An On-Line Algorithm for Improving Performance in Navigation

Avrim Blum[*]
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
avrim@theory.cs.cmu.edu

Prasad Chalasani
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
chal@cs.cmu.edu

Abstract

We consider the following scenario. A point robot is placed at some start location s in a 2-dimensional scene containing oriented rectangular obstacles. The robot must repeatedly travel back and forth between s and a second location t in the scene. The robot knows the coordinates of s and t but initially knows nothing about the positions or sizes of the obstacles. It can only determine the obstacles' locations by bumping into them. We would like an intelligent strategy for the robot so that its trips between s and t both are relatively fast initially, and improve as more trips are taken and more information is gathered.

In this paper we describe an algorithm for this problem with the following guarantee: in the first $k \leq n$ trips, the average distance per trip is at most $O(\sqrt{n/k})$ times the length of the shortest s-t path in the scene, where n is the Euclidean distance between s and t. We also show a matching lower bound for deterministic strategies. These results generalize known bounds on the one-trip problem. Our algorithm is based on a novel method for making an optimal tradeoff between search effort and the goodness of the path found. We improve this algorithm to a "smooth" variant having the property that for *every* $i \leq n$, the robot's ith trip length is $O(\sqrt{n/i})$ times the shortest s-t path length.

A key idea of this paper is a method for analyzing obstacle scenes using a tree structure that can be defined based on the positions of the obstacles.

1 Introduction

This paper addresses an abstraction of the following type of scenario. Imagine you have just moved to a new city. You are at your home and must travel to your office, but you do not have a map (let's assume you know the coordinates of your office; you just do not know the street layout). Several papers in recent literature have discussed strategies that can be used to plan one's route in this type of situation so that the distance traveled is not too much longer than the shortest path. But now, suppose you have reached your office, spent the day there, and it is time to go home. You could retrace your path, but you now have some information about the city (what you saw on your way to work in the morning) and would like to do better. The next morning you have even more information and so on. What is a strategy that allows your path taken each time to be good, and to improve with experience? Perhaps you might even design your paths explicitly so as to gain more information for future trips.

Specifically, we consider the scenario (examined in [17, 7, 11, 10]) where there is a start point s and target t in a 2-dimensional plane filled with non-overlapping, axis-parallel rectangular obstacles,

[*]This material is based upon work supported under NSF National Young Investigator grant CCR-9357793 and a Sloan Foundation Research Fellowship.

having corners at integral coordinates. A point robot begins at s, and knows its current position and that of the target, but it does *not* know the positions and extents of the obstacles; it only finds out of their existence as it bumps into them. In the problem considered in previous papers, the robot's goal is to travel from s to t as quickly as possible. We call this the *one-trip* problem. For this problem, if n is the Euclidean s-t distance, [7] presents an algorithm that guarantees an $O(\sqrt{n})$ ratio of the distance traveled to the shortest path length, which is known to be optimal for deterministic algorithms [17]. Here, we consider the situation where the robot may be asked to make *multiple* trips between s and t. We would like an intelligent strategy for the robot so that its trips between s and t both are as fast as can be hoped for initially, and improve as more trips are made and more information is gathered. For instance, after making one trip that achieves the above $O(\sqrt{n})$ ratio the robot has some partial information about the scene. Can it exploit this information to improve its ratio on the second trip? Can it continue to exploit new information gained on future trips? It is important to note that partial information may not help if it is somehow not sufficiently relevant. Thus the challenge is to perform as well as possible on each trip given the information gained so far, and at the same time acquire information that will be useful for improving on later trips. This makes the multi-trip problem more difficult than the single trip problem.

The multi-trip problem has aspects of both a machine learning and an on-line algorithms problem. As in machine learning settings, we would like our algorithm to improve its performance with experience. As in standard on-line algorithms settings (e.g., [16]), decisions the robot makes now may affect the costs it experiences in the future. However, our scenario also exhibits key differences. In particular, unlike typical on-line algorithms problems, here the algorithm may have *partial information* about the future — namely, the positions of obstacles that lie ahead that it has already encountered. There is also a value associated with information gathering in our setting in the sense that such information may (or may not) prove to be useful on the future trips made. One contribution of our work is a method for analyzing problems of this sort and quantifying the information that is most relevant in this setting.

We study the case of oriented rectangular obstacles for two main reasons. First, scenes containing such obstacles are complex enough to embody many of the strategic issues that arise in path planning. For example, the question of when one should "give up" on a difficult region in the scene and move to a new region that might be more promising; also the question of which information is worth gathering. Second, if one allows arbitrarily shaped obstacles, it is known [7] that one cannot perform much better in the worst case than a simpleminded depth-first-search strategy. Thus, such scenes do not allow one to demonstrate theoretically the value of a useful approach by the performance guarantees achieved.

An extended abstract of this paper appears as [6].

1.1 Results and goodness measures

Given the basic scenario described above, the first question to be addressed is the measure of success to use. Clearly we do not want to give high marks to a solution in which the robot makes an artificially long first trip and then subsequently "improves" on future trips. Instead we would like an algorithm that performs as well as possible at all times. For this reason, we will analyze our algorithms using a type of "competitive analysis". The idea of competitive analysis is to compare the performance of one's algorithm to the best one could hope to do if there were no missing information (in our case, if a map of the scene was known). For instance, for the one-trip problem, Papadimitriou and Yannakakis [17] showed that for any deterministic algorithm and integer $n > 0$, there exist scenes having Euclidean s-t distance n (the width of the thinnest obstacle is taken as 1 unit), forcing the algorithm to travel $\Omega(\sqrt{n})$ times the length of the shortest path. Subsequently, Blum, Raghavan and Schieber (BRS) [7] showed an algorithm having a performance guarantee that matches this lower bound.

For the multi-trip problem we consider two similar measures of performance. In the *cumulative* measure, we compare the total distance traveled on the first k trips to the length of the optimal path. Our first main result is a deterministic strategy having the property that given $k < n$, it guarantees that the total distance traveled in the first k trips is at most $O(L\sqrt{nk})$, where L is the length of the shortest s-t path. (If $k \geq n$ the distance becomes $O(Lk)$.) We also show that up to constant factors this is the best guarantee achievable by a deterministic strategy. In particular, for any deterministic strategy and any n and $k \leq n$, there exist scenes which force the strategy to travel distance $\Omega(L\sqrt{nk})$ on the first k trips. One problem with the cumulative measure is that it does not force the algorithm to perform as well as possible on *each* trip. For this reason, we also consider a *per-trip* measure in which we separately bound the cost of each trip. Our second main result is an improvement on the cumulative algorithm having the property that for *all* $i < n$, the ith trip of the robot has length at most $O(L\sqrt{n/i})$. This is optimal in the sense that (up to constant factors) it meets the cumulative lower bound simultaneously for all i.

1.2 Main ideas and the basic strategy

The core of our results (and the bulk of the paper) is a method for achieving a smooth *search-quality tradeoff*: smoothly trading off in a single trip the exploration cost with the goodness of the path found. Specifically, we design an algorithm that given any $k \leq n$, searches a distance $O(L\sqrt{nk})$ and finds an s-t path of length at most $O(L\sqrt{n/k})$. In other words, at a cost of only t times the cost of the BRS algorithm ($t = \sqrt{k}$ in our case) we find a path that is a factor of t better than the BRS guarantee. In addition, our method for achieving this tradeoff has the property that it can be performed in a "piecemeal" fashion (somewhat like the piecemeal learning of [5]). In particular, the searching can be performed a little bit at a time on each trip. This latter property is what allows us to turn our cumulative algorithm into one that is more like a learning algorithm, with optimal per-trip performance. As more trips are made, better searches are performed, and cheaper paths are found for the future trips. An example of an exploratory trip achieving the desired tradeoff is given in Fig. 1.

Our main idea for achieving this search-quality tradeoff is a method for analyzing an obstacle scene and determining which pieces of information are the most important. In particular, we show that a *tree* structure can be defined in the scene, where the nodes are portions of certain obstacles and the edges are short paths from a node to its children. This tree can be tailored to the search cost and path quality desired. Our search algorithm is essentially an online strategy to traverse a sequence of trees optimally, and the path found is a concatenation of specific root-to-leaf paths from each tree. Besides its use for achieving a search-quality tradeoff on a single trip, the tree structure enables us to spread the search over several trips: since there is a "short" path from the root to each node, we can suspend our tree-traversal on one trip and resume the exploration on a later trip by moving quickly to the point where we stopped. The tree structure is defined formally in Section 5.

1.3 Related Work

Versions of the multi-trip problem have been addressed in the framework of reinforcement learning. Thrun [18] describes heuristics for path improvement in scenes containing (possibly concave) obstacles and presents empirical results. Koenig and Simmons [12] consider a similar problem on graphs. In other machine learning literature, Chen [9] considers how the computation *time* for path-planning in a *known* scene can be improved by making use of (portions of) solutions to previous path planning problems in the same scene.

Betke, Rivest and Singh [5] consider a related problem of completely exploring an environment, but with the restriction that the robot must return to the start to refuel every d steps for some distance d. They call this *piecemeal learning* and provide algorithms for the case of a bounded region with axis-parallel rectangular obstacles.

Lumelsky and Stepanov [13, 14, 15] describe some very simple algorithms that can be used

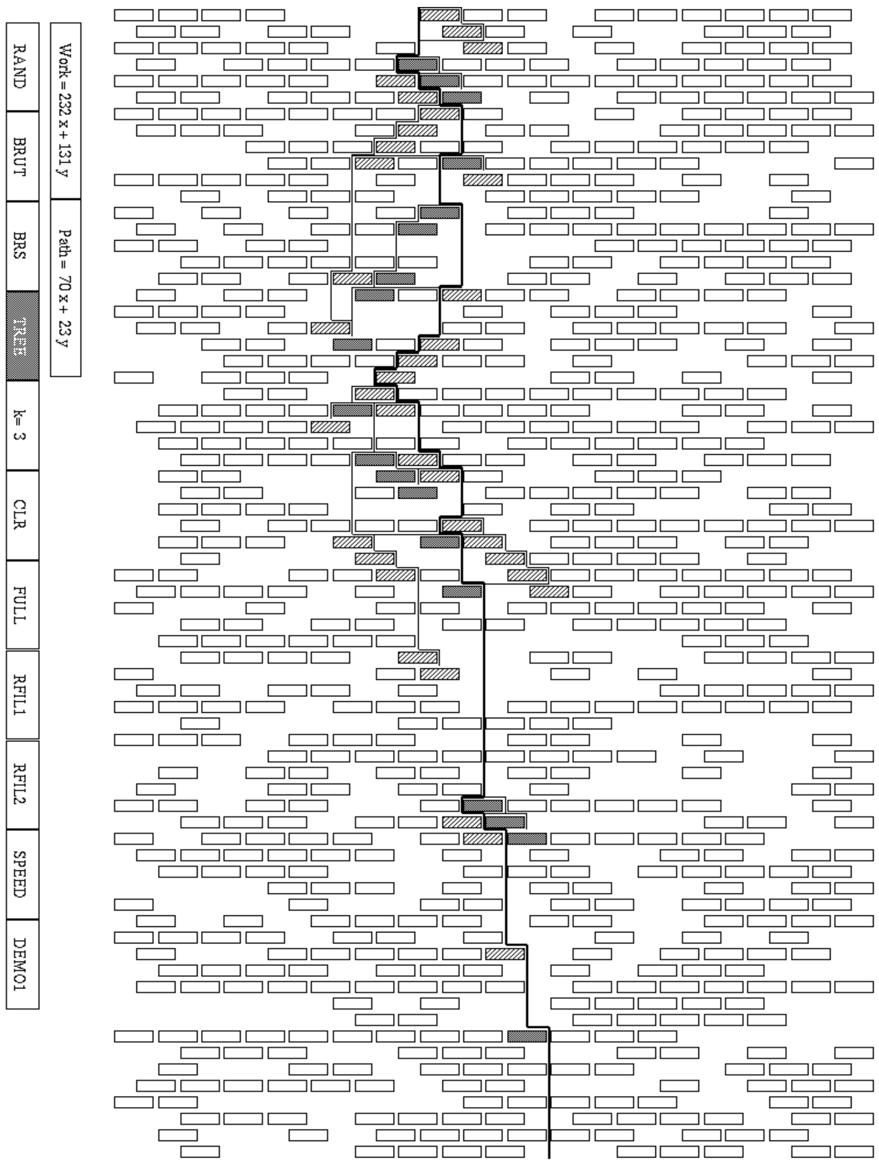

Figure 1: An example of an initial search trip, $k = 3$. The thick line shows the s-t path found (s is at center top, t is at the bottom), and the thin and thick lines are the search path. The robot occasionally will back up, which accounts for the dead ends. Obstacles hit are shaded. In each "fence group" (see Section 5.1) fences 1 and 3 are lightly shaded and fence 2 is darkly shaded. This figure is a screen dump of a demonstration program that allows a user to create a "simple scene" (see Section 4) and then run various algorithms on it.

to solve the one-trip problem for *arbitrary* (non-convex) obstacles, having the property that the distance traveled is at most the Euclidean s-t distance plus 1.5 times the sum of the perimeters of all the obstacles.

2 The model

Let $\mathcal{S}(n)$ denote the class of scenes in which the Euclidean distance between s and t is n. We define s to be at the origin $(0, 0)$. As mentioned above, we assume that the width and height of each obstacle is at least 1 (this in essence defines the units of n) and for simplicity assume that the x-coordinates of the corners of obstacles are integral. Thus no more than n obstacles can be placed side by side between s and t. We assume that when obstacles touch, the point robot can move between them.

To simplify the exposition, for most of this paper we will take t to be the infinite vertical line (a "wall") $x = n$ and require the robot only to get to any point on this line; this is the Wall Problem of [7]. Our algorithms are easily extended to the case where t is a point, using the "Room Problem" algorithms of [7] or [3], and we describe this modification in Section 8.

We model the robot as having only *tactile* sensors; that is, it discovers an obstacle only when it "bumps" into it. It will be convenient to assume, however, that when the robot hits an obstacle, it is told which corner of the obstacle is nearest to it, and how far that corner is from its current position. As in [7], our algorithms can be modified to work without this assumption with only a constant factor penalty. We describe these modifications toward the end of the paper.

Consider a robot strategy R for making k trips between s and t. Let $R_i(S)$ be the distance traveled by the robot in the ith trip, in scene S. Let $L(S)$ be the length of the shortest obstacle-free path in the scene between s and t. We define the *cumulative* k-trip competitive ratio as

$$\rho(R, n, k) = \max_{S \in \mathcal{S}(n)} \frac{R^{(k)}(S)}{kL(S)},$$

where $R^{(k)}(S) = \sum_{i=1}^{k} R_i(S)$ is the *total* distance traveled by the robot in k trips. That is, $\rho(R, n, k)$ is the ratio between the robot's average distance traveled in k trips, and L. We define the *per-trip* competitive ratio for the ith trip as

$$\rho_i(R, n) = \max_{S \in \mathcal{S}(n)} \frac{R_i(S)}{L(S)}.$$

Given this notation, our main results can be described as follows. First, we show for any k, n, and deterministic algorithm R, that $\rho(R, n, k) = \Omega(\sqrt{n/k})$. Second, we describe a deterministic algorithm that given $k \leq n$ achieves $\rho(R, n, k) = O(\sqrt{n/k})$. Finally, we show an improvement to that algorithm that achieves $\rho_i(R, n) = O(\sqrt{n/i})$ for *all* $i \leq n$. Notice that the latter algorithm is optimal in that it matches the lower bound simultaneously for all k. I.e., $\frac{1}{k}\sum_{i=1}^{k} \sqrt{n/i} = O(\sqrt{n/k})$. The simplest of these results is the lower bound, which we describe first.

Conventions. We will use the words up, down, left, and right to mean the directions $+y, -y, -x, +x$ respectively. When we say point A is above, below, behind, or ahead of a point B we will mean that A is in the $+y, -y, -x, +x$ direction respectively from B. Finally, vertical (horizontal) motion is parallel to the y (respectively, x) axis. At any point in time, the current coordinates of the robot (which are known to the robot) are denoted by (x, y).

3 A Lower Bound for k Trips

Theorem 1 (k-trip Cumulative Lower Bound) *For $k \leq n$, the ratio $\rho(R, n, k)$ is at least $\Omega(\sqrt{n/k})$, for any deterministic algorithm R.*

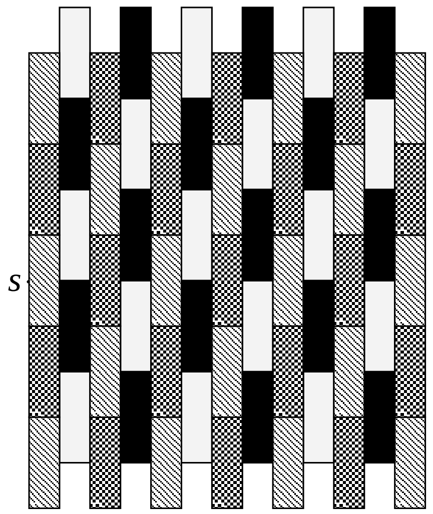

Figure 2: A 4-coloring of the brick pattern for the lower bound.

Proof: Since R is deterministic, an adversary can simulate it and place obstacles in $\mathcal{S}$ as follows. Recall that s is the point $(0, 0)$.

The adversary first places obstacles of fixed height $2h \geq \sqrt{n}$ and width 1, in a full "brick pattern" on the entire plane, as shown in Fig. 2, with s at the center of the left-side of an obstacle. (Recall that the point robot can "squeeze" between bricks). The adversary simulates R on this scene, notes which obstacles it has touched at the end of k trips, then removes all other obstacles from the scene. This is the final scene that the adversary creates for the algorithm, and say it contains M obstacles. The brick pattern ensures that R must have hit at least one brick at every integer x-coordinate, so $M \geq n$. Further, this arrangement forces the robot to hit a brick at every integer x-coordinate on *every* trip. Whenever it hits a brick, it must move vertically up or down a distance h, so its total k-trip distance $R^{(k)}$ is at least nkh.

We now show that there is a path from s to the wall of length at most $O(\sqrt{R^{(k)}h})$. Imagine the full brick pattern to be built out of four kinds of bricks (red, blue, yellow and green, say) arranged in a periodic pattern as shown in the figure. This arrangement has the following property: for each color, to go from a point on an obstacle of that color to a point on any other obstacle of the same color, the robot must move a distance at least h. Out of the M obstacles hit by the robot, at least $M/4$ must have the same color, say blue. So regardless of how the robot moved, since it has visited $M/4$ blue obstacles, we have $R^{(k)} \geq Mh/4$, which implies $M \leq 4R^{(k)}/h$.

We claim there is a non-negative integer $j \leq \sqrt{M}$ such that at most $\sqrt{M}$ obstacles have centers at the y-coordinate jh. This is because a given obstacle intersects at most one y-coordinate of the form jh, and there are M obstacles. Thus, there is a path to t that goes vertically to the y-coordinate jh, then horizontally along this y-coordinate, going around at most $\sqrt{M}$ obstacles. The total length of this path is at most $2h\sqrt{M} + 2h\sqrt{M} + n$, which is at most $6h\sqrt{M}$ since $n \leq M$ and $\sqrt{n} \leq 2h$. Since $M \leq 4R^{(k)}/h$, this path is in fact of length at most $6\sqrt{4hR^{(k)}}$. Thus the k-trip ratio is at least $R^{(k)}/(6k\sqrt{4hR^{(k)}})$. Recalling that $R^{(k)} \geq nkh$, this is at least $\frac{1}{12}\sqrt{n/k} = \Omega(\sqrt{n/k})$. ■

It is not hard to see that this lower bound also holds for the case where t is a point rather than a wall.

4 The k-trip Cumulative Algorithm: Preliminaries

We now give some preliminary observations needed for our algorithm.

We begin by assuming for simplicity that the algorithm knows the length L of the shortest obstacle-free path from s to t. In Section 5.1 we show that this assumption can be removed by using a standard "guessing and doubling" trick. One simple observation is that the shortest obstacle-free s-t path must lie entirely within a *window* of height $2L$ centered at s, since any s-t path that leaves the window must be longer than L. *In the remainder of the paper we will refer to the rectangular region of height $2L$ centered vertically at s, and extending horizontally between s and t as "the window".* This observation immediately leads to an easy algorithm to achieve a cumulative k-trip ratio of $O(1)$ for $k \geq n$:

> **First trip:** Using a depth-first-search, explore the entire window. This can be done by walking a total distance of $O(Ln)$. Compute the shortest obstacle-free s-t path (of length L).
>
> **Remaining trips:** Use the shortest path.

Clearly the average trip length is $O(L)$, so the cumulative n-trip ratio is $O(1)$.

Thus, the cases $k = 1$ (the BRS algorithm) and $k \geq n$ can be done with known methods. In fact, at the high level, our optimal cumulative strategy for $1 \leq k \leq n$ trips is similar to the n-trip algorithm just described:

> **First trip:** Somehow perform an "exploratory" walk of length $O(L\sqrt{nk})$, in such a way that an s–t path P of length $O(L\sqrt{n/k})$ is discovered.
> **Remaining $k-1$ trips:** Use the path P.

The average trip length of this algorithm is $O(\frac{1}{k}(L\sqrt{nk} + (k-1)L\sqrt{n/k})) = O(L\sqrt{n/k})$, so the cumulative k-trip ratio is $O(\sqrt{n/k})$. Thus, as mentioned in the introduction, the key question is how to find a path that is a factor $\Omega(\sqrt{k})$ better than the BRS guarantee while traveling a distance that is only $O(\sqrt{k})$ longer.

In order to make the main ideas clear, we first describe our algorithm for a class of scenes we call *simple scenes* that capture most of the difficulties in designing online navigation algorithms (for both the one-trip and k-trip problems). In Section 6 we show how to extend this algorithm to handle the general case. A scene is *simple* if (a) all obstacles have the same height $2h$ and width 1, and (b) the obstacle corners have coordinates of the form (i, jh) for integer i and j. For instance, the obstacles in the lower bound of Section 3 form a simple scene. Observe that in a simple scene one can move unimpeded vertically along any integer x-coordinate without encountering any obstacles.

Notice that if $h \leq L/\sqrt{nk}$ then a brute-force strategy that moves forward when possible and otherwise arbitrarily goes around any obstacle encountered will hit at most n obstacles and therefore travel a distance at most $O(nL/\sqrt{nk}) = O(L\sqrt{n/k})$, which is our desired bound. Thus, we may assume in what follows that $h > L/\sqrt{nk}$.

Conventions. For convenience, in a simple scene we define the *position* of an obstacle A to be the coordinates of the midpoint of the left edge of the obstacle. A horizontal path whose y coordinate is a multiple of h is said to *hit* an obstacle if it hits the obstacle's center (as opposed to grazing the top or bottom edge), i.e., if it reaches the obstacle's position. We reiterate that we will use the phrase "the window" to refer to the rectangular region of height $2L$ centered vertically at s, and extending horizontally between s and t. As mentioned above, we will assume in what follows that $h > L/\sqrt{nk}$.

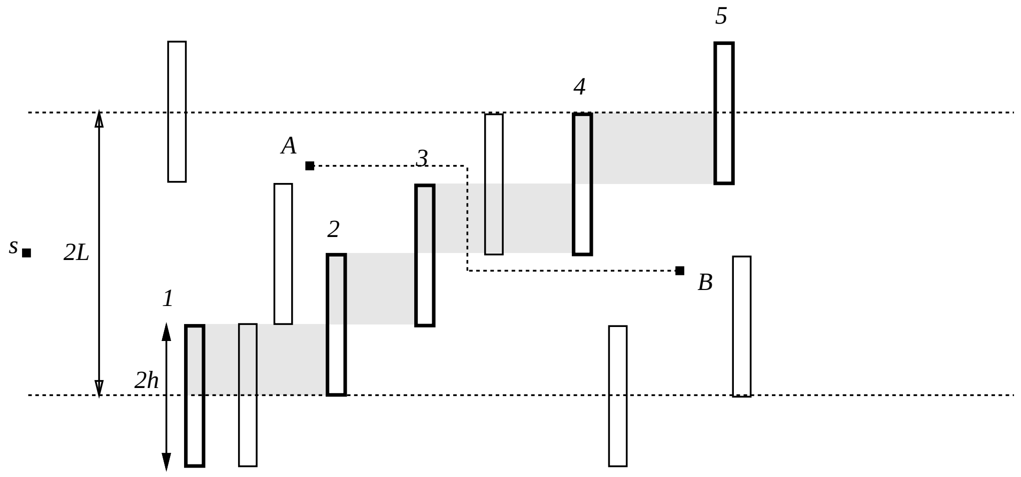

Figure 3: A fence in a simple scene. The obstacles $\langle 1,2,3,4,5\rangle$ with thick boundaries form a fence. The dashed line connecting points A and B crosses the fence. The shaded regions are the bands of the fence. Note: since $L \geq n$, the window in this figure should really be taller than its width. However for the sake of clarity, in this and all figures in this paper the vertical dimension has been compressed considerably.

5 The Algorithm for Simple Scenes

5.1 Fences and the One-trip BRS Algorithm

One key notion used in both the one-trip BRS algorithm and our k-trip algorithm is that of a *fence*.[1] An *up fence* F is a sequence of M obstacles at points $(X(1), Y(1)), (X(2), Y(2)), \ldots, (X(M), Y(M))$ such that $Y(1) \leq -L$, $Y(M) \geq L$, and for $m = 1, 2, \ldots, M-1$:

$$\begin{aligned} X(m) &\leq X(m+1) \qquad (1)\\ Y(m+1) &= Y(m) + h \qquad (2)\end{aligned}$$

See Fig. 3. A *down fence* has the same definition except $Y(1) \geq L$, $Y(M) \leq -L$, and Equation 2 is replaced by $Y(m+1) = Y(m) - h$. The mth obstacle (counting from the left) of fence F_i is denoted by $F_i(m)$ and its coordinates are $(X_i(m), Y_i(m))$. For each m, the rectangular region of height h whose opposite corners are $(X(m), Y(m))$ and $(X(m+1), Y(m+1))$ is called a *band.* A fence can thus be viewed as a contiguous sequence of bands extending across the window.

A point P is said to be *left* (resp. *right*) of a fence F if an imaginary horizontal line from P to the left (resp. right) does not intersect any obstacle of F. A path is said to *cross the fence* if it connects some point left of the fence to some point right of the fence, *and* stays inside the window. Any path that crosses a fence has vertical length at least h since it must completely cross some band (see Fig. 3).

It is easy to see how a robot can find a fence with vertical motion at most $2L$. Specifically, starting from the bottom of the window, an up fence can be found as follows:

> **Repeat until** at top of the window (i.e. $y = +L$): walk to the right until an obstacle is hit, then move up to the top of the obstacle.

The one-trip BRS algorithm restricted to simple scenes (and assuming L is known) reduces to the following:

> Initially, walk from s down to the bottom of the window. Until the wall is reached, walk to the right, alternately building up and down fences across a window of height $2L$ centered at s.

[1] This is called a *sweep* in [7].

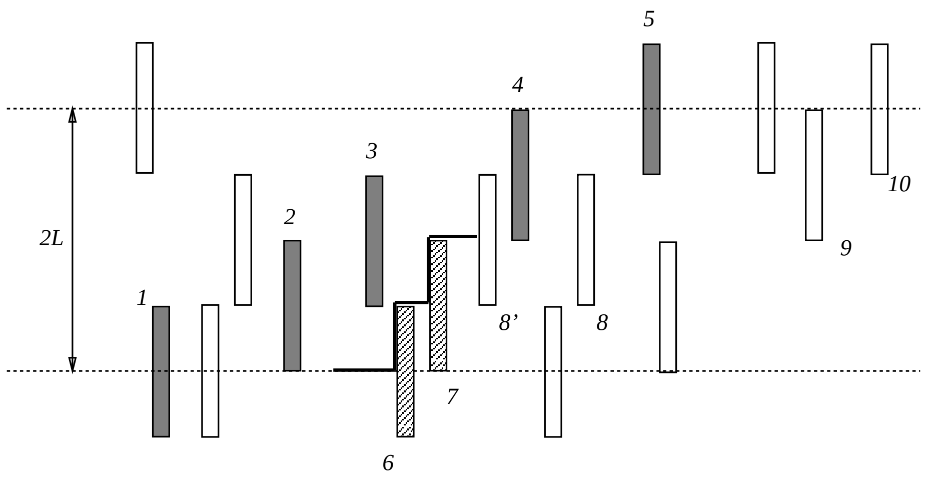

Figure 4: The fences $F_1 = \langle 1, 2, 3, 4, 5 \rangle$ and $F_2 = \langle 6, 7, 8, 9, 10 \rangle$ are disjoint. F_1 and $F_3 = \langle 6, 7, 8', 9, 10 \rangle$ are not disjoint since the band of F_3 between $8'$ and 9 overlaps the band of F_1 between 3 and 4. In fact, one can cross both F_1 and F_3 at a total cost of only h by traveling between $8'$ and 4. A greedy strategy for constructing a fence such as F_2 disjoint from the previously found fence F_1 might be to go up and over obstacle $8'$ until obstacle 4 is hit, and then down around 4 to reach obstacle 8. However, this type of strategy might be expensive, as shown in the next figure.

The robot never walks backward in the BRS algorithm, so its total horizontal cost is n, and since $L \geq n$, this cost is only a small order term. Note that every obstacle hit by the robot is part of some fence. Thus every time the robot spends $2L$ (vertically) to build a fence, it is also forcing the optimal offline path to spend at least h to cross the fence. So if $h \geq L/\sqrt{n}$, the competitive ratio is $O(\sqrt{n})$. The case $h < L/\sqrt{n}$ is even easier to handle: the robot hits at most n obstacles (since they have width 1 and the robot never walks backwards), so its total vertical cost is at most $nh < L\sqrt{n}$.

We say that two fences are *disjoint* if their bands do not intersect each other (see Fig. 4). Because the bands of disjoint fences do not overlap, any path that crosses t disjoint fences must pay (vertically) at least th. For the k-trip problem an intuitively reasonable approach is to extend the BRS idea as follows: On each trip, make new fences that are *disjoint* from previous fences. If on each trip one could find new disjoint fences "cheaply" ($O(L)$ cost), *and* one could cross old fences cheaply ($O(h)$ cost), then this would result in an optimal algorithm. However, we know of no way to find new disjoint fences this cheaply. The naive strategy of extending each new fence greedily and using previously found paths to bypass obstacles that enter existing fences can be too expensive in certain scenes. Examples of such scenes are given in [8].

Our approach, as hinted in Section 4, is to give up on trying to create new disjoint fences on each trip, and instead to try to find a group of disjoint fences all at once on one trip. Specifically, we do the following. Suppose that $h = aL/\sqrt{nk}$ for some $a \geq 1$ (recall that $2h$ is the obstacle height, and $a < 1$ is an easy case to handle). Then our strategy is:

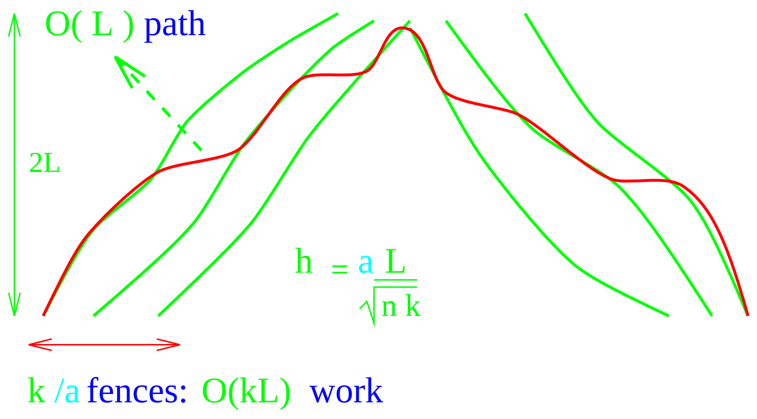

Figure 5: High-level view of the optimal k-trip strategy. First trip: create groups of fences with short group-crossing paths. Remaining trips: follow these short paths.

First trip: Build a sequence of fence groups in which each group consists of $\lceil \frac{k}{a} \rceil$ disjoint fences (alternate between groups of up-fences and groups of down-fences) until the wall is reached. Ensure that

(a) the cost of building each group is $O(kL)$,

(b) an $O(L)$ length path crossing each group (i.e. going from the x-coordinate of the leftmost obstacle of the leftmost fence to the x-coordinate of the rightmost obstacle of the rightmost fence) is found, and

(c) the right end of each group-crossing path is the left end of the next group-crossing path.

Remaining $k-1$ trips: Follow the group-crossing paths to the wall .

The first trip is shown schematically in Fig. 5. To see why the above strategy achieves an $O(\sqrt{n/k})$ ratio, assuming we can somehow satisfy (a), (b) and (c), notice that the *average* online cost per trip to get past each fence group is $O(L)$ (amortizing the $O(kL)$ building cost). Crossing each group costs the optimal offline path at least $(k/a)h = L/\sqrt{n/k}$, so the average online trip length is within an $O(\sqrt{n/k})$ factor of optimal, as desired.

A Search-Quality Tradeoff. Since each fence group costs the offline optimum path at least $L/\sqrt{n/k}$ to cross, the robot will find at most $\sqrt{n/k}$ groups before reaching the wall. Thus the total length of its first trip is $O(kL\sqrt{n/k}) = O(L\sqrt{nk})$, and the total length of the group-crossing paths is $O(L\sqrt{n/k})$. Therefore, this achieves the search-quality tradeoff mentioned earlier.

The Doubling Strategy when L is unknown. Note that if L is not known, we can just begin with a guess of $L = n$, and if the wall has not been reached after building $\sqrt{n/k}$ fence groups, we can double our guess and repeat the entire procedure. Thus there is only a constant factor penalty for not knowing L.

The advantage of building an entire collection of fences on one trip is that this allows the robot to make more effective use of its movements. In [8] a detailed example is given where building fences one-by-one on consecutive trips can be too expensive. In fact a crucial property of the fence collections we will define is that the ith obstacle of a fence is always easily reachable either from the preceding obstacle on the same fence or the corresponding obstacle of the fence above.

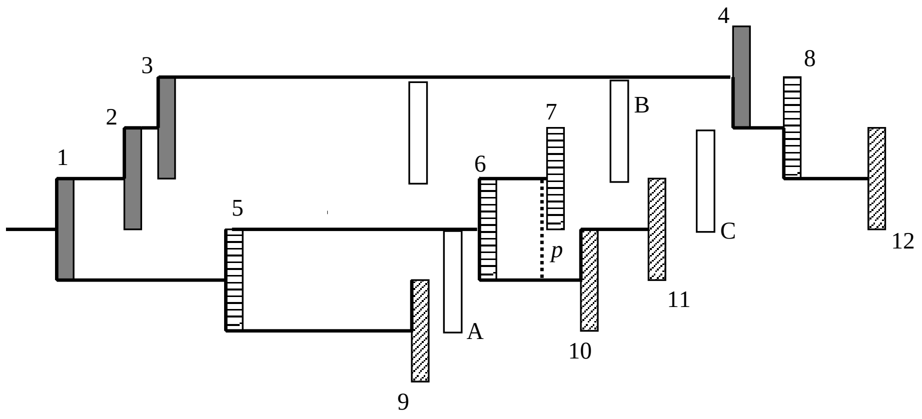

Figure 6: A scene showing a 3×4 fence-tree, where the root $F_1(1)$ is obstacle 1. The shaded obstacles are the nodes of the tree, and the dark lines are the edges. Differently-shaded obstacles constitute different fences; other obstacles are not part of any fence. For instance, obstacle 4 is node $F_1(4)$, obstacle 7 is node $F_2(3)$, and obstacle 8 is node $F_2(4)$. Note that the tree defines 3 disjoint fences with 4 obstacles each. If we had failed to follow the Fence-Tree Rules and instead had made $F_2(4)$ be up-right from $F_2(3)$ (that is, obstacle B) then fences 1 and 2 would not be disjoint.

5.2 Fence-trees

We would like to build a collection of $G = \lceil \frac{k}{a} \rceil$ up-fences across the window, and find an $O(L)$ length path crossing the collection, while paying a cost of only $O(kL)$. Our key idea is to define a *tree* structure whose nodes are obstacles in the scene, and whose edges are "short" paths between the nodes. These nodes will constitute the desired collection of fences, and the path from the root node to the rightmost node is the desired $O(L)$ length path that crosses the collection. Furthermore, traversing all edges of this tree with a cost of $O(kL)$ is equivalent to building the desired collection of fences.

In order to define the tree structure we introduce some notation and terminology. The nodes of this tree will be denoted by $F_i(m)$, for $i = 1, 2, \ldots, G$ and $m = 1, 2, \ldots, M$ (M is roughly $2L/h$ and will be fully specified later). The reason for this notation is that $F_i(m)$ will turn out to be the mth obstacle of the ith fence. The coordinates of obstacle $F_i(m)$ are denoted by $(X_i(m), Y_i(m))$. We say an obstacle Q is *down-right* (*up-right*) from an obstacle P if Q is the first obstacle hit when moving to the right from the *bottom* (*top*) of P. The following rules then define the $G \times M$ fence-tree with root $F_1(1)$ (an example is given in Fig. 6).

Fence-Tree Rules.

1. For $i = 2, 3, \ldots, G$, $F_i(1)$ is down-right from $F_{i-1}(1)$.
2. For $m = 2, 3, \ldots, M$, $F_1(m)$ is up-right from $F_1(m-1)$.
3. For $i = 2, 3, \ldots, G$:

```
    For $m = 2, 3, \ldots, M$:
        If $X_i(m-1) \geq X_{i-1}(m)$,
            then $F_i(m)$ is up-right from $F_i(m-1)$
            else $F_i(m)$ is down-right from $F_{i-1}(m)$.
```

(E.g., in Fig. 6, obstacle 7 is up-right from obstacle 6, and obstacle 8 is down-right from obstacle 4.)

Thus each node $F_i(m)$ (except the root node $F_1(1)$) is defined to be either up-right from

$F_i(m-1)$ or down-right from $F_{i-1}(m)$. We will call that "defining obstacle" of $F_i(m)$ its *parent*, and call the up-right or down-right path from the parent to $F_i(m)$ (consisting of a vertical portion of height h followed by a horizontal portion) an *edge* in the tree. That is, if rule 2 is used or if $F_i(m-1)$ is to the right of $F_{i-1}(m)$ then $F_i(m-1)$ is the parent of $F_i(m)$ and otherwise $F_{i-1}(m)$ is the parent of $F_i(m)$.

The fence tree rules define a natural *binary rooted tree* structure. The tree structure is visually apparent in Fig. 6. The $G \times M$ fence-tree in fact defines exactly the group of fences that we want to build:

Theorem 2 (Fence-Tree) *Let P be an obstacle with y coordinate $-L$ in a simple scene, and let $M \geq \lceil \frac{2L}{h} \rceil + \lceil \frac{k}{a} \rceil$. Then, the obstacles in a $G \times M$ fence tree with root P form G disjoint up-fences with M obstacles each, with P as the first obstacle of the leftmost fence.*

Proof: It is easy to see that the obstacles $F_1(m)$ defined by Rule 2 constitute a fence F_1. In addition, for all $i > 1$, the obstacles $F_i(m)$, $m = 1, 2, \ldots, M$ constitute a fence F_i since each is to the right of and exactly h higher than the previous obstacle. By Rule 1, the initial obstacle of each fence is down-right from the initial obstacle of the fence F_{i-1} above it. Therefore, the fences are disjoint if and only if for each $m = 2, 3, \ldots, M$, $F_i(m)$ is to the right of $F_{i-1}(m)$, and this is guaranteed by Rule 3.

It is easy to verify that the value $M = \lceil \frac{2L}{h} \rceil + \lceil \frac{k}{a} \rceil$ is sufficient to ensure that even the fence F_G that starts $Gh = h\lceil \frac{k}{a} \rceil$ below the bottom of the window extends at least $2L$ above the obstacle P, and thus all fences cross the window. ■

Recall that we wanted to define the fence group so that there is a cheap ($O(L)$ length) path that crosses the group. One such path is the path from the root of the tree to the rightmost obstacle on the rightmost fence (strictly speaking, this path does not cross the rightmost fence, but it can be cheaply extended to one that does). In fact, the unique path in the tree from the root $F_1(1)$ to each node has length at most $O(L)$, as we show below.

Lemma 3 *In a $G \times M$ fence-tree in a simple scene, where $G = \lceil \frac{k}{a} \rceil$ and $M = \lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil$,*
(a) the unique path in the tree from the root to each node has length $O(L)$, and
(b) the total length of all edges is $O(kL)$.

Proof: Recall first that $k \leq n$ and $L \geq n$. Since the unique tree path from the root to any given node always proceeds to the right, the total horizontal cost of this path is at most n. On this path, each down-edge leads to a lower fence (and there are only $G = \lceil \frac{k}{a} \rceil$ fences), and each up-edge leads to an obstacle on the same fence (and there are only $M = \lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil$ obstacles per fence). So, the total vertical cost of this path is at most $h(G + M)$. Thus the total length of any such path is at most $n + h(G + M) = n + h(2\lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil)$, which is $O(L)$ since $h = aL/\sqrt{nk} \leq aL/k$.

To bound the total length of all edges, note that each edge can be associated with a unique node, namely the one on its right (the child). Thus, the sum of the *vertical* portions of all edges is at most $h(GM - 1) = O(kL)$. By fence rules 2 and 3, the length of the horizontal portion of the edge associated with $F_i(m)$ is no more than the horizontal distance between $F_i(m)$ and its predecessor $F_i(m-1)$ on the same fence. Thus the sum of the *horizontal* portions of the edges is a most Gn: n for each fence. This is also $O(kL)$. ■

Thus if the robot traverses all edges of this tree it will have found not only a group of disjoint fences but also a cheap path that crosses all of them.

As noted in the proof above, there are GM obstacles in the fence-tree, and $GMh \leq kMh = O(kL)$. Thus we would like the robot to traverse the fence-tree with a cost proportional to h times the number of obstacles in the fences, or a cost proportional to the total length of the tree edges. We remark here that the fence-tree must be traversed online; a simple approach based on depth-first-traversal may not be efficient since the algorithm does not know where exactly the nodes are: the robot can locate $F_i(m)$ only after it has located both its potential parents $F_{i-1}(m)$ and $F_i(m-1)$ or at least after it has determined whether or not $X_i(m-1) \geq X_{i-1}(m)$. So, intuitively the difficulty is that before visiting a node such as $F_i(m)$ we need to visit both potential parent nodes, which may be in very different parts of the tree. (Actually, one could imagine an algorithm that attempted to visit nodes before finding both parents, and only later verified whether or not those nodes were actually legally part of the tree; our algorithm does not do this.)

Conventions. In the subsequent sections, we will find it convenient to associate an edge in the tree with the obstacle at its right end, and we define the coordinates of an edge to be the coordinates of its associated obstacle. We say an edge belongs to a fence F_i if its associated obstacle belongs to F_i. When we say an object A, such as an edge or obstacle, is left (right) of another object B we will mean that the x-coordinate of A is strictly smaller (greater) than that of B. We will often identify a fence with its rightmost obstacle; thus when we say "fence F_i is to the left of obstacle P" we mean that the last obstacle of F_i is left of P. We use (X_i, Y_i) to denote the coordinates of the rightmost obstacle of F_i, and $|F_i|$ will denote the number of obstacles currently in F_i. To simplify the wording of our algorithms we will assume that $X_0 = \infty$, and $|F_0| = M + 1$.

5.3 Finding the Fence-tree

Our algorithm builds a tree using a *conservative* strategy in the following sense. It adds a new edge to the current partial tree only when such an edge is certain to be part of the final tree being built. In addition, the algorithm visits a node $F_i(m)$, $i > 1, m > 1$, only after both its possible parents $F_{i-1}(m)$ and $F_i(m-1)$ have been visited. At any stage our algorithm will be located at the rightmost node found so far for some partial fence, and then either adds a (down-right or up-right) edge from the current fence, or "jumps" to another fence.

It is reasonable to wonder whether an efficient fence-tree-traversal strategy exists that only walks along the tree edges when jumping from one fence to another. We do not know of any such strategy; our algorithm may often shortcut to another fence without necessarily walking along tree edges. Even with this freedom to walk outside the tree, it is important to note that a bad order of visiting the nodes of the fence-tree may make the jumps prohibitively expensive.

The key problems in designing a traversal strategy therefore are (a) deciding the order in which the nodes will be discovered, and (b) designing the jump procedures. Procedure FindFenceTree in Figure 7 finds the desired fence-tree, using a recursive procedure Raise described in Fig. 8. In these procedures, it should be understood that if the "wall" $x = n$ is reached at any time, the robot halts and the procedures terminate. The procedure JumpDownLeft (i) and JumpDownRight (i) take the robot from the last obstacle of the current fence F_i to the last obstacle of the next lower fence F_{i+1}. These procedures are described in Figs 9 and 10. In all the procedures, the "retrace to $F_i(j)$" statements are executed by simply retracing the path used to reach the current position from $F_i(j)$.

We start with an intuitive description of the algorithm. The algorithm begins (FindFenceTree) by finding the first obstacle in each fence and placing itself at the first obstacle of the top-most one. It then calls the recursive procedure Raise$(1, 0)$. In general for $q < i$, the job of Raise (i, q) is to raise all fences i and lower that are currently behind F_q, as far as possible given the constraints imposed by the location of F_q. (For $i = 1, q = 0$, this means to raise all the fences until they each have M obstacles).

The Raise procedure is a bit complicated, so is perhaps best described through the example of Fig. 6. In this example, Raise $(1, 0)$ is first called at obstacle 1, and the algorithm knows only about obstacles 1, 5, and 9. Raise begins by adding new obstacles to its current fence (in line 4) so long as these are legal with respect to constraints imposed from above, until it has overshot the fence below it. In the example, these are obstacles 2, 3, and 4. Once it reaches obstacle 4, Raise realizes it is to the right of the fence F_2 below it (formally, the conditions of the inner while loop become satisfied), and will try to make sufficient progress on F_2 (and any others that are behind and below F_1, in this case F_3). Unfortunately, because the current obstacle (number 4) of fence F_1 is too high relative to

```
procedure FindFenceTree
   Move to the right until at an obstacle; this defines $F_1(1)$
   for $i := 2$ to $G$ do
       Add a down-right edge to define$F_i(1)$;
   end
   Retrace to $F_1(1)$;
   Raise $(1, 0)$;
end
```

Figure 7: The main procedure for finding the fence-tree.

obstacle 5 on fence F_2, the algorithm cannot simply add a down-right edge (formally, Down(j) is not satisfied) from obstacle 4 to discover the next obstacle of fence F_2. So, the algorithm runs the JumpDownLeft(1) procedure to reach the last obstacle (number 5) of fence F_2, and calls Raise $(2, 1)$ recursively to raise that fence. This call to Raise begins by finding obstacle 6. At this point the robot is to the right of the fence below it (the condition of the inner while loop is satisfied) and it is just high enough above the last obstacle of F_3 (i.e. Down(j) is satisfied) so it adds the down-right edge to obstacle 10 (line 8). At this point it goes back to obstacle 6 (since Up(i) is still satisfied) and makes the up-right edge to obstacle 7. Now, Up(i) is no longer satisfied because the current fence has ”bumped into” the constraint imposed by the fence above it. So, Raise$(1, 0)$ drops down to line 17, where it calls JumpDownRight (2) to get back to obstacle 10 (using the path indicated by ”p” in the figure) and then recursively calls itself to work on raising that fence. Finally, that recursive call ends with obstacle 11, we pop out of both levels of recursion, and at the very top level we retrace our path all the way to obstacle 4, finally adding down-right edges to obstacles 8 and 12 in the inner while loop.

We now give a formal analysis of the formal algorithm given in Figures 7-10. In order to establish the correctness and bound the cost of these procedures, we need to show that certain pre- and post-conditions hold whenever they are invoked. For ease of reference we use mnemonic names for the various conditions:

Up(i): $i \geq 1$ and an up-right edge is legal from fence F_i, i.e., either $X_i < X_{i-1}$ and $|F_i| < |F_{i-1}| - 1$, or $X_i \geq X_{i-1}$ and $|F_i| < |F_{i-1}|$.

Down(i): $i < G$ and a down-right edge is legal from F_i, i.e., $X_{i+1} < X_i$ and $|F_{i+1}| = |F_i| - 1$.

Ord(i, j): if $i \leq j$, then $X_i \leq \ldots \leq X_j$.

At(i): means that the robot is at the last known obstacle of fence F_i.

Eq(i, j) means “if $i \leq j$ then $|F_i| = \ldots = |F_j|$”.

Unch(i, j) stands for “if $i \leq j$ then the values of $|F_i|$ through $|F_j|$ have not changed since the start of the procedure or while loop under consideration.”

AtNewest: means that the robot is at the newest obstacle found so far.

AlmostOrd: No fence has more than one obstacle to the right of a lower fence. That is, for $i = 1, 2, \ldots, G$, if $|F_i| = m$, then $X_i(m - 1) \leq X_j$ for all $j > i$.

We prepend a condition by NOT to signifiy that the logical negation of the condition holds. For easy reference, in Fig. 13 we define several collections of conditions that will be useful in the correctness proof of the algorithm.

We use the next several lemmas to establish the correctness of the procedure FindFenceTree (Theorem 9).

```
procedure Raise $(i, q)$
   while (Up($i$)) do
         Retrace to $F_i$; $j := i$;
         Add an up-right edge to $F_i$;
         while ($j < G$ and $X_j > X_{j+1}$) do
               Retrace to $F_j$;
               if (Down($j$))
                  Add a down-right edge;    $j := j + 1$;
               else
                    JumpDownLeft $(j)$;
                    Raise $(j + 1, j)$;
               fi
         od
   od
   Let $F_d$ be the current fence;
   if $(d < G$ and $X_{d+1} < X_q)$
     JumpDownRight $(d)$;
     Raise $(d + 1, q)$;
   fi
end
```

Figure 8: Recursive procedure Raise used by FindFenceTree˙

```
procedure JumpDownLeft $(j)$
   Move left along tree edges until $x$-coordinate = $X_{j+1}$;
   Move vertically down to last obstacle of $F_{j+1}$;
end
```

Figure 9: Procedure JumpDownLeft to jump from current fence F_j to the next lower fence F_{j+1} when $X_{j+1} < X_j$

```
procedure JumpDownRight $(d)$
   Move vertically down until on a previously found tree edge;
   Follow tree edges to the right until at last obstacle of $F_{d+1}$;
end
```

Figure 10: Procedure JumpDownLeft to jump from current fence F_d to the next lower fence F_{d+1} when $X_{d+1} \geq X_d$

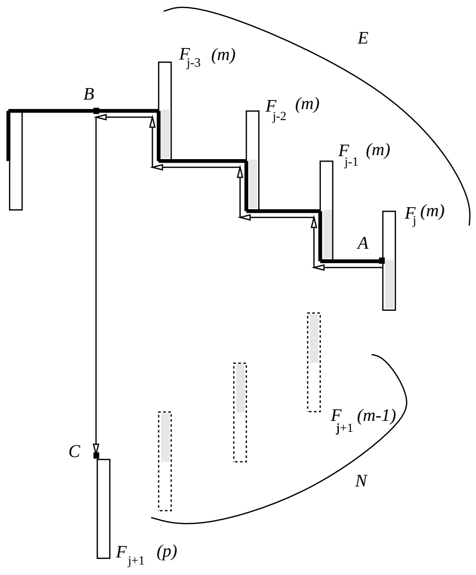

Figure 11: Showing a use of procedure JumpDownLeft to jump from $F_j(m)$ to $F_{j+1}(p)$. Solid-boundary rectangles are obstacles found so far in the tree. The set N of rectangles with dotted boundaries are obstacles that will be found on fence F_{j+1} immediately after this procedure completes. Thick solid lines are tree edges. The thin solid line is the path followed when executing the procedure. The procedure starts from A (obstacle $F_j(m)$), retraces tree edges to the left to point B, then goes vertically down to C, at the top of the final obstacle of F_{i+1}. The set E is the set of edges retraced in AB. The length of BC is no more than the total length of the edges in E plus the heights of the obstacles in N.

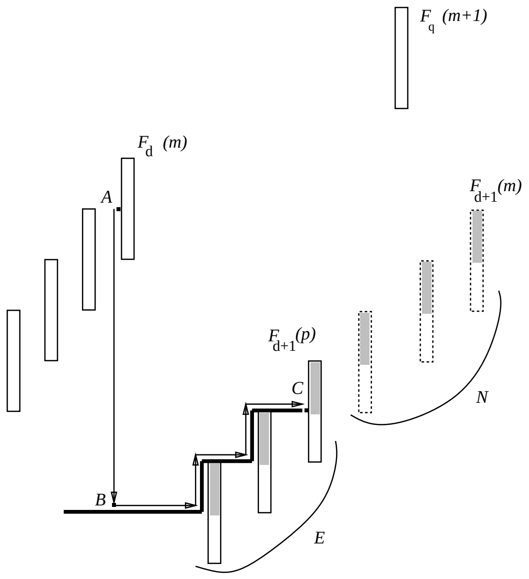

Figure 12: Showing a use of procedure JumpDownRight to jump from $F_d(m)$ to to $F_{d+1}(p)$. Solid-boundary rectangles are obstacles found so far in the tree. The set N of dotted rectangles represents obstacles that will be found on fence F_{d+1} immediately following this procedure. Thick solid lines are tree edges. The thin solid line shows the path followed when executing the procedure. The procedure starts from A (obstacle $F_d(m)$), goes vertically down to a tree edge (point B), then follows the edges to the right to the final obstacle of F_{d+1} (point C). The set E is the set of edges followed in BC. The length of AB is no more than the lengths of the edges in E plus the heights of the obstacles in N.

PreRaise(i, q):

1. Eq($q + 1, i - 1$),
2. At(i),
3. $i > q$,
4. $X_i < X_q$,
5. Ord($q + 1, G$),
6. If $q + 1 < i$, then NOT Up$(q + 1)$,
7. Up(i).

OuterWhile(k) (F_c is the current fence):

1. Unch$(1, i - 1)$,
2. Eq(i, c) and $c \geq i$,
3. Ord(i, G),
4. AtNewest holds after the first (if any) iteration of the loop.

PreJDL(j):

1. At(j),
2. $X_{j+1} < X_j$,
3. Up$(j + 1)$.

PostRaise(i, q) (F_c is the current fence):

1. Unch$(1, i - 1)$,
2. Eq$(q + 1, c)$,
3. Ord$(q + 1, G)$,
4. If $c < G$, $X_{c+1} \geq X_q$,
5. NOT Up$(q + 1)$,
6. AtNewest.

InnerWhile(k) (F_c is the current fence):

1. Unch$(1, i - 1)$,
2. Eq(i, j),
3. Ord$(j + 1, G)$,
4. AtNewest.
5. If $c > j$, then all PostRaise$(j + 1, j)$ conditions hold.

PreJDR(d):

1. At(d),
2. $X_d \leq X_{d+1} < X_q$,
3. Eq$(q + 1, d)$,
4. NOT Up$(q + 1)$,
5. Up$(d + 1)$.

Figure 13: Various conditions required for the formal proof.

Lemma 4 *Whenever* Raise (i, q) *is called, the PreRaise*(i, q) *conditions hold. Moreover, the procedure terminates, and the PostRaise*(i, q) *conditions hold at that time.*

Proof: It is easy to verify that when FindFenceTree makes the first call Raise $(1, 0)$, the PreRaise$(1, 0)$ conditions hold. We claim that if the PreRaise(i, q) conditions hold when Raise (i, q) is called, then Raise (i, q) terminates and the PostRaise(i, q) conditions hold. We prove this by induction. The base case is $i = G$, i.e., an invocation of the form Raise (G, q): in this case there are no recursive calls to Raise, the inner while loop is not entered, and JumpDownLeft and JumpDownRight are not called. The only effect of Raise (G, q) is that up-right edges are added (line 4) to fence F_G until Up(G) is not true. It is easy to check that Raise (G, q) terminates with all the PostRaise(G, q) conditions holding.

Next, let us inductively assume that the claim holds for all calls to Raise $(j, .)$, for $j = i + 1, \ldots, G$. Consider an invocation of Raise (i, q) at some point when the PreRaise(i, q) conditions hold. Before the outer while loop is entered, the current fence is F_c where $c = i$. It is easily verified that the PreRaise(i, q) conditions imply that all the OuterWhile conditions hold at this point. By Lemma 5, when the outer while loop is exited, all the OuterWhile conditions continue to hold, *and* NOT Up(i) holds. At this point, if the “If” condition on line 16 fails, then we claim that all the PostRaise(i, q) conditions hold. Most of these conditions are easy to check, so we will only argue the less trivial ones. To argue that condition (2) Eq$(q + 1, c)$ holds, note that NOT Up(i), combined with Ord$(q + 1, G)$ and $i > q$ (from PreRaise(i, q)) imply Eq$(i - 1, i)$. This, combined with Eq$(q+1, i-1)$ (in PreRaise(i, q)) and Eq(i, c) (in OuterWhile) implies Eq$(q+1, c)$. Condition (5) NOT Up$(q + 1)$ follows from Eq$(q + 1, c)$ (which implies Eq$(q + 1, i)$) and NOT Up(i).

Thus if the condition on line 16 fails, then Raise (i, q) terminates with the PostRaise(i, q) conditions holding, and the lemma is proved. But if the condition on line 16 is true upon exit of the outer while loop, then we claim that the conditions PreJDR(d) hold. Again we will only show the arguments for the non-trivial ones among these: Condition (2) $X_d \le X_{d+1} < X_q$ follows from Ord(i, G) and $c \ge i$ (in OuterWhile), which imply $X_d \le X_{d+1}$ (where $d = c$) and from the truth of the If condition. The reasoning to show that (5) Up$(d + 1)$ holds is as follows. Since AtNewest holds (from OuterWhile), this means that the robot has just found the new obstacle on F_c. Since obstacle number $|F_d|$ on F_{d+1} can only be found after the corresponding obstacle on F_d, this means that $|F_{d+1}| < |F_d|$. From the condition $X_d \le X_{d+1}$ that we just argued, this implies that Up$(d+1)$ must hold.

Now when JumpDownRight (d) is invoked on line 17, by Lemma 8, the robot ends up at (the most-recently discovered obstacle of) F_{d+1}. At this point we claim that the conditions PreRaise$(d+1, q)$ hold. In particular, condition (3) $d + 1 > q$ holds since $d \ge i$ (from OuterWhile) and $i > q$ (from PreRaise(i, q)). (This actually implies that $d > q$, a fact we will use below). Condition (4) Ord$(q + 1, G)$ is one of the PreRaise(i, q) conditions, which we assumed to hold. The fact that $d > q$ implies $q + 1 < d + 1$, and NOT Up$(q + 1)$ was already argued above, so (5) holds. The remaining PreRaise$(d + 1, q)$ conditions are easy to check.

By induction assumption, therefore, procedure Raise $(d + 1, q)$ terminates with the conditions PostRaise$(d + 1, q)$ holding. Of these conditions, all but condition (1) depend only on q and the number c of the current fence at the end of the procedure. So the conditions PostRaise(i, q) (2) through (6) hold. Finally, condition (1) Unch$(1, i - 1)$ holds because it is an OuterWhile condition, and this completes the proof. ■

Lemma 5 *The conditions OuterWhile are invariants for the outer while loop.*

Proof: Suppose all the OuterWhile conditions hold just before entry of the outer while loop. If the outer while loop is entered, an obstacle is added to the current fence $F_c = F_i$ via an up-right edge at line 4, and at this point $j = i$. At this stage it is trivial to verify that the InnerWhile conditions hold. By Lemma 6, upon exit of the inner while loop, all the invariants InnerWhile continue to hold, *and* either $j = G$, or $j < G$ and Ord$(j, j + 1)$ holds. At this point we claim that the OuterWhile conditions hold: Condition (1) Unch$(1, i-1)$ is an InnerWhile condition. We argue condition (2) Eq(i, c) as follows. If $c = j$, then Eq(i, j) (from InnerWhile) implies Eq(i, c). Otherwise, $c > j$, in which case from the InnerWhile conditions, all the PostRaise$(j+1, j)$ conditions hold. In particular, Eq$(j+1, c)$ holds and Up$(j+1)$ is false. Since $c > j$ we must also have $j < G$ and Ord$(j, j+1)$ (the failure of the conditions of the inner while loop), i.e., $X_j \leq X_{j+1}$. Since Up$(j+1)$ is false, this must mean that $|F_j| = |F_{j+1}|$. Thus we have Eq(j, c). This, combined with Eq(i, j) from the InnerWhile conditions, implies Eq(i, c). Condition (3) Ord(i, G) is argued as follows. The InnerWhile conditions Ord$(j+1, G)$ and Eq(i, j), and the condition Ord$(j, j+1)$ that holds because the inner while loop was just exited, imply Ord(i, G). Finally, condition (4) AtNewest follows from the fact that just before returning to the start of the outer while loop, either an up-right edge was added at line 4, or the inner while loop was executed, and AtNewest is one of its invariants. ■

Lemma 6 *The conditions InnerWhile are invariants for the inner while loop.*

Proof: Suppose all the conditions InnerWhile hold at the start of an iteration of the inner while loop. If the loop is entered, then clearly $X_j > X_{j+1}$. If at this point Down(j) holds, then a down-right edge is added, and j is incremented to $j+1$. At this point it is easy to check that all the InnerWhile conditions continue to hold. On the other hand, if Down(j) does not hold, then we claim that the conditions PreJDL(j) hold: (1) At(j) is clearly true. (2) $X_{j+1} < X_j$ holds as we observed above. (3) Up$(j+1)$ holds since $X_{j+1} < X_j$ and NOT Down(j) holds.

By Lemma 7, after JumpDownLeft is executed, the robot is at F_{j+1}. At this point we claim that all the conditions PreRaise$(j+1, j)$ hold. For instance, condition (5) Ord$(j+1, G)$ holds since by assumption it held when the inner while loop was entered (being one of the InnerWhile conditions), and before this invocation of Raise $(j+1, j)$, no new obstacles were discovered on any fence. The remaining Raise $(j+1, j)$ conditions are trivially checked.

By Lemma 4, after Raise is executed, all the conditions PostRaise$(j+1, j)$ will hold. At this point we claim that all the conditions InnerWhile continue to hold: (1) Unch$(1, i-1)$ is maintained since fences F_j and above are unaffected by Raise$(j+1, j)$ (this is the Unch$(1, j)$ condition in PostRaise$(j+1, j)$), and $j \geq i$. (2) Eq(i, j) is maintained for the same reason. Conditions (3) Ord$(j+1, G)$ and (4) AtNewest are also PostRaise$(j+1, j)$ conditions. Finally, we just argued above that the PostRaise$(j+1, j)$ conditions hold, and this is condition (5) of InnerWhile. ■

We establish below that the procedures JumpDownLeft and JumpDownRight work correctly if they are invoked under appropriate conditions.

Lemma 7 *Whenever* JumpDownLeft (j) *is invoked, the conditions PreJDL*(j) *hold, and the procedure terminates with the robot at the last known obstacle of* F_{j+1}.

Proof: That the conditions PreJDL(j) hold whenever JumpDownLeft (j) is invoked can easily be seen from the proofs of Lemmas 4 and 6. The PreJDL(j) condition $X_{j+1} < X_j$ implies that the motion in line 2 (following tree edges to the left) leads to a point where the x-coordinate is X_{j+1} (see Fig. 11). Since edges followed to the left only lead to the same fence or a higher one, this implies that the point at the end of the motion in line 2 is vertically above (and not below) the last known obstacle P of F_{j+1}. Thus moving vertically down in line 3 leads to obstacle P. ■

Lemma 8 *Whenever the procedure* JumpDownRight (d) *is invoked, the conditions PreJDR*(d) *hold, and the procedure terminates with the robot at the last known obstacle of* F_{d+1}.

Proof: That the conditions PreJDR(d) hold whenever JumpDownRight (d) is invoked can easily be seen from the proof of Lemma 4. Since $X_d \leq X_{d+1}$, moving vertically down from F_d will lead to a tree edge that is to the left of F_{d+1}. So following the tree edges to the right will lead to the most-recently discovered obstacle of F_{d+1}. ■

From the above Lemmas it is easy to prove that the procedure FindFenceTree finds the desired fence-tree:

Theorem 9 *When executing procedure* FindFenceTree*, the robot either finds a complete $G \times M$ fence-tree, or reaches the wall (the line $x = n$) after having found a collection of i partial fences $F_1, F_2, \ldots, F_i$ that satisfy the fence-tree rules.*

Proof: Recall that we have set $|F_0|$ to be $M + 1$, and X_0 to be infinite. After the first obstacle on fence $F_1, F_2, \ldots, F_G$ is found in line 4 of FindFenceTree, the robot returns to $F_1(1)$. At this point it is easy to check that the PreRaise$(1, 0$ conditions are satisfied. When Raise $(1, 0)$ is invoked in line 7, by Lemma 4, the robot completes the procedure (if it hasn't reached the wall) with the PostRaise$(1, 0)$ conditions holding. In particular, condition (4) implies that the current fence upon completion of the procedure must be $F_c = F_G$ (since otherwise $X_{c+1} \geq X_0$, which is impossible since $X_0 = \infty$). Also, conditions (2) Eq$(1, c)$ and (5) NOT Up(1) imply that $|F_1| = |F_2| = \ldots = |F_G| = |F_0| - 1 = M$. ■

Finally we establish an invariant that will be useful later.

Lemma 10 *The AlmostOrd invariant holds throughout any execution of* Raise (i, q).

Proof: The only two statements of the procedure which could possibly result in a violation of the AlmostOrd invariant are 4 (where an up-right edge is added) and 8 (where a down-right edge is added). But whenever line 4 is reached, Ord(i, G) holds: this follows from Ord$(q + 1, G)$, which is one of the OuterWhile conditions (which we prove below to be invariants for the outer while loop), and $i > q$, which is a PreRaise(i, q) condition. Thus even if the new obstacle added to F_i in line 4 is to the right of (the last obstacle of) a lower fence, this would be the only such obstacle of F_i. Similarly, whenever a down-right edge is added from fence F_j (thereby adding an obstacle to F_{j+1}), Ord$(j + 1, G)$ holds: this is one of the InnerWhile conditions, which we show below to be invariants for the inner while loop. Thus even if the new obstacle added to F_{j+1} in line 8 is to the right of some lower fence, it would be the only such obstacle of F_{j+1}. ■

5.4 Cost Analysis

Recall from Subsection 5.2 that we would like our fence-tree-finding algorithm to travel a distance of no more than $O(kL)$. The next Theorem establishes this.

Theorem 11 *For $G = \lceil \frac{k}{a} \rceil$ and $M = \lceil \frac{k}{a} \rceil + \lceil \frac{2L}{h} \rceil$, the algorithm* FindFenceTree *for finding a $G \times M$ fence-tree in a simple scene has total cost $O(kL)$.*

Proof: From Lemma 3, it suffices to show that the total distance traveled by the robot while executing FindFenceTree is bounded by some constant times the total length of all edges plus the heights of all obstacles in the fence-tree. There are four kinds of motions performed by the algorithm:

- Adding an up-right edge (line 4 of Raise),
- Adding a down-right edge (line 8 of Raise),
- Retracing an old path (lines 3, 6 of Raise),
- Jumping from a fence to the next lower one, using procedures JumpDownLeft and Jump-DownRight.

Consider a specific iteration of the outer while loop of Raise. If the robot is not at F_i at the start of this iteration, it executes "Retrace to F_i" at line 3. This motion consists simply of retracing the paths it walked while executing the remaining lines of this loop, during the *previous* iteration of the loop. So the retracing motion at line 3 in a given iteration of the outer while loop can be charged off to the non-line-3 motions executed during the previous iteration of the outer loop. Similarly, the retracing motion at line 6 in a given iteration of the inner while loop can be charged off to the non-line-6 motions executed during during the previous iteration of this loop. Thus it suffices to bound the cost of the remaining three kinds of motions. Clearly, the motion required to add up-right and down-right edges can be charged off to the edges created, so we only need to bound the cost of the procedures JumpDownLeft and JumpDownRight. Lemmas 12 and 13 below establish that the total cost of these procedures is $O(kL)$, which implies our theorem. ■

Lemma 12 *The total distance traveled during all invocations of* JumpDownLeft *is* $O(kL)$.

Proof: Consider a call to JumpDownLeft (j) at line 10 of Raise (i, q), and suppose the robot is at obstacle $F_j(m)$ when this procedure is invoked. The edge-following motion in line 2 of this procedure can be charged to the set E of edges followed. The right ends of all these edges are at obstacle number m of different fences. Note that just before JumpDownLeft (j) is invoked, Up$(j + 1)$ is true (this is a PreJDL(j) condition), and when Raise $(j + 1, j)$ is completed after JumpDownLeft (j), Up$(j + 1)$ is *not* true (this is a PostRaise$(j + 1, j)$ condition). This means that the procedure Raise $(j + 1, j)$ has discovered a collection N of *new* obstacles on fence F_{j+1}, so that $|F_{j+1}|$ would be at least $m - 1$. The cost of the vertical motion in line 3 of JumpDownLeft (j) is clearly no more than the total length of the edges E plus the heights of the new obstacles N discovered by the subsequent call to Raise. We can thus charge the total cost of this specific invocation of JumpDownLeft (j) to the set E of edges and the set N of new obstacles. Now we need to argue that the sets E and N of future calls to JumpDownLeft will not overlap with those of the present call. Since the obstacles N are *new* obstacles discovered on F_{j+1} just after the present call to JumpDownLeft (j), the only future calls to JumpDownLeft whose N-sets can possibly overlap with the N set of the present one are calls to JumpDownLeft from the same fence F_j, i.e., calls to JumpDownLeft (j). However, as we observed above, the Raise $(j+1, j)$ executed just after the present JumpDownLeft (j) finds the obstacles on N before any future call to JumpDownLeft (j) is made. Moreover, it is easy to see that the execution of Raise $(j + 1, j)$ does not involve any calls to JumpDownLeft (j). Therefore the N-sets of different calls to JumpDownLeft do not overlap.

Now we show that the E-sets of different calls to JumpDownLeft do not overlap. Again consider a specific invocation of JumpDownLeft (j) from obstacle $F_j(m)$. As we noted above, all the edges in E have at their ends the m'th obstacle of different fences. Therefore the only future invocations of JumpDownLeft whose E sets can possibly overlap with the current E set are those from obstacle m of some fence F_u below F_j. We established before that an invocation of JumpDown-Left (u) is only made when the conditions PreJDL(u) hold, and in particular (a) $X_{u+1} < X_u$ and (b) Up$(u + 1)$ must hold. However, just after the present execution of JumpDownLeft (j), Raise $(j + 1, j)$ is executed, and at that point the PostRaise$(j + 1, j)$ conditions hold. In particular, for any fences F_u below F_j such that $X_u < X_j$, Up(u) does not hold. Therefore when (if at all) a future call to JumpDownLeft (u) is made from obstacle m of a fence F_u below F_j, $X_{u+1} \geq X_j(m)$ must hold at that time. In such a future invocation of, in line 2, edges are followed to the left until the x-coordinate equals X_{u+1}, so these edges would be to the *right* of $X_j(m)$, and so would not overlap with the edges E of the present set (all of which are to the *left* of X_j). Thus the "charge sets" E and N for different calls to JumpDownLeft (j) will not overlap. ■

Lemma 13 *The total distance traveled during all invocations of* JumpDownRight *is* $O(kL)$.

Proof: Consider an invocation of JumpDownRight (d) at line 17 of Raise (i, q). We will present a charging scheme where different invocations of JumpDownRight (d) will be charged to distinct portions of the fence-tree. When JumpDownRight (d) is invoked, Up$(d + 1)$ is true (this is a PreJDR(d) condition). Subsequent to this invocation of JumpDownRight (d), Raise $(d + 1, q)$ is invoked, and when that procedure completes, the PostRaise$(d+1, q)$ conditions imply that Up$(d+1)$ is no longer true. This means that Raise $(d + 1, q)$ has found a collection N of new obstacles on fence F_{d+1}, and then $|F_{d+1}|$ would equal $|F_d| = m$ (this is a PostRaise$(d + 1, q)$ condition). The edge-following motion in line 3 of JumpDownRight (d) can be charged to the set E of edges that are followed. The vertical motion in line 2 of JumpDownRight (d) is clearly no more than the lengths of the edges E plus the heights of the obstacles N, so this motion can be charged to the edge-set E and the obstacle-set N (see Fig. 12).

Note that since Up$(d+1)$ is not true after the Raise procedure completes just after this invocation JumpDownRight (d), the next invocation of JumpDownRight (d) can only occur from obstacle $m+1$ or later of F_d. Since presently $X_d < X_q$, and NOT Up$(q+1)$ and Eq$(q+1,d)$ hold, the obstacle $m+1$ of F_d will be to the right of X_q. So the edge-set E followed by any future invocation of JumpDownRight (d) will be distinct from the set E of the present invocation. Also, since N is the set of *new* obstacles discovered just after the present invocation of JumpDownRight (d), the set N of any future call to JumpDownRight (d) will not overlap with the set N of the present one. In fact, since we are the charging the motion of JumpDownRight (d) only to obstacles and edges associated with fence F_{d+1}, the sets N and E of any future call to JumpDownRight (d') will also not overlap with the sets N and E of the present call. ■

6 Extension to Arbitrary Axis-Parallel Rectangular Obstacles

We now show how to extend the search algorithm to scenes with arbitrary axis-parallel rectangular obstacles (for brevity we call such scenes *general* scenes). That is, we will show how to explore a distance of $O(L\sqrt{nk})$ and find a path of length $O(L\sqrt{n/k})$. Fortunately it turns out that algorithm FindFenceTree, interpreted appropriately, can be used unchanged for these scenes. However the procedures JumpDownRight and JumpDownLeft must be modified since for general scenes vertical motion is not always unobstructed. In fact if all obstacles have width 1 (but arbitrary heights and positions) then even these procedures remain unchanged. In the next two subsections, we define the notions of *τ-post* and a *τ-fence,* which are the analogues of "obstacle" and "fence" for general scenes. As stated earlier, we will assume throughout that all obstacles have their corners at integer coordinates.

6.1 τ-Posts

Throughout this section we will denote the value $L/\sqrt{nk}$ by τ. We assume $k \leq n$ so $\tau \geq 1$. Recall that in a simple scene if the obstacles have height less than $2\tau = 2L/\sqrt{nk}$ then they can be considered "small" – in the sense that the simple strategy of just moving horizontally forward (walking around any obstacles on the way by the shortest route) achieves the optimal ratio of $O(\sqrt{n/k})$ on *each* trip. This motivates the following definition in a general scene. A point P on the left side of an obstacle is called a *τ-post* if the obstacle extends vertically at least τ above and below P. We will use the term τ-post to refer either to the entire segment of height 2τ or just to the center of that segment. Roughly speaking when the robot encounters a τ-post P while moving horizontally, the obstacle encountered is "big", otherwise it is "small".

6.2 τ-Fences

We define a τ-fence as the generalization of the fence defined in Section 5.1. The definitions (and notations) for *up τ-fence, down τ-fence, band*, as well as the definition of a point being *left* or *right* of a fence, and of a path *crossing* a fence, remain the same as in simple scenes, except that we replace the word "obstacle" with "τ-post" throughout, and replace h by τ in the relations (1) and (2) of Section 5.1. Note that consecutive τ-posts of a fence may lie on the *same* obstacle, since the inequality (1) is not strict. The band between two such posts is empty. We say two τ-fences are *disjoint* if their *non-empty* bands are disjoint. Thus a collection of k disjoint τ-fences costs at least $k\tau$ to cross. In Fig. 14, the sequence of τ-posts $\langle F_1^1, F_1^2, F_1^3, F_1^4 \rangle$ form a τ-fence F_1. Note that F_1^2, F_1^3 are on the same obstacle, and that the three fences in the figure are disjoint.

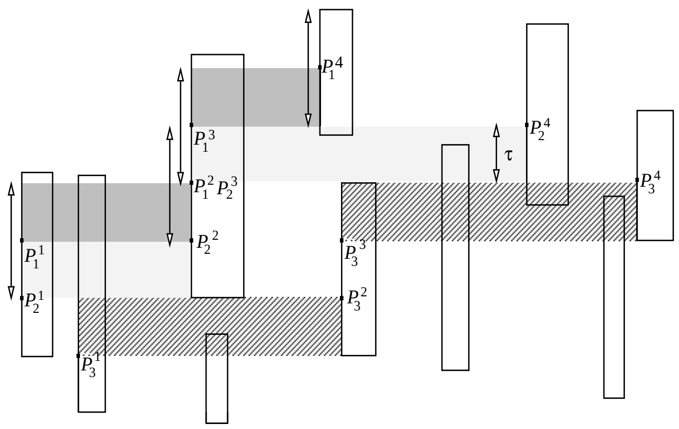

Figure 14: A collection of 3 disjoint fences with 4 posts each. The solid rectangles are the obstacles. The bands of different fences are shaded differently. For convenience, post $F_i(m)$ is denoted P_i^m.

For future reference we define a (right) τ-*path* as the path of the robot when it moves to the right along a fixed horizontal line $y = y_0$ until it hits a τ-post or the wall, moving around any non-post obstacle on its way. For instance in Fig. 15, the path from A to τ-post $F_1(2)$ is a τ-path. Observe that a τ-path has vertical motion at most 2τ at every (integer) x-coordinate on the path, so:

Fact 1 *A τ-path between two points (x, y) and $(x + \delta x, y)$ has length at most $\delta x + 2\tau\ \delta x$.*

6.3 The Initial Search Trip

Roughly speaking, a general scene is treated as if it is a simple scene with obstacles of height $2h = 2\tau = 2L/\sqrt{nk}$. Recall that for simple scenes, the initial trip consists of building groups of G fences of M obstacles each (where $G = \lceil \frac{kT}{h} \rceil$ and $M = \lceil \frac{kT}{h} \rceil + \lceil \frac{2L}{h} \rceil$), where each group must be built "cheaply" (i.e. with cost $O(kL)$) and must have a known "short" (cost $O(L)$) path crossing it. Analogously for general scenes we have $h = \tau$ and we would like to build groups k τ-fences with $M = k + \lceil \frac{2L}{\tau} \rceil$ τ-posts each. We will now pay more attention to our progress in the x direction and will build each group with cost at most $O(kL + k\tau\ \Delta x)$, and give each group a group-crossing path of length $O(L + \tau\ \Delta x)$. Here Δx is the x-distance between the leftmost τ-post $F_1(1)$ and right-most τ-post $F_k(M)$ in the group.

These bounds are sufficient for our purposes for the following reason. Since a fence costs $\tau = L/\sqrt{nk}$ to cross, there can be at most $\sqrt{nk}$ disjoint τ-fences in the window between s and t, so the algorithm will find at most $\sqrt{n/k}$ groups of k fences each. Since the x-motions do not overlap between the groups of fences, the Δx terms add to at most n, so the total distance traveled is $O(kL\sqrt{n/k} + nk\tau) = O(L\sqrt{nk})$. In addition, the concatenation of the group-crossing paths has total cost $O(L\sqrt{n/k} + n\tau) = O(L\sqrt{n/k})$.

6.4 Extending FindFenceTree to General Scenes

Once we fix the τ-post $F_1(1)$ in a scene, the three fence-tree definition rules given for simple scenes can be used in a general scene to define a group of $G = k$ τ-fences with M τ-posts each, with the following interpretation. Firstly, "τ-post" replaces the word "obstacle" everywhere. Secondly, an up-right edge from a τ-post $F_i(m)$ is simply a path that goes up to the top of the τ-post, then right along a τ-path until a τ-post is reached; this τ-post is $F_i(m + 1)$. A down-right edge is a similar path that leads to the τ-post $F_{i+1}(m)$. With this interpretation, the algorithm FindFenceTree can be used unchanged for general scenes; only the jump procedures must be changed to handle arbitrary obstacle widths, since the vertical motions (in line 3 of JumpDownLeft and line 2 of

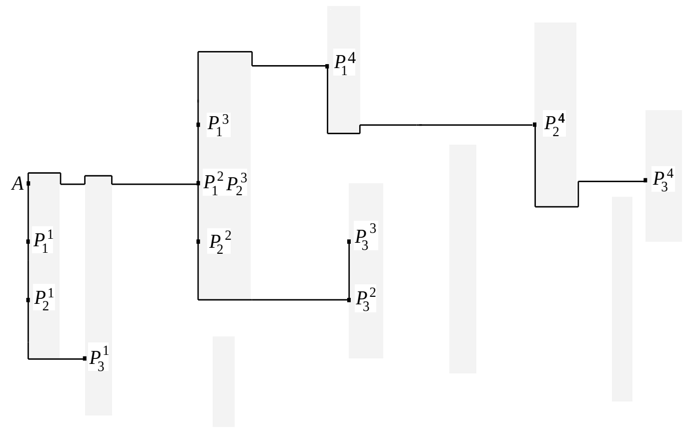

Figure 15: A 3×4 τ-fence-tree. The shaded rectangles are the obstacles, and solid lines are tree edges. The fences corresponding to this tree are shown in Fig. 14. For convenience, post $F_i(m)$ is denoted P_i^m.

JumpDownRight) may no longer be possible. The modified jump algorithms are described in the next two subsections, and we will show that they work correctly, i.e., that the analogues of Lemmas 7 and 8 hold. We will also show that these procedures are not expensive, i.e., that the analogues of Lemmas 12 and 13 hold. Given the correctness of the jump procedures, it is easy to verify that the Raise procedure works correctly, i.e., satisfies Lemma 4, and consequently that the procedure FindFenceTree does indeed find a $k \times M$ τ-fence-tree, if it exists, with the given post $F_1(1)$ as root. Furthermore, the invariant AlmostOrd also holds (Lemma 10throughout the execution of FindFenceTree.

We must show that this algorithm is still "cheap" in a general scene. Note that if the robot does not encounter any "small" obstacles when adding the various edges, then (assuming the jump procedures are cheap) our previous arguments suffice. We begin with the fairly straightforward argument that in general the cost of going around "small" obstacles is not too large. To do this, we show the analogue of Lemma 3, namely, that the total cost of the tree edges and the cost of the path from the root to the rightmost node are both within our required bounds.

Lemma 14 *Suppose there is a $k \times M$ τ-fence-tree consisting of fences $F_1, F_2, \ldots, F_k$, with $M = k + \lceil \frac{2L}{\tau} \rceil$, where $\tau = L/\sqrt{nk}$. Let Δx be the x-distance from $F_1(1)$ to $F_k(M)$. Then:*
(a) The unique path in the tree from $F_1(1)$ to $F_k(M)$ has length at most $4L + 3\tau\Delta x$;
(b) The total length of all the edges in the fence-tree is at most $k(3L + 3\tau\Delta x)$

Proof: Since $k \le n$ it follows that $k\tau = kL/\sqrt{nk} \le L$. This implies $M\tau = 2L + k\tau \le 3L$.

Part (a). There are exactly $(M + k - 2)$ edges in the tree path from $F_1(1)$ to $F_k(M)$. Since the vertical portion of each edge has length τ, and the τ-path portions of the edges do not overlap in the x-direction, the total length of these edges is at most $(M + k - 2)\tau + 2\tau\Delta x + \Delta x$, which is at most $(4L + 3\tau\Delta x)$ from the inequalities above.

Part (b). Note that we can associate each edge with a unique post, namely the one at the right-end of the edge. For any given post $F_i(m)$ other than $F_1(1)$, the x-distance to its parent is at most the x-distance δx to its predecessor $F_i(m-1)$ on the same fence. So the edge associated with this post has length at most $(\tau + 2\tau\delta x + \delta x)$. The sum of the δx terms over all posts of the fence F_i is the x-distance between the first and last posts of F_i, which is at most Δx. So the total length of the edges associated with the M posts of a fence is at most $(M\tau + 2\tau\Delta x + \Delta x)$, which sums to $k(M\tau + 2\tau\Delta x + \Delta x)$ for k fences. This last expression is at most $k(3L + 3\tau\Delta x)$ from the previous inequalities. ■

```
Procedure JumpDownRight $(d)$
    1. Let $m = |F_d|$ and $p = |F_{d+1}|$.
    2. Move greedy down-left until down-left of top of $F_{d+1}(p)$.
    3. Move greedy right-down until:
        • at $\tau$-post $F_{d+1}(p)$, or
        • on a tree edge. In this case, follow tree-edges to the right until at $F_{d+1}(p)$.
```

Figure 16: General procedure for jumping down from F_d to F_{d+1} when $X_d \leq X_{d+1}$.

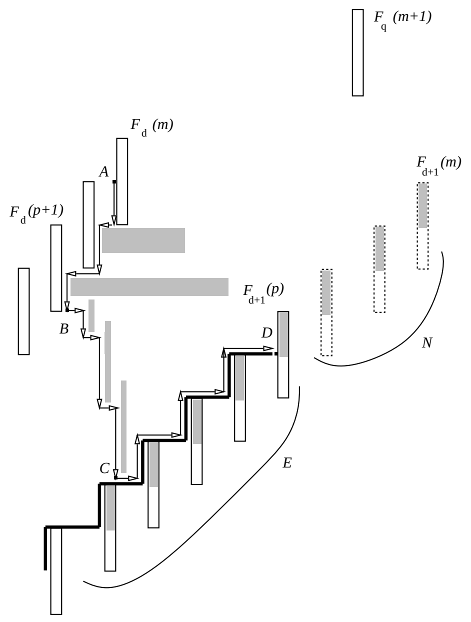

Figure 17: A use of the generalized procedure JumpDownRight to jump from $F_d(m)$ to $F_{d+1}(p)$, for clarity shown in a scene where the fence posts correspond exactly to obstacles of height 2τ. Shaded rectangles with no boundaries are obstacles that are not part of any fence. Solid-boundary rectangles are nodes found so far in the tree. Dotted-boundary rectangles are posts on F_{d+1} that will be found immediately following this procedure. Thick solid lines are tree edges. The thin arrow line shows the path followed when executing the procedure.

Thus, just as in simple scenes, we only need to argue that the total cost of each jump procedure is at most a constant times the total length of the tree edges plus the heights of all τ-posts in the tree. In the next two subsections we show how these procedures can be modified to handle arbitrary obstacle widths, and prove that they are not too expensive. As before, our approach will be to argue that for each jump procedure, no portion of the tree is charged too often for different executions of that procedure.

6.5 Modifying JumpDownRight

To describe the modifications, it will be useful to introduce the notion of a *greedy down-left* path: it is a path that repeatedly goes "*down* till it hits an obstacle, then to the *left* corner of the obstacle". Other greedy paths are defined similarly. Also, a point (x, y) will be said to be *down-left* of another point (x_0, y_0) if $x \leq x_0$ and $y \leq y_0$. As in the case of simple scenes, we will find it convenient to associate an edge in the fence-tree with the post at its right end. Our modified procedure is shown in Fig. 16, and a typical path walked while executing this procedure is shown in Fig. 17.

We first establish the correctness of this modified procedure, i.e., the analogue of Lemma 8.

Lemma 15 *If the general procedure* JumpDownRight (d) *is called when the PreJDR*(d) *conditions hold, then after the procedure is completed, the robot will be at the last known* τ*-post of* F_{d+1}.

Proof: It will be useful to consult Fig. 17 which shows a typical path followed while executing this procedure. From the PreJDR(d) conditions, the robot is intially at (the last known τ-post of) F_d, so the initial y-coordinate is $Y_d(m)$. The PreJDR(d) conditions $X_d \leq X_{d+1}$ and Up$(d+1)$ also imply that $|F_{d+1}| < |F_d|$, or $p < m$, which means that the bottom of the τ-post $F_d(m)$ is no lower than the top of the τ-post $F_{d+1}(p)$. Therefore the greedy down-left path in step 2 will not encounter any tree-edges, since even a down-right leading to the destination τ-post $F_{d+1}(p)$ can only originate at τ-post number p or lower of F_d, which must be lower than $F_d(m)$. Note that the path in step 2 is bounded on the left by the τ-posts of F_i. Also, at the end of this step, the robot's y-coordinate is the same as the top of the τ-post $F_{d+1}(p)$ to which the robot is jumping. Even if the greedy right-down path of step 3 goes only to the right, it will hit this τ-post. In the *worst* case, the motion in step 3 is just vertically down until a tree edge (down-right or up-right) is reached. From the definition of the fence-tree, it is easy to see that following the tree edges to the right must lead to $F_{d+1}(p)$. ■

In the lemma below, we show that the cost of all calls to JumpDownRight can be charged off to the lengths of all edges in the fence-tree.

Lemma 16 *The total cost of all calls to* JumpDownRight *is at most a constant times the total length of all edges in the fence-tree, plus the heights of all* τ*-posts in the tree.*

Proof: As in the case of simple scenes (Lemma 13) we will present a charging scheme where the cost of different invocations of JumpDownRight is charged to distinct portions of the fence-tree. Consider a particular call to JumpDownRight (d) from Raise (i, q), to jump from $F_d(m)$ to $F_{d+1}(p)$ (that is, when this procedure is called, $|F_d| = m$ and $|F_{d+1}| = p$). See Fig. 17. By a reasoning similar to the one in the proof of Lemma 13, we can see that there is a set N of new obstacles that will be added to F_{d+1} by the Raise $(d+1, q)$ procedure that is invoked just after this invocation of JumpDownRight (d). After these new obstacles are added to F_{d+1}, $|F_{d+1}|$ would equal $|F_d| = m$. Let E be the set of edges of F_{d+1} (if any) that are followed in step 3 of JumpDownRight.

Clearly the total horizontal and vertical motion of this procedure (in steps 2 and 3) is no more than twice the total length of the edges in E plus the heights of the obstacles in N. (This bound is actually quite loose but will suffice for our purposes). As before it is easy to argue that the sets N and E of any future call of JumpDownRight (d) will not overlap with the corresponding sets of the present call. ■

6.6 Modifying JumpDownLeft

The general procedure JumpDownLeft is shown in Fig. 18, and a sample path executed by that procedure is shown in Fig. 19.

We first establish the correctness of this general procedure, i.e., the analogue of Lemma 7.

Lemma 17 *If the general procedure* JumpDownLeft (j) *is called under the conditions PreJDL*(j), *then the procedure terminates with the robot at the last known* τ*-post of* F_{j+1}.

Proof: Let m and p be the quantities defined in the procedure. See Fig. 19 for a typical path of this procedure. The PreJDL(j) condition $X_{j+1} < X_j$ implies that following tree edges to the left in step 2 will lead to a point where $x = X_{j+1} = X_{j+1}(p)$. The PreJDL(j) conditions $X_{j+1} < X_j$ and Up$(j+1)$ imply that $p = |F_{j+1}| < m - 1$, so the $(m-1)$st posts of fences F_j and above are higher than the top of the destination post $F_{j+1}(p)$. Suppose F_r is the highest fence reached in step 2, i.e.,

```
Procedure JumpDownLeft $(j)$
  1. Let $m = |F_j|$, $p = |F_{j+1}|$.
  2. Follow tree edges to the left until $x = X_{j+1}(p)$.
  3. Go greedy down-left until robot is either
       • down-left of top of $F_{j+1}(p)$, or
       • down-left of top of $F_u(m-1)$ for some $u \leq j$. In this case:
         (a) While $u \leq j$ do the following:
               i. Go greedy right-down until at $F_u(m-1)$ or on a tree edge.
              ii. If at a tree edge, follow edges to right until at $F_u(m-1)$.
             iii. $u := u + 1$.
         (b) Go greedy down-left until down-left of top of $F_{j+1}(p)$.
  4. Go greedy right-down until at $F_{j+1}(p)$ or on a tree edge. If at a tree edge, follow
     edges to right until at $F_{j+1}(p)$.
```

Figure 18: General procedure for jumping down from F_j to F_{j+1} when $X_j > X_{j+1}$.

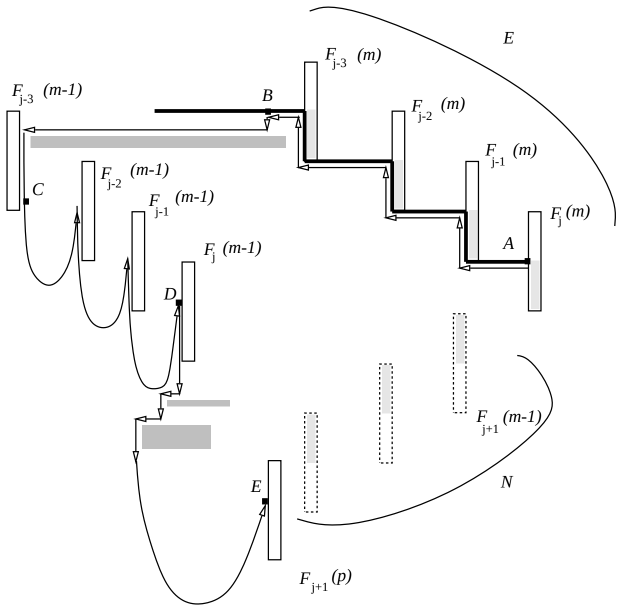

Figure 19: A use of procedure JumpDownLeft to jump from $F_j(m)$ to $F_{j+1}(p)$, for clarity shown in a scene where the fence posts correspond exactly to obstacles of height 2τ. Shaded rectangles with no boundaries are obstacles that are not part of any fence. Solid-boundary rectangles are posts found so far in the tree. Dotted-boundary rectangles are posts of F_{j+1} that will be found immediately following this procedure. Thick solid lines are tree edges. The thin arrow line is the path followed when executing the procedure. Curved arrows represent motions executed during steps 3(a)(i) and 4 of the procedure.

the last edge retraced has its right end on a τ-post of F_r. By the AlmostOrd invariant, no fence can have more than one post to the right of a lower one, so only the last edge retraced in step 2 can be an up-right edge; the others must be *down-right* edges. The same invariant implies that the x-coordinate of the robot at the end of step 2 (i.e. $X_{i+1}(p)$) lies in the interval $[X_u(m-1), X_u(m)]$, for each $u = r, r+1, \ldots, j$. This means that in step 3 when the robot goes greedily down-left, the robot must either reach a point down-left of the top of some post $F_u(m-1)$, or else reach a point down-left of $F_{j+1}(p)$, the destination post. In the former case, the robot enters the while loop of step 3(a). We claim that in each iteration of this while loop, the robot jumps down to post $(m-1)$ of the next lower fence until it reaches $F_j(m-1)$. This motion is similar to that of procedure JumpDownRight, so we can reason as in the proof of Lemma 15 (correctness of JumpDownRight), to show this claim. Once the robot is at $F_j(m-1)$, step 3(b) will take the robot to a point down-left of the top of $F_{j+1}(p)$. Finally step 4 is similar to step 3 in the general version of JumpDownRight, so by reasoning as in the proof of Lemma 15, we can show that the robot will eventually be at $F_{j+1}(p)$. ■

Lemma 18 *The total cost of all calls to* JumpDownLeft *is at most a constant times the total length of all the tree edges, plus the heights of all the τ-posts in the fence-tree.*

Proof: Consider a call to JumpDownLeft (j). As in the case of simple scenes, we can use a charging scheme where different calls to JumpDownLeft will be charged to distinct portions of the fence-tree. As in the proof of Lemma 12, let E be the set of edges followed in step 2 of the procedure, and let N be the set of $m-1-p$ "new" obstacles that will be added to F_{j+1} by the Raise procedure invoked just after this procedure completes. By the same reasoning as in that Lemma, the E and N sets of this call to JumpDownLeft will not overlap with the corresponding sets of any future call. The edge-following motion of the robot during step 2 of the procedure can be charged to the edge-set E, and so we only need to account for the motions in the remaining steps of the procedure.

If step 3 takes the robot vertically down to the τ-post $F_{j+1}(p)$, then as in the case of simple scenes this vertical motion can be charged to the edge-set E and the set N. In this case we are done with the proof. However, in general scenes there are two possibilities for the motion of the robot during this step:

- (A) The robot may reach some point down-left of the top of $F_{j+1}(p)$. Let (x_1, y_1) be the coordinates of the robot at this point. In this case, step 3 is done, and we charge the vertical motion to the total heights of the τ-posts in N plus the lengths of the edges in E. Also in this case, while going greedy down-left, the robot cannot go to the left of post $F_j(p+1)$, since the bottom of this post has the same y-coordinate as the top of $F_{j+1}(p)$. (If the robot was forced to go to the left of $F_j(p+1)$, case (B) below would occur). Therefore the horizontal motion in this case is no more than $X_{j+1}(p) - X_j(p+1)$, which in turn is no more than the lengths of the portions of the edges of F_{j+1} that lie between $x = X_j(p+1)$ and $x = X_{j+1}(p)$; we can charge the horizontal motion to this edge-set E_1. We now claim that the edge-set E_1 of this call to JumpDownLeft (j) will not overlap with the E_1 set of any future call to JumpDownLeft (j). As we argued in the proof of Lemma 12, by the time any future call to JumpDownLeft (j) is made, F_{j+1} would have at least $m-1$ obstacles. Since $X_j(m) > X_{j+1}(p)$, the E_1 set of such a call would lie entirely to the right of $x = X_j(m+1) > X_j(m)$, which is to the right of $x = X_{j+1}(p)$ (the right-boundary of the present E_1 set). Therefore the edge-sets E_1 of different calls to JumpDownLeft (j) cannot overlap.

- (B) The robot may reach some point down-left of the top of post $(m-1)$ of some fence F_j or above. In this case the robot enters the while loop of step 3(a), and repeatedly moves down to the $(m-1)$st post of the next lower fence until it reaches $F_j(m-1)$. The motion in each iteration of this while loop is similar to the motion in the JumpDownRight procedure. The only difference is that instead of going from post number m of one fence to a lower-numbered obstacle of the next lower fence, in this case the robot goes to the *same*-numbered post of the next lower fence. Consider some iteration of this while loop, where the robot is jumping down from $F_u(m-1)$ to $F_{u+1}(m-1)$. Let E_2 be the set of edges of F_{u+1} contained in the region between the lines $x = X_u(m-1)$ and $x = X_{u+1}(m-1)$. The vertical motion in step 3(a)(i) is no more than the height of the destination post, plus the heights of the posts associated with the edges E_2. The horizontal motion in steps 3(a)(i) and 3(a)(ii) is no more than the lengths of the edges in E_2. In any future call to JumpDownLeft the E_2 set corresponding to fence F_{u+1} must lie entirely to the right of $F_u(m)$, which in turn is to the right of $F_{u+1}(m-1)$ (the right-boundary of the present E_2 set). Thus in any future call to JumpDownLeft, the E_2 set associated with F_{u+1} will not overlap with the E_2 set of the present call. Similarly, as argued in the case of simple scenes, in any future call to JumpDownLeft from post m of a fence below F_j the edge-set E followed in step 2 will lie entirely to the right of $X_j(m)$, so none of the E_2 sets of the that call will overlap with those of the present call. After exiting the while loop of step 3(a), the robot executes step 3(b): go greedy down-left until down-left of top of $F_{j+1}(p)$. The charging for this step is similar to case (A) above, except that we do not need to charge the vertical motion to the edge-set E since this step starts at $F_j(m-1)$. The same argument as in case (A) establishes that the charge-sets for this step will not overlap with the corresponding charge-sets of any future call to JumpDownLeft.

Finally in step 4, the robot performs a motion similar to the one in the (generalized) procedure JumpDownRight, and we can use a charging-scheme similar to the one used there. The argument to show that the portions of the tree charged for step 4 of this call to JumpDownLeft (j) do not overlap with the corresponding charge-sets of any future call is similar to the one used above for case (B) of step 3. ■

7 An incremental algorithm

We describe here an improvement of our cumulative algorithm, so that the per-trip ratio on the i'th trip, for all $i \leq n$, is $O(\sqrt{n/i})$. Let us for simplicity say that we know L. From the earlier results in this paper, we know that by searching a distance at most $cL\sqrt{nk}$ we can find an s-t path of length at most $c'L\sqrt{n/k}$, for some constants c, c' and any $k \leq n$.

Let us suppose that at the end of i trips we know an s-t-path π of length at most $c'L\sqrt{n/i}$ (for the base case, simply use the BRS algorithm). What we now want to do is to search with cost at most $cL\sqrt{n2i}$ and find a path of length at most $c'L\sqrt{n/2i}$. Let us denote by Π the path we would have traveled if we did this entire search in one trip using the algorithm of the previous sections. In order to maintain a per-trip ratio of $O(\sqrt{n/i})$, we spread the work of Π over the next i trips as follows. Each trip consists of two phases: The first is a *search* phase, where we walk an additional portion of Π of length $\frac{1}{i}cL\sqrt{n2i} = cL\sqrt{2n/i}$, starting from where we left off on the previous trip. We can always do this because the fences are in a tree structure, so that the last point in Π during the previous search can always be reached from the start point by a known short path whose length adds only a small constant factor to the total trip length. Once the search phase is completed, we "give up" and enter the *follow* phase, where we complete the trip by joining (by a greedy path) the known path π of length $c'L\sqrt{n/i}$, and following it to t. Thus our trip length is still $O(L\sqrt{n/i})$. Since in each such search-follow trip we traverse a portion of Π of length $cL\sqrt{2n/i}$, and the length of Π is at most $cL\sqrt{2ni}$, after i trips we will have completely walked the path Π. So after the first $2i$ trips we have a path of length at most $c'L\sqrt{n/2i}$. This reestablishes our invariant. Thus, we have the following theorem:

Theorem 19 *There is a deterministic algorithm R that for every $i \leq n$ achieves a per-trip ratio on the i'th trip, $\rho_i(R, n)$, of $O(\sqrt{n/i})$.*

8 Modification for Point-to-Point Navigation

Our algorithms can be extended to the case where t is a point rather than a wall, with the same bounds, up to constant factors, as follows. Let us assume for simplicity that the shortest path length L is known. As before, if we do not know L, we can use the standard "guessing and doubling" approach and suffer only a constant factor penalty in performance. On the first trip, the robot can get to t using the optimal point-to-point algorithms of [7] or [3], with a single-trip ratio of $O(\sqrt{n})$. Once at t, the robot creates a greedy up-left path and a greedy down-left path from t, within a window of height $4L$ centered at t. Note that the highest post in a $k \times M$ τ-fence-tree is $M\tau \leq 3L$ above the root (which is always distance L below t) and the lowest post is $k\tau \leq L$ below the root. So the robot is guaranteed to stay within a window of height $4L$ centered at t. Thus after the first trip, these greedy paths play the role of a wall; once the robot hits one of these paths, it can reach t with an additional cost that is only a low-order term in the total cost.

9 Modification for a Purely Tactile Robot

We assumed so far that whenever our robot hits an obstacle, it is told how far the nearest corner of the obstacle is. This information is used only to tell the robot whether or not there is a τ-post at the point of encounter. With only a constant factor penalty (see the analysis in [1]) the robot can obtain this information on its own, using the standard doubling strategy: Move up a distance 1, then down 2, then up 4, and so on, each time moving double the previous distance.

10 Conclusion and Open Problems

The core result of this paper is an algorithm that performs a smooth tradeoff between search effort and the goodness of the path found. This algorithm may be of interest independently of the performance-improvement problem. For instance when a robot has more time or fuel available, one would like it to spend more effort and find a better route. The fence-tree structure is central to this search algorithm. Intuitively, one can think of the fence-tree as representing the collection of those obstacles in the scene which are responsible for making the scene difficult to cross from s to t. Thus the fence-tree in a sense captures the "essence" of a scene, as far as the difficulty (i.e., cost) of crossing the scene is concerned. It would be interesting to explore whether an analogous structure can be defined in more general scenes. This might lead to a generalization of our results to such scenes.

At a higher level, our approach in designing a "learning" navigation algorithm was to start with an algorithm that achieves the above-mentioned cost/performance tradeoff, and convert that to a more incremental algorithm by spreading the work over several trips. This high-level idea may well be useful in designing performance-improvement algorithms for other tasks.

There are several other interesting research directions that can be explored. For instance, can randomization provide a better or simpler algorithm? For the one trip problem, the best lower bound known is $\Omega(\log \log n)$ by Karloff, Rabani and Ravid [11], and the best upper bound is $O(n^{4/9} \log n)$ by Berman, Blum, Fiat, Karloff and Rosen and Saks [2]. What about extending our multi-trip results to more general scenes? Recently, Berman and Karpinski [4] have designed a randomized $O(n^{3/4})$-competitive single-trip algorithm for 2-dimensional scenes containing arbitrary convex obstacles within which a unit circle can be inscribed. Achieving an $O(\sqrt{n})$ ratio for such scenes seems considerably harder. A good first step might be to consider scenes with rectangular obstacles in arbitrary orientations (i.e. not necessarily axis-parallel).

A related problem is the question of how the robot can efficiently visit several destinations in a scene, improving performance wherever possible. One difficulty here is devising a useful performance measure (depending on the location of the destinations, one may be able to use previous information to varying degrees) that appropriately captures the essence of the problem.

References

[1] R. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Information and Computation*, 106(2):234–252, 1993.

[2] P. Berman, A. Blum, A. Fiat, H. Karloff, A. Rosen, and M. Saks. Randomized robot navigation algorithms. Unpublished Manuscript.

[3] E. Bar-Eli, P. Berman, A. Fiat, and P. Yan. On-line navigation in a room. In *Proc. 3rd ACM-SIAM SODA*, 1992.

[4] P. Berman and M. Karpinski. Wall problem with convex obstacles. Unpublished Manuscript, July 1994.

[5] M. Betke, R. Rivest, and M. Singh. Piecemeal learning of an unknown environment. In *Proc. 6th ACM Conf. on Computational Learning Theory*, pages 277–286, 1993.

[6] A. Blum and P. Chalasani. An online algorithm for improving performance in navigation. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 2–11, 1993.

[7] A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrain. In *Proc. 23rd ACM STOC*, 1991.

[8] P. Chalasani. *Online Performance-improvement algorithms*. PhD thesis, Carnegie Mellon University, 1994.

[9] P. Chen. Improving path planning with learning. In *Prof. 9th Int'l Workshop on Machine Learning*, 1992.

[10] E.G. Coffman and E.N. Gilbert. Paths through a maze of rectangles. *Networks*, vol. 22, no. 4, pp. 349–367, July 1992.

[11] H. Karloff, Y. Rabani, and Y. Ravid. Lower bounds for randomized k-server and motion-planning algorithms. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 278–288, 1991.

[12] S. Koenig and R.G.Simmons. Complexity analysis of real-time reinforcement learning. In *Proc. AAAI*, pages 99–105, 1993.

[13] V. Lumelsky. Algorithmic issues of sensor-based robot motion planning. In *26th IEEE Conference on Decision*, pages 1796–1801, 1987.

[14] V. Lumelsky. Algorithmic and complexity issues of robot motion in an uncertain environment. *Journal of Complexity*, 3:146–182, 1987.

[15] V. Lumelsky and A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Trans. on Automatic Control*, 31:1058–1063, 1986.

[16] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for on-line problems. *Journal of Algorithms*, 11:208–230, 1990.

[17] C. Papadimitriou and M. Yannakakis. Shortest paths without a map. In *Proc. 16th ICALP*, 1989.

[18] S. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Carnegie Mellon University, 1992.

This research was sponsored in part by National Science Foundation (NSF) grant no. CCR-0122581.