Research Contribution

Shortest Rectilinear Paths among Weighted Rectangles[1]

C. D. Yang*, T. H. Chen* and D. T. Lee*

We consider the problem of a rectilinear shortest path among weighted obstacles. Instead of restricting a path to avoid obstacles totally, we allow it to pass through them at extra costs. The extra costs are represented by the weights of the obstacles. We aim to find a shortest rectilinear path between two distinguished points among a set of disjoint, weighted rectangles. By using a plane sweep approach and a data structure called the *weighted segment tree*, we obtain an algorithm that runs in optimal $\Theta(n \log n)$ time and $\Theta(n)$ space, where n is the number of rectangles.

1. Introduction

One of the fundamental problems in computational geometry is the problem of finding the shortest path between two points in a plane. The problem varies according to the metric used and the types of obstacle treated. Most previous results [1-7, 9] deal with problems on the assumption that the path does not cross any of the obstacles, namely, that the path is *collision-free*. In the real world, finding a shortest path that totally avoids any obstacle may be undesirable and sometimes inadequate. In these circumstances, obstacles can be considered as regions that can be penetrated at extra costs. For example, consider the case of going from city A to city B with city C in between. We can choose either to go though city C or to by-pass it, depending on how we measure the cost. The distance between cities A and B when passing directly through city C may be shorter than when by-passing it, but may involve an extra cost, such as a delay caused by heavy traffic. If we treat this extra cost as a weight associated with each obstacle, then the problem becomes one of finding a minimal-cost path between two given points s and t in the presence of weighted obstacles.

Mitchell and Papadimitriou [8] first introduced the 'weighted region problem' and described an $O(n^8L)$ algorithm, where L is the precision of the problem instance (including the number of bits required to specify the largest integer among the weights and the coordinates of vertices) and n is the number of the weighted regions. The algorithm finds a shortest path in the Euclidean metric, that is, a path which minimizes the sum of the path lengths multiplied by the respective weight factors of the regions through which the path passes. In this paper we will consider the rectilinear case, in which the obstacles are disjoint (open) rectangles and the distance metric is the L_1-metric or Manhattan distance. This paper is a natural generalization of the previous work done by deRezende *et al.* [2], and is organized as follows. In Section 2, we introduce the weighted segment tree structure and briefly describe operations on the structure. In Section 3 we introduce our notation and give some preliminary results. In Section 4, we describe our main result, a $\Theta(n \log n)$ time algorithm.

[1]Supported in part by the National Science Foundation under Grants CCR 8420814 and CCR 8901815.

*Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, ILL 60208.

2. Weighted Segment Trees

The conventional segment tree is a static structure designed to handle intervals on real line whose extremes belong to a fixed set on n abscissae [10]. The abscissae are fixed and can be normalized by replacing each of them according to rank in left-to-right order. Without loss of generality, we may consider these abscissae as integers in the range $[1, n+1)$. The segment tree is a rooted binary tree. Given integers l and r with $l<r$, the segment tree $T(l, r)$ is recursively built as follows. It consists of a root v, with attributes $v.B=l$ and $v.E=r$, where $v.B$ and $v.E$ are mnemonics for 'beginning' and 'end,' respectively, and if $r-l>1$, of a left subtree $T(l, \lfloor (v.B+v.E)/2 \rfloor)$, denoted as $ls(v)$, and a right subtree $T(\lfloor (v.B+v.E)/2 \rfloor, r)$, denoted as $rs(v)$. The attributes $v.B$ and $v.E$ of node v define the interval $[v.B, v.E)$, called a *standard interval*. The standard intervals pertaining to the leaves of $T(l, r)$ are called the *elementary intervals*. For $r-l>3$, an arbitrary interval (b, e), with integer $b<e$, will be partitioned into a collection, called a *canonical covering* of (b, e), of at most $\lceil \log_2 (r-l) \rceil + \lfloor \log_2 (r-l) \rfloor - 2$ standard intervals of $T(l, r)$.

In addition to the attributes *u.B*, *u.E*, and *u.M* associated with each node, where *u.M* is $\lfloor (u.B+u.E)/2 \rfloor$, there are two variables *u.w* and *u.e* in the weighted segment tree. The first represents the weight of the interval [*u.B*, *u.E*) and the second is defined later.

Basic operations performed on the weighted segment tree structure include *addw, resetw, setw*, and *getw*: 'adding a weight to a given interval,' 'resetting (to 0) the weight of a given interval,' 'setting the weight of an interval,' and 'getting the accumulated weight of a given elementary interval.' The first, *addw*, is implemented by adding the given weight to every node in the canonical covering of the interval. Afterwards, the accumulated weight of an elementary interval can be queried (by calling *getw*) by summing up all the weights stored in the nodes along the root-to-leaf path. The difficulty lies in the *resetw* operation. Suppose we want to reset the weight of an elementary interval. We cannot simply reset the weights of all the nodes on the path, because they might be commonly used by other elementary intervals whose weights are not supposed to be reset. Thus, we employ the idea of compensating for the accumulated weight of the nodes by adding to the node corresponding to the elementary interval an equivalent negative weight (making the total weight 0). In general, if we want to reset the weight of an interval, we will perform compensation operations at the nodes in the canonical covering of the interval and place a reset mark at each of these nodes.
The reset mark is used to signify that the weights of the nodes below the marked node have all been reset and become useless. Consequently, the query operation is modified so as to accumulate the weights of the nodes on the root-to-leaf path until a marked node is encountered. The operation of adding weight to a given interval is also modified as follows. If a marked node is encountered on the path from the root to a node in the canonical covering of the interval, the node is first 'unmarked' without changing its weight, and the effect of the 'marking' is pushed downward to its two children by resetting their weights and marking them. The *setw* operation can be implemented by calling consecutively the *resetw* and *addw* operation on the same given interval. The modification follows the rule that all attributes *on and above* a (reset) marked node are valid, and that the rest are invalid and considered to be all zeros.

The following are the descriptions of the operations:

```
resetw (v: interval, u: node, wsum: int)
  begin
    if u is marked then begin
(3)     ls(u).w=rs(u).w=0
        mark ls(u) and rs(u)
        unmark u
(6) end
    /*now, u must be an unmarked node
(7) if (v.B≤u.B and u.E≤v.E)
    /*u is in the canonical covering
    then begin
        mark u
        /*and store the compensation value
        u.w= -(wsum)
      end
    else begin
        wsum=wsum+u.w
        if (v.B<u.M) then
        resetw (v, ls(u), wsum)
        if (u.M<v.E)
        resetw (v, rs(u), wsum)
      end
  end
addw (v: interval, u: node, weight: int)
  begin
    if u is marked then begin
(3)     ls(u).w=rs(u).w=0
        mark ls(u) and rs(u)
        unmark u
(6) end
(7) if (v.B≤u.B and u.E≤v.E)
    then u.w=u.w+weight
    else begin
        if (v.B<u.M) then
        addw (v, ls(u), weight)
        if (u.M<v.E) then
        addw (v, rs(u), weight)
      end
  end
getw (ev: int, u: node)
  begin
    if (u.B=ev and u.E=ev+1)
    then return (u.w)
    if (u is reset-marked)
    then return (u.w)
    if (ev<u.M) then
      return (u.w+getw (ev, ls(u))
    if (ev>u.M) then
      return (u.w+getw (ev, rs(u))
  end
setw (v: interval, u: node, weight: int)
  begin
    resetw (v, u, 0)
    addw (v, u, weight)
  end
```

Lemma 1. Each operation **getw** (*ev*, *root*), **resetw** (*v*, *root*, 0) **setw** (*v*, *root*, *weight*) and **addw** (*v*, *root*, *weight*), runs in $O(\log n)$ time.
[proof]: Immediate.◇.

We illustrate the above operations by an example given in Fig. 1. For simplicity, let the elementary interval $[i, i+1)$ represent integer i, and let the query interval $[a, b]$ be an interval covering integers from a to b. Initially, the weight of every node is zero and all are unmarked. Figure 1 shows the final weighted segment tree after a sequence of **addw** and **resetw** operations. If we then perform **getw** (7, root), we will get 2.

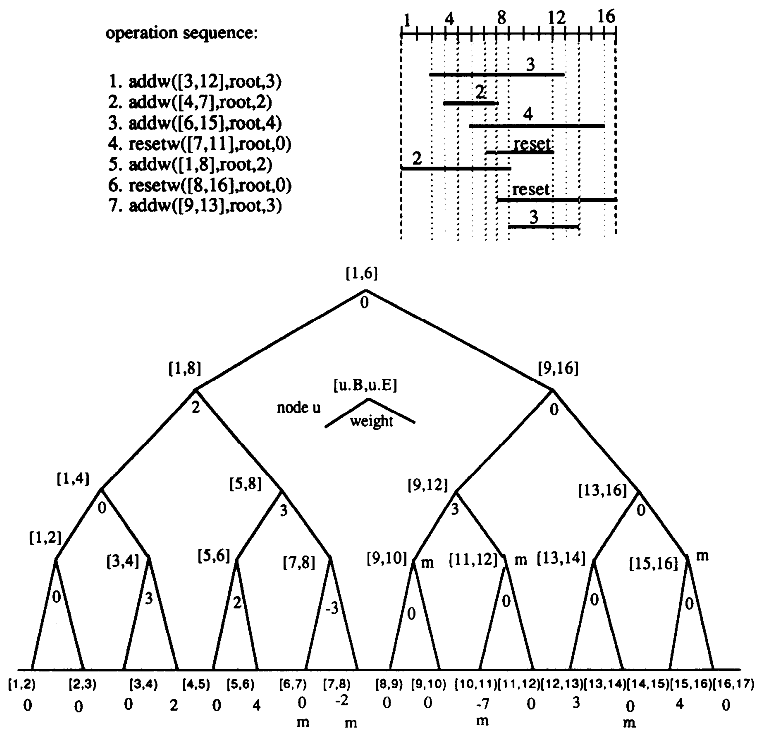

Fig. 1 Illustration of weighted segment tree and the associated operations addw, resetw and getw.

3. Preliminaries

Definition 1. A *rectilinear path* Π_{st} is a path connecting two points s and t, that consists of only horizontal or vertical line segements, and its length is denoted by $|\Pi_{st}|$.

Unless otherwise specified, we use the term *path* to mean a rectilinear path.

Definition 2. Let Π_{st} be denoted as q_1, p_1, q_2, p_2, q_3, p_3, . . . q_k, p_k, where q_i is a path outside any obstacle, and p_i is a path completely within some obstacle R_i. q_1 or p_k may be of zero length. Then the *weighted length* of Π_{st}, denoted $dw(\Pi_{st})$, is defined as $dw(\Pi_{st})=\Sigma_{i=1}^{k}(|q_i|+|p_i|)+\Sigma_{i=1}^{k}(R_i.w*|p_i|)$, where $R_i.w$ denotes the weight of R_i.

Definition 3. A *shortest path* Π_{st}^* between s and t is the one that has the smallest weighted length among all possible (Π_{st})'s. The *weighted shortest path length*, $dw(\Pi_{st}^*)$, is denoted as dw_{st}^*, for short.

Note that Π_{st}^* is not unique. Without loss of generality, we let point s be the origin of the XY-coordinate system and point t be located to the right of the Y-axis.

Definition 4. Π_{st} is said to be *monotone* in the X-axis if for all vertical lines V the intersection of Π_{st} is either empty, a point on Π_{st}, or a line segment in Π_{st}. Similarly, Π_{st} is said to be monotone in Y-axis if for all horizontal lines H the intersection of H and Π_{st} is either empty, a point on Π_{st}, or a line segment in Π_{st}.

Definition 5. An *X-path* is a directed path that goes only in the $+X$, $+Y$, or $-Y$ direction; $(-X)$, Y, and $(-Y)$*-paths* are defined similarly.

Notice that an X-path or a $(-X)$-path is monotone in X-axis and a Y-path or a $(-Y)$-path is monotone in Y-axis.

Definition 6. An *XY-path* is one that is an X-path and also a Y-path. $X(-Y)$, $(-X)Y$, and $(-X)(-Y)$*-paths* are defined similarly.

Definition 7. A *Y-preferred* XY-path is an XY-path that always goes in the $+Y$ direction if possible, without crossing any obstacle. X, $(-X)$ and $(-Y)$*-preferred* AB-paths, where $A=\{X, (-X)\}$ and $B=\{Y, (-Y)\}$, are defined similarly.

Definition 8. The *X-region* of a point s, denoted as X-region(s), is the region bounded by the Y-preferred XY-path, denoted as $P1$, and the $(-Y)$-preferred $X(-Y)$-path, denoted as $P2$, starting from s. $(-X)$, Y and $(-Y)$ regions are similarly defined. (Fig. 2).

3.1 Monotonicity of the Shortest Path

Lemma 2. If a path that does not cross any obstacles, Π_{st}, exists and is an XY-path, an $X(-Y)$-path, a $(-X)$ Y-path or a $(-X)(-Y)$-path, then $\Pi_{st}=\Pi_{st}^*$.

The following lemma shows the *monotonicity* of Π_{st}^*.

Lemma 3. Π_{st}^* must be a U-path if $t\in U$-region(s) where $U\in\{X, Y, -X, -Y\}$.

[proof]: The proof follows the same arguments given

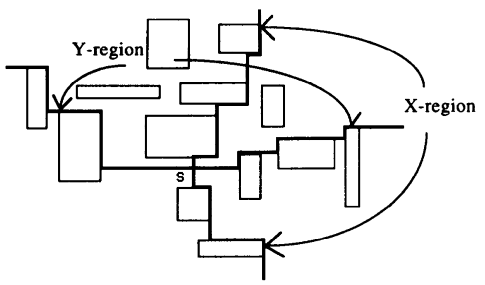

Fig. 2

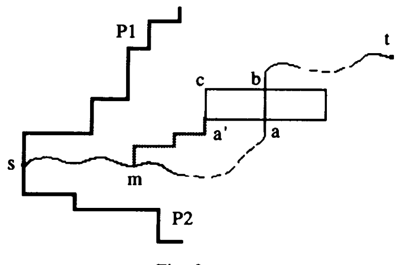

Fig. 3

by deRezende *et al.* [2]. ◇

Lemma 4. Suppose $\Pi^*_{st} \cap R_i = \Pi_{ab}$, and $\Pi_{ab} \cap int(R_i) \neq \phi$, where a and b are two distinct points on the boundary of R_i, and $int(R_i)$ denotes the interior of R_i. Then the subpath Π_{ab} must be either a horizontal or a vertical segment.

[proof]: Immediate. ◇

A path Π_{st} containing a horizontal or vertical segment that lies in $int(R_i)$ is said to have a *horizontal* or *vertical penetration* at R_i, respectively. Unless otherwise specified, we assume from now on that $t \in X$-region(s), and $R_i \in X$-region(s), for all i.

Lemma 5. There exists Π^*_{st} without any vertical penetration.

[proof]: Suppose there were a vertical penetration at R_i in Π^*_{st}. Assume that R_i is penetrated from point a on the bottom edge to point b on the top edge (Fig. 3). Let the lower left corner and upper left corner of R_i be denoted a' and c, respectively. We first form a $(-Y)$-preferred $(-X)(-Y)$-path P from a'. P will intersect either P_1 or Π_{sa} at some point m. We can construct a new path $\Pi^{new'}$ by concatenating Π_{sm}, $\Pi_{ma'}$, $\overline{a'a}$, $\overline{ab}$ and Π_{bt}, denoted $\Pi^{new'} = \Pi_{sm} \| \Pi_{ma'} \| \overline{a'a} \| \overline{ab} \| \Pi_{bt}$. $\Pi^{new'}$ is not longer than Π^*_{st}. Now, replacing $\overline{a'a} \| \overline{ab}$ on $\Pi^{new'}$ by $\overline{a'c} \| \overline{cb}$, we get a better path Π^{new} that avoids the vertical penetration.

Similarly, Π^*_{st} does not have a vertical penetration if $t \in (-X)$-region(s), and Π^*_{st} has no horizontal penetrations if $t \in Y$-region(s) or $t \in (-Y)$-region(s).

Definition 9. A point a' is the *left projection* of a point $a \in X$-region(s) if a' is horizontally visible from a and lies on the path $P1$ or $P2$ or on the right side of some rectangle.

Lemma 6. There exists a shortest path Π^*_{st} that passes through the left projection of t.

[proof]: Immediate. ◇

Lemma 6 implies that Π^*_{st} can always be rewritten as $\Pi^*_{st'} \| \overline{t't}$, where t' is the left projection of t. Let $R_i.e = \overline{ab}$ with a above b denote the right side of rectangle R_i. Assume t' is on $R_i.e$. According to *Lemmas* 3 and 4, $\Pi^*_{st'}$ has either a horizontal penetration at R_i or a vertical segement along $R_i.e$. In the former case, the problem is the same as if R_i were removed. Of course, the length of Π'^*_{st} is off by $W_i * R_i.w$, where W_i denotes the horizontal width of R_i. In the latter case, Π^*_{st} has a 'turn' at t', and because of the monotonicity property, $\Pi^*_{st'}$ must pass through either a or b. The problem thereafter reduces to that of finding Π^*_{sa} or Π^*_{sb}. Similar arguments apply to the left projections of a and b. Thus, we need to have a mechanism to determine if and where a penetration is to occur at R_i for each R_i.

For a point q on $R_i.e$, let $u\,\Pi^*_{sq}$, $d\,\Pi^*_{sq}$ and $p\,\Pi^*_{sq}$ denote, respectively, the shortest paths from s to q that pass through a, b, and $int(R_i)$. We define $f_u(q)$ and $f_d(q)$ as follows:

Definition 10. $f_u(q) = dw(u\,\Pi^*_{sq}) - dw(p\,\Pi^*_{sq})$
$f_d(q) = dw(d\,\Pi^*_{sq}) - dw(p\,\Pi^*_{sq})$

Definition 11. A point q on $R_i.e$ is called a *penetrating point* if $f_u(q) > 0$ and $f_d(q) > 0$. The set $PR_i = \{q \mid q$ is a penetrating point of $R_i\}$ is called the *penetration set* of R_i. The upper limit of PR_i, denoted as $PR_i.u$, is the maximum of the Y-coordinates of the points in PR_i, and the lower limit of PR_i, denoted $PR_i.l$, is the minimum of the Y-coordinates of the points in PR_i.

Lemma 7. The penetration set PR_i of R_i, if non-empty, is a vertical continuous open interval $(PR_i.l, PR_i.u)$ corresponding to a subsegment of $R_i.e$.

[proof]: Assume that PR_i consists of more than one interval, called *penetrating subintervals (Fig. 4). Consider a point $p \notin PR_i$* that lies between two consecutive penetrating subintervals. There exists a corner-turning shortest path Π^*_{ps} along $R_i.e$ that must go either up or down, passing through one of the penetrating subintervals. When it reaches a point in PR_i, it *must* penetrate R_i. But then this path can be replaced by one that penetrates R_i directly at p and makes an appropriate turn to reach the point at which the former path exits from R_i. That is, p is a penetrating point, a contradiction. The lemma is proved. ◇

Lemma 8. For any two points q and r, q above r. on $R_i.e$ where $PR_i \neq \phi$, we have $(i) f_u(q) \leq f_u(r)$ and $(ii) f_d(q) \geq f_d(r)$.

[proof]: It follows from the triangle inequality. ◇

With the information of PR_i for each rectangle in X-region(s), we can construct Π^*_{st} by tracing backwards from t as follows. We first find the left projection t' of t on R_i, and decide whether to make a turn or a penetration. For a turn, we known whether Π^*_{ts} should turn upwards to a or downwards to b along $R_i.e$. We continue finding the left projection of the point a or b, and

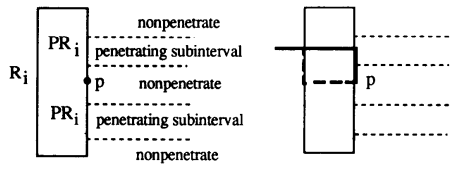

Fig. 4 There is only one penetrating interval for each rectangle.

repeat thus process until we hit $P1$, $P2$, or s. For a penetration, we continue finding the left projection of t' as if R_i were nonexistent, and repeat the same process. The first left projection point at which a turn is made is called a *nonpenetrating left projection.* Findry the nonpenetrating left projection in this manner appears to be a time-consuming process. It turns out that the nonpenetrating left projection of a given point t can be calculated easily by a plane sweep method, as described below.

4. The Algorithm

Let V be a set of vertical segments containing those on $P1$ and $P2$, $R_i.e$ for all R_i and the point t (regarded as a degenerated segment), and let C denote the set of Y-coordinates of the endpoints of all the segments in V. For each R_i, define $\overline{PR_i}$ to be the smallest open interval (l, u), l, $u \in C$, that encloses the penetrating interval $(PR_i.l, PR_i.u)$. $\overline{PR_i} = \phi$, that is, $l \geq u$, if $PR_i = \phi$. We are now ready to describe the plane sweep method, which also computes $\overline{PR_i}$ for each R_i. We shall sweep the X-region(s) from left to right by a vertical sweep line L stopping at each segment in V. The sweep line L contains a set of *cut points*, $\{c_i\}$, $i = 1, 2, \ldots, m$, whose Y-coordinates are exactly those in C, and these cut points are sorted by their Y-coordinates. Associated with each cut point c on L, we have two attributes, $c.e$ and $c.w$. The first denotes the index of some segment in V to the left of L that contains the first nonpenetrating left projection c' of c. The second, $c.w = \sum_{j=1}^{k} R_j.w * W_j$, where W_j is the width of R_j penetrated by $\overline{cc'}$, and $c.w = 0$ if no R_j exists, in other words, c' is the left projection of c.

With these two attributes, $c.e$ and $c.w$, of c maintained on L, we can determine the weighted length of Π^*_{sc} for each cut point c on L as $dw^*_{sc} = dw^*_{sc'} + |\overline{c'c}| + c.w$, where c' is on the vertical segment $c.e$ and $dw^*_{sc'}$ is obtained as follows (Fig. 5):

(1) if $c' \in R_i.e$ and $c'.y \geq PR_i.u$, then
$dw^*_{sc'} = dw^*_{sa} + |\overline{c'a}|$,
where a is the upper endpoint of $R_i.e$

(2) if $c' \in R_i.e$ and $c'.y \geq PR_i.l$, then
$dw^*_{sc'} = dw^*_{sb} + |\overline{c'b}|$,
where b is the upper endpoint of $R_i.e$

(3) if c' is on $P1$ or $P2$, then
$dw^*_{sc'} = |c'.x - s.x| + |c'.y - s.y|$.

Definition 12. Let $u.e$ be an attribute associated with each internal node u in the weighted segment tree T.

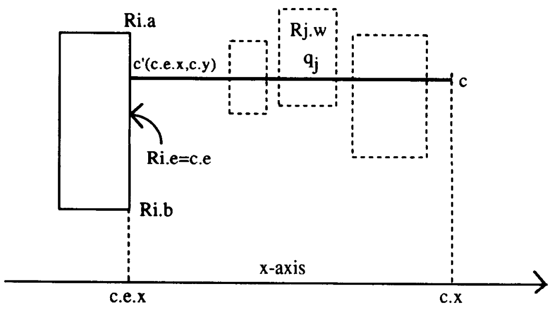

Fig. 5 Illustration of a cut point c and its attributes.

Define **sete**, **resete** and **adde** to be functions identical to **setw**, **resetw** and **addw**, respectively, but applying on the attribute $u.e$ instead of $u.w$. **gete** is the same as **getw** except that it extracts information from $u.e$ instead of $u.w$.

Functions **sete** and **gete** are designed to maintain $c.e$ for each cut point c. We are now ready to describe the algorithm.

Initially, the variables $u.w$ and $u.e$ for each node u in the weighted segment tree are set to zero, and every node is unmarked.

Let $R_i.e = \{c_i = b, c_{i+1}, \ldots, c_j = a\}$ denote all the cut points on $R_i.e$. Recall that **getw** (c_i, root) [1]returns $c_i.w$, the accumulated weight of the elementary interval $[c_i, c_{i+1})$, in $O(\log n)$ time. $f_u(c_k)$, by definition, is equal to $dw^*_{sa} + |\overline{ac_k}| - dw^*_{s,c'_k} - |\overline{c'_k, c_k}| - \mathbf{getw}(c_k, root)$, and $f_d(c_k) = dw^*_{sb} + |\overline{bc_k}| - dw^*_{s,c'_k} - |\overline{c'_k, c_k}| - \mathbf{getw}(c_k, root)$, where c'_k is the projection of c_k on $c_k.e$ and $c_k.e$ is obtained by calling **gete**(c_k, root). The detail is given in Algorithm **SRP0**.

4.1 Algorithm SRP0

1. Partition the plane into four regions according to the given query point s.

2. Determine the region in which the given query point t falls. (assume t falls in the X-region(s)).

3. Construct the sweep line L, consisting of cut points $CC = \{c_i\}$, $i = 1, 2, \ldots, m$, sorted in ascending order of their Y-coordinates and all $c_i.y \in C$ and create the weighted segment tree T for CC in L.

4. Sort the vertical segments in V.

5. Start sweeping from s to t from left to right, stopping at each segment in V. For each sweeping step, do the following:

5.1 If the current L contains $R_i.e$, do the following: (let $R_i.e = \overline{ab}$ with a above b, and let I denote the cut point list of L on $R_i.e$, that is, $I = \{c_i = b, c_{i+1}, \ldots, c_{j-1}, c_j = a\}$)

5.1.1 Use binary search to find intervals $[c_x, c_{x+1})$ and $(c_{y-1}, c_y]$, $x < y$, on I containing $PR_i.l$ and $PR_i.u$, respectively, as follows. Take the middle cut point $c_k \in I$. If $f_d(c_k) > 0$, then c_x is below c_k; otherwise c_x is equal to

[1]We use **getw**(c_i, v) to denote **getw**($c_i.y$, v).

or above c_k. In the former case, reduce the search domain to $\{c_i, \ldots, c_{k-1}\}$; in the latter, consider $\{c_k, \ldots, c_j\}$. The search continues until $f_d(c_x) \le 0$ and $f_d(c_{x+1}) > 0$ for some $i \le x \le j$. Do the same to locate $(c_{y-1}, c_y]$ so that $f_u(c_y) \le 0$ and $f_u(c_{y-1}) > 0$. Let $l = c_x$ and $u = c_y$. $\overline{PR_i}$ exists iff u and l are found and $u > l$. Let $\overline{PR_i} = (l, u)$.

5.1.2 If $\overline{PR_i}$ *exists, then for all* c in intervals $[u, c_{j-1}]$, and $[c_{i+1}, l]$ perform **resetw** operation and set $c.e$ to $R_i.e$ and for all c in (l, u), perform the **addw** operation. If $\overline{PR_i}$ does not exist, then for all $c \in [c_{i+1}, c_{j-1}]$. perform the **resetw** operation and set $c.e$ to $R_i.e$. Calculate and store in R_i the information dw^*_{sa}, dw^*_{sb}, and $\overline{PR_i}$, if it exists.

5.2 If the current sweep line contains a vertical edge ve on $P1$ or $P2$, perform the **resetw** operation and set $c.e$ to ve for those cut points c in ve.

5.3 When the sweep line L reaches the point t, we can start tracing back from t by first finding the nonpenetrating left projection of t (say c). The rectangle R_i containing $c.e$ is the first place that the shortest path should make a turn. By comparing $\overline{PR_i}$ with $t.y$, we determine whether it should turn to a or to b. Now continue the recursive steps from the points a or to b. Now continue the recursive steps from the points a or b until we hit $P1$ or $P2$ at some point, say m. The shortest path from s to t will be the path concatenated by the path from s to m on $P1$ or $P2$ and the segments found in each recursive step.

In the algorithm **SRP0**, we spend $O(\log n)$ time evaluating $f_d(c_k)$ and $f_u(c_k)$ (the **getw** and **gete** operations), and therefore $O(\log^2 n)$ time locating the penetrating interval of each vertical segment. Thus the total time complexity of the algorithm is $O(n \log^2 n)$.

Theorem 1. There exists an algorithm that solves the shortest path problem in the presence of n weight rectangles in $O(n \log^2 n)$ time and $O(n)$ space.

4.2 Extended Weighted Segment Tree

We shall show below that the time complexity can be improved further by modifying the underlying data structure and using a *global* search to find the penetrating interval in $O(\log n)$ time in each sweeping step. The idea is as follows. For each interval $[u.B, u.E)$ in the canonical covering of interval I, we expect to know in $O(1)$ time whether it contains a penetrating point. If so, we need to know which subtree of node u to search further.

We therefore maintain in each node u four attributes *u.law*, *u.raw*, *u.lae* and *u.rae*; *u.law* is the *left accumulated weight* of those nodes along the leftmost subpath from u to the leftmost leaf, and *u.raw* is the *right accumulated weight* of those nodes along the rightmost subpath from u to the rightmost leaf. The weight of u is included in these two new attributes. That is, if v is the leftmost or rightmost leaf of the subtree rooted at u, its accumulated weight $v.w$ can be obtained in $O(1)$ time from *u.law* or *u.raw* respectively.

u.lae and *u.rae*, like *u.law* and *u.raw*, store the indexes, *lm.e* and *rm.e*, of the leftmost and rightmost leaves *lm* and *rm* of u.

That is, left
$L(u)$ = set of nodes on the leftmost subpath below and including u and
$R(u)$ = set of nodes on the rightmost subpath below and including u.
We have $u.law = \Sigma_{v \in L(u)} (v.w)$, $u.raw = \Sigma_{v \in R(u)} (v.w)$, $u.lae = \Sigma_{v \in L(u)} (v.e)$, and $u.rae = \Sigma_{v \in R(u)} (v.e)$.

After finding these attributes in advance, we can locate the intervals $[c_k, c_{x+1})$ and $(c_{y-1}, c_y]$ containing $PR_i.u$ and $PR_i.l$, respectively, in $O(\log n)$ time within those subtrees rooted at the canonical covering nodes for interval $R_i.e$, as described below.

1. Initialize all *u.law*, *u.raw*, *u.lae*, and *u.rae* at zero.

2. Modify the algorithm **resetw** by adding between lines (6) and (7) the following:

```
if (v.B ≤ u.B) then u.law = -(wsum)
if (v.E ≥ u.E) then u.raw = -(wsum)
```

This is to ensure that if the leftmost (or rightmost) leaf of the subtree $T(u)$ rooted at u is in the interval to be reset, the left (or right) accumulated weight is adjusted accordingly. Make similar modifications to **resete** for *u.lae* and *u.rae*.

3. By the same token, modify the algorithm *addw* by adding between lines (6) and (7) the following,

```
if (v.B ≤ u.B) then u.law = u.law + weight
if (v.E ≥ u.E) then u.raw = u.raw + weight
```

Make similar modifications to **adde** for *u.lae* and *u.rae*.

4. To be consistent with the rule that all attributes on and above a marked node are valid, add the following after line (3) in algorithms **addw** and **resetw**.

$ls(u)$.law = $ls(u)$.raw = $rs(u)$.law = $rs(u)$.raw = 0.

Add similar statements for attributes *lae* and *rae* in algorithms **adde** and **resete**.

5. To find $\overline{PR_i} = (u, l)$ of a rectangle R_i, a new operation **bi_search** as described below is used to find an elementary interval $[c_{y-1}, c_y)$ that contains $PR_i.u$.

5.1 Find the canonical covering C of $R_i.e$.

5.2 Examine the node $v \in C$ from right to left in the tree (that is, from high to low ordinate), and check $f_u(v.B)$. Note that $v.B$ corresponds to a cut point c_k and $c_k.w$ can be calculated from *v.law*, as described in **bi_search**. If $f_u(v.B) > 0$, then $PR_i.u$ lies in the subtree $T(v)$ and we do the next step. If $f_u(v.B) \le 0$, check the next node in C. If the nodes in C are exhausted, then PR_i does not exist.

5.3 Suppose $T(v)$ contains $PR_i.u$. Use the **subsearch** operation to perform a standard binary search on $T(v)$ to find the elementary interval $[c_{y-1}, c_y)$ where $f_u(c_{y-1}) > 0$ and $f_u(c_y) \le 0$. Thus, let u be c_y. Similarly, use f_d function to find an elementary interval $[c_x, c_{x+1})$ containing $PR_i.l$ and let l be c_x. Note that $\overline{PR_i}$ is nonempty iff $u>l$.

```
bi_search (u: node, v=[g, h]: interval, wsum: int, esum: int, U: int, done: boolean)
/*v=interval of cut points on segment $\overline{ab}$, a above b.*/
/*wsum and esum are accumulated values above u, exclusively.*/
/*let $SA[a]=dw(\Pi_{sa})$.*/
/*let RW=(Weight of the rectangle encountered)*(its Width)*/
begin
  if (g.y≤u.B) and (u.E≤h.y) then
  /*u is a canonical covering node*/
  begin
    if f(SA[a], u, wsum, esum, RW, a)>0 then
    begin
      U=subsearch (u, wsum, esum)
      done=TRUE
    end
  end
  else begin
      wsum=u.w+wsum
      esum=u.e+esum
      /*search right subtree first*/
      if not done and (u.M<h.y) then
        bi_search (rs (u), v, wsum, esum, U, done)
      if not done and (g.y<u.M) then
        bi_search (ls (u), v, wsum, esum, U, done)
    end
end
subsearch(u: node, wsum: int, esum: int): int
begin
  if u is a leaf then return (u.B)
    wsum=wsum+u.w
    esum=esum+u.e
  if f(SA[a], rs(u), wsum, esum, RW, a)>0 then
    return(subsearch(rs(u), wsum, esum))
  else return(subsearch(ls(u), wsum, esum))
end
f(sa: int, u: node, wsum: int, esum: int, RW: int, a: point): int
begin
  e=esum+u.lae/*get c.e for c=u.B*/
  /*let e=(e.a, e.b) with e.a above e.b*/
  dw=min (SA[e.a]+|e.a.y−u.B|, SA[e.b]
        +|u.B−e.b.y|)
  return((sa+|a.y−u.B|)−((wsum+u.law)+(|a.x-
  −e.a.x|+dw))
end
```

The functions, **bi_search**, **subsearch** and **f** shown above are designed to find $PR_i.u$. Modifications can be made to find $PR_i.l$.

4.3 The SRP Algorithm in $\Theta(n \log n)$

[Algorithm SRP] We obtain the optimal algorithm SRP by modifying algirithm **SRP0** in steps 4 and 5.1.1 as follows:

4. Create the extended weighted segment tree T for CC in L.

5.1.1 According to Lemma 8, to find $\overline{PR_i}$, we use operation **bi_search** to find intervals $[c_x, c_{x+1})$ and $(c_{y-1}, c_y]$ on L that contain $PR_i.l$ and $PR_i.u$, respectively. Let $u=c_y$ and $l=c_x$, PR_i exists if and only if $u>l$. $\overline{PR_i}=(l, u)$.

Theorem 2. The algorithm **SRP** runs in $\Theta(n \log n)$ time and $\Theta(n)$ space, which are both optimal.

[proof]: Since step 5.1.1 can be done in $O(\log n)$ time, the time complexity of the algorithm follows. The space requirement of the algorithm is obviously $O(n)$, since each node in the extended weighted segment tree contains $O(1)$ attributes and there are $O(n)$ nodes. The optimality of the algorithm follows from the proof given by Clarkson *et al.* [1]. ◇

Theorem 3. The problem of the Shortest Rectilinear Path among Weighted Rectangles can be solved optimally in $\Theta(n \log n)$ time and $\Theta(n)$ space, where n is the number of rectangular obstacles.

5. Conclusion

In this paper, we first introduced the weighted segment tree structure, which enabled us to find the shortest rectilinear path among weighted rectangles in $O(n \log^2 n)$ time. Then, by maintaining more information on the weighted segment tree, we were able to solve the problem optimally in $O(n \log n)$ time and $O(n)$ space. The problems with weighted rectilinear obstacles or, weighted simple polygonal obstacles are interesting and worth further research.

References

1. CLARKSON, K. L., KAPOOR, S. and VAIDYA, P. M. Rectilinear shortest paths through polygonal obstacles in $O(n \log^2 n)$ time, *Proc. 3rd ACM Symposium on Computational Geometry*, Waterloo, Ontario (June 1987), 251-257.
2. DEREZENDE, P. J., LEE, D. T. and WU, Y. F. Rectilinear shortest paths with rectangular barriers, *Discrete and Computational Geometry*, **4** (1989), 41-53.
3. GUIBAS, L. J. and HERSHBERGER, J. Optimal shortest path queries in a simple polygon, *Proc. 3rd ACM Symposium on Computational Geometry*, Waterloo, Ontario (June 1987), 50-63.
4. LARSON, R. C. and LI, V. O. Finding minimum rectilinear distance paths in the presence of barriers, *networks*, **11**, (1981), 285-304.
5. LEE, D. T. Proximity and reachability in the plane, PhD Dissertation, University of Illinois, 1978.
6. LEE, D. T. and PREPARATA, F. P. Euclidean shortest paths in the presence of rectilinear barriers, *Networks*, **14** (1984), 393-410.
7. LOZANO-PEREZ, T. and WESLEY, M. A. An algorithm for planning collision-free paths among polyhedral obstacles, *Comm. ACM*, **22** (1979), 560-570.
8. MITCHELL, J. and PAPADIMITRIOU, C. The weighted region problem: Finding shortest paths through a weighted planar subdivision, *J. ACM*, to appear.
9. MITCHELL, J. Shortest rectilinear paths among obstacles *Technical report NO. 739, School of Operations Research and Industrial Engineering, Cornell University*, April 1987.
10. PREPARATA, F. P. and SHAMOS, M. I. Computational Geometry, *Springer-Verlag*, NY (1985), reading.

(Received February 13, 1990; revised June 25, 1990)