root to t (inclusive).

Let us now see how we can perform each of the above operations fast.

- **MaxUpTo(p):** Thanks to the heap ordering, finding the maximum over the priorities of the elements of S preceding and including p if it belongs to S reduces into descending the tree guided by p, and computing the maximum over the quantities

$$c_l.max \;-\; c_l.debt \;-\; \sum_{\text{ancestor } a \text{ of } c_l} a.debt$$

  of the left child c_l at each tree node where the path we are following makes a right turn; note that each such child is the root of a subtree, whose leaves all correspond to elements of S preceding p. Finally, if p is in S, we also take into account p's priority, i.e.,

$$\tau_p.max \;-\; \tau_p.debt \;-\; \sum_{\text{ancestor } a \text{ of } \tau_p} a.debt,$$

  where τ_p is the tree leaf corresponding to p.

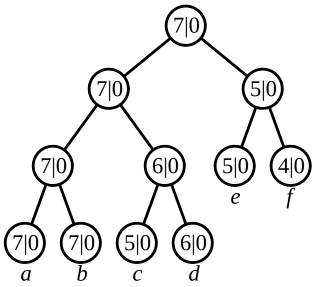

Figure 12

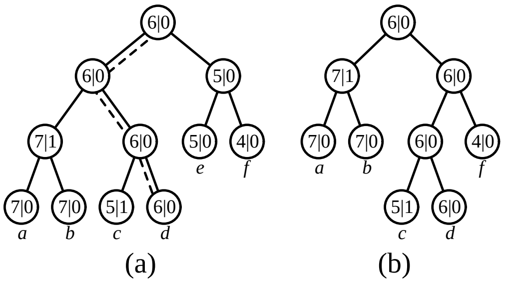

Figure 13

- **SubUpTo(k, p):** As described earlier, we descend the tree guided by p, and we add k to the *debt* field of the left child of every tree node where we take a right turn. After reaching a leaf, we walk up the same path while making sure that the *max* field of each of the nodes in the path satisfies the heap invariant (6), updating the field appropriately if needed.

- **Remove(p):** The removal consists of two steps. First, we reduce the priority of p, so that it does not contribute to the heap ordering at all. Unless the entire tree is just a single node, in which case we just discard it and terminate, the tree leaf τ_p corresponding to p has a parent that has another child, say τ_q, corresponding to another element q of S. Then we compute $d = \tau_p.max - \tau_p.debt - \tau_q.max + \tau_q.debt$: if $d \leq 0$, the priority of p is no more than that of q, and we proceed to the second step; otherwise, we add d to $\tau_p.debt$ (which is sufficient to reduce the priority of p so that it equals the priority of q), and then we move up from τ_p to the root, if needed, updating the *max* fields, so that equality (6) holds. In the second step, we remove the tree leaf τ_p, and reduce the 3-node subtree rooted at τ_p's parent, say t, into a single leaf that inherits τ_q's fields, except that its *debt* field is set to $t.debt + \tau_q.debt$. The removal may require rebalancing, and local updating of the *max* fields if a rotation occurs.

Figure 12 shows the initial tree for the set $S = \{a(3,5),\ b(5,6),\ c(1,6),\ d(4,7),\ e(6,8),\ f(2,8)\}$ of nodes in the grid (by $p(r,c)$, we denote the node p located in row r and column c of the grid), and respective priorities 7, 7, 5, 6, 5, and 4. (The nodes have been ordered in S by increasing column and decreasing row for nodes in the same column.) The letter below each leaf of the tree indicates the corresponding node in S; the two numbers in each tree node correspond to its *max* and *debt* fields respectively. Figure 13 shows the transformations that the tree of Figure 12 undergoes after a **SubUpTo**$(1, (6,7))$ operation (Figure 13(a)), and then after a **Remove**(e) operation (a double rotation at the tree root; Figure 13(b)).

The above discussion implies that performing each of the three operations as described above takes time linear in the height of the priority search tree, that is, time logarithmic in the number of leaves of the tree, since the tree is balanced. Summarizing, we have:

Lemma 5.1. *Given a set S of size m, a priority search tree of the type we introduced in this section enables us to perform each of the* ***MaxUpTo( )***, ***SubUpTo( )***, *or* ***Remove( )*** *operation on S in $O(\log m)$ time. The priority search tree requires $O(m)$ time to construct and $O(m)$ space to store.*

We close this section by mentioning that the priority search tree we introduced supports other operations in logarithmic time as well: insertion, computing the maximum over the priorities of all the elements of S *between* two (potential) elements of S (the equivalent of **MinYinXRange( )** described in [5]), and the equivalents of **MinXinRectangle( )** and **MaxXinRectangle( )** described in [5]. The equivalent of **EnumerateRectangle( )**, however, may take as much as $\Theta\big((s+1)\log m\big)$ time, if s out of the m elements of S are reported.

5.3. Computing the optimal solution in $L(p,q)$ for all the appropriate pairs (p,q) in $O(n^2 \log n)$ time.

In order to be able to perform the first step of the algorithm described in Section 5.1 in $O(n^2 \log n)$ time, we build a priority search tree T_u for each node u that can be connected down. The tree is built on the set S_u of nodes in the quadrant B of u (except u) that can be connected right. The priority of such an element (node) t equals the optimal solution in the rectangle $R(1, u.row; u.column, max\_col)$ under the restrictions shown in Figure 14 (t is assumed to be the leftmost node connected right), which is

$$\texttt{d\_down}[u][t.column-1] \;+\; \texttt{r\_right}[t][u.row] \;+\; t.b.$$

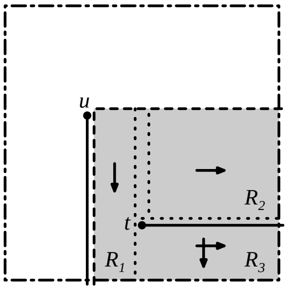

Figure 14

The three terms correspond to the optimal solutions in the rectangles R_1, R_2, and R_3 respectively. The elements of S_u are ordered by increasing column (and decreasing row, if two nodes happen to be in the same column), which can be done in linear time using radix-sort. According to Lemma 5.1, the tree T_u of a node u requires $O(n)$ time to construct and $O(n)$ space to store.

Figure 12 depicts the tree T_u that corresponds to the node u in Figure 15. Note that the node z is not associated with any tree leaf; as it cannot be connected right, it is not an element of S_u on which T_u is built. Figure 16 shows optimal connection patterns that justify the priorities associated with nodes b and c.

Let us now see how we can use T_u to compute the optimal solution in $M(u,v)$ for all the nodes v that define such a configuration with u in $O(n \log n)$ total time. As suggested by Figure 9, these are the nodes in the quadrant B of u (except u) that can be connected right, that is, the elements of S_u.

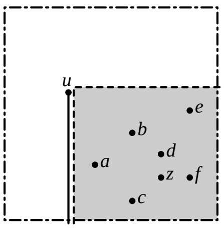

Figure 15

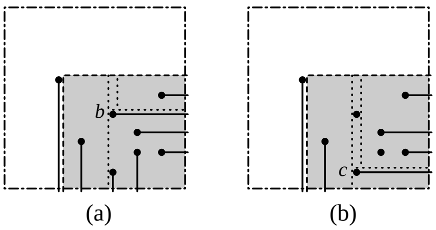

Figure 16

1. We copy the elements of S_u in a new list, and we radix-sort them by decreasing row in linear time (it is worth noting that we have at most one such node per row).
2. Next, we process each such node, say v, in order as follows:

   (i) we compute the optimal solution in $M(u, v)$; this is precisely the maximum of the optimal solutions associated with v and all the nodes to the left and below v that are associated with T_u's leaves. In light of the ordering established in the set S_u and the operations of our priority search tree, the optimal solution is the value returned by $\mathbf{MaxUpTo}(v)$ in T_u.

   (ii) next, we perform a $\mathbf{SubUpTo}(1, v)$ operation in T_u to reduce the optimal solutions associated with each of the nodes to the left and below v by 1; note that v contributed a term $+1$ in the optimal solution in the rectangle R_2 corresponding to each of these nodes (compare Figures 10 and 14).

   (iii) finally, we remove the node v from the set S_u (actually, the leaf associated with v in T_u) by performing a $\mathbf{Remove}(v)$ operation.

Since the number of elements of S_u is $O(n)$ initially, and always decreases as we process them, each of the **MaxUpTo( )**, **SubUpTo( )**, and **Remove( )** operations takes $O(\log n)$ time to execute (Lemma 5.1); therefore, the total time required to find the optimal solutions in $M(u, v)$ for all the appropriate vs is $O(n \log n)$. In other words, the first step of the algorithm outlined at the beginning of Section 5.1 can be performed in $O(n^2 \log n)$ time.

In a similar fashion, we can execute the second step of the algorithm in $O(n^2 \log n)$, assuming that the optimal solutions for the pairs considered in the first step have been tabulated for easy lookup. This time, we build a priority search tree T_q for each node q that can be connected right. The tree is built on the set S_q of nodes in the quadrant D of q that can be connected down, plus a dummy node located at (0,0). The priority of the dummy node is equal to the optimal solution for the configuration shown in Figure 17(a), that is,

$$\text{optimal down-left-right solution in } R(1, q.row; 1, max\_col) \ + \ \texttt{r\_left}[q][max\_row]. \tag{7}$$

The priority of any other element (node) t of S_q is equal to the optimal solution in the region shown in Figure 17(b) under the indicated restrictions (t is assumed to be the highest node connected down), which is

$$t.a + \big(\texttt{r\_left}[q][max\_row] - \texttt{r\_left}[q][t.row]\big)$$
$$+ \text{ optimal solution in } M(t, q). \tag{8}$$

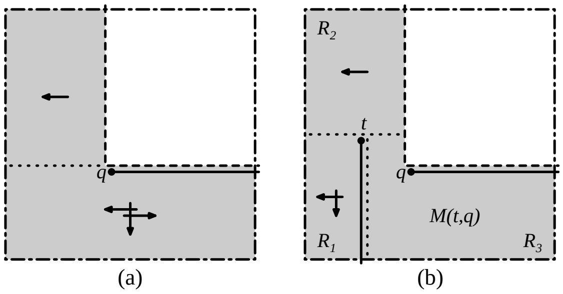

Figure 17

The three terms correspond to the optimal solutions in the rectangles R_1, R_2, and R_3 respectively. The elements of S_q are ordered by increasing row (and decreasing column, if two nodes happen to be in the same row), which again can be done in linear time using radix-sort.

Since the size of S_q is $O(n)$, Lemma 5.1 implies that the tree T_q of the node q requires $O(n)$ time to construct and $O(n)$ space to store, and yields the optimal solution in $L(p,q)$ for all the nodes p that define with q such a configuration (Figure 5(a)) in $O(\log n)$ time; these are the nodes in the quadrant C of q (except q) that can be connected left. Indeed, for each such p, we perform a **MaxUpTo(**p**)** to compute the maximum among the optimal solutions associated with each of the nodes in S_q located below the row of p. If the value returned is val, then the optimal solution in $L(p,q)$ (compare Figures 11 and 17, and expressions (3) and (7), and (4) and (8)) is

$$val \;-\; \big(\texttt{r\_left}[q][max\_row] - \texttt{r\_left}[q][p.row]\big).$$

Since the **MaxUpTo( )** operation takes time logarithmic in the size of S_q, the total time required to find the optimal solution in $L(p,q)$ for each such pair p and q is $O(\log n)$. In other words, the second step of the algorithm outlined at the beginning of this section can be performed in $O(n^2 \log n)$ time.

Summarizing, the priority search trees help us find the optimal solution in $L(p,q)$ (and similarly in $L(q,p)$) for all appropriate p and q in $O(n^2 \log n)$. Then, taking into account expression (2), we have

Lemma 5.2. *Given n nodes in the grid, the maximum number of them that can be connected to the grid boundary by means of non-intersecting line segments forming a connection pattern that is not vertically partitioned can be computed in $O(n^2 \log n)$ time and $O(n^2)$ space.*

6. Disallowing Near-Misses.

When dealing with near-misses, it is important to know whether two nodes are in adjacent rows or columns, although the exact number of empty rows or columns between them does not really matter. In other words, we can merge any positive number of successive rows (columns resp.) into a single row (column resp.) without altering the problem configuration. We can, therefore, assume without loss of generality that the given nodes are all contained in the rectangle with vertices at the grid points $(1,1)$, $(max\_row, 1)$, (max_row, max_col), and $(1, max\_col)$, where

$$max\_row \;\le\; 2n \qquad\qquad \text{and} \qquad\qquad max\_col \;\le\; 2n. \tag{9}$$

Our basic strategy will be the same as that for the general problem; so, below, we identify the places where near-misses may arise, and appropriately modify the general algorithm to safeguard against them.

6.1. Vertically Partitioned Connection Patterns.

We start by pointing out that, since the connections in each of the quadrants of the given grid nodes do not involve opposite directions, no near-miss can possibly occur in the corresponding optimal solutions, and therefore the technique of Section 4.1 can be applied without modifications.

The difference, however, is that it is no longer true that the optimal connection pattern which is vertically partitioned, say, about column c, consists of the optimal up-down-left connection pattern in $R(1, max\_row; 1, c)$ and the optimal up-down-right connection pattern in $R(1, max\_row; c+1, max\_col)$; the reason is that these two partial optimal solutions may very well exhibit a near-miss in columns c and $c+1$. We need therefore to consider the relative positions of the nodes, if any, in columns c and $c+1$. To facilitate that we assume that we have precomputed arrays `highest[ ]` and `lowest[ ]`, whose entries are defined as follows:

$$\texttt{highest}[k] \;=\; \begin{cases} +\infty, & \text{if no node exists in column } k; \\ v.row, & \text{where } v \text{ is the \textit{highest} node in column } k. \end{cases}$$

$$\texttt{lowest}[k] \;=\; \begin{cases} -\infty, & \text{if no node exists in column } k; \\ v.row, & \text{where } v \text{ is the \textit{lowest} node in column } k. \end{cases}$$

The arrays can be computed in $O(n)$ time and require $O(n)$ space.

Additionally, in order to ensure that no near-miss exists at columns c and $c+1$ (where the vertically partitioning is exhibited), we need to compute optimal solutions in $R(1, max\_row; 1, c)$ (and $R(1, max\_row; c+1, max\_col)$ resp.) under the additional restriction that no node in column c (column $c+1$ resp.) is connected up or down. In particular, we use the following notation:

$l\_complete[k]$:	optimal up-down-left solution in $R(1, max\_row; 1, k)$.
$l\_non\text{-}up[k]$:	optimal up-down-left solution in $R(1, max\_row; 1, k)$, where no node in column k is connected up.
$l\_non\text{-}down[k]$:	optimal up-down-left solution in $R(1, max\_row; 1, k)$, where no node in column k is connected down.

and similarly

$r\_complete[k]$:	optimal up-down-right solution in $R(1, max\_row; k, max\_col)$.
$r\_non\text{-}up[k]$:	optimal up-down-right solution in $R(1, max\_row; k, max\_col)$, where no node in column k is connected up.
$r\_non\text{-}down[k]$:	optimal up-down-right solution in $R(1, max\_row; k, max\_col)$, where no node in column k is connected down.

It turns out that computing the optimal solution over all connection patterns that are vertically partitioned about column c reduces to considering the following three cases:

(i) if $\texttt{highest}[c] \;<\; \texttt{lowest}[c+1]$: Then, connecting the highest node in column c up and the lowest node in column $c+1$ down creates a near-miss (see Figure 18(a)); so, these two connections are incompatible, and therefore the optimal solution in this case is

$$vp\_opt(c) \;=\; \max\{\; l\_complete[c] \;+\; r\_non\text{-}down[c+1],\; l\_non\text{-}up[c] \;+\; r\_complete[c+1] \;\}.$$

(ii) if $\texttt{lowest}[c] \;>\; \texttt{highest}[c+1]$: Similarly to the previous case, connecting the lowest node in column c down and the highest node in column $c+1$ up creates a near-miss (see Figure 18(b)). The optimal solution is

$$vp\_opt(c) \;=\; \max\{\; l\_complete[c] \;+\; r\_non\text{-}up[c+1],\; l\_non\text{-}down[c] \;+\; r\_complete[c+1] \;\}.$$

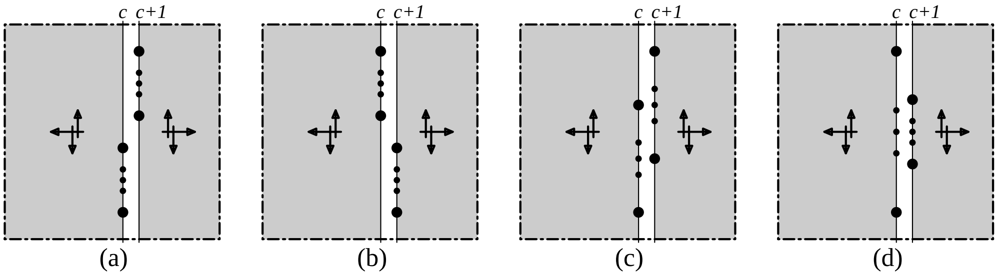

Figure 18

(iii) Otherwise, no near-miss can possibly exist at columns c and $c+1$ (see Figures 18(c) and (d), and symmetric cases); therefore,

$$vp\_opt(c) \;=\; l\_complete[c] \;+\; r\_complete[c+1].$$

(Note that, thanks to the way we defined the entries `highest[ ]` and `lowest[ ]` for a column that does not contain any nodes, if no nodes exist in column c or $c+1$, we end up in case (iii), as desired.) The above procedure allows us to compute the optimal solution $vp\_opt(c)$ of all vertically partitioned connection patterns about a specific column c in constant time, provided that the corresponding *complete*, *non-up*, and *non-down* solutions are available. After this has been done for all columns c ($1 \le c < max\_col$), the optimal solution over all vertically partitioned patterns is simply

$$\max_{1 \le c < max\_col} \{\ vp\_opt(c)\ \}.$$

We show next how to compute the $l\_complete[k]$, $l\_non\text{-}up[k]$, and $l\_non\text{-}down[k]$ solutions for all k: $1 \le k < max\_col$ in $O(n)$ time (and $O(n)$ space to store the values). Computing the $r\_complete[k]$, $r\_non\text{-}up[k]$, and $r\_non\text{-}down[k]$ solutions ($1 \le k < max\_col$) is an easy extension: the columns are processed in a similar fashion from right to left. In light of the above discussion and Lemma 4.1, we conclude that:

Lemma 6.1. *Given n nodes in the grid, the maximum number of them that can be connected to the grid boundary by means of non-intersecting line segments that form a vertically partitioned connection pattern and do not exhibit near-misses can be computed in $O(n^2)$ time and $O(n)$ space.*

6.1.1. Computing the *l_complete*[ ], *l_non-up*[ ], and *l_non-down*[ ] solutions.

Before getting into the details of the method that we use, we prove the following easy lemma, which helps us simplify the final expressions.

Lemma 6.2. *For any column k of the grid ($1 \le k < max\_col$),*

$$l\_non\text{-}up[k] + 1 \;\ge\; l\_complete[k] \qquad \text{and} \qquad l\_non\text{-}down[k] + 1 \;\ge\; l\_complete[k].$$

Similarly,

$$r\_non\text{-}up[k] + 1 \;\ge\; r\_complete[k] \qquad \text{and} \qquad r\_non\text{-}down[k] + 1 \;\ge\; r\_complete[k].$$

Proof: We concentrate on the first of the four inequalities. Consider an optimal connection pattern in $R(1, max\_row; 1, k)$ that corresponds to $l\_complete[k]$. If column k does not contain any nodes, or the connection pattern does not contain a connection up from the highest node in column k, then

$$l\_non\text{-}up[k] \;\geq\; l\_complete[k] \;>\; l\_complete[k] - 1,$$

according to the definition of $l\_non\text{-}up[k]$. Otherwise, removing that connection produces a connection pattern with $l\_complete[k] - 1$ connections, where no node in column k is connected up; thus,

$$l\_non\text{-}up[k] \;\geq\; l\_complete[k] - 1. \qquad \blacksquare$$

In general, the approach is to process the columns from left to right, computing all three solutions *l_complete*[ ], *l_non-up*[ ], and *l_non-down*[ ] for each column before moving to the next one; the reason is that the value of each of the three solutions for a specific column also involves the values of the other quantities at the previous column. However, for simplicity, we present the methods to compute *l_complete*[ ], and *l_non-up*[ ] and *l_non-down*[ ] in separate paragraphs.

Computing the *l_complete*$[k]$ solution $(1 \leq k < max\_col)$: If $k = 1$ (that is, we consider the nodes in the first column only), all the nodes can be connected left. Hence,

$$l\_complete[1] \;=\; \text{(number of nodes in column 1)}.$$

For $k > 1$, we distinguish the following three cases:

1. *If there exist no nodes in column k:* Then, the optimal solution in $R(1, max\_row; 1, k)$ is the same as that in $R(1, max\_row; 1, k-1)$, i.e.,

   $$l\_complete[k] \;=\; l\_complete[k-1].$$

2. *If there exists exactly 1 node, say v, in column k:* Interestingly, if node v can be connected up or down without restriction, we can always connect it to the boundary without creating a near-miss; indeed, we connect v up, if $v.row > \texttt{lowest}[k-1]$, otherwise we connect it down. Therefore,

   $$l\_complete[k] \;=\; l\_complete[k-1] + 1.$$

   Note that we do not need to consider the case where v is connected left, since, due to the optimality of $l\_complete[k-1]$, the quantity $l\_complete[k]$ cannot exceed $l\_complete[k-1] + 1$.

3. *If there exist at least 2 nodes in column k:* Then, the connection pattern either involves a connection left from a node q in column k, or no such connection exists. In the former case, the connection associated with q precludes a near-miss, and the corresponding optimal solution is

   $$c\_opt_1 \;=\; \max_q \{\; q.a + q.d - 1 \;\},$$

   where q ranges over all nodes in column k that can be connected left (the term -1 takes care of the fact that q's connection has been counted both in $q.a$ and $q.d$). In the latter case, nodes in column k can be connected up or down only, and the pattern is vertically partitioned about column $k-1$. We can therefore apply the rules that we came up with in Section 6.1 to combine partial optimal solutions to compute the optimal solution of a vertically partitioned pattern, where $c = k-1$. In this case, the right-hand side portion of the grid contains column k only, and the nodes can be connected up or down only; since, there are at least 2 nodes in column k,

$$r\_complete[k] \;=\; 2 \qquad \text{and} \qquad r\_non\text{-}up[k] \;=\; r\_non\text{-}down[k] \;=\; 1.$$

Then, cases (i), (ii), and (iii) in Section 6.1 translate into the following three case statements respectively, where we also used Lemma 6.2 to eliminate the max{ } operators.

$$c\_opt_2 \;=\; \begin{cases} l\_non\text{-}up[k-1]+2, & \text{if } \texttt{highest}[k-1] \;<\; \texttt{lowest}[k], \\ l\_non\text{-}down[k-1]+2, & \text{if } \texttt{lowest}[k-1] \;>\; \texttt{highest}[k], \\ l\_complete[k-1]+2, & \text{otherwise.} \end{cases}$$

Overall, the optimal solution in this case is

$$l\_complete[k] \;=\; \max\{\; c\_opt_1,\; c\_opt_2 \;\}.$$

Computing the $l\_non\text{-}up[k]$ and $l\_non\text{-}down[k]$ solutions ($1 \le k < max\_col$): As in the case of the $l\_complete[1]$ value,

$$\begin{aligned} l\_non\text{-}up[1] &= \text{(number of nodes in column 1)} \\ l\_non\text{-}down[1] &= \text{(number of nodes in column 1)} \end{aligned}$$

since all the nodes can be connected left. For $k > 1$, we distinguish the following two cases:

1. *If there exist no nodes in column k:* Then, clearly,

$$\begin{aligned} l\_non\text{-}up[k] &= l\_complete[k-1] \\ l\_non\text{-}down[k] &= l\_complete[k-1] \end{aligned}$$

2. *If there exists at least 1 node in column k:* Again, either a node in column k is connected left, or no such connection exists in the connection pattern. In the former case, and assuming that q is the highest node in column k connected left, we have that the corresponding $l\_non\text{-}up$ solution is

$$nu\_opt_1 \;=\; \max_q\{\; q.a + d(q.row, q.column - 1) \;\}.$$

In the latter case, if $\texttt{highest}[k-1] \ge \texttt{lowest}[k]$, we can safely connect the lowest node in column k down. If $\texttt{highest}[k-1] < \texttt{lowest}[k]$, however, the highest node in column $k-1$ and the lowest node in column k would create a near-miss if they are connected up and down respectively; so, these connections cannot coexist, and the optimal solution is

$$\max\{\; l\_complete[k-1],\; l\_non\text{-}up[k-1]+1 \;\},$$

the first term corresponding to a connection pattern where the lowest node in column k is not connected down, and the second term corresponding to a pattern where the highest node in column $k-1$, if any, is not connected up. According to Lemma 6.2, the above maximum is equal to $l\_non\text{-}up[k-1]+1$, and therefore,

$$nu\_opt_2 \;=\; \begin{cases} l\_complete[k-1]+1, & \text{if } \mathtt{highest}[k-1] \geq \mathtt{lowest}[k]; \\ l\_non\text{-}up[k-1]+1, & \text{otherwise.} \end{cases}$$

Summarizing,

$$l\_non\text{-}up[k] \;=\; max\{\; nu\_opt_1,\; nu\_opt_2 \;\}.$$

Similarly, for the $l\_non\text{-}down$ solution, we have

$$l\_non\text{-}down[k] \;=\; max\{\; nd\_opt_1,\; nd\_opt_2 \;\},$$

where

$$nd\_opt_1 \;=\; \max_q \{\; a(q.row, q.column-1) + q.d \;\}$$

over all nodes q in column k that can be connected left, and

$$nd\_opt_2 \;=\; \begin{cases} l\_complete[k-1]+1, & \text{if } \mathtt{lowest}[k-1] \leq \mathtt{highest}[k]; \\ l\_non\text{-}down[k-1]+1, & \text{otherwise.} \end{cases}$$

It should be obvious that computing the optimal solution over all vertically partitioned patterns while disallowing near-misses at the same time requires an additional $O(n)$ space, as opposed to the general algorithm described in Section 4: we need to store all three (partial) solutions $l\_complete[\;]$, $l\_non\text{-}up[\;]$, and $l\_non\text{-}down[\;]$ (and their right-side counterparts) instead of just one as in Section 4; additionally, for each node (located at the grid vertex (r,c)), we need to store the values of $a(\;,\;)$ and $d(\;,\;)$ at the grid vertex $(r,c-1)$, and by symmetry the values of $b(\;,\;)$ and $c(\;,\;)$ at the grid vertex $(r,c+1)$ (which will be needed to compute the $r\_non\text{-}up[c]$ and $r\_non\text{-}down[c]$ solutions). Therefore, according to Lemma 4.1, the total space required at this step is $O(n)$.

Finally, from a time complexity standpoint, computing the $l\_complete[k]$, $l\_non\text{-}up[k]$, and $l\_non\text{-}down[k]$ for all k ($1 \leq k < max\_col$) takes $O(n)$ time: as shown above, the value of each of the above quantities for column k takes $O(n_k+1)$ time given the corresponding values for column $k-1$ (n_k is the total number of nodes in column k); therefore, if we process the columns in increasing order, the overall time will be $O(n)$, since we have normalized the given configuration to fit in a $2n \times 2n$ square (see conditions (9)).

6.2. Connection patterns that are not Vertically Partitioned.

Unlike Section 6.1, only minor changes need to be introduced in the algorithm described in Section 5.3, so that no near-misses are present. It is important to observe that a connection pattern that contains nodes p and q as shown in Figure 5, which are connected to the left and right side respectively, cannot exhibit a vertical near-miss. Horizontal near-misses may however exist, but they can result in the following two places only:

1. **a near-miss could be formed by the connections of nodes p and q:** To eliminate such a possibility, we need to consider only those among the pairs $(p,\ q)$ (that the general algorithm collects; see Section 5.3), where p and q either are on the same column, or otherwise are at least one row away.
2. **at the rectangle $R(1, q.row; 1, max\_col)$, while considering the configuration of Figure 17(a):** Then, in expression (7), we need only use the corresponding *complete* solution (which precludes the existence of near-misses; see Section 6.1).

Clearly, the above changes do not require any additional computation time or space, and therefore, in light of Lemma 5.2,

Lemma 6.3. *Given n nodes in the grid, the maximum number of them that can be connected to the grid boundary by means of non-intersecting line segments, which form a connection pattern that is not vertically partitioned and does not exhibit near-misses can be computed in $O(n^2 \log n)$ time and $O(n^2)$ space.*

7. Concluding Remarks.

Our results are summarized in the following theorem.

Theorem 7.1. *Given n nodes in the grid, the maximum number of them that can be connected to the boundary of the grid by means of non-intersecting line segments can be computed in $O(n^2 \log n)$ time and $O(n^2)$ space. Furthermore, the time and space complexities do not change under the additional constraint that no near-misses are exhibited in the connection pattern.*

Although in this paper, we concentrated on finding the maximum number of non-intersecting connections, the algorithm can be slightly modified to yield an optimal connection pattern as well; we simply need to maintain pointers from each optimal solution to the optimal (partial) solutions that contributed to it (in this case, we may have to store all $O(n^2)$ solutions $a(i, j)$ and their counterparts in the remaining three quadrants; see Section 4.1). The optimal connection pattern can then be computed in $O(n)$ time by tracing back these pointers.

Of course, the immediate open question is whether the time and/or space complexity of the described algorithms can be improved. Note that speeding up the algorithm to compute the optimal solution over the connection patterns that are not vertically partitioned will immediately speed up the entire algorithm.

Finally, as is the case with the near-misses, additional constraints in the connection pattern yield interesting variants of the problem considered. A variant that seems more complicated requires that the intersection of the *interiors* of the line segments that connect nodes to the grid boundary be empty, thus allowing line segments to abut to each other.

Acknowledgements. I would like to express my thanks to Bhaskar Dasgupta of the Computer Science Dept, University of Minnesota for suggesting the problem and for helpful discussions.

8. References.

1. Y. Birk and J.B. Lotspiech, "A Fast Algorithm for Connecting Grid Points to the Boundary with Nonintersecting Straight Lines," *Proc. 2nd Annual Symposium on Discrete Algorithms* (1991), 465–474.
2. J. Bruck and V.P. Roychowdhury, "How to Play Bowling in Parallel on the Grid," *Journal of Algorithms* 12 (1991), 516–529.
3. K. Hwang and F.A. Briggs, *"Computer Architecture and Parallel Processing"* (1985), McGraw Hill, New York.
4. S.Y. Kung, S.N. Jean, and C.W. Chang, "Fault-Tolerant Array Processors using Single-Track Switches," *IEEE Transactions on Computers* 38 (1989), 501–514.
5. E.M. McCreight, "Priority Search Trees," *SIAM Journal on Computing* 14 (1985), 257–276.
6. K. Mehlhorn, *"Data Structures and Algorithms"* Vol. 3 (1984), Springer-Verlag, Berlin.
7. R. Raghavan, J. Cohoon, and S. Sahni, "Manhattan and Rectilinear Wiring," *Tech. Report 81-5*, Computer Science Dept., University of Minnesota, Minneapolis (1981).
8. V.P. Roychowdhury and J. Bruck, "On Finding Non-intersecting Paths in Grids and its Application in Reconfiguring VLSI/WSI Arrays," *Proc. 1st Annual Symposium on Discrete Algorithms* (1990).
9. A.D. Singh, "Interstitial Redundancy: An Area Efficient Fault Tolerance Scheme for Large Area VLSI Processor Arrays," *IEEE Transactions on Computers* 37 (1988), 1398–1410.