JOURNAL OF ALGORITHMS **22,** 57–92 (1997)
ARTICLE NO. AL950798

Connecting the Maximum Number of Nodes in the Grid to the Boundary with Nonintersecting Line Segments*

Leonidas Palios

The Geometry Center, University of Minnesota, Minneapolis, Minnesota 55454

Received February 4, 1994

Given a finite set S of nodes in a rectangular grid, we consider the problem of finding the maximum size subset of S such that the nodes in the subset can be connected to the boundary of the grid by means of nonintersecting 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 (S. Y. King *et al.*, Fault-tolerant array processors using single-track switches, *IEEE Trans. Comput.* **38** (1989), 501–514). The problem has been investigated by Bruck and Roychowdhury, who described an algorithm to find the maximum number of compatible connections of n given nodes in the grid in $O(n^3)$ time and $O(n^2)$ space (J. Bruck and V. P. Roychowdhury, How to play bowling in parallel on the grid, *J. Algorithms* **12** (1991), 516–529). In this paper, we improve their result by describing an $O(n^2 \log n)$ time and $O(n^2)$ space algorithm; instrumental in this improvement is the introduction of a new type of priority search trees which is of interest in its own right. Finally, we extend the algorithm to handle the additional constraint that *near-misses* are disallowed; this is the first algorithm to resolve this case, and, like the general algorithm, it runs in $O(n^2 \log n)$ time and requires $O(n^2)$ space. © 1997 Academic Press

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 [4], and [5], [10]). Unfortunately, due to faults during the manufacturing process, it is often the case that array processors

*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.

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 *et al*. [5]: the array processor consists of a two-dimensional array of processing elements with double-row-column spare ones placed around them; 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, while no two compensation paths intersect. Then, 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 finite set of nodes located at vertices of a rectangular 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.

(We use the expression ‘‘the boundary of the grid’’ to refer to the boundary of a rectangle that encloses all the given nodes; since the set of nodes is finite such a rectangle always exists.) 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 Fig. 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 Fig. 2, where not both nodes p and q can be connected to the boundary by means of nonintersecting vertical or horizontal line segments. The problem was first addressed by Kung *et al*. [5], who formulated it as a maximum independent set problem and adapted an algorithm by Bron and Kerbosch [2] to solve it. Later, Roychowdhury and Bruck gave an $O(n^2)$ time algorithm, where n is the number of the given nodes in the grid [9]. A year later, Birk and Lotspiech described an $O(n \log n)$ time algorithm [1], which can be proven optimal by means of a reduction from the element uniqueness problem.

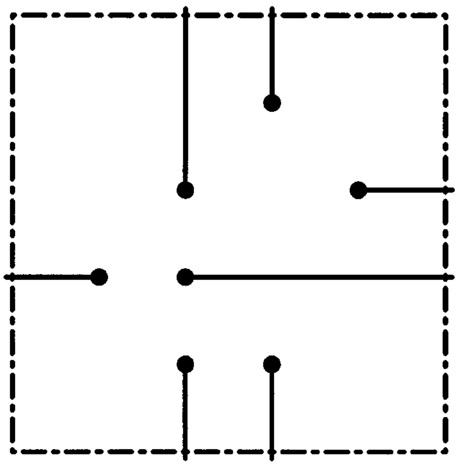

FIGURE 1

Problem P1 leads to a *maximization* version, namely:

P2: Given a finite set S of nodes in a rectangular grid, compute the size of a *maximal* subset of S such that each node in the subset can be connected to the boundary of the grid by means of a line segment parallel to one of the grid axes and no two such line segments intersect.

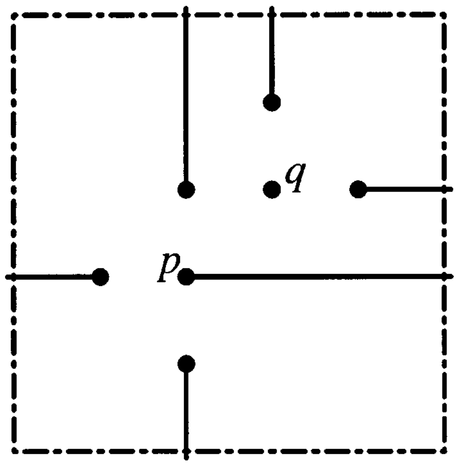

FIGURE 2

An algorithm for this problem was presented by Bruck and Roychowdhury in [3]; it relies on appropriate partitions of the grid into a constant number of rectilinear polygons, whose corresponding optimal connection patterns are independent of each other and are easier to compute. 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 [3]; the introduction and use of the following two key elements, however, allows us to achieve the improved performance:

(i) a new way to compute some of the needed optimal partial solutions, which relies on appropriate recursive expressions; the idea is so powerful that in some cases it leads to methods faster by an order of magnitude.

(ii) a new type of priority search trees 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, we note that the related problem of computing the maximum size of a *rectilinear wiring* pattern in a bounded portion of the grid is NP-complete, even in the case where the number of bends is restricted to one (*Manhattan wiring*) [8].

We also consider the variant of problem P2, where we forbid *near-misses* (see [5]), 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 [5] 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. Interestingly, the changes do not affect the asymptotic performance of the general algorithm, i.e., the modified algorithm 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. Section 3 summarizes the fundamental observations on which the algorithm of Bruck and Roychowdhury as well as our improved version rely. Our algorithm is described and analyzed in Section 4. Section 5 deals with the case where near-misses are disallowed, discussing how the algorithm for the general case needs to be modified. Finally, Section 6 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

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

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 $O(\min\{n + \Delta_r + \Delta_c, n \log n\})$ 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*. However, for a given node in the grid, it is likely that not all four of the directions may lead to feasible connections; think of other nodes located in the same row or column. Let R be a simple rectilinear polygon in the grid, which has been associated with some (or maybe all) of the above four directions. Then, by a *connection pattern* in R, we refer to a set of nonintersecting 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 $(1 \leq c \leq max\_col)$ such that no connection (line segment) intersects the *interior* of the vertical strip bounded by the columns c and $c + 1$. See also [3]. 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 Fig. 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 connec-

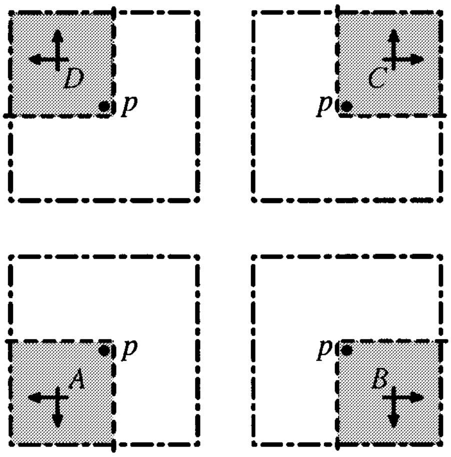

FIGURE 3

tions of p and q; note that neither the connections of p and r nor those of q and s form a near-miss.

Notation. Both the algorithm of Bruck and Roychowdhury and our algorithm rely on combining optimal solutions in portions of the grid to compute the optimal solution in the entire grid. In most cases, these grid portions are rectangles; for convenience, we use the expression $R(r_1, r_2; c_1, c_2)$ (where $r_1 \leq r_2$ and $c_1 \leq c_2$) to denote the rectangular

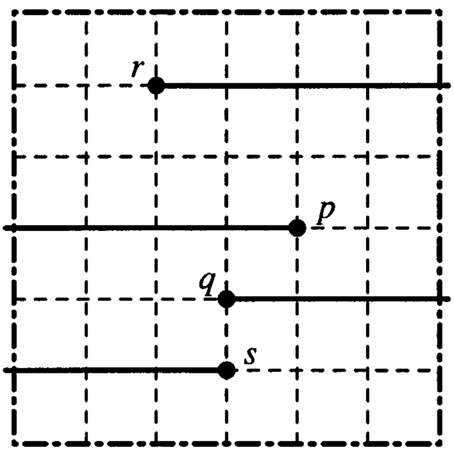

FIGURE 4

portion of the grid enclosed by and including rows r_1 and r_2, and columns c_1 and c_2. Such a rectangle may be defined in terms of a node, say, t, in the grid; then, we use $t.row$ and $t.column$ to denote t's row and column respectively. For example, the quadrant A of a node t can be denoted as $R(1, t.row; 1, t.column)$ (see Fig. 3); in figures, the convention is that row indices increase from bottom to top, and column indices increase from left to right.

Although, the problem statement allows for connections along all four possible directions (*up*, *down*, *left*, and *right*), we may seek the optimal solution in (a portion of) the grid where connections along only some of these directions are allowed. When referring to such a case in text, we qualify the solution with the associated allowed directions; for example, an "optimal *left−down* solution" involves line segments connecting nodes to the left and bottom boundary sides of the grid only. In figures, the 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, for a node t that is located at the vertex (r, c) of the grid, we define the following quantities, which we will need later in the description of the algorithm:

$$\mathit{left}(t)[k] = \begin{cases} 0, & \text{if } k = 0 \text{ or } k = r; \\ \text{optimal left solution in } R(k, r-1; 1, c-1), & \\ & \text{if } 1 \le k < r; \\ \text{optimal left solution in } R(r+1, k; 1, c-1), & \\ & \text{if } r < k \le \mathit{max\_row}. \end{cases}$$

$$\mathit{right}(t)[k] = \begin{cases} 0, & \text{if } k = 0 \text{ or } k = r; \\ \text{optimal right solution in } R(k, r-1; c+1, \mathit{max\_col}), & \\ & \text{if } 1 \le k < r; \\ \text{optimal right solution in } R(r+1, k; c+1, \mathit{max\_col}), & \\ & \text{if } r < k \le \mathit{max\_row}. \end{cases}$$

$$\mathit{down}(t)[k] = \begin{cases} 0, & \text{if } k = 0 \text{ or } k = c; \\ \text{optimal down solution in } R(1, r; k, c-1), & \\ & \text{if } 1 \le k < c; \\ \text{optimal down solution in } R(1, r; c+1, k), & \\ & \text{if } c < k \le \mathit{max\_col}. \end{cases}$$

(Note that the first parameter in the above expressions is a node in the grid, whereas the second one is an integer, a row or column index.) In Figs. 5a and 5b, we give a graphical representation of the values $\mathit{left}(t)[k]$ and $\mathit{right}(t)[k]$ for $k > r$, while in Fig. 5c, a graphical representation of

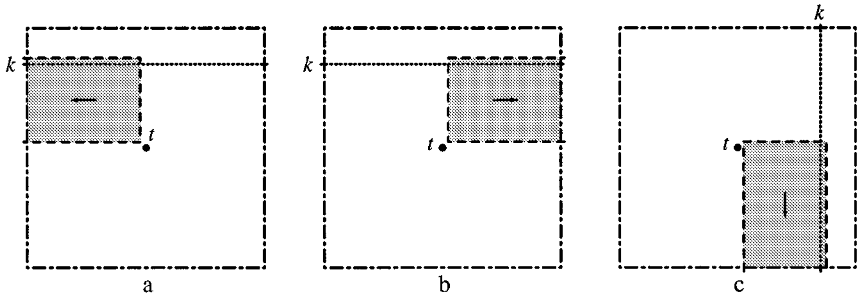

FIGURE 5

down$(t)[k]$ for $k > c$. It is important to observe that for $1 \le k_1 \le k_2 < r$ or $r < k_1 \le k_2 \le max\_row$

$$\text{optimal left solution in } R(k_1, k_2; 1, c-1) = \mathit{left}(t)[k_2] - \mathit{left}(t)[k_1 - 1].$$

Similar equalities hold for *right*$(t)[\ ]$ and *down*$(t)[\ ]$ as well.

3. FOUNDATIONS OF THE ALGORITHM

In this section, we summarize and formalize the fundamental observations of Bruck and Roychowdhury ([3]), on which their algorithm as well as the improved version we present in this paper rely. Readers familiar with [3] are encouraged to at least review the formulas.

Bruck and Roychowdhury observed that the optimal connection pattern for the entire grid either has a vertical partitioning or is of one of the two symmetric configurations depicted in Fig. 6 (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 nonintersecting line segments is

$$\begin{aligned}\max\{&\text{optimal solution over all vertically partitioned connection patterns},\\ &\text{optimal solution over all connection patterns as shown in Fig. 6a},\\ &\text{optimal solution over all connection patterns as shown in Fig. 6b}\}.\end{aligned} \tag{2}$$

We consider these three terms in order in the following paragraphs.

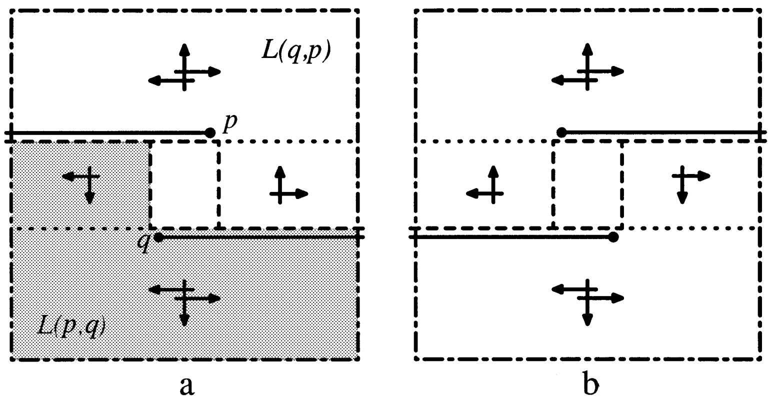

FIGURE 6

A. Vertically Partitioned Connection Patterns

According to its definition, a vertically partitioned connection pattern is characterized by a column $c\,(1 \leq c < max\_col)$ 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 (Fig. 7). Since the solutions in the two rectangles are clearly

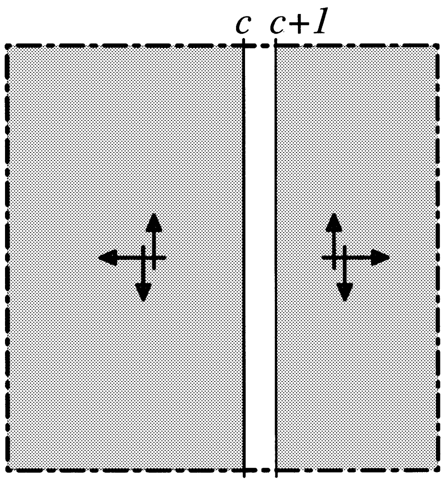

FIGURE 7

independent of each other, the optimal solution corresponding to a vertically partitioned connection pattern in the grid is

$$\max_{1 \le c < max\_col} \{\text{optimal } \textit{up-down-left} \text{ solution in } R(1, max\_row; 1, c) + \text{optimal } \textit{up-down-right} \text{ solution in } R(1, max\_row; c+1, max\_col)\}. \quad (3)$$

B. Connection Patterns as Shown in Fig. 6a

It is clear 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 the connection patterns that correspond to the configuration of Fig. 6a as

$$\max_{p,q} \{\text{optimal solution in } L(p, q) + \text{optimal solution in } L(q, p)\}, \quad (4)$$

where the optimal solutions in $L(p, q)$ and $L(q, p)$ satisfy the connection restrictions indicated in the figure, and the maximum is computed over all pairs of nodes p and q such that (i) p and q can be connected left and right respectively, and (ii) $p.row > q.row$ and $p.column \ge q.column$. For such a pair (p, q), let us concentrate on how to express the optimal solution in $L(p, q)$ (an expression for the optimal solution in $L(q, p)$ can be found similarly). Only the two configurations shown in Fig. 8 are possible, depending on whether the optimal connection pattern contains a

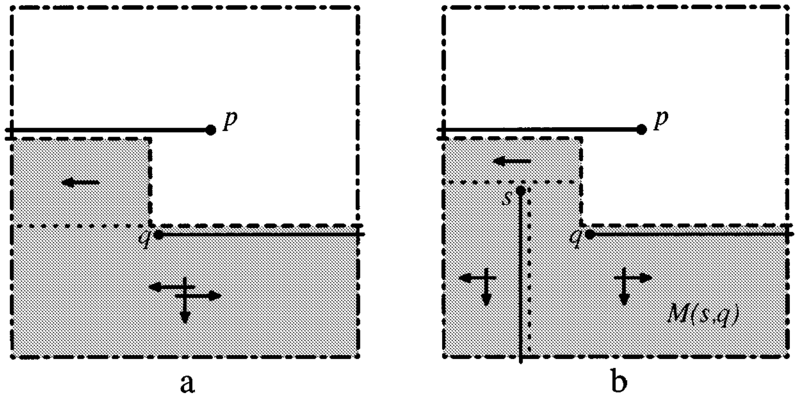

FIGURE 8

node in the rectangle $R(q.row+1, p.row-1; 1, q.column-1)$ which is connected down; the maximum of the corresponding optimal solutions yields the optimal solution that we seek. The optimal solution in $L(p,q)$ that corresponds to the configuration of Fig. 8a is precisely

$$\begin{aligned}&\text{optimal } \textit{down–left–right} \text{ solution in } R(1, q.row; 1, max\_col)\\&\quad + \textit{left}(q)[p.row-1]. \qquad (5)\end{aligned}$$

(See also Fig. 5a.) In the configuration of Fig. 8b, the node s is meant to be the highest node connected down in the optimal connection pattern in $L(p,q)$; the corresponding optimal solution is

$$\begin{aligned}&s.a + (\textit{left}(q)[p.row-1] - \textit{left}(q)[s.row])\\&\quad + \text{optimal solution in } M(s,q), \qquad (6)\end{aligned}$$

where $s.a$ denotes the optimal *down–left* solution in the quadrant A of s (see Fig. 3), and $M(s,q)$ denotes the L-shaped region at the bottom right corner of the grid bounded by the connections of s and q. Therefore, the optimal solution in $L(p,q)$ over all connection patterns compatible with the configuration of Fig. 8b is

$$\begin{aligned}&\max_s\{s.a + (\textit{left}(q)[p.row-1] - \textit{left}(q)[s.row])\\&\qquad + \text{optimal solution in } M(s,q)\}, \qquad (7)\end{aligned}$$

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

Finally, in order to compute the optimal solution in $M(s,q)$ under the connection restrictions indicated in Fig. 8b, we consider all possible instances of the configuration shown in Fig. 9, where the node t is the leftmost node in $M(s,q)$ that is connected right; for each such case, the corresponding optimal solution in $M(s,q)$ is given by

$$\textit{down}(s)[t.column-1] + \textit{right}(t)[q.row] + t.b,$$

where $t.b$ is the optimal *down–right* solution in the quadrant B of t (see also Figs. 3 and 5). Therefore, the optimal solution in $M(s,q)$ is

$$\max_t\{\textit{down}(s)[t.column-1] + \textit{right}(t)[q.row] + t.b\}, \qquad (8)$$

where t may be q or any node in the rectangle $R(1, q.row-1; s.column+1, q.column-1)$ that can be connected right.

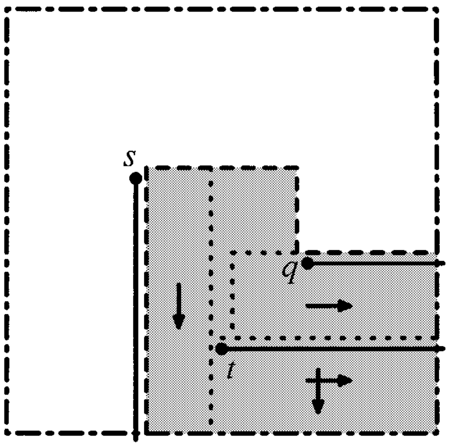

FIGURE 9

C. Connection Patterns as Shown in Fig. 6b

This case is almost identical to case B. Since the corresponding configurations are left-to-right mirror images, the entire discussion of case B applies here, except that the directions *left* and *right* need to be interchanged.

4. THE ALGORITHM

We now present the techniques and data structures that enable us to improve the running time of the algorithm, and we analyze their complexity in terms of the size n of the input set of nodes in the grid. Very briefly, the algorithm begins with a preprocessing phase (Section 4.1), computes the optimal solution over all vertically partitioned connection patterns (Section 4.2), the optimal solution over all connection patterns as shown in Fig. 6a (Section 4.3), and in a similar fashion the optimal solution over all patterns as shown in Fig. 6b. Then, the maximum among these three values yields the desired quantity in accordance with expression (2).

4.1. *The Preprocessing Phase*

In this phase, we precompute and store in tables (for constant time access) some quantities that will be useful. First, as expressions (7) and (8) indicate, we need the optimal solutions in the quadrants of all the nodes. Bruck and Roychowdhury also precompute these values; their approach involves computing each of them 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 4.1.1); 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 $O(n^2)$.

Next, for each node t, we precompute the values of *left*(t)[ ], *right*(t)[ ], *down*(t)[ ], as well as those needed for the computation of the optimal solution in $L(q, p)$ and connection patterns as shown in Fig. 6b. For a specific node, each of these arrays can be computed in $O(n)$ time and requires $O(n)$ space to store. Therefore, precomputation of the values of all these arrays for all the nodes takes $O(n^2)$ time and $O(n^2)$ space.

Finally, we also compute arrays *leftmost*[ ], *rightmost*[ ], *lowest*[ ], and *highest*[ ]: *leftmost*$[k]$ (*rightmost*$[k]$ resp.) is equal to the column of the leftmost (rightmost resp.) node located in row k; if no node exists in row k, it is set equal to $n + 1$ (0 resp.). The entries for the other two arrays are defined similarly over the columns of the grid. The entries of all four arrays can be computed in $O(n)$ time and space.

4.1.1. 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 *all* $O(n^2)$ *grid vertices* together takes less time than computing the corresponding solution for each node independently. Let $a(i, j)$ ($0 \leq i \leq max\_row$, $0 \leq j \leq max\_col$) 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 nonintersecting line segments; clearly, the optimal *left–down* solution in the quadrant A of a node located at the intersection of row r and column c is precisely the value $a(r, c)$ (see Fig. 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,

$$a(i, 0) = 0, \qquad \text{for all } i : 0 \leq i \leq max\_row;$$

$$a(0, j) = 0, \qquad \text{for all } j : 0 \leq j \leq max\_col.$$

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

1. No node in the ith 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

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

where we remind that *leftmost*$[k]$ is equal to the index of the column on which the leftmost node in row k is located.

2. No node in the jth 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 jth column of $R(0, i; 0, j)$, if any such nodes exist. In other words, the optimal solution is

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

where *lowest*$[k]$ is equal to the index of 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 ith row *and* a connection left from a node in the jth column of $R(0, i; 0, j)$; such a connection pattern, however, contains 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,

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

Given that the values of *leftmost*[ ] and *lowest*[ ] are available after the preprocessing phase, the computation of $a(i, j)$ takes constant time if the values of $a(i-1, j)$ and $a(i, j-1)$ are already known. 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, $max\_col + 1 \leq n + 1$ entries suffice, as illustrated by the use of the array `t[ ]` in the piece of C code included in Fig. 10 that

```
/*  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];    /* t[j] = opt.  solution in quadrant A */
            if (++k >= n)       /* increment index of node_ptr[ ]; next node */
                break;
        }
    }
```

FIG. 10. Computing the optimal *left–down* solution in the quadrant A of all the nodes.

does the entire computation. The contents of t[ ] immediately before $a(r, c)$ is computed are:

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

4.2. *Vertically Partitioned Connection Patterns*

In light of the normalization conditions (1), the expression (3) implies that the optimal solution over all vertically partitioned connection patterns can be found in $O(n)$ time, provided that we have precomputed (and tabulated in $O(n)$ space) the optimal *up–down–left* solutions in the rectangles $R(1, max\_row; 1, c)$ and the optimal *up–down–right* solutions in $R(1, max\_row; c+1, max\_col)$ for all c such that $1 \le c < max\_col$. We restrict our attention to computing the former set of solutions; the latter set can be computed in an almost identical fashion, except that the grid columns need to be processed from right to left. As in Section 4.1.1, the idea of using recursive definitions leads to a very efficient method. Indeed, in Section 4.2.1, we show how (after the preprocessing phase) the optimal *up–down–left* solutions in $R(1, max\_row; 1, c)$ for all c can be computed in $O(n)$ time by sweeping the grid from left to right.

The above discussion and the results in Sections 4.1.1 and 4.2.1 yield:

LEMMA 4.1. *Given a set S of n nodes in the grid, $O(n^2)$ time and $O(n)$ space suffice to compute the size of a maximal subset of S such that each node* *in the subset can be connected to the grid boundary by means of a line segment parallel to one of the grid axes and these line segments do not intersect and form a vertically partitioned connection pattern.*

4.2.1. Computing the Optimal Up–Down–Left Solutions in All the Rectangles $R(1, max\_row; 1, k)$ Where $1 \leq k < max\_col$.

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,

$$\begin{aligned}&\text{optimal } \textit{up–down–left} \text{ solution in } R(1, max\_row; 1, 1)\\ &\quad = (\text{number of nodes in column } 1).\end{aligned}$$

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

1. None of the nodes in the kth column are connected left: This implies that the nodes in the kth 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

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

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 kth 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 kth column is connected left: Figure 11 shows the configuration of the connection pattern if a node p in the kth 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

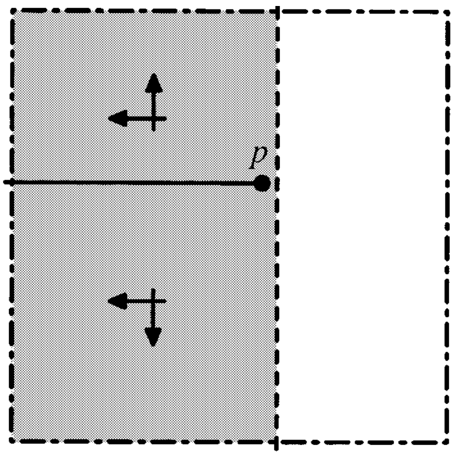

FIGURE 11

has been counted in both $p.a$ and $p.d$ (see Fig. 3). As a result, the optimal solution in this case is

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

over all nodes p in the kth 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 kth column. Since the optimal solutions in the quadrants of each node have been precomputed, the entire computation therefore takes $O(n)$ time (we take into account normalization conditions (1)), if we sweep the grid from left to right computing the optimal *up–down–left* solutions in $R(1, max\_r; 1, k)$ by increasing k.

4.3. *Connection Patterns as Shown in Fig.* 6*a*

The expression (4) implies that the optimal solution over all connection patterns as shown in Fig. 6a can be computed in $O(n^2)$ time provided that we have found the optimal solutions in $L(p, q)$ and $L(q, p)$. Again, we concentrate on the computation of the optimal solutions in $L(p, q)$ for all appropriate pairs (p, q); the optimal solutions in $L(q, p)$ can be found similarly. We show next how to carry out this computation in $O(n^2 \log n)$ instead of $O(n^3)$ as described in [3].

The basic idea is to compute together all the optimal solutions in the $L(p,q)$s for each specific node q and all ps; to do that, we augment the partial solutions that contribute to the above optimal solutions in such a way that the optimal solution in $L(p,q)$ is equal to the sum of the corresponding (augmented) partial solutions minus a corrective term (which depends only on p and q). Crucial in performing this task efficiently is the use of a new type of priority search trees, which we introduce in Section 4.3.1; 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 tree is taking, $O(n)$ space suffices.)

4.3.1. 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 row and column indices). 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 the right choice for data structure to represent the set S may be a priority search tree, that is, a balanced binary search tree with an additional heap ordering. Unfortunately, the standard priority search tree described in [6] (see also [7]) 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 $\Omega(\log^2 m)$ total cost.

We therefore use a different type of priority search trees. The basic structure is a balanced binary search tree whose leaves from left to right correspond to the elements of S in order. The novelty lies in the way the heap ordering of the priorities is stored and maintained in the tree: each tree node is associated with two fields, namely *max* and *debt*, 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 i elements of S reduces to finding the (at most $2\log i$) maximal subtrees whose sets of leaves partition the set of the i leftmost leaves of our priority search tree, and adding k to the *debt* field of their roots. Initially, all the *debt* fields equal 0; the *max* field of each tree leaf is initialized to the priority of the associated element of S, while the *max* field of an internal tree node t is initialized according to the heap ordering condition

$$t.max = \max\{c_l.max - c_l.debt,\ c_r.max - c_r.debt\}, \tag{9}$$

where c_l and c_r are the two children of t. A subtle point is that now the current priority of an element of S associated with a leaf t of the tree equals $t.max$ minus the sum of the *debt* fields of all the tree nodes on the path from the root to t (inclusive). Clearly, building and initializing the tree takes time linear in the size of S.

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.

• *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 (9), 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 (9) 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 13a shows the changes in the *max* and *debt* fields after a *SubUpTo*$(1,(6,7))$ operation at the tree of Fig. 12; note that $(6,7)$ should be placed between c

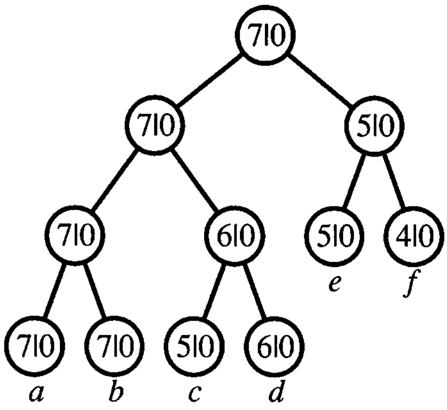

FIGURE 12

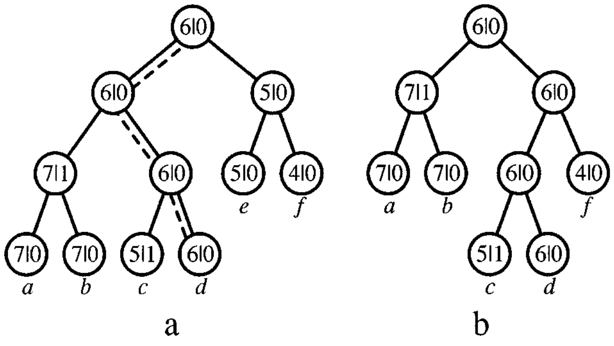

FIGURE 13

and d in the ordered set S. Figure 13b shows the effect of a subsequent *Remove*(e) operation: a double rotation at the tree root.

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 4.2. *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( )* operations 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 [6]), and the equivalents of *MinXinRectangle*( ) and *MaxXinRectangle*( ) described in [6]. The equivalent of *EnumerateRectangle*( ), however, may take as much as $\Theta((k+1)\log m)$ time, if k out of the m elements of S are reported.

4.3.2. Computing the Optimal Solution in L(p, q) for All the Appropriate Pairs (p, q)

Let us now see how we can compute the optimal solutions in $L(p,q)$ for all the appropriate pairs (p,q) in $O(n^2 \log n)$ total time. Since the expression (7) relies on the optimal solution in $M(s,q)$ (see Fig. 8b), we first compute the optimal solution in $M(u,v)$ for all the appropriate pairs (u,v); thanks to the priority search tree that we described in Section 4.3.1, we can do that in $O(n^2 \log n)$ total time.

The idea is that, for each node u that can be connected down, we compute the optimal solutions in $M(u, v)$ for all appropriate vs together; these are the nodes in the quadrant B of u (except u) that can be connected right (Fig. 8b). In particular,

0. We build a priority search tree T_u (of the type described in Section 4.3.1) on the set S_u of nodes in the quadrant B of u (except u) that can be connected right; note that the latter requirement implies that S_u contains at most one node per row. 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. The priority of such an element (node) t equals the optimal solution in the rectangle $R(1, u.row;\ u.column + 1, max\_col)$ under the connection restrictions shown in Fig. 14 (t is assumed to be the leftmost node connected right), which is

$$init(t) = down(u)[t.column - 1] + right(t)[u.row] + t.b; \qquad (10)$$

the three terms correspond to the optimal solutions in the rectangles R_1, R_2, and R_3, respectively. Figure 12 depicts the tree T_u that corresponds to the node u in Fig. 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 comply with Fig. 14 and justify the priorities 7 and 5 assigned to the nodes b and c, respectively.

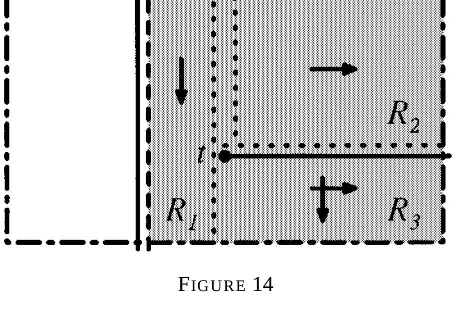

FIGURE 14

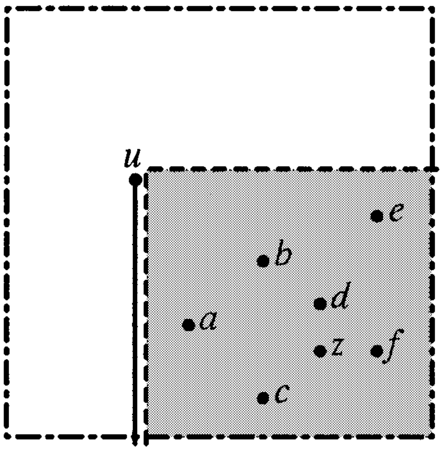

FIGURE 15

Then, the optimal solutions in $M(u, v)$ for all the nodes v that define such a configuration with u (as suggested by Fig. 8b, these are the nodes in the quadrant B of u (except u) that can be connected right, that is, the elements of S_u) are computed as described below:

1. We process the nodes x that define a region $M(u, x)$ with u by decreasing row. Since these nodes are precisely the nodes in S_u, we copy the elements of S_u in a list Q, and we radix-sort them by decreasing row in linear time.

2. Then, each node, say v, in Q is processed in order as follows:

a. We perform a $MaxUpTo(v)$ operation in T_u, which, according to the ordering of S_u, returns the maximum over the priorities of v and all the nodes associated with tree leaves preceding v; the value returned is the optimal solution in $M(u, v)$.

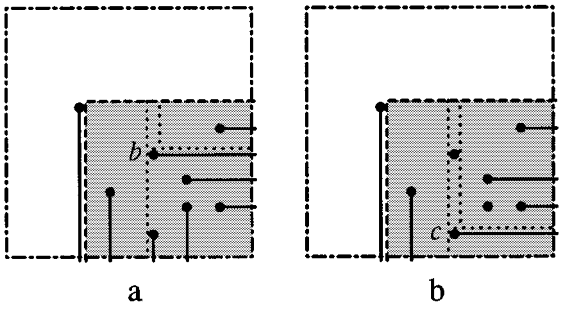

FIGURE 16

b. Next, we perform a $SubUpTo(1, v)$ operation in T_u to reduce the priorities of all the nodes associated with tree leaves preceding v.

c. Finally, we remove the node v from the set S_u (actually, the leaf associated with v in T_u) by performing a $Remove(v)$ operation.

The correctness of the procedure follows from the following argument: Let us assume that we are processing node q from Q. Then, because of the ordering of the leaves in T_u and the removal operation in Step 2c, the nodes associated with the leaves preceding q are precisely the nodes in the rectangle $R(1, q.row - 1; u.column + 1, q.column - 1)$. Executing Step 2a yields

$$\max_t \{\text{current priority of } t\} \tag{11}$$

as the optimal solution in $M(u, q)$, where, as indicated, the maximum is computed over all nodes t in $R(1, q.row - 1; u.column + 1, q.column - 1)$ and q as well. After having been assigned to its initial value $init(w)$ according to expression (10), the priority of a node w decreased by 1 every time step 2b was executed for a node v in Q such that w lies in $R(1, v.row - 1; u.column + 1, v.column - 1)$, or equivalently for every node in Q that lies in the rectangle $R(w.row + 1, u.row; w.column + 1, max\_col)$. Therefore, taking into account the definition of $right(\ )[\ ]$ in Section 2, and the fact that the nodes in Q came from S_u, which contains only nodes that can be connected right, we find that the current priority of a node w when q is being processed is

$$init(w) - (right(w)[u.row] - right(w)[q.row]). \tag{12}$$

Combining expressions (10), (11), and (12), and substituting s for u yields precisely expression (8) for the optimal solution in $M(s, q)$.

From a complexity point of view, the above procedure takes $O(n \log n)$ time to find the optimal solutions in $M(u, v)$ for a fixed u and all the appropriate vs; the tree T_u requires $O(n)$ time to build, while each of the $MaxUpTo(\ )$, $SubUpTo(\ )$, and $Remove(\ )$ operations takes $O(\log n)$ time to execute (Lemma 4.2), since the number of elements of S_u is $O(n)$ initially and always decreases. Repeating the procedure for all nodes u that can be connected down will yield the optimal solutions in all the $M(u, v)$ that we need in $O(n^2 \log n)$ total time. All these values are then stored in an array (of $O(n^2)$ size) for constant time access.

We are now ready to compute the optimal solutions in $L(p, q)$, for all the appropriate pairs (p, q). Again, for each node q that can be connected right, we compute the optimal solutions in $L(p, q)$ for all appropriate ps together. We perform the following steps:

0. 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 rectangle $R(q.row + 1, max\_row; 1, q.column - 1)$ that can be connected down, plus a dummy node located at $(0, 0)$. 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; note that the dummy node will be the first node in the ordered list. The priority of the dummy node is equal to the optimal solution for the configuration shown in Fig. 17a, that is,

$$\begin{aligned}&\text{optimal } \mathit{down\text{-}left\text{-}right} \text{ solution in } R(1, q.row; 1, max\_col)\\&\quad + \mathit{left}(q)[max\_row]. \qquad (13)\end{aligned}$$

(Note that 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 in a fashion similar to that described in Section 4.2.1.) The priority of any other element (node) s of S_q is equal to the optimal solution in the region shown in Fig. 17b under the indicated restrictions (s is assumed to be the highest node connected down), which is

$$\begin{aligned}&s.a + (\mathit{left}(q)[max\_row] - \mathit{left}(q)[s.row])\\&\quad + \text{optimal solution in } M(s, q); \qquad (14)\end{aligned}$$

the three terms correspond to the optimal solutions in the rectangles R_1 and R_2 and in $M(s, q)$, respectively.

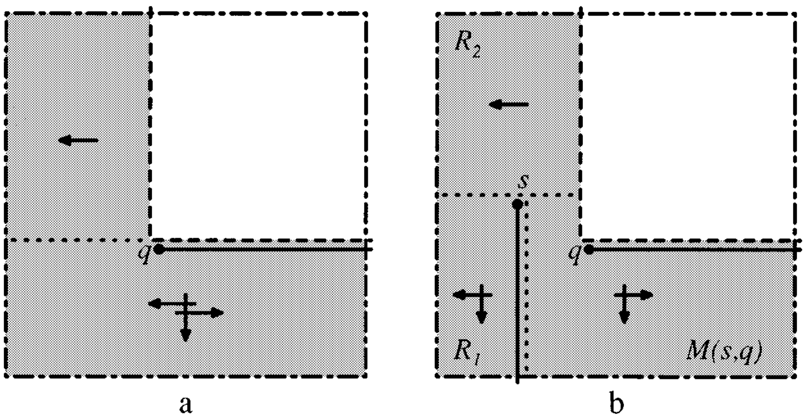

FIGURE 17

1. For each node p in the quadrant C of q (except q) that can be connected left, we perform a $\mathit{MaxUpTo}(p)$ 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 we report

$$val - (\mathit{left}(q)[\mathit{max\_row}] - \mathit{left}(q)[p.\mathit{row} - 1]) \tag{15}$$

as the optimal solution in $L(p, q)$.

The correctness of the procedure follows from the comparison of Figs. 8 and 17, and expressions (5) and (13), and (6) and (14) taking into account expression (15) as well. Since the size of S_q is $O(n)$, Lemma 4.2 implies that the tree T_q of the node q requires $O(n)$ time to construct and $O(n)$ space to store. Additionally, the $\mathit{MaxUpTo}(\ )$ operation takes time logarithmic in the size of S_q, so 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, after having computed the optimal solutions in $M(u, v)$, additional $O(n^2 \log n)$ time suffices for the computation of the optimal solutions in $L(p, q)$ for all the appropriate pairs (p, q).

Summarizing, the optimal solutions in $L(p, q)$ (and similarly in $L(q, p)$) for all appropriate p and q can be found in $O(n^2 \log n)$ time and can be tabulated in $O(n^2)$ space. Then, the maximum in expression (4) can be computed in $O(n^2)$ time, and therefore

LEMMA 4.3. *Given a set S of n nodes in the grid, $O(n^2 \log n)$ time and $O(n^2)$ space suffice to compute the size of a maximal subset of S such that each node in the subset can be connected to the grid boundary by means of a line segment parallel to one of the grid axes and these line segments do not intersect and they form a connection pattern as shown in Fig. 6a.*

4.4. *Putting the Pieces Together*

In Sections 4.2 and 4.3, respectively, we showed how to efficiently compute the first and second terms in expression (2); the third term can be computed in a fashion similar to the second one and in the same time and space complexity. Then, Lemmas 4.1 and 4.3 imply that

THEOREM 4.1. *Given a set S of n nodes in the grid, $O(n^2 \log n)$ time and $O(n^2)$ space suffice to compute the size of a maximal subset of S such that each node in the subset can be connected to the boundary of the grid by means of a line segments parallel to one of the grid axes and no two such line segments intersect.*

5. DISALLOWING NEAR-MISSES

Unlike the general problem, we can no longer blindly eliminate all rows and columns that do not contain any nodes from the given set. The reason is that in this way nodes may be brought in adjacent rows or columns, which, since near-misses are disallowed, will impose additional restrictions that did not exist in the original problem configuration. It is crucial to observe, however, that the exact number of empty rows or columns between two nodes does not really matter; in other words, we can merge any positive number of successive empty rows (columns resp.) into a single empty row (column resp.) without altering the final optimal solution. 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

$$max\_row \leq 2n - 1 \qquad \text{and} \qquad max\_col \leq 2n - 1. \tag{16}$$

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. Interestingly, however, the time and space complexity of the resulting algorithm matches that of the general algorithm in the asymptotic sense, and we are able to prove the counterparts of Lemmas 4.1 and 4.3, which imply that

THEOREM 5.1. *Given a set S of n nodes in the grid, $O(n^2 \log n)$ time and $O(n^2)$ space suffice to compute the size of a maximal subset of S such that each node in the subset can be connected to the boundary of the grid by means of a line segments parallel to one of the grid axes and no two such line segments intersect or exhibit near-misses.*

5.1. *The Preprocessing Phase*

First, it is easy to see that, since the connections in each of the quadrants of a node do not involve opposite directions (see Fig. 3), no near-miss can possibly occur in the corresponding optimal solutions. Therefore, the method of Section 4.1.1 can be applied without modifications. Moreover, the same applies in the computation of the values of *left*(t)[ ], *right*(t)[ ], and *down*(t)[ ] as well as the corresponding quantities needed for the optimal solutions in $L(q, p)$. Finally, the values of the arrays *leftmost*[ ], *rightmost*[ ], *lowest*[ ], and *highest*[ ] depend solely on the location of the given nodes, and not the allowed directions.

In summary, even if near-misses are disallowed, the methods of Section 4.1 do not change. Note however that the interesting portion of the grid may have up to twice as many rows/columns as nodes; this implies for instance that the arrays we use are of size $2n$ and that the value of *leftmost*$[k]$ (*lowest*$[k]$ resp.) is set equal to $2n$ if the k-th row (column resp.) does not contain any nodes.

5.2. *Vertically Partitioned Connection Patterns*

If near-misses are not allowed, 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 therefore 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-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-down$[k]$:	Optimal *up–down–left* solution in $R(1, max\_row; 1, k)$, where no node in column k is connected down.

Similarly

r_complete$[k]$:	Optimal *up–down–right* solution in $R(1, max\_row; k, max\_col)$.
r_non-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-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* $1 \leq highest[c] < lowest[c+1] < 2n$: Then, connecting the highest node in column c up and the lowest node in column $c+1$ down creates a near-miss (see Fig. 18a); so, these two connections are incompatible, and therefore the optimal solution in this case is

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

(ii) *if* $1 \le highest[c+1] < lowest[c] < 2n$: Similarly to the previous case, connecting the lowest node in column c down and the highest node in columns $c+1$ up creates a near-miss (see Fig. 18b). The optimal solution is

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

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

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

(Note that, if no nodes exist in column c or $c+1$, then we end up in case (iii), as desired.) The above procedure allows us to compute the optimal solution $up\_opt(c)$ 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 ($1 \le c \le max\_col$), the optimal solution over all vertically partitioned patterns is simply

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

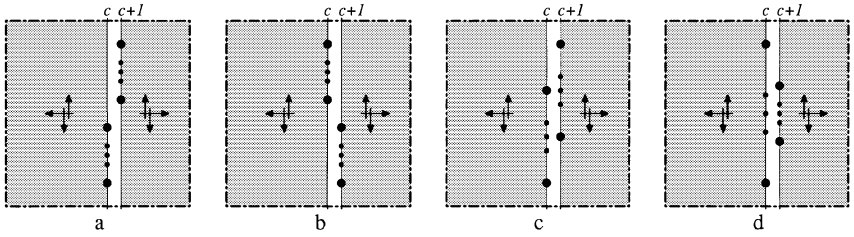

FIGURE 18

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 such that $1 \leq k < max\_col$ 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 $(1 \leq k < max\_col)$ 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 5.1. *Given a set S of n nodes in the grid, $O(n^2)$ time and $O(n)$ space suffice to compute the size of a maximal subset of S such that each node in the subset can be connected to the grid boundary by means of a line segment parallel to one of the grid axes and these line segments do not intersect or exhibit near misses and they form a vertically partitioned connection pattern.*

5.2.1. *Computing the $l\_complete[\ ]$, $l\_non\text{-}up[\ ]$, and $l\_non\text{-}down[\ ]$ 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 5.2. *For any column k of the grid $(1 \leq k < max\_col)$,*

$$l\_non\text{-}up[k] + 1 \geq l\_complete[k]$$

and

$$l\_non\text{-}down[k] + 1 \geq l\_complete[k].$$

Similarly,

$$r\_non\text{-}up[k] + 1 \geq r\_complete[k]$$

and

$$r\_non\text{-}down[k] + 1 \geq r\_complete[k].$$

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

$$l\_non\text{-}up[k] \geq l\_complete[k]$$

according to the definition of $l\_non\text{-}up[k]$; therefore, $l\_non\text{-}up[k] + 1 \geq l\_complete[k]$. 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 $l\_complete[\ ]$, $l\_non\text{-}up[\ ]$, and $l\_non\text{-}down[\ ]$ 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 $l\_non\text{-}up[\ ]$ and $l\_non\text{-}down[\ ]$ in separate paragraphs.

Computing the $l\_complete[k]$ *solution* $(1 \le k < max\_col)$. 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 $v.row > lowest[k-1]$, 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 the beginning of Section 5.2 to combine partial optimal solutions in order 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 5.2 translate into the following three case statements, respectively, where we also used Lemma 5.2 to eliminate the max{ } operators.

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

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,

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

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,

$$l\_non\text{-}up[k] = l\_complete[k-1]$$
$$l\_non\text{-}down[k] = l\_complete[k-1].$$

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

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

where $d(i, j)$ is equal to the optimal *left–up* solution in the rectangle $R(i, max\_row; 1, j)$ (compare to the definition of $a(i, j)$ in Section 4.1.1). In case no node in column k is connected left, we distinguish two cases. If $\mathit{highest}[k-1] \geq \mathit{lowest}[k]$, we can safely connect the lowest node in column k down, and the corresponding *non-up* solution is

$$\mathit{l\_complete}[k-1] + 1.$$

If $\mathit{highest}[k-1] < \mathit{lowest}[k]$, 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

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

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 5.2, the above maximum is equal to $\mathit{l\_non\text{-}up}[k-1] + 1$, and therefore, the optimal solution if no node in column k is connected left is

$$\mathit{nu\_opt}_2 = \begin{cases} \mathit{l\_complete}[k-1] + 1, & \text{if } \mathit{highest}[k-1] \geq \mathit{lowest}[k]; \\ \mathit{l\_non\text{-}up}[k-1] + 1, & \text{otherwise}. \end{cases}$$

Summarizing,

$$\mathit{l\_non\text{-}up}[k] = \mathit{max}\{\mathit{nu\_opt}_1, \mathit{nu\_opt}_2\}.$$

Similarly, for the *l_non-down* solution, we have

$$\mathit{l\_non\text{-}down}[k] = \mathit{max}\{\mathit{nd\_opt}_1, \mathit{nd\_opt}_2\},$$

where

$$\mathit{nd\_opt}_1 = \max_q \{a(q.\mathit{row} - 1, q.\mathit{column} - 1) + q.d\}$$

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

$$\mathit{nd\_opt}_2 = \begin{cases} \mathit{l\_complete}[k-1] + 1, & \text{if } \mathit{lowest}[k-1] \leq \mathit{highest}[k]; \\ \mathit{l\_non\text{-}down}[k-1] + 1, & \text{otherwise}. \end{cases}$$

It should be obvious that computing the optimal solution over all vertically partitioned patterns while at the same time disallowing near-misses 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-up*[ ], and *l_non-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 $a(r-1, c-1)$ and $d(r+1, c-1)$, and by symmetry the values of $b(r-1, c+1)$ and $c(r+1, c+1)$ (which will be needed to compute the *r_non-up*$[c]$ and *r_non-down*$[c]$ solutions). Therefore, according to Lemma 4.1, the total space required at this step is $O(n)$.

Finally, from a complexity standpoint, computing the *l_complete*$[k]$, *l_non-up*$[k]$, and *l_non-down*$[k]$ for all k $(1 \le k < max\_col)$ 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 $O(n)$ in light of the normalization conditions (16).

5.3. *Connection Patterns That Are Not Vertically Partitioned*

Unlike Section 5.2, only minor changes need to be introduced in the algorithm described in Section 4.3, 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 Fig. 6, 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 come up 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) as shown in Fig. 6, 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 Fig. 17a: Then, in the expression (13), we need only use the corresponding *complete* solution in $R(1, q.row; 1, max\_col)$ (which precludes the existence of near-misses; see Section 5.2).

Clearly, the above changes do not increase the asymptotic time or space complexity of the algorithm, and therefore, in light of Lemma 4.3 and the similarity of cases (a) and (b) of Fig. 6, we have

LEMMA 5.3. *Given a set S of n nodes in the grid, $O(n^2 \log n)$ time and $O(n^2)$ space suffice to compute the size of a maximal subset of S such that each node in the subset can be connected to the grid boundary by means of a line segment parallel to one of the grid axes and these line segments do not intersect or exhibit near-misses and they form a connection pattern that is not vertically partitioned.*

6. CONCLUDING REMARKS

In this paper, we considered the problem of connecting nodes in the grid to the grid boundary by means of nonintersecting line segments along one of the grid axes; we showed that given a set of n nodes, the size of a maximal subset containing nodes that can be connected to the boundary of the grid by means of such line segments can be computed in $O(n^2 \log n)$ time and $O(n^2)$ space. Furthermore, we can achieve the same time and space complexity under the additional constraint that no near-misses are exhibited in the connection pattern.

Instrumental in the algorithm is a new type of priority search trees; its key feature is that, given an ordered set S, it allows us to increase or decrease the priorities of the elements in an *interval* of S by the same amount in time logarithmic in the size of S. In fact, one can achieve time complexity logarithmic in the size of the interval, provided that we have a "finger" in one of the elements of the interval. This is the case in the problem we deal with in this paper, since the intervals we consider always contain the first element of the current set S; however, this observation does not improve the asymptotic time complexity of the algorithm in the worst case.

Although we concentrated on finding the maximum number of nonintersecting connections, the algorithm can be easily 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 4.1.1). 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/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, different restrictions in the connection pattern yield interesting variants of the problem considered. An interesting variant requires that the intersections 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.

ACKNOWLEDGMENTS

I express my thanks to Bhaskar Dasgupta of the Computer Science Department at the University of Minnesota for suggesting the problem and for helpful discussions.

REFERENCES

1. Y. Birk and J. B. Lotspiech, A Fast Algorithm for Connecting Grid Points to the Boundary with Nonintersecting Straight Lines,'' *in ''Proc. 2nd Annual Symposium on Discrete Algorithms*,'' 1991, pp. 465–474.
2. C. Bron and J. Kerbosch, Algorithm 457–Finding all cliques of an undirected graph, *Comm. ACM* **16** (1973), 575–577.
3. J. Bruck and V. P. Roychowdhury, How to play bowling in parallel on the grid, *J. Algorithms* **12** (1991), 516–529.
4. K. Hwang and F. A. Briggs, ''Computer Architecture and Parallel Processing,'' McGraw–Hill, New York, 1985.
5. S. Y. Kung, S. N. Jean, and C. W. Chang, Fault-tolerant array processors using single-track switches, *IEEE Trans. Comput.* **38** (1989), 501–514.
6. E. M. McCreight, Priority search trees, *SIAM J. Comput.* **14** (1985), 257–276.
7. K. Mehlhorn, ''Data Structures and Algorithms,'' Vol. 3, Springer-Verlag, Berlin, 1984.
8. R. Raghavan, J. Cohoon, and S. Sahni, ''Manhattan and Rectilinear Wiring,'' *Technical Report 81-5*, Computer Science Dept., University of Minnesota, Minneapolis, 1981.
9. V. P. Roychowdhury and J. Bruck, ''On finding non-intersecting paths in grids and its application in reconfiguring VLSI/WSI arrays,'' *in Proc. 1st Annual Symposium on Discrete Algorithms*, 1990.
10. A. D. Singh, Interstitial redundancy: An area efficient fault tolerance scheme for large area VLSI processor arrays, *IEEE Trans. Comput.* **37** (1988), 1398–1410.