RECTILINEAR PATH PROBLEMS AMONG RECTILINEAR OBSTACLES REVISITED*

CHUNG-DO YANG†, D. T. LEE‡, AND C. K. WONG§

Abstract. We present efficient algorithms for finding rectilinear collision-free paths between two given points among a set of rectilinear obstacles. Our results improve the time complexity of previous results for finding the shortest rectilinear path, the minimum-bend shortest rectilinear path, the shortest minimum-bend rectilinear path and the minimum-cost rectilinear path. For finding the shortest rectilinear path, we use graph-theoretic approach and obtain an algorithm with $O(m \log t + t \log^{3/2} t)$ running time where t is the number of extreme edges of given obstacles, and m is the number of obstacle edges. Based on this result we also obtain an $O(N \log N + (m+N) \log t + (t+N) \log^2 (t+N))$ running time algorithm for computing the L_1 minimum spanning tree of given N terminals among rectilinear obstacles. For finding the minimum-bend shortest path, the shortest minimum-bend rectilinear path and the minimum-cost rectilinear path, we devise a new dynamic-searching approach and derive algorithms that run in $O(m \log^2 m)$ time using $O(m \log m)$ space or run in $O(m \log^{3/2} m)$ time and space.

Key words. rectilinear shortest path, minimum-bend path, path preserving graph, computational geometry, rectilinear obstacles

AMS subject classifications. 68U05, 68Q25, 68P05, 68R10

1. Introduction. The problem of finding paths among obstacles has been extensively studied in the past. As the integrated circuits draw more research interest, finding rectilinear paths using different criteria has become an important variation of the traditional shortest path problem in automated circuit design. Both measures, the *number of bends* and the *length* of a rectilinear path, are two important factors while routing between two points among a set of rectilinear obstacles. Many efficient algorithms about finding rectilinear paths with respect to these two factors separately or jointly have been obtained [3, 4, 8, 10, 11, 12, 13, 14, 16, 18]. Two of the best results for finding shortest path (SP) are due to Clarkson, *et al.* [3], which runs in $O(m \log^{3/2} m)$ time, where m is the number of obstacle edges, and due to Wu *et al.* [16], which runs in $O(m \log t + t^2 \log t)$ time where t is the number of extreme edges of given obstacles. Mitchell [11] proposed a wave-front approach to find the shortest rectilinear path in $O(m \log^2 m)$ time which is later reported to run in $\theta(m \log m)$ time after some modifications [12]. An edge on the boundary of an obstacle is an *extreme* edge if its two adjacent edges lie on the same side of the line containing the edge. Results from [3, 11, 12] are also applicable to any polygonal obstacles. Here we only focus on rectilinear paths. In this paper we shall present an $O(m \log t + t \log^{3/2} t)$ time algorithm which combines features of both results to find a shortest path. Finding the minimum spanning tree (MST) among N terminals can also be solved in $O(N \log N + (m + N) \log t + (t + N) \log^2 (t + N))$ time based on the same method, which improves the $O(N \log N + (m + t^2) \log t)$ time algorithm due to Wu, *et al.* [16].

* This work was supported in part by the National Science Foundation under the Grants CCR-8901815, and CCR-9309743, and by the Office of Naval Research under the Grant N00014-93-1-0272.

† IBM Microelectronics, EDA Physical Design, E. Fishkill Facility, Hopewell Junction, New York 12533 (yangcd@fshvmfk1.vnet.ibm.com).
‡ Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60208 (dtlee@eecs.nwu.edu).
§ IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, New York 10598 (wongck@watson.ibm.com).

With regard to the number of bends, Ohtsuki [13] proposed an $O(m \log^2 m)$ time and $O(m)$ space algorithm for finding a minimum-bend path (MBP). By combining their algorithms with the data structure from Imai and Asano [6], finding a minimum-bend path can be solved in $\theta(m \log m)$ time and space.

When both measures are considered jointly one may define problems by assigning different optimization priorities to length and bends. Recently, Yang, Lee and Wong [17] presented algorithms to find a minimum-bend shortest path (MBSP), a shortest minimum-bend path (SMBP) and also a minimum-cost path (MCP) in $O(mt + m \log m)$ time, where t is the number of the extreme edges. We shall show below that this class of problems can be solved in $O(m \log^{3/2} m)$ time by applying a *hybrid* approach, called *dynamic-searching*, which is a combination of graph-theoretic approach and the continuous-search (e.g., wave-front) approach. We first generate a graph just for guidance purpose and then by traversing the graph construct the corresponding rectilinear paths on the fly, thereby obtaining our goal path.

In the following, we first define our problems and give some preliminaries in section 2. In section 3, we show the faster algorithm for SP and MST. In section 4, we focus on solving MBSP, SMBP and MCP. We introduce the dynamic-searching approach, the guidance graph and then present our algorithms. The conclusion follows in the last section.

2. Preliminaries. A *rectilinear path*, Π_{pq}, is a path connecting two points p and q, which consists of only horizontal and vertical line segments.

Given two terminals, s and d, and a set of rectilinear obstacles, we define the following problems:

Problem SP is to find a path with the shortest distance. Such a path is called the shortest path, denoted as *sp*.

Problem MBSP is to find a path with a minimum number of bends among all the shortest paths from s to d. Such a path is called the minimum-bend shortest path, denoted as *mbsp*.

Problem SMBP is to find the shortest one among all the minimum-bend paths from s to d. Such a path is called the shortest minimum-bend path, denoted as *smbp*.

Problem MCP is to find the minimum-cost path from s to d where the cost is a non-decreasing function f of the number of bends and the length of a path. Denote such a path *mcp*.

Each of the input obstacles is specified by a sequence of edges in clockwise order. An *extreme edge* is an obstacle edge with both of its adjacent edges lying on the same side of the line containing it. An *extreme point* is an end point of an extreme edge. Let m be the number of obstacle edges and t be the number of extreme edges. Let $EXTM$ be the set of extreme points and let V be the set of obstacle vertices. Note that for a convex rectilinear polygon, there are only four extreme edges and eight extreme points. Many of the following discussion will be on a point set, $\{s, d\} \bigcup EXTM$, which will be denoted as $EXTMSD$.

We now define a basic graph generated by Clarkson, *et al.*[3], which will be used in both of our algorithms for SP and for MBSP, SMBP and MCP.

DEFINITION 1. Let R_{pq} denote the closed rectangle with segment $\overline{pq}$ as its diagonal. Given a set of points, S, a point $q \in S$ is a *staircase point* of a point $p \in S$ if and only if R_{pq} does not contain any point in S other than p and q. Let $SC(p)$ denote the set of staircase points of point p.

We recall an important property employed by Clarkson, *et al.*[3], and refer to their algorithm as Algorithm C from here on. That is, a shortest path from one point p to some other point q can always be replaced by subpaths going from p to one of its staircase points and the same holds for subsequent subpaths. With this property, Algorithm C obtains a graph on the vertex set, $V \bigcup \{s, d\}$, by adding all the edges between points and their staircase points. Such a graph, which preserves shortest paths, may contain $O(m^2)$ edges and $O(m)$ vertices, which yields a basic $O(m^2)$ time path-finding algorithm. Then by introducing *Steiner points* to the graph, they reduce the graph to have $O(m \log m)$ vertices and edges, referred to as Algorithm C1. Algorithm C1 constructs the graph in a recursive manner. A cutting line that divides the vertices in two halves is first introduced. Then all the vertices are projected onto this line, creating *Steiner points*. Edges are added between vertices and their corresponding Steiner points, and between consecutive Steiner points. Recursively perform the cutting, projecting and adding edges, on the subsets of vertices that lie on both sides of the cutting line. Since each vertex can create at most $O(\log m)$ Steiner points, the number of vertices of the graph is $O(m \log m)$. The edges, which are either horizontal or vertical, are also added during the creation of Steiner points. Therefore, it is not difficult to see that there are $O(m \log m)$ edges. The time complexity to search this graph for a shortest path is thus $O(m \log^2 m)$.

To further improve the time complexity, they use a *grouping* technique to generate more edges but fewer vertices, i.e., $O(m \log^{3/2} m)$ edges and $O(m \log^{1/2} m)$ vertices, for the graph and are able to reduce the time complexity of the searching to $O(m \log^{3/2} m)$. This algorithm is referred to as Algorithm C2. The grouping technique is to virtually cut the space into strips before they generate Steiner points such that each strip contains only $O(\sqrt{\log m})$ points. In each strip, only two Steiner points are created (the highest and the lowest) in order to provide connections across strips. Inside a strip, an edge is created for each pair of vertices that lie on different sides of the cutting line. It can be proved that such a graph still preserves the shortest path. Note that this refined graph contains some *oblique* edges due to the direct connections inside each strip. This property makes this graph different from the previous one when we use it in our algorithm later. We refer the reader to [3] or [10] for details of these algorithms.

By applying Fredman and Tarjan's $O(|E| + |V| \log |V|)$ shortest path algorithm [5] on the final graph, they obtain the shortest path in $O(m \log^{3/2} m)$ time.

On the other hand, the algorithm of Wu, *et al.*[16], referred to as Algorithm W, finds the rectilinear shortest path among rectilinear obstacles by first constructing a so-called "*track graph*" based on the extreme edges. Horizontal and vertical tracks, which are projected from extreme edges are the edges and the intersections of these tracks are the vertices of the track graph. The graph plus the graph representing the obstacle vertices and edges, is of size $O(m + t^2)$, hence yielding an $O(m \log t + t^2 \log t)$ time algorithm.

Depending on the values of t and m, Algorithm W may be asymptotically more efficient than Algorithm C2 and vice versa.

Before we present our algorithm, which is better than both, let us give some preliminary results.

DEFINITION 2. A *U-shaped* subpath (or *U-subpath* for short) consists of three segments, s_1, s_2 and s_3 such that s_1 and s_3 lie on the same side of the line containing s_2; segment s_2 is referred to as the *U-segment* of the U-subpath. A *staircase* path is a path containing no U-subpath.

LEMMA 2.1. (*Figure 1*) *Any sp, mbsp, smbp or mcp, $\Pi_{s,d}$, from s to d can be*

FIG. 1. *Divide a path into staircase subpaths.*

divided into a sequence of subpaths $\pi_i, i = 1, \ldots, m$, such that all π_i's are staircase paths and the start and end points of all π_i are points in EXTMSD.

Proof. If the U-segments of all the U-subpaths of $\Pi_{s,d}$ contain extreme points, then each U-subpath can be cut into subpaths at these extreme points and each subpath will satisfy the property as claimed. Consider now a U-subpath of $\Pi_{s,d}$ denoted as s_1, s_2 and s_3, such that its U-segment does not contain any extreme point (as in Figure 1). We can always *shorten* the U-subpath by shrinking s_1 and s_3 and moving s_2 accordingly without incurring any bends. A strictly better *sp, smbp, mbsp* or *mcp* can therefore be obtained, which yields a contradiction. That is, any *sp, mbsp, smbp* or *mcp* can be represented by a sequence of staircase subpaths starting and ending at points in *EXTMSD*. □

The above lemma implies that it is sufficient to just consider staircase paths while deriving efficient algorithms.

3. A Faster Algorithm for SP.

3.1. A Smaller Path-Preserving Graph. Our algorithm is based on the following observation that it is sufficient to generate a shortest-path-preserving graph by just considering the extreme points and their projection points when using Algorithm C1 or C2.

DEFINITION 3. Given a set of obstacles, define *the projection point set $\mathcal{P}(p)$* of a point p to be $\{q | q$ is on some obstacle boundary and $\overline{qp}$ is a horizontal or vertical collision-free segment$\}$.

Let $PJ = \{q | q \in \mathcal{P}(p), p \in \mathit{EXTMSD}\}$ and $I = \mathit{EXTMSD} \bigcup PJ$ denote the set of *essential points*, which are used in the construction of the backbone of the shortest-path-preserving graph. They are not, however, the only vertices in the final graph.

DEFINITION 4. Define the graph $SPG0$ to be a graph with vertex set equal to I and edge set equal to $\{(a, b)|\ a, b \in I,\ a \in SC(b)\}$. The cost of edge (a, b) is the rectilinear distance between points a and b.

The graph $SPG0$ is the backbone of the graphs that we generate later. $SPG0$ contains in the worst case $O(|I|)$ (or $O(t)$) vertices and $O(|I|^2)$ (or $O(t^2)$) edges. We adopt the graph reduction method used in Algorithm C1 by introducing Steiner points. Let SPG_r denote the reduced graph we generate. Let SPG_{rr} denote the refined, reduced graph when the method of Algorithm C2 is used. We have the following lemma whose proof is similar to that given in [3, 10, 14].

LEMMA 3.1. *SPG_r has $O(|I|\log|I|)$ vertices and edges, and can be computed in $O(m\log|I|+|I|\log|I|)$ time. SPG_{rr} has $O(|I|\log^{1/2}|I|)$ vertices and $O(|I|\log^{3/2}|I|)$ edges, and can be computed in $O(m\log|I|+|I|\log^{3/2}|I|)$ time.*

DEFINITION 5. Define *the boundary graph BG* to be a graph with vertex set equal to $EXTM\bigcup PJ$ and edge set equal to the set of edges connecting p and q in the vertex set that lie consecutively on the boundary of an obstacle. The cost of the edge (p,q) is defined to be the shorter rectilinear path length along the boundary of the obstacle connecting points p and q.

Let SPG be the union of SPG_{rr} and BG. The following lemma is easily established.

LEMMA 3.2. *SPG contains $O(t\log^{1/2}t)$ vertices and $O(t\log^{3/2}t)$ edges.*

We now prove that SPG is a shortest-path-preserving graph. Since the extreme points are in SPG, from Lemma 2.1 it is sufficient to show that for any shortest path all its subpaths connecting points in $EXTMSD$ are embedded in SPG.

LEMMA 3.3. *If there exists a shortest staircase path RP connecting two points $p,q\in EXTMSD$, then there is a path P in SPG connecting p and q with the same length as RP.*

FIG. 2. *Situations while dragging paths.*

Proof. Denote the abscissa and ordinate of a point p as $p.x$ and $p.y$ respectively. Without loss of generality, let $p.x\le q.x$ and $p.y\le q.y$ as shown in Figure 2(a). Let the sequence of adjacent segments in RP be denoted as $h_1,v_1,h_2,v_2,\ldots,h_g,v_g$ where h_i's are horizontal segments and v_i's are vertical segments. h_1 and v_g can be empty respectively. Apparently, the path is in R_{pq}. If there is no obstacle intersecting R_{pq}, then p and q are staircase points of each other, and P is an edge in SPG. Suppose that there are obstacles in R_{pq}. We perform the following dragging operations to h_i from $i = 1$ to $i = g$:

(1) Drag h_i downward until it either is as low as p, hits some obstacle, or is as low as h_{i-1}. In the last case, merge h_i and h_{i-1} to be one horizontal edge.

(2) If $h_i = (a, b)$, $a.x < b.x$, hits some obstacle on edge (r, u) but $r \neq a$, then break h_i into two segments, (a, r) and (r, b) and perform dragging operations on (a, r) (Figure 2(b)).

(3) At each dragging, the adjacent vertical segments are adjusted accordingly.

We then perform the same dragging operations to the vertical segments on RP except that we drag them rightward and adjust horizontal segments accordingly.

Denote as P the resultant path (Figure 2(c)) after performing these operations. It is not hard to see that P has the following properties: first, it has the same length as RP; second, it is a staircase path; third, all the turning points between h_i and v_{i-1} (after re-ordering segments on P as $\{h_1, v_1, \ldots, h_{g'}, v_{g'}\}$) will be an *upper-left* corner of an obstacle. Since we focus only on paths between points in $EXTMSD$, we may assume that there is no extreme point on P. Otherwise, those subpaths formed by cutting P at those extreme points can be considered respectively.

Consider a horizontal edge $h_i = (a, b)$, $1 < i \leq m'$ (treat all the other horizontal segments similarly and all vertical ones symmetrically). As in Figure 2(d), point a is either p or a corner vertex of an obstacle as shown before. Let (a, r) be the edge of obstacle o containing a.

(1) If the boundary of o turns downward at r and $r.x \leq b.x$, then r is an extreme point on (a, b), which violates our assumption.

(2) If it turns upward at r, $r = b$ is a vertex of o.

(3) If it turns upward or downward at r where $r.x > b.x$ and h_{i+1} exists, then v_i must align with an *extreme* edge γ (due to the rightward dragging we performed on v_i) of a possibly different obstacle and b is its projection on o. If h_{i+1} does not exist in this case, then v_i must connect q, which also makes b a projection point. Thus, b is a projection point on o. Therefore h_i is part of an edge in the boundary graph.

All edges in P satisfying (2-3) are either connecting extreme points to the projections or connecting points in $EXTM \bigcup PJ$ along the boundary of obstacles. The first kind of edges are all in SPG_{rr}, and the second kind of edges are in BG and thus also included in SPG. Note that the dragging operations may be performed symmetrically by shifting horizontal segments upwards and vertical segments leftward. Since all the edges or subpaths of P are embedded by SPG, the lemma is proved. □

We conclude with the following theorem.

THEOREM 3.4. *SPG is a shortest-path-preserving graph.*

Now we summarize the algorithm to construct SPG and find the shortest rectilinear path.

Algorithm FindSRP:

1. Sort all the obstacle edges on their X and Y coordinates.

2. By plane-sweeping horizontally and vertically, find the projection set PJ. Projection points are recorded in order in lists associated with every obstacle edge.

Each such list can be implemented as a balanced binary search tree[1].

3. Find the boundary graph BG along all obstacle boundaries by using $EXTM$ and PJ as its vertex set. This is done by traversing the obstacle boundary and linking up all the extreme points and the projection points in lists associated with obstacle edges.

4. Find SPG_{rr} as in Algorithm C2 using $EXTMSD \bigcup PJ$ as the vertex set.

5. Merge SPG_{rr} and BG to obtain SPG.

6. Find the shortest path from s to d on the graph SPG using Fredman and Tarjan's algorithm [5].

THEOREM 3.5. *The Algorithm FindSRP finds the shortest rectilinear path from s to d among rectilinear obstacles in $O(m \log t + t \log^{3/2} t)$ time using $O(m + t \log^{3/2} t)$ space.*

Proof. Steps 1 and 2 take $O(m \log t)$ time. For step 3, computing BG can be done in $O(m)$ time by traversing along the obstacle boundaries. For step 4, construction of SPG_{rr} can be computed in $O(m \log t + t \log^{3/2} t)$ time. The reader is referred to [3, 10, 14]. Since SPG_{rr} contains $O(t \log^{3/2} t)$ edges and $O(t \log^{1/2} t)$ vertices, and BG contains $O(t)$ edges and vertices, the combined SPG can be computed in time linear in the size of SPG. Finally, in step 6, applying Fredman and Tarjan's shortest path algorithm on SPG that runs in $O(|E| + |V| \log |V|)$ time, we are able to find the shortest rectilinear path in $O(t \log^{3/2} t)$ time. Overall, the time needed is of $O(m \log t + t \log^{3/2} t)$. The space needed is $O(m + t \log^{3/2} t)$. The correctness of the algorithm follows immediately. $\square$

3.2. Finding MST Among Obstacles. With the same approach, one can handle multiple input points and generate a similar graph where the minimum spanning tree (MST) is preserved. Given N terminals, Wu, *et al* [16] obtained the MST in $O(N \log N + (m + t^2) \log t)$ time based on the *track graph*. Here we simply let the essential point set be the same as we used before with all the N input terminals included. As mentioned earlier, based on Algorithm C1, we can generate SPG_r containing only $O(|I| \log |I|)$ edges and vertices for the essential point set I. Combining this graph with the boundary graph we have a different shortest-path-preserving graph, denoted SPG_m, with $O((t + N) \log(t + N))$ edges and vertices.

Since the pairwise shortest paths are all retained in SPG_m, the MST is also retained. We then adopt the algorithm presented by Wu *et al.* [15, 16] that runs in $O(|E_G| \log |V_G|)$ time for finding an MST among a set of points, S, on a graph $G = (V_G, E_G)$ with $S \subseteq V_G$.

THEOREM 3.6. *The minimum spanning tree of N terminals among obstacles can be computed in $O(N \log N + (m + N) \log t + (t + N) \log^2 (t + N))$ time, where m is the number of obstacle edges and t is the number of the extreme edges of obstacles.*

4. Faster Algorithms for MBSP, SMBP and MCP. We now deal with problems, MBSP, SMBP and MCP. We shall refer to any of *mbsp, smbp* or *mcp* as an *optimal path.*

DEFINITION 6. A point p is said to *1-dominate* a point q if the X- and Y-coordinates of p and q satisfy the condition, $p.x \geq q.x$ and $p.y \geq q.y$. In other words, point p lies in the 1st quadrant when point q is placed at the origin. 2,3,4-*dominating* relations are defined symmetrically.

Redefining the *essential point set* in Definition 4 to be $V \bigcup \{s, d\}$, and using Algorithm C1, we obtain a graph, referred to as *guidance graph*, and denoted as GG. The guidance graph will be used to *guide* our path finding algorithm. The graph was also exploited by Lee *et al.* [10] to solve shortest path problems when the obstacles are *weighted.* We summarize some properties of GG as follows.

LEMMA 4.1. [3, 10, 14] GG has the following properties:
(1) It has $O(m \log m)$ edges and vertices.
(2) V is a subset of the vertex set of GG.
(3) All edges in GG are horizontal or vertical.
(4) For any two points p and q in V, if R_{pq} is empty, then there is a shortest path between p and q in GG.
(5) It can be constructed in $O(m \log m)$ time using $O(m \log m)$ space.

We show below that the guidance graph contains at least a path which is homotopic to an optimal path.[1] Lemma 2.1 implies that if for any staircase subpath, π, of an optimal path between two points, p and q, in V, we can find a path in GG between p and q that is homotopic to π, then there is always a path in GG that is homotopic to the optimal path. We introduce the *corridor* of a staircase path between two points p and q in V. We consider only the staircase (first type) where point q 1-dominates point p. The corridor of a staircase (second type) where q 2-dominates p is defined similarly. Without loss of generality, a staircase in the following discussions refers to the first type. The second type of staircase can be treated similarly. Let π_{pq} denote the staircase path in R_{pq} for $p, q \in V$ and q 1-dominates p. Let $D(\pi_{pq})$ and $U(\pi_{pq})$ denote the sets of vertices in V that lie in R_{pq} and below and above π_{pq} respectively.

FIG. 3. *The corridor of a path.*

DEFINITION 7. (Figure 3) Given two points $p, q \in V$, with q 1-dominating p, and a staircase path π from p to q, define $corridor(p, q, \pi)$ to be the *open* rectilinear region enclosed by two staircase paths from p to q, denoted P_1 and P_2, where P_1 and P_2 satisfy the following:
P_1 passes through and makes upward turns at points in $U(\pi_{pq})$ that are not 4-dominated by any other point in $U(\pi_{pq})$ and P_2 passes through and makes rightward turns at points in $D(\pi_{pq})$ that are not 2-dominated by any other point in $D(\pi_{pq})$.

One can see that the $corridor(p, q, \pi)$ does not contain any points of V and all staircase paths between p and q in $corridor(p, q, \pi)$ are homotopic to each other.

LEMMA 4.2. *There exists a path π from s to d in GG such that π is homotopic to an optimal path from s to d.*

[1] Two paths are homotopic to each other if one can be continuously dragged to become the other without crossing any points in V.

FIG. 4. *A path in the corridor and GG.*

Proof. From lemma 2.1, we can focus on a staircase subpath between two points p and q in V. Let π^*_{pq} denote a staircase subpath from p to q of an optimal path. Let $u_{i,i=1,\ldots,k}$ and $v_{i,i=1,\ldots,t}$ represent the points in V on P_1 and P_2 that define $corridor(p, q, \pi^*_{pq})$ respectively (Figure 4). v_i 1-dominates v_j if $i > j$ and u_k 1-dominates u_l if $k > l$. Now let $z_{i,i=1,\ldots,r}$ be any *maximal* sequence such that z_i is in $u_{i,i=1,\ldots,k}$ or in $v_{i,i=1,\ldots,t}$, and z_i 1-dominates z_j if $i > j$. Apparently, all z_i's are in V and for every two consecutive elements z_i and z_{i+1}, there is no other point in $R_{z_i z_{i+1}}$. According to the definition of GG, GG provides the connection between z_i and z_{i+1} and thus there is a path in GG connecting p and q through $z_{i,i=1,\ldots,r}$. We therefore conclude that there is a path from s to d embedded in GG that is homotopic to any optimal path from s to d. ◻

FIG. 5. *Two pushed paths from p in a corridor.*

DEFINITION 8. (Figure 5) Define a *pushed staircase path* from p to q in $corridor(p, q, \pi)$ to be a staircase path from p to q, such that the path only makes turns alternately at points on P_1 and P_2. Any two-segment subpath, connecting a horizontal and a vertical segment, is called an *L-subpath.* A *canonical path* from s to d is a concatenation of pushed staircase paths connecting two points p and q with the last segment of each staircase path overlapping the leading segment of the next pushed staircase path.

LEMMA 4.3. *There are at most four different kinds of pushed staircase paths from p to q in the corridor between p and q for any $p, q \in V$.*

Proof. (Figure 5) Starting from either p or q, there are at most two pushed staircase paths, one starting horizontally and the other starting vertically. Once we decide on the first segment of the pushed path from either p or q, the rest of it is unique. □

LEMMA 4.4. *There exists an optimal path from s to d such that any of its staircase subpaths between two consecutive points $p, q \in EXTMSD$ is a pushed path.*

Proof. Consider a subpath between p and q of an assumed optimal path. Let it be Π^*_{pq}. Without loss of generality, let q 1-dominate p. Consider $corridor(p, q, \Pi^*_{pq})$ as a polygon containing Π^*_{pq}. Since there are no points in V inside the corridor, we can, starting from p, drag all the horizontal segments of Π^*_{pq} upward and vertical segments rightward until they hit the boundary of the corridor without increasing the length or the number of bends. Consequently we obtain a pushed path from p to q with the same length and number of bends as Π^*_{pq}. This applies to all such subpaths in an optimal path. The lemma is proved. □

This lemma implies that there exists an optimal path which is a canonical path.

LEMMA 4.5.
For a pushed staircase path Π from p to q, there is a path P_{GG} in GG from p to q passing through all the essential points on Π.

FIG. 6. *An L-subpath.*

Proof. According to the definition of pushed path, Π is a sequence of horizontal or vertical segments or L-subpaths connecting points in V. Since every horizontal or vertical segment between two points, u and v in V on path Π, is obviously an edge in GG, we can focus on proving that each L-subpath is also supported in GG.

Without loss of generality (Figure 6), consider an L-subpath which goes from $r \in D(\Pi_{pq})$ upward and then turns rightward to $v \in U(\Pi_{pq})$ without containing any other points in V. If R_{rv} is empty, then we are sure that the connection between r and v is supported in GG. Otherwise, there are some essential points on P_2 falling in R_{rv}. Order them as $h_i, \{i = 1, \ldots, k\}$ such that R_{r,h_1}, $R_{h_i,h_{i+1}}$ and $R_{h_k,v}$ are all empty. Based on the properties of GG, between every two consecutive points of $\{r, h_1, h_2, \ldots, h_k, w\}$ there exists a path in GG. Therefore, the L-subpath between r and v is also supported. This completes the proof. □

We call the path P_{GG} described in Lemma 4.5 a *target path.* We intend to search on GG a target path and convert it to a canonical path by some *dragging* operations. Since GG contains only vertical or horizontal edges, we always append a vertical or horizontal segment to the end of the computed pushed path when we advance in the graph searching process. The last segment of a pushed path may be subject to dragging operation. When we reach d, we have a pushed path, which will be an optimal path. By focusing only on dragging the target path, we define the *dragging* operations applied to segments of a path.

DEFINITION 9. Let $last(\pi)$ be the last staircase subpath of π from p to v, where p, $v \in V_{GG}$. A segment w of $last(\pi)$ is *fixed* if either
(1) w is horizontal (resp. vertical) and cannot be dragged any further vertically (resp. horizontally) without crossing any point in V, or
(2) it is the first segment of $last(\pi)$.
It is *floating* otherwise.

When we advance along a target path on GG, the fixed portion of the path remains fixed thereafter and need not be considered again. Only the floating segment of the path needs to be considered for possible dragging, as defined below.

FIG. 7. *Dragging operations while advancing on e.*

DEFINITION 10. Let π_{sp} be a pushed path from s to p, and let π'_{sq} be the path formed by concatenating to π_{sp} an edge $e \in E_{GG}$ from p to q, where e is either horizontal or vertical (Figure 7). Let π_{sq} be the pushed path dragged from π'_{sq} by the following dragging operations. Let w and w_{new} be the last segments of π_{sp} and π_{sq} respectively, and assume they are horizontal. The vertical case can be defined similarly. Define the *dragging operations* on π'_{sq} as follows:

- **(1)** q is to the right of p (e is horizontal) (Figure 7(a)): $w_{new} = w||e$ is of the same type (fixed or floating) as w, where '||' denotes path concatenation. Note that w is fixed if it borders an obstacle above.
- **(2)** q is to the left of p: Since w is horizontal, this is not possible.
- **(3)** q is above p (e is vertical):
  - **(3.1)** w is fixed: Let w_{new} be e. w_{new} is fixed if it borders an obstacle to the right.
  - **(3.2)** w is floating: (Figure 7(b)) Drag w upward and adjust π accordingly:
    - **(3.2.1)** If w can be dragged to q without hitting a point in V, then w_{new} is the dragged w.
    - **(3.2.2)** Otherwise, we ignore this advancing step.
- **(4)** q is below p (e is vertical): (Figure 7(c))
  - **(4.1)** If w borders an obstacle below it, we have a U-subpath here. We now have a new staircase subpath (of the second type) with w being the

leading segment, which is fixed, and $last(\pi_{sq}) = w||e$.

(4.2) Otherwise, we ignore it.

LEMMA 4.6. *A target path in GG can be successfully transformed by the dragging operations defined above to be a canonical path which is an optimal path.*

Proof. We can follow Lemma 4.5 and just focus on an L-subpath of an optimal path between two points in V. If each L-subpath from u to v can be formed by dragging a corresponding subpath on the target path from u to v, then the lemma is proved. According to Lemma 4.5 the corresponding subpath on the target path consists of horizontal and vertical edges which form a staircase path from u to v. There is no point of V in the area between this subpath and our L-shaped optimal subpath from u to v. Clearly, we will not hit any point in V when we do the dragging. Operations (1) and (3) are sufficient to drag edges in such a staircase path to form the L-subpath. Operation (3.2.2) ignores the case when we hit some point during dragging. Step (4.1) is just for making turns around a boundary edge and forming a U-subpath. Step (4.2) ignores the cases not belonging to the target path. □

4.1. The Algorithm and Its Complexity. Now we are ready to describe our algorithm: we apply Dijkstra's shortest path searching algorithm[1] on the graph GG to find the optimal path. While searching on GG and computing the pushed path, more than one pushed path may be generated when we reach a vertex from different edges. The information of the pushed paths leading to a vertex will be computed, and compared. Only the *best path(s)* obtained thus far will be retained at each vertex. We discuss below only the algorithm for MBSP. Modifications to the algorithms for solving SMBP or MCP are straightforward.

Algorithm MBSP

1. Construct the guidance graph GG, and preprocess V for the dragging operations. This can be done using method in [2], which supports $O(\log m)$ query time using $O(m \log m)$ preprocessing time and linear space.
2. Apply Dijkstra's algorithm to find the optimal path according to metric vector $v = (d(\pi), b(\pi))$, where $d(\pi)$ and $b(\pi)$ denote the length and the number of bends of π respectively. Let π_u be a pushed path obtained so far from s to a vertex, u, on GG. There may be other pushed paths computed while reaching u from other edges and have been stored in u. We thus store at u the *type* of staircase of the last staircase subpath. Do the following:
   - **2.1** If π_u is strictly worse than any other stored in u, discard π_u.
   - **2.2** If π_u is strictly better (with lexicographically smaller metric) than some of the pushed paths stored in u, we discard the worse ones and store π_u at u.
   - **2.3** If two have the same metric and the same type, compare their last segments. If their last segments overlap, then discard the one with the longer last segment. Otherwise (If they are not of the same type), keep both of them in u.
   - **2.4** Calculate the metric vector of a pushed path advancing to each neighbor v of u via edge (u, v) in GG and put that into the queue used by the Dijkstra's algorithm. Note that there are at most eight pushed paths stored at each vertex u. That is, for each type of path there are two pushed paths, with the last segment leading to u either horizontally or vertically.

When we reach a vertex u, we ignore a pushed path if there exists one with smaller metric. When two pushed paths have the same metric, yet reaching u in different directions, we simply keep them all in u and proceed with the next advancing. We are able to compare and ignore some of those pushed paths that are of the same type and reach u in the same direction. See Lemma 4.7 below. The number of pushed paths that need to be stored in each vertex u is therefore at most eight, i.e., four pushed paths from vertex v that i-dominates u, $i = 1, 2, 3, 4$, and the last segment to u can be either vertical or horizontal.

FIG. 8. *One path subsumes the other.*

LEMMA 4.7. *Consider two pushed staircase paths π_1 and π_2 with the same metric vector from s to some vertex u in GG. If the last segment, w_1, of π_1 is longer than the last segment, w_2, of π_2 and w_1 contains w_2 (Figure 8), then any pushed path generated using π_1 will not be better than the pushed path generated using π_2. The same holds for other types of paths.*

Proof. Without loss of generality, let w_1 and w_2 be horizontal. Let the other end point of w_2 be m (u is the other end point). If w_1 is fixed, then obviously π_2 can replace π_1 in any further case. If w_1 is floating, then we may obtain a pushed path from π_1 by dragging w_1 upward. Wherever w_1 goes, the area swept over by it is empty. Since w_2 is shorter, it can also be dragged to the same position as w_1 with the same vertical distance offset. The dragging adds the same vertical distance offset to both paths and yet incurs no extra bend to either path. That is, any pushed path π_1 can grow to is no better than the best pushed path grown from π_2. The lemma is proved. $\square$

THEOREM 4.8. *The problems, MBSP, SMBP and MCP can be solved in $O(m \log^2 m)$ time, using $O(m \log m)$ space.*

Proof. According to lemma 4.6, while traversing the target path and applying the dragging operations on the way, we can get an optimal path from s to d. Lemma 4.7 guarantees that when a path is discarded, there must be some other path that subsumes it. This ensures that the final path is an optimal path. As to the running time, according to Lemma 4.1, the time needed to generate the graph GG is $O(m \log m)$. The time used for searching on GG from s to d follows the complexity of Dijkstra's algorithm which is $O(|E| \log |E|)$ on a graph $G(V, E)$. The time for searching on GG is therefore $O(m \log^2 m)$. When we advance on GG through an edge, we perform dragging operations on at most eight pushed paths. Since each dragging operation can be done in $O(\log m)$ time [2] with $O(m \log m)$ time and $O(m)$ space preprocessing, the time complexity of the algorithm is $O(m \log^2 m)$. We need space for storing the guidance graph ($O(m \log m)$) and the pushed paths. Each pushed path has size less than that of its corresponding target path on GG and an edge of GG can be traversed (appended and dragged) by at most eight pushed paths from one of its end points. Therefore, the size of all the pushed paths kept while searching will be $O(E_{GG})$ or $O(m \log m)$. □

4.2. Solving MBSP, SMBP and MCP in $O(m \log^{3/2} m)$ time. The algorithm can be modified to run in $O(m \log^{3/2} m)$ time as follows. First, we find a more suitable guidance graph and a more efficient searching algorithm. Algorithm C2, as mentioned, can construct a refined, reduced graph, SPG_{rr}, providing the same connections as GG, with $O(m \log^{3/2} m)$ edges and $O(m \log^{1/2} m)$ vertices in $O(m \log^{3/2} m)$ time. We adopt this graph as the new *guidance graph*, denoted GG', and use the shortest path searching algorithm of Fredman and Tarjan, *et al.*[5], which runs in time $O(|E| + |V| \log |V|)$. The algorithm thus spends only $O(m \log^{3/2} m)$ time on graph searching. However, one of the properties in Lemma 4.1 does not hold any more, that is, the edges in GG' are not necessarily horizontal and vertical. This has effects on our dragging operations since we can only deal with horizontal and vertical segments in our dragging. We thus do some adjustments to the graph and compute substitutes for all such oblique edges.

FIG. 9. *The Z-substitute.*

From [3], each oblique edge in GG' is constructed from a set of horizontal and vertical edges in GG, which forms a *Z-shaped subpath* consisting of three segments where the first and the third lie on the different sides of the line containing the second one. (Figure 9). We call these three segments the *Z-substitute* of the original oblique edge. We can perform dragging similarly on these three segments when we encounter an oblique edge while searching on the new guidance graph and transform the target path to a pushed path.

In order to handle oblique edges, we add one dragging operation to the previous definition.

DEFINITION 11. Define dragging operations as follows: (assume $last(\pi)$ is a staircase path.)

(1-4) The same as operations (1-4) in Definition 10.

(5) For an oblique edge, replace it by the three segments of the corresponding Z-substitute. Apply the same operations as (1-4) to each of them in sequence.

Besides reducing searching time on the guidance graph GG', we must devise a better way to perform dragging, since spending $O(\log m)$ time for each dragging consumes overall $O(|E_{GG'}|\log m)$ time, which becomes $O(m\log^{5/2} m)$.

In the following, we compute the *hit vertices* for each edge (at most one in each direction) when we generate the the graph GG' in $O(m\log^{3/2} m)$ time. Thereafter, while searching and computing the path, we are able to obtain in $O(1)$ time the hit vertex for each dragging operation, thus realizing a total of $O(m\log^{3/2} m)$ time. Note that the dragging operations are applied to the last segment of a pushed path (i.e., the w_{new} in the dragging definition) which may consist of several edges of the graph GG'. For example, consider that we form w_{new} by concatenating a horizontal last segment w with a horizontal edge e on GG. We simply let the hit vertex of w_{new} be the lower one of the hit vertices of w and e when we drag w_{new} upward. Dragging horizontal segments downward or dragging vertical segments can be done similarly.

DEFINITION 12. An *upward* (resp. *downward*) *hit vertex, h,* of a horizontal edge, e, in GG is the farthest of all the obstacle vertices that e can be dragged upward (resp. downward) to hit without crossing the *interior* of any obstacle. A *leftward* or *rightward hit vertex* of a vertical edge in GG is defined symmetrically. The hit vertex of an edge, e, in GG' is defined as follows.
(1) If e is also in GG then e inherits all its hit vertices in GG.
(2) If e is not in GG, then it is an oblique edge. The hit vertices of e are defined to be the hit vertices of the three segments of the Z-substitute of e. For each vertical segment, g, the hit vertex in each direction is the nearest hit vertex of all the edges in GG that compose g.

Note that a hit vertex of an edge can be undefined if the edge cannot be dragged to hit any vertex or can only be dragged to hit a portion of the obstacle boundary which does not contain any vertex. We attempt to store the Z-substitute for each oblique edge in GG' and compute its hit vertices when we construct GG'. We will embed the computation of the hit vertices of all the edges on GG' in the recursive steps where the edges of GG' are constructed. Following is the algorithm for constructing GG' including the computation of hit vertices. Graph GG and the hit vertices of its edges are generated as intermediate products. The reader is referred to [3, 14] for details of the original algorithm. Here we only put emphasis on how we compute the hit vertices.

Algorithm FindGG'

1. Sort all vertices in $V' = V\bigcup\{s,d\}$ vertically and horizontally.
2. Find the median vertical cut line L that divides V' in halves.
3. Find the projection points, if visible, from all the vertices to the cut line L. Call those projection points the *potential Steiner points*. Call the edges connecting vertices to their projections the *potential horizontal edges*. An obstacle edge that is cut by L is marked as a potential horizontal edge with a distinction as to whether it borders the obstacle below or above. For two vertices on the same side of L and with the same ordinate, we only generate one potential Steiner point and one potential horizontal edge from the one nearer to L, if it exists. (Note that these potential points and edges are in GG. They are called *potential* since not all of them will be inserted into GG'.)
4. (Find the hit vertices of the potential horizontal edges.)
   Initialize an empty stack for storing those potential horizontal edges that are to the left of L. Scanning these edges from top to bottom, we compute the

FIG. 10. *Finding the hit vertices for the potential edges.*

downward hit vertices. Let the edge we encounter be e and let ve be the vertex that produces e (Figure 10(a)). Let the edge at the top of the stack be te.

(a) e is longer than te:

- **(a.1)** If e borders an obstacle boundary below it, then pop all the edges off from the stack leaving the hit vertices of them undefined, and let ve be the downward hit vertex of e.
- **(a.2)** Otherwise, push e onto the stack.

(b) e is not longer than te:

- **(b.1)** If e borders an obstacle boundary below it, then keep popping an edge te off from the stack until the stack is empty. Let ve be the downward hit vertex of te if te is not shorter than e and let the hit vertex of te be undefined, otherwise. Let the downward hit vertex of e be ve.
- **(b.2)** Otherwise, keep popping an edge te off from the stack as long as te is longer than e, and let the hit vertex of te be ve. Push e onto the stack.

Find the upward hit vertices in a similar way. Do the same for the potential horizontal edges on the right of L. Call the edges connecting those consecutive potential Steiner points on L the *potential vertical edges.*

5. (Find the hit vertices of the potential vertical edges.)
At each recursive step, we compute the hit vertices of all the potential vertical edges generated on the cut lines. Let L be the cut line under consideration. We look at the two nearest existing cut lines on each side of L, which were generated in previous recursive steps. Let them be L_L and L_R which are on the left side and right side of L respectively (Figure 10(b)).

(a) If there is no previous cut line on either side, i.e., the first cut line and those leftmost and rightmost cut lines during the recursive cutting process, then we simply scan over all the points on that side and find the leftward or rightward hit vertices directly. Otherwise, do the following.

(b) The leftward hit vertices of the edges on L are obtained either from vertices between L and L_L, if it exists, or from the leftward hit vertices on the potential vertical edges on L_L, if there is no vertex between L and L_L. The rightward hit vertices of all the potential vertical edges on L can be found in a similar way (from the vertices between L and L_R and those rightward hit vertices on L_R.)

6. (Find the hit vertices of the edges in GG'.)
Cut the plane into horizontal strips of size $O(\sqrt{\log m})$ each and decide which potential Steiner points and which potential edges are added to GG' (see [3, 14]). For those potential edges that are added to GG', let them have the same hit vertices. Some oblique edges which connect pairs of points on different sides of L are generated in each recursive step. We record with each oblique edge, its Z-substitute which consists of three segments: two potential horizontal edges from its end points to L and one vertical segment which is composed of all consecutive potential vertical edges between the potential Steiner points on L of the two end points. We regard the three segments in a Z-substitute as three edges while dragging and thus we need to compute the hit vertices of them respectively. The hit vertices of the two horizontal edges are those computed in Step 4. The hit vertex of the vertical segment is obtained by comparing the hit vertex of each individual potential vertical edge obtained in Step 5 and selecting the nearest.

7. Do Steps 2 through 6 recursively to the vertex sets on the left side and the right side of L respectively.

Algorithm FindGG' generates the same graph structure as in [3]. In addition, the hit vertices of all the horizontal and vertical edges on GG' and of the Z-substitutes of all the oblique edges are computed.

LEMMA 4.9. *The algorithm FindGG' runs in $O(m\log^{3/2} m)$ time and space and the hit vertices of all the edges in GG' are correctly computed.*

Proof. The complexity is the same as Algorithm C2 since in each recursive step we compute the hit vertices in time linear in the number of points processed. The only step we need to address is the computation of Z-substitutes and their hit vertices for each pair of vertices inside a strip. Inside each strip, as the accumulated distances are computed [3, 14, 10], we construct a hit vertex table recording the nearest hit vertex in each direction between every pair (not necessarily consecutive) of potential Steiner vertices on L in the strip. This information can later be used for computing the hit vertices of a vertical segment in a Z-substitute. The time complexity therefore is still kept within the same bound. For the correctness of computing the hit vertices, we distinguish two cases as follows:

(1) When a vertex, h, is the downward hit vertex of a potential horizontal edge, e:
If there is any other potential horizontal edge, e', between h and e, it must not be shorter than e. If e' is as long as e then it must not border an obstacle below. According to the algorithm, for these cases, e' will be pushed onto a stack. When h is scanned, we pop all edges on top of the stack before we meet an edge shorter than the edge from h to L, which certainly includes e. The case when h is the upward hit vertex of e is proved similarly.

(2) When a vertex, h, is the leftward hit vertex of a potential vertical edge, e:
Let e be on a cut line L. If h falls between L and L_L, then we can certainly find

it during the scan. If h falls to the left of L_L, then h must be a hit vertex of some potential vertical edge on L_L. We can assume that h has been recorded as the leftward hit vertex of this potential vertical edge. The reason is that since e can be dragged over L_L to hit h, the two vertices that have projections on the endpoints of e must have projections on L_L as well. These two projections are two potential Steiner points on L_L (denoted as a and b in Figure 10(b)). Therefore, there must be one or more potential vertical edges on L_L between a and b, and one of them will hit h when dragged leftward. Hence by scanning all the hit vertices on L_L and vertices between L_L and L, we can correctly compute the hit vertices of edges on L. The proof for computing the rightward hit vertices is similar. □

After we have computed all the hit vertices of all the edges of the new guidance graph, the dragging operations can be redefined to be the following:

DEFINITION 13. The dragging operations are defined to be those specified in Definitions 10 and 11 and the following corresponding operations:
(1) The upward hit vertex of w_{new} is the lower one of the upward hit vertices of w and e.
(2) The same as in Definition 10.
(3.1) The upward hit vertex of w_{new} is the upward hit vertex of e.
(3.2.1) If the upward hit vertex of h is higher than q then we drag w to q and let it be w_{new}. The upward hit vertex of w_{new} is that of w.
(3.2.2) The same as in Definition 10.
(4) and (5) are the same as in Definition 11.

One can see that such dragging operations have the same effects as do the previously defined ones. However, we do not perform segment dragging queries any more. The algorithm is the same except the dragging operations are modified. The space needed is $O(|E_{GG'}|)$ for storing GG'.

THEOREM 4.10. *Problems MBSP, SMBP and MCP can be solved in* $O(m \log^{3/2} m)$ *time and space.*

5. Conclusion. In this paper, we have presented two results improving upon previous ones on finding rectilinear paths among obstacles. We have shown that a *smaller* shortest-path-preserving graph for shortest rectilinear paths among rectilinear obstacles is sufficient and thus obtained a more efficient algorithm for problem *sp*. As a by-product, a faster algorithm for finding the minimum spanning tree of a set of terminals among obstacles is also obtained. Furthermore we have presented a dynamic-searching approach which computes optimal paths dynamically while searching on a guidance graph. Problems MBSP, SMBP and MCP can be solved efficiently using this approach. It is not clear whether a *better* guidance graph can be obtained to yield a more efficient algorithm for these problems. The problems of how one can improve the time complexities of these algorithms remain open. The results of this paper cannot be extended to non-rectilinear cases in an obvious way.

Acknowledgement. The authors wish to thank the anonymous referee for the comments that greatly helped improve the presentation of the paper.

REFERENCES

[1] A. AHO, J. HOPCROFT, J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] B. Chazelle, *An Algorithm for segment dragging and its implementation*, Algorithmica, 3 (1988), pp. 205—221.

[3] K. L. Clarkson, S. Kapoor, and P. M. Vaidya, *Rectilinear shortest paths through polygonal obstacles in $O(n \log^{3/2} n)$ time*, manuscript, submitted for publication, 1989.

[4] P. J. deRezende, D. T. Lee, and Y. F. Wu, *Rectilinear shortest paths with rectangular barriers*, Discrete and Computational Geometry, 4 (1989), pp. 41—53.

[5] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM, 34 (1987), pp. 596—615.

[6] H. Imai and T. Asano, *Dynamic segment intersection search with applications*, Proc. 25th IEEE Symp. on Foundations of Computer Science, Singer Island, Florida, 1984, pp. 393—402.

[7] R. C. Larson and V. O. Li, *Finding minimum rectilinear distance paths in the presence of barriers*, Networks, 11 (1981), pp. 285—304.

[8] D. T. Lee, *Proximity and reachability in the plane*, Ph. D. Dissertation, Department of Computer Science, University of Illinois, 1978.

[9] D. T. Lee and F. P. Preparata, *Euclidean shortest paths in the presence of rectilinear barriers*, Networks, 14 (1984), pp. 393—410.

[10] D. T. Lee, C. D. Yang and T. H. Chen, *Shortest rectilinear paths among weighted obstacles*, International J. Computational Geometry & Applications, 1 (1991), pp. 109—124.

[11] J. S. B. Mitchell, *Shortest rectilinear paths among obstacles*, Algorithmica, 8 (1992), pp. 55—88.

[12] ———, *An optimal algorithm for shortest rectilinear path among obstacles*, presented in the First Canadian Conference on Computational Geometry, Montreal, Canada, Aug. 1989.

[13] T. Ohtsuki, *Gridless routers — New wire routing algorithm based on computational geometry*, International Conference on Circuits and Systems, China, 1985.

[14] P. Widmayer, *Network design issues in VLSI*, manuscript, 1989.

[15] Y. F. Wu, P. Widmayer and C. K. Wong, *A faster approximation algorithm for the Steiner problem in graphs*, Acta Informatica, 23 (1986), pp. 223—229.

[16] Y. F. Wu, P. Widmayer, M. D. F. Schlag, and C. K. Wong, *Rectilinear shortest paths and minimum spanning trees in the presence of rectilinear obstacles*, IEEE Trans. Comput., C-36 (1987), pp. 321—331.

[17] C. D. Yang, D. T. Lee and C. K. Wong, *On bends and lengths of rectilinear paths: A graph-theoretic approach*, International J. Computational Geometry & Applications, 2 (1992), pp. 61—74.

[18] ———, *On bends and distances of paths among obstacles in two-layer interconnection model*, IEEE Trans. Comput., Vol. 43, No. 6, June 1994, pp. 711-724.