The Field D* Algorithm for Improved Path Planning and Replanning in Uniform and Non-Uniform Cost Environments

Dave Ferguson and Anthony Stentz

Technical Report CMU-TR-RI-05-19
Carnegie Mellon University
{dif,tony}@cmu.edu

Abstract. We present an interpolation-based planning and replanning algorithm for generating smooth paths through non-uniform cost grids. Most grid-based path planners use discrete state transitions that artificially constrain an agent's motion to a small set of possible headings (e.g. $0, \frac{\pi}{4}, \frac{\pi}{2}$, etc). As a result, even the 'optimal' grid planners produce unnatural, suboptimal paths for the agent to traverse. Our approach uses linear interpolation during planning to calculate accurate path cost estimates for arbitrary positions within each grid cell and to produce paths with a range of continuous headings. Consequently, it is particularly well suited to planning smooth, least-cost trajectories for mobile robots. In this paper, we present a number of applications and results, a comparison to related algorithms, and several implementations on real robotic systems.

1 Introduction

In mobile robot navigation, we are often provided with a grid-based representation of our environment and tasked with planning a path from some initial robot location to a desired goal location. Depending on the environment, the representation may be binary (each grid cell contains either an obstacle or free space) or may associate with each cell a cost reflecting the difficulty of traversing the respective area of the environment.

Many algorithms exist for planning paths on such grids. Dijkstra's algorithm computes paths from every grid cell to a goal location [1]. A* uses a heuristic to focus the search from a particular start location towards the goal and thus produces a path from a single location to the goal very efficiently [2,10]. D*, Incremental A*, and D* Lite are extensions of A* that incrementally repair solution paths when changes occur in the underlying graph [15,5,4,3]. These incremental algorithms have been used extensively in robotics for mobile robot navigation in unknown or dynamic environments.

However, almost all of these approaches are limited by the small, discrete set of possible transitions they allow between grid cells. For instance, when planning on a 2D grid, it is common to plan from the center of each grid cell and only allow transitions to the centers of adjacent grid cells. This restricts the agent's heading to increments of $\frac{\pi}{4}$, which results in paths that are suboptimal in length and difficult to traverse in practice.

In this paper we present Field D*, an interpolation-based planning and replanning algorithm that alleviates this problem. This algorithm extends D* and D* Lite to use linear interpolation to efficiently produce globally-smooth paths. The paths are optimal with respect to a linear interpolation assumption and very effective in practice. This algorithm is currently being used by a number of fielded robotic systems in both indoor and outdoor environments.

We begin by discussing the limitations of paths produced using classical grid-based planners and mention recent approaches that attempt to overcome some of these limitations. We then present a linear interpolation-based method for obtaining more accurate cost approximations of grid points and introduce Field D*, a novel planning and replanning algorithm that uses this method. Next, we provide a number of optimizations, a few example illustrations and applications, and results comparing our algorithm to D* Lite, a member of one of the most widely-used families of grid-based planning and replanning algorithms in robotics. We conclude with discussion and extensions.

2 Limitations of Classical 2D Path Planning

Consider a ground vehicle navigating an outdoor environment. We can represent this environment as a 2D traversability grid in which cells are given a cost of traversal reflecting the difficulty of navigating the respective area of the environment. With such a grid we can extract a graph for path planning quite easily: assign a node to each cell center, with edges connecting the node to each adjacent cell center (node). The cost of each edge is a combination of the traversal costs of the two cells it connects and the length of the edge. Fig. 1(a) shows the node and edge extraction process for one cell in a uniform 2D grid.

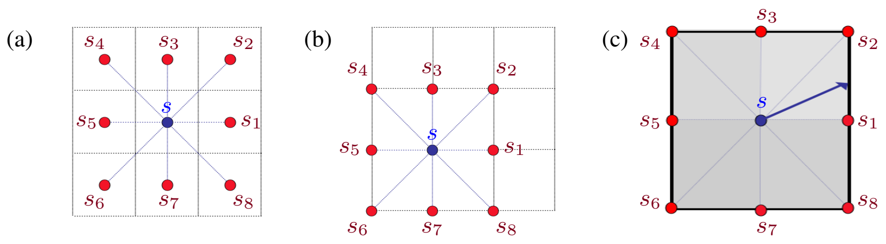

Fig. 1. (a) A standard 2D grid used for path planning, in which nodes reside at the center of each grid cell. The arcs emanating from the center node represent all the possible actions that can be taken from this node. (b) A modified representation used by Field D*, in which nodes reside at the corners of grid cells. (c) The optimal path from node s must intersect one of the edges $\{\overrightarrow{s_1s_2}, \overrightarrow{s_2s_3}, \overrightarrow{s_3s_4}, \overrightarrow{s_4s_5}, \overrightarrow{s_5s_6}, \overrightarrow{s_6s_7}, \overrightarrow{s_7s_8}, \overrightarrow{s_8s_1}\}$.

We can then plan over this graph to generate paths from the robot's initial location to a desired goal location, using any of the algorithms already mentioned. Unfortunately, paths produced using this graph are restricted to headings of $\frac{\pi}{4}$ increments. This means that the final solution path may be suboptimal in path cost, involve unnecessary turning, or both.

For instance, consider a robot facing its goal position in a completely obstacle-free environment (see Fig. 2). Obviously, the optimal path is a straight line between the robot and the goal. However, if the robot's initial heading is not a multiple of $\frac{\pi}{4}$, the path returned by traditional planners would require that the robot first turn to attain the nearest grid heading, move some distance along this heading, and then turn $\frac{\pi}{4}$ in the opposite direction of its initial turn and continue to the goal. Not only does this path have clearly suboptimal length, it contains possibly expensive or difficult turns that are purely artifacts of the limited representation.

Sometimes it is possible to alleviate this problem by post-processing the path. Usually, given a robot location s, one finds the furthest point p along the solution path for which a straight line path from s to p is collision-free, then replaces the original path to p with this straight line path. However, this does not always work, as illustrated by Fig. 3. Indeed, for non-uniform cost environments such smoothing can often *increase* the cost of the path.

Recently, researchers have looked at more sophisticated methods of obtaining smooth paths through grids. Konolige [6] presents an interpolated planner that first uses discrete grid planning to construct a cost-to-goal value function over the grid and then interpolates this result to produce a smooth path from the initial position to the goal. This method results in smoother paths for agents to traverse but does not incorporate the smoothed path cost into the planning process. Consequently,

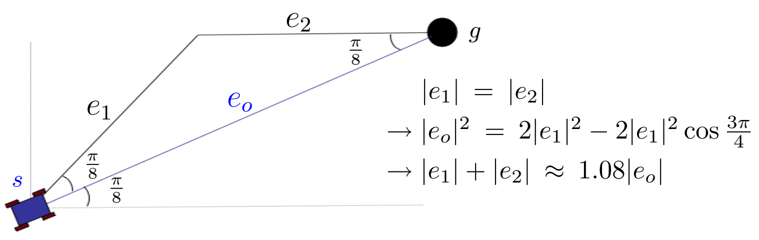
e_2
g
$\frac{\pi}{8}$
e_1
e_o
$|e_1| = |e_2|$
$\rightarrow |e_o|^2 = 2|e_1|^2 - 2|e_1|^2 \cos \frac{3\pi}{4}$
$\rightarrow |e_1| + |e_2| \approx 1.08|e_o|$
s
$\frac{\pi}{8}$
$\frac{\pi}{8}$

Fig. 2. A uniform 2D grid-based path (e_1 plus e_2) between two grid nodes can be up to 8% longer than an optimal straight-line path (e_0). Here, the desired straight-line heading is $\frac{\pi}{8}$ and lies perfectly between the two nearest grid-based headings of 0 and $\frac{\pi}{4}$. This result is independent of the resolution of the grid.

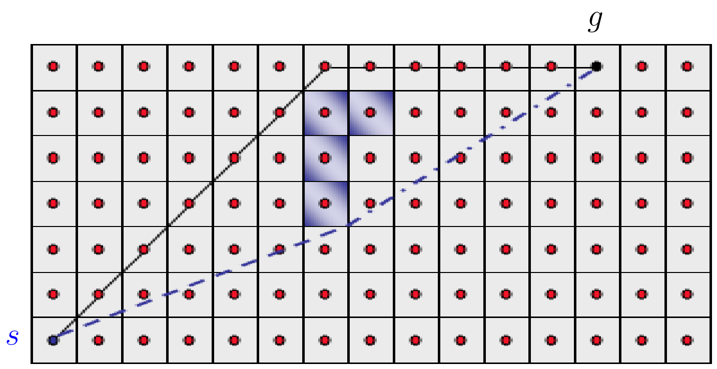

Fig. 3. 2D grid-based paths cannot always be smoothed in a post-processing phase. Here, the grid-based path from s to g (top, in black) cannot be smoothed because there are four obstacle cells (shaded). The optimal path is shown dashed (in blue).

the resulting path is not necessarily as good as the path the algorithm would produce if continuous costs were calculated during planning. Further, this approach provides no replanning functionality.

Philippsen [11,12] presents an algorithm based on Fast Marching Methods [13] that computes a value function over the grid by growing a surface out from the goal to every region in the environment. The surface expands according to surface flow equations, and the value of each grid point is computed by combining the values of two neighboring grid points. This approach incorporates the interpolation step into the planning process, producing low cost, interpolated paths. Philippsen has shown that this approach generates nice paths in indoor environments [11,12]. However, the search is not focussed towards the robot location (such as in A*) and assumes that the transition cost from a particular grid node to each of its neighbors is constant. Consequently, it is not as applicable to navigation in outdoor environments, which are often best represented by large, non-uniform cost grids.

The idea of using interpolation to produce better value functions for discrete samples over a continuous state space is not new. This approach has been used in dynamic programming for some time to compute the value of successors that are not in the set of samples [7–9]. However, as LaValle points out [9], when the action space is also continuous, this becomes difficult, as solving for the value of a state now requires minimizing over an uncountably infinite set of successor states.

The approach we present here is an extension of the widely-used D* family of algorithms that uses linear interpolation to produce globally smooth, low-cost paths. It relies upon an efficient, closed-form solution to the above minimization problem for 2D grids, which we will introduce in

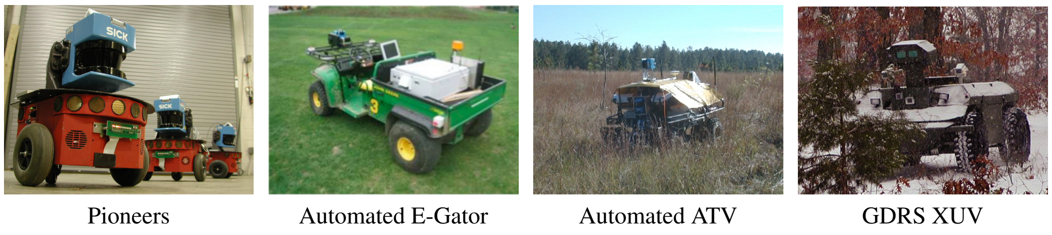

Fig. 4. Some robots that currently use Field D* for path planning. These range from indoor planar robots (the Pioneers) to outdoor robots able to operate in harsh terrain (the XUV).

Section 3. As with D* and D* Lite, our approach focusses its search towards the most relevant areas of the state space during both initial planning and replanning. It is very effective in practice and is currently employed as the path planner in a wide range of fielded robotic systems (see Fig. 4).

3 Improving Cost Estimation through Interpolation

The key to our algorithm is a novel method of computing the path cost of each grid node s given the path costs of its neighboring nodes. By the path cost of a node we mean the cost of a path from the node to the goal. In classical grid-based planning this value is computed as

$$g(s) = \min_{s' \in nbrs(s)} (c(s, s') + g(s')),$$

where $nbrs(s)$ is the set of all neighboring nodes of s (see Fig. 1), $c(s, s')$ is the cost of traversing the edge between s and s', and $g(s')$ is the path cost of node s'.

This calculation assumes that the only transitions possible from node s are straight-line trajectories to one of its neighboring nodes. This assumption results in the limitations of grid-based plans discussed earlier. However, consider relaxing this assumption and allowing a straight-line trajectory from node s to any point on the boundary of its grid cell. If we knew the value of every point s_b along this boundary, then we could compute the optimal value of node s simply by minimizing $c(s, s_b) + g(s_b)$, where $c(s, s_b)$ is computed as the distance between s and s_b multiplied by the traversal cost of the cell in which s resides. Unfortunately, there are an infinite number of such points s_b and so computing $g(s_b)$ for each of them is not possible.

It is possible, however, to provide an approximation to $g(s_b)$ for each boundary point s_b by using linear interpolation. To do this, we first modify the graph extraction process discussed earlier. Instead of assigning nodes to the centers of grid cells, we assign nodes to the *corners* of each grid cell, with edges connecting nodes that reside at corners of the same grid cell (see Fig. 1(b)). Given this modification, the traversal costs of any two equal-length segments of an edge will be the same. This differs from the original graph extraction process in which the first half of an edge was in one cell and the second half was in another cell, with the two cells possibly having different traversal costs. In the modified approach the cost of an edge that resides on the boundary of two grid cells is defined as the minimum of the traversal costs of each of the two cells.

We then treat the nodes in our graph as sample points of a continuous cost field. The optimal path from a node s must pass through an edge connecting two consecutive neighbors of s, for example $\overrightarrow{s_1 s_2}$ (see Fig. 1(c)). The path cost of s is thus set to the minimum cost of a path through any of these edges, which are considered one at a time. To compute the path cost of node s using edge $\overrightarrow{s_1 s_2}$, we use the path costs of nodes s_1 and s_2 and the traversal costs c of the center cell and b of the bottom cell (see Fig. 5).

In order to compute this cost efficiently, we assume the path cost of any point s_y residing on the edge between s_1 and s_2 is a linear combination of $g(s_1)$ and $g(s_2)$:

$$g(s_y) = yg(s_2) + (1 - y)g(s_1),$$

where y is the distance from s_1 to s_y (assuming unit cells). This assumption is not perfect: the path cost of s_y may not be a *linear* combination of $g(s_1)$ and $g(s_2)$, nor even a function of these path costs. However, this linear approximation works well in practice, and allows us to construct a closed form solution for the path cost of node s.

Given this approximation, the path cost of s given s_1, s_2, and cell costs c and b can be computed as

$$\min_{x,y}[bx + c\sqrt{(1-x)^2 + y^2} + g(s_2)y + g(s_1)(1 - y)],$$

where $x \in [0, 1]$ is the distance traveled along the bottom edge from s before cutting across the center cell to reach the right edge a distance of $y \in [0, 1]$ from s_1 (see Fig. 5). Note that if both x and y are zero in the above equation the path taken is along the bottom edge but its cost is computed from the traversal cost of the center cell.

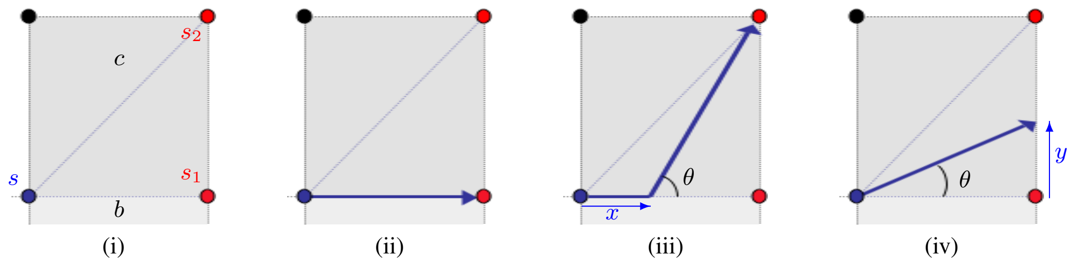

Fig. 5. Computing the path cost of node s using the path cost of two of its neighbors, s_1 and s_2, and the traversal costs c of the center cell and b of the bottom cell. Illustrations (ii) through (iv) show the possible optimal paths from s to edge $\overrightarrow{s_1 s_2}$.

Let $(\bar{x}, \bar{y})$ be a pair of values for x and y that solve the above minimization. Because of the linear interpolation at least one of these values will be either zero or one. Intuitively, if it is less expensive to partially cut through the center cell than to traverse around the boundary, then it is least expensive to completely cut through the cell. Thus, if there is any component to the cheapest solution path from s that cuts through the center cell, it will be as large as possible, forcing $\bar{x} = 0$ or $\bar{y} = 1$. If there is no component of the path that cuts through the center cell, then $\bar{y} = 0$.

We can prove this simply as follows.

First, set $f = g(s_1) - g(s_2)$, which can be thought of as the cost of traversing the right edge[1]. It is also useful to think of the original problem as trying to plan a path from s to s_2 with the minimum possible cost. This follows because if $g(s_1) < g(s_2)$ then it is always cheapest to compute the path cost of s by taking a direct route to s_1, which is equivalent to having $f < 0$ so that we want to travel along the entirety of the right edge (i.e. we want to travel from s_1 to s_2 along the right edge, so the planning problem is to get to s_1 with the minimum path cost). Since this just gives us the path cost of getting to s_2, we then have to add $g(s_2)$ to our result to get the full path cost of s.

[1] If $f > r$, where r is the traversal cost of the right cell, then the path cost $g(s_1)$ is clearly suboptimal. We assume in our derivation that both $g(s_1)$ and $g(s_2)$ are optimal with respect to our linear interpolation assumption.

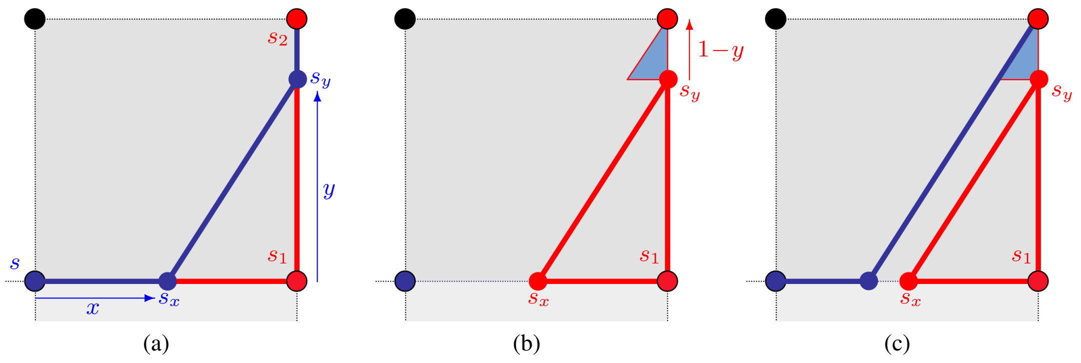

Fig. 6. (a) Imagine the blue path is the optimal path, travelling along bottom edge to s_x then across center cell to s_y, then up right edge to s_2. Notice the triangle formed between vertices s_x, s_1, and s_y. (b) We can create a scaled version of this triangle with a vertical edge length of $1 - y$. (c) Combining the hypothenuses of the two triangles shown in (b) produces a lower-cost path than the one shown in (a), forcing a contradiction.

Now, assume the optimal path involved travelling along parts of both the bottom and right edges *and* some cutting across part of the center cell (i.e., $\bar{x} \in (0, 1)$ and $\bar{y} \in (0, 1)$). Thus, the path travels from s along the bottom edge some distance $x \in (0, 1)$ to point s_x, then cuts across the center cell to arrive at a point s_y on the right edge some distance $y \in (0, 1)$ from s_1, then travels along the right edge to s_2 (see Fig. 6(a)). Since this path is optimal, the cost of taking the straight-line path from s_x through the center cell to s_y must be cheaper than going from s_x along the bottom edge to s_1 then up the right edge to s_y. Thus, we have the following relationship:

$$c\sqrt{(1-x)^2 + y^2} \le (1-x)b + yf$$

The straight-line path between s_x and s_y, along with the line between s_x and s_1 and the line between s_1 and s_y, define a right angled triangle. We know that since the cost of the weighted hypotenuse $\overrightarrow{s_x s_y}$ is cheaper than the combined costs of the weighted sides $\overrightarrow{s_x s_1}$ and $\overrightarrow{s_1 s_y}$, this will be the case if we were to scale the size of the triangle by any amount (maintaining the same ratios of side lengths).

Now, assume (w.l.o.g) that $(1 - y) < x$, so that s_y is closer to s_2 than s_x is to s.

Then let's construct a scaled version of this triangle with a vertical edge of length $(1 - y)$ (see Fig. 6(b)). The horizontal edge of this triangle will have length $(1-y)\frac{(1-x)}{y}$. Since this is a scaled version of our original triangle, the weighted cost of the hypotenuse of this new triangle is cheaper than the combined weighted costs of the horizontal and vertical edges[2]. But this means we could combine the hypotenuse of this new triangle with our previous hypotenuse and construct a path that went from s along the bottom edge a distance of $x - (1-y)\frac{(1-x)}{y}$ then straight to s_2, and the cost of this path would be *less* than the cost of our original (optimal) path (see Fig. 6(c)). This is a contradiction. Hence no optimal path involves traveling along sections of both the bottom and right edges *and* a component cutting across part of the center cell[3].

[2] Note that we have drawn this new triangle above our previous triangle only to show how they could be combined to form a single triangle. The costs of the vertical and horizontal edges of the new triangle are derived from the values f and b, respectively.

[3] Note that it may be possible that the optimal path to s_2 involves travelling along the vertical edge from s for some distance, then cutting across to s_2. However, this possibility will be examined when computing the path cost from s to neighbors s_2 and s_3 (see Figure 1). We can thus restrict our attention when computing

```
ComputeCost($s, s_a, s_b$)
01. if ($s_a$ is a diagonal neighbor of $s$)
02.     $s_1 = s_b; s_2 = s_a;$
03. else
04.     $s_1 = s_a; s_2 = s_b;$
05. $c$ is traversal cost of cell with corners $s, s_1, s_2$;
06. $b$ is traversal cost of cell with corners $s, s_1$ but not $s_2$;
07. if $(\min(c, b) = \infty)$
08.     $v_s = \infty;$
09. else if $(g(s_1) \leq g(s_2))$
10.     $v_s = \min(c, b) + g(s_1);$
11. else
12.     $f = g(s_1) - g(s_2);$
13.     if $(f \leq b)$
14.         if $(c \leq f)$
15.             $v_s = c\sqrt{2} + g(s_2);$
16.         else
17.             $y = \min(\frac{f}{\sqrt{c^2 - f^2}}, 1);$
18.             $v_s = c\sqrt{1 + y^2} + f(1 - y) + g(s_2);$
19.     else
20.         if $(c \leq b)$
21.             $v_s = c\sqrt{2} + g(s_2);$
22.         else
23.             $x = 1 - \min(\frac{b}{\sqrt{c^2 - b^2}}, 1);$
24.             $v_s = c\sqrt{1 + (1 - x)^2} + bx + g(s_2);$
25. return $v_s$;
```

Fig. 7. The Interpolation-based Path Cost Calculation

Thus, the path will either travel along the entire bottom edge to s_1 (Fig. 5(ii)), or will travel a distance x along the bottom edge then take a straight-line path directly to s_2 (Fig. 5(iii)), or will take a straight-line path from s to some point s_y on the right edge (Fig. 5(iv)). Which of these paths is cheapest depends on the relative sizes of c, b, and the difference f in path cost between s_1 and s_2: $f = g(s_1) - g(s_2)$. Specifically, if $f < 0$ then the optimal path from s goes straight to s_1 and will have a cost of $\min(c, b) + g(s_1)$ (Fig. 5(ii)). If $f = b$ then the cost of a path using some portion of the bottom edge (Fig. 5(iii)) will be equivalent to the cost of a path using none of the bottom edge (Fig. 5(iv)). We can solve for the $\bar{y}$ of the latter path (equal to $1 - \bar{x}$ for the former path) that minimizes the path cost as follows. Firstly, let $k = f = b$. The path cost of s is

$$c\sqrt{1 + y^2} + k(1 - y) + g(s_2)$$

Taking the differential with respect to y and setting this equal to zero yields:

$$\bar{y} = \sqrt{\frac{k^2}{c^2 - k^2}}$$

Whether the bottom edge or the right edge is used we end up with the same calculations and path cost computations. So all that matters is which edge is cheaper. If $f < b$ then we use the right edge and compute the path cost as above (with $k = f$), and if $b < f$ we use the bottom edge and substitute $k = b$ and $\bar{y} = 1 - \bar{x}$ into the above equation. The resulting algorithm for computing the minimum path cost of s given *any* two consecutive neighbors s_a and s_b is provided in Fig. 7.

the path cost of s using neighbors s_1 and s_2 to paths that fully reside within the triangle defined by vertices s, s_1, and s_2.

```
key(s)
  01. return [min(g(s), rhs(s)) + h(s_start, s); min(g(s), rhs(s))];

UpdateState(s)
  02. if s was not visited before, g(s) = ∞;
  03. if (s ≠ s_goal)
  04.     rhs(s) = min_{(s′,s″)∈connbrs(s)} ComputeCost(s, s′, s″);
  05. if (s ∈ OPEN) remove s from OPEN;
  06. if (g(s) ≠ rhs(s)) insert s into OPEN with key(s);

ComputeShortestPath()
  07. while (min_{s∈OPEN}(key(s)) <̇ key(s_start) OR rhs(s_start) ≠ g(s_start))
  08.     remove state s with the minimum key from OPEN;
  09.     if (g(s) > rhs(s))
  10.         g(s) = rhs(s);
  11.         for all s′ ∈ nbrs(s) UpdateState(s′);
  12.     else
  13.         g(s) = ∞;
  14.         for all s′ ∈ nbrs(s) ∪ {s} UpdateState(s′);

Main()
  15. g(s_start) = rhs(s_start) = ∞; g(s_goal) = ∞;
  16. rhs(s_goal) = 0; OPEN = ∅;
  17. insert s_goal into OPEN with key(s_goal);
  18. forever
  19.     ComputeShortestPath();
  20.     Wait for changes in cell traversal costs;
  21.     for all cells x with new traversal costs
  22.         for each state s on a corner of x
  23.             UpdateState(s);
```

Fig. 8. The Field D* Algorithm (basic D* Lite version).

4 Field D*

Once equipped with this interpolation-based path cost calculation for a given node in our graph, we can plug it into any of a number of current planning and replanning algorithms to produce smooth paths. Fig. 8 presents our most recent formulation of *Field D**, an incremental replanning algorithm incorporating these interpolated path costs. This version of Field D* is based on D* Lite (with differences highlighted in red)[4].

In this figure, $connbrs(s)$ contains the set of consecutive neighbor pairs of state s: $connbrs(s) = \{(s_1,s_2), (s_2,s_3), (s_3,s_4), (s_4,s_5), (s_5,s_6), (s_6,s_7), (s_7,s_8), (s_8,s_1)\}$, where s_i is positioned as shown in Fig. 1(c). Apart from this construct, notation follows from the D* Lite algorithm: $g(s)$ is the current path cost of state s (its *g-value*), $rhs(s)$ is the one-step lookahead path cost for s (its *rhs-value*), *OPEN* is a priority queue containing inconsistent states (i.e., states s for which $g(s) \neq rhs(s)$) in increasing order of *key* values (line 01), s_{start} is the initial agent state, and s_{goal} is the goal state. $h(s_{start}, s)$ is a heuristic estimate of the cost of a path from s_{start} to s. Because the key value of each state contains two quantities a lexicographic ordering is used, where key(s) $\dot{<}$ key(s') iff the first element of key(s) is less than the first element of key(s') or the first element of key(s) equals the first element of key(s') and the second element of key(s) is less than the second element of key(s'). For more details on the D* Lite algorithm and this terminology, see [4,3].

This is an unoptimized version of Field D*. In the following section we discuss a number of optimizations that significantly improve the overall efficiency of planning and replanning with this algorithm.

[4] As opposed to the original, graph-based version of D* Lite, lines {20 - 22} tailor Field D* to grids. Also, because paths intersect edges and not just nodes, $h(s_{start}, s)$ must be small enough that when added to the cost of any edge emanating from s it is still not greater than a minimum cost path from s_{start} to s.

Once the cost of a path from the initial state to the goal has been calculated, the path is extracted by starting at the initial position and iteratively computing the cell boundary point to move to next. Because of our interpolation technique, it is possible to compute the path cost of *any* point inside a grid cell, not just the corners, which is useful for both extracting the path and getting back on track if execution is not perfect (which is usually the case for real robots). See Section 6 for more on path extraction.

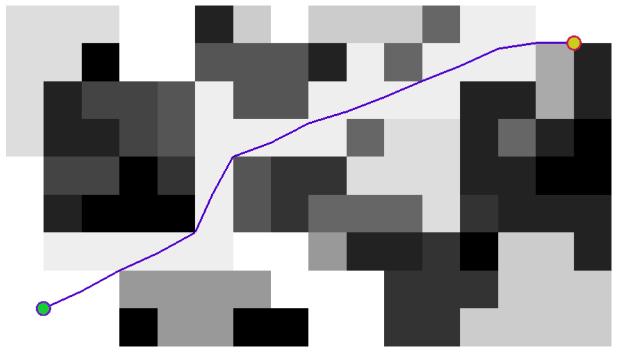

Fig. 9. A close-up of a path planned using Field D* showing individual grid cells. Notice that the path is not limited to entering and exiting cells at corner points.

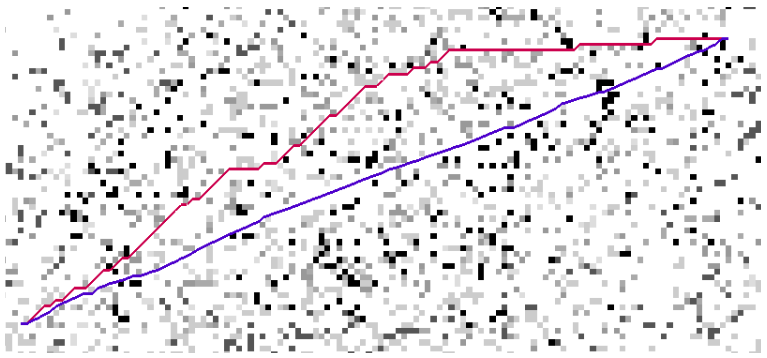

Fig. 10. Paths produced by D* Lite (top) and Field D* (bottom) in a 150×60 non-uniform cost environment.

Figures 9, 10, and 11 illustrate Field D* applied to three non-uniform cost environments. In each of these figures, darker areas represent regions that are more costly to traverse. Unlike paths produced using classical grid-based planners, paths produced using Field D* are not restricted to a small set of headings. As a result, Field D* provides smoother, lower-cost paths through both uniform and non-uniform cost environments.

5 Optimizations

As with other members of the D* family of algorithms, there are a number of optimizations that can be made to the basic algorithm to significantly improve its efficiency. Firstly, we can reduce the

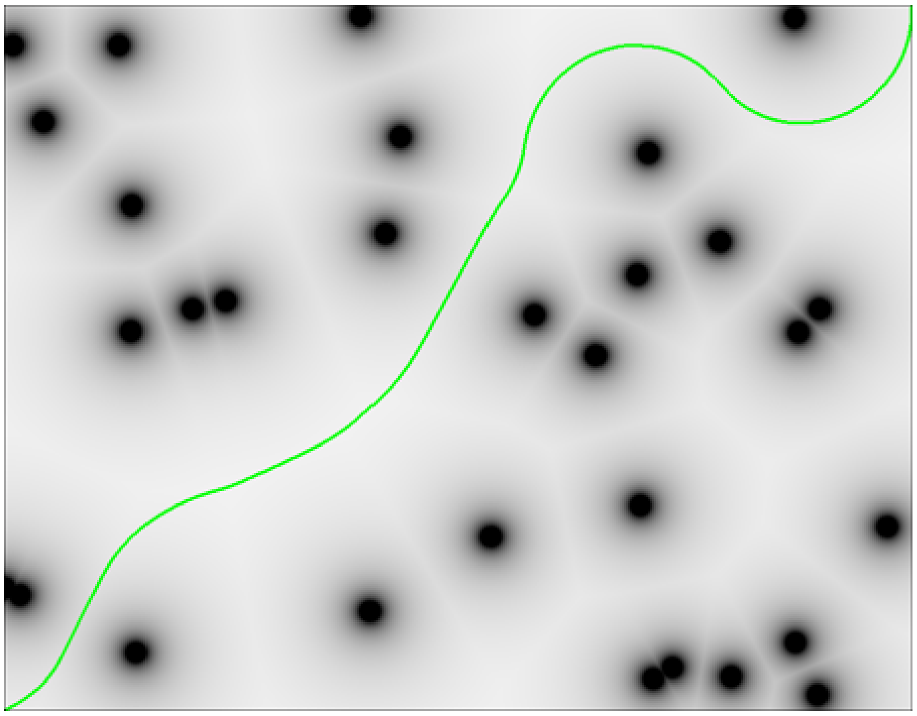

Fig. 11. Planning through a potential field of obstacles. At high grid resolutions, Field D* produces smooth curves through both uniform and non-uniform cost environments; this is not generally true of standard grid-based planners.

amount of computation required when updating the neighbors of a popped state (lines $\{09$ - $14\}$) by only considering those states actually affected by the new value of the popped state and how these states are affected.

To do this, we keep track of a backpointer for each state, specifying from which states it currently derives its path cost. Since, in Field D*, the successor of each state is a point on an edge connecting two of its neighboring states, this backpointer needs to specify the two states that form the endpoints of this edge. We use $bptr(s)$ to refer to the most *clockwise* of the two endpoint states relative to state s. For example, if the current best path from state s intersects edge $\overrightarrow{s_1 s_2}$ (see Fig. 1), then $bptr(s) = s_1$. We also make use of two new operators, $cknbr(s, s')$ and $ccknbr(s, s')$, that, given a state s and some neighboring state s' of s, return the next neighboring state of s in the clockwise and counter-clockwise directions, respectively. Thus, using the state labels from Fig. 1, $cknbr(s, s_1) = s_8$, $cknbr(s, s_8) = s_7$, etc, and $ccknbr(s, s_1) = s_2$, $ccknbr(s, s_2) = s_3$, etc. Thus, if $bptr(s) = s_1$ then $ccknbr(s, bptr(s)) = s_2$.

This gives us the algorithm presented in Fig. 12. Some sections of the algorithm are clearly not as efficient as they could be (e.g. the repeated path cost calculation in lines $\{12$ - $13\}$ and again in lines $\{15$ - $16\}$ and $\{23$ - $24\}$) and have been presented in the current form only for clarity.

We can also save a significant amount of computation by optimizing how changes to the traversal costs of individual cells are dealt with. In the naive implementation of the algorithm, when the traversal cost of a cell changes we recompute the path cost for any state that resides at one of the

```
**key**$(s)$
  return $[\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))]$;

**UpdateState**$(s)$
  if $(g(s) \neq rhs(s))$ insert $s$ into *OPEN* with key($s$);
  else if ($s \in$ *OPEN*) remove $s$ from *OPEN*;

**ComputeShortestPath**$()$
  while ($\min_{s \in OPEN}$(key($s$)) $\dot{<}$ key($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
    peek at state $s$ with the minimum key on *OPEN*;
    if $(g(s) > rhs(s))$
      $g(s) = rhs(s)$;
      remove $s$ from *OPEN*;
      for all $s' \in nbrs(s)$
        if $s'$ was not visited before
          $g(s') = rhs(s') = \infty$;
        if ($rhs(s') >$ ComputeCost($s', s, ccknbr(s', s)$))
          $rhs(s') =$ ComputeCost($s', s, ccknbr(s', s)$);
          $bptr(s') = s$;
        if ($rhs(s') >$ ComputeCost($s', s, cknbr(s', s)$))
          $rhs(s') =$ ComputeCost($s', cknbr(s', s), s$);
          $bptr(s') = cknbr(s', s)$;
        UpdateState($s'$);
    else
      $g(s) = \infty$;
      for all $s' \in nbrs(s)$
        if ($bptr(s') = s$ OR $bptr(s') = cknbr(s', s)$)
          $rhs(s') = \min_{s'' \in nbrs(s')}$ ComputeCost($s', s'', ccknbr(s', s'')$);
          $bptr(s') =$ argmin$_{s'' \in nbrs(s')}$ComputeCost($s', s'', ccknbr(s', s'')$);
          UpdateState($s'$);
      UpdateState($s$);
**Main**$()$
  $g(s_{start}) = rhs(s_{start}) = \infty; g(s_{goal}) = \infty$;
  $rhs(s_{goal}) = 0$; *OPEN* $= \emptyset$;
  insert $s_{goal}$ into *OPEN* with key($s_{goal}$);
  forever
    ComputeShortestPath();
    Wait for changes in cell traversal costs;
    for all cells $x$ with new traversal costs
      for each state $s$ on a corner of $x$
        if $s$ was not visited before, $g(s) = \infty$;
        if $(s \neq s_{goal})$
          $rhs(s) = \min_{s' \in nbrs(s)}$ ComputeCost($s, s', ccknbr(s, s')$);
          UpdateState($s$);
```

Fig. 12. The Field D* Algorithm (after initial optimizations).

corners of the cell. This can be hugely expensive, often far more so than replanning once the traversal costs have been updated.

However, we can reduce this computation considerably by making two alterations to the algorithm. Firstly, when the traversal cost of a cell increases, we only need to update the rhs-value of states that relied upon the old cell cost. Secondly, when the traversal cost of a cell decreases, we can avoid recomputing the rhs-value for *any* state (at least initially) by making one small approximation to the algorithm. Basically, instead of recomputing rhs-values for each of the corner states, we simply put the state with the minimum current rhs-value onto the *OPEN* list. Since this state may have its rhs-value equal to its g-value, we need to modify the algorithm so that such states are processed as if their path costs have decreased (i.e. as if their rhs-values were in fact lower than their g-values). Then, when this state is popped off the *OPEN* list, it will have a chance to update the rhs-values of the other corner states based on the new traversal cost of the center cell.

Unfortunately, it turns out that this second method no longer guarantees optimal path costs with respect to our linear interpolation assumption. This is because it is possible in theory that one of the

```
**ComputeShortestPath**()
04. while ($\min_{s \in OPEN}$(key($s$)) $\dot{<}$ key($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
05.     peek at state $s$ with the minimum key on OPEN;
06.     if ($g(s) \geq rhs(s)$)
07.         $g(s) = rhs(s)$;
08.         remove $s$ from OPEN;
09.         for all $s' \in nbrs(s)$
10.             if $s'$ was not visited before
11.                 $g(s') = rhs(s') = \infty$;
12.             $rhs_{old} = rhs(s')$;
13.             if ($rhs(s') >$ ComputeCost($s', s, ccknbr(s', s)$))
14.                 $rhs(s') =$ ComputeCost($s', s, ccknbr(s', s)$);
15.                 $bptr(s') = s$;
16.             if ($rhs(s') >$ ComputeCost($s', s, cknbr(s', s)$))
17.                 $rhs(s') =$ ComputeCost($s', cknbr(s', s), s$);
18.                 $bptr(s') = cknbr(s', s)$;
19.             if ($rhs(s) \neq rhs_{old}$)
20.                 UpdateState($s'$);
21.     else
22.         $rhs(s) = \min_{s' \in nbrs(s)}$ ComputeCost($s, s', ccknbr(s, s')$);
23.         $bptr(s) =$ argmin$_{s' \in nbrs(s)}$ComputeCost($s, s', ccknbr(s, s')$);
24.         if ($g(s) < rhs(s)$)
25.             $g(s) = \infty$;
26.             for all $s' \in nbrs(s)$
27.                 if ($bptr(s') = s$ OR $bptr(s') = cknbr(s', s)$)
28.                     if ($rhs(s') \neq$ ComputeCost($s', bptr(s'), ccknbr(s', bptr(s'))$))
29.                         if ($g(s') < rhs(s')$ OR $s' \notin$ OPEN)
30.                             $rhs(s') = \infty$;
31.                             UpdateState($s'$);
32.                         else
33.                             $rhs(s') = \min_{s'' \in nbrs(s')}$ ComputeCost($s', s'', ccknbr(s', s'')$);
34.                             $bptr(s') =$ argmin$_{s'' \in nbrs(s')}$ComputeCost($s', s'', ccknbr(s', s'')$);
35.                             UpdateState($s'$);
36.         UpdateState($s$);

**UpdateCellCost**($x, c$)
37. if ($c$ is greater than current traversal cost of $x$)
38.     for each state $s$ on a corner of $x$
39.         if either $bptr(s)$ or $ccknbr(s, bptr(s))$ is a corner of $x$
40.             if ($rhs(s) \neq$ ComputeCost($s, bptr(s), ccknbr(s, bptr(s))$))
41.                 if ($g(s) < rhs(s)$ OR $s \notin$ OPEN)
42.                     $rhs(s) = \infty$;
43.                     UpdateState($s$);
44.                 else
45.                     $rhs(s) = \min_{s' \in nbrs(s)}$ ComputeCost($s, s', ccknbr(s, s')$);
46.                     $bptr(s) =$ argmin$_{s' \in nbrs(s)}$ComputeCost($s, s', ccknbr(s, s')$);
47.                     UpdateState($s$);
48. else
49.     $rhs_{min} = \infty$;
50.     for each state $s$ on a corner of $x$
51.         if $s$ was not visited before, $g(s) = rhs(s) = \infty$;
52.         else if ($rhs(s) < rhs_{min}$)
53.             $rhs_{min} = rhs(s)$; $s^\star = s$;
54.     if ($rhs_{min} \neq \infty$)
55.         insert $s^\star$ into OPEN with key($s^\star$);

**Main**()
56. $g(s_{start}) = rhs(s_{start}) = \infty$; $g(s_{goal}) = \infty$;
57. $rhs(s_{goal}) = 0$; OPEN $= \emptyset$;
58. insert $s_{goal}$ into OPEN with key($s_{goal}$);
59. forever
60.     ComputeShortestPath();
61.     Wait for changes in cell traversal costs;
62.     for all cells $x$ with new traversal costs $c$
63.         UpdateCellCost($x, c$);
```

Fig. 13. The Field D* Algorithm (optimized version).

corner states of the cell does not use the state with minimum rhs-value for one of its backpointers, yet still uses the traversal cost of the cell for its optimal action. As an example, consider Fig. 5(iii) and imagine the traversal cost of the bottom cell has decreased and the corner state of that cell with minimum rhs-value is not s or s_1. Fortunately, the probability of this situation arising is extremely low and if it does, the difference in path cost for the affected state s is not extreme (and is bounded by the maximum difference between path (iii) and the cheapest of paths (ii) and (iv)). For our results, we ran both this optimized version and the original and found there to be no difference in the overall path cost for any of our runs.

There is also a novel, significant optimization we can make to the general D* Lite family of algorithms. In the optimized version of D* Lite, when a state s is popped whose cost has increased (so that $rhs(s) > g(s)$), each affected neighbor of s recomputes its rhs-value. However, there is a chance that some of these neighbors will recompute their rhs-values several times, as their new successor states may be popped with increased costs, and so on. We can avoid these multiple rhs-value updates by just setting the new rhs-value of the neighbor state to infinity, rather than recomputing its true value. Since the state will be inserted into the *OPEN* list with a key value based on its g-value, not its rhs-value, this will not affect its priority. The one exception to this is when the state is already in the *OPEN* list with its g-value greater than or equal to its rhs-value. To account for this possibility, we thus check if this is the case, and if so, we recompute a new rhs-value for the state. States that are popped with increased path costs then recompute new rhs-values, as in the basic version of the algorithm. We have found this optimization to be extremely useful, both in updating traversal costs of cells and during replanning.

This final, optimized version of the algorithm is presented in Figure 13 and was used for generating the results presented in the following section[5]. As with our earlier versions of the algorithm, some sections of this version are not as efficient as they could be (e.g. there are repeated path cost calculations in lines {13 - 14} and again in lines {16 - 17}, {30 - 31}, and {42 - 43}). This is solely for ease of presentation; these sources of inefficiency should not appear in any implementation for which speed is a concern.

It is also possible to avoid the bulk of the processing involved in the ComputeCost() function at runtime by precomputing the result of the minimization step (Fig. 7 lines {13 - 24}) for every combination of c, b, and f, and storing these values in a lookup table. If we use integers to represent our g-values, and we have a discrete set of possible cell traversal costs, the number of elements in this table will be finite. In fact, the number of elements in the table will be:

$$\mathcal{N}_c^2 \times \mathcal{M}_c,$$

where $\mathcal{N}_c$ is the number of distinct traversal costs (including the infinite cost of traversing an obstacle cell) and $\mathcal{M}_c$ is the maximum traversal cost of any *traversable*, i.e. non-obstacle, cell.

The construction of this lookup table and the altered version of ComputeCost() are shown in Fig. 14. In this figure, $\mathcal{N}_c$ and $\mathcal{M}_c$ are as described above, and $cellcosts$ is an array specifying, for each distinct traversal cost identifier, the actual traversal cost associated with that identifier. The interpolation costs are stored in the lookup table $\mathcal{I}$. Differences between this version and the algorithm presented in Fig. 7 are highlighted in red.

We have also used this approach to create an efficient implementation of Field D* for the Mars Exploration Rovers. However, for this application we ignored the possibility of using the bottom cell b (Fig. 5(iii)) and only considered paths involving the center cell, so that our lookup table was indexed by just c and f. We also used a small set of non-linearly spaced cell traversal cost values (e.g. values of 255, 375, 510, 640, 1020, 1275, 1785). The memory required for storing the corresponding

[5] We have left out the key() and UpdateState() functions as they are the same as in Figure 12.

```
**ConstructInterpolationTable**($\mathcal{N}_c, \mathcal{M}_c, cellcosts$)
c_i = 0;   → $c_i = 0$;
while ($c_i < \mathcal{N}_c$)
    $c = cellcosts[c_i]$;
    $b_i = 0$;
    while ($b_i < \mathcal{N}_c$)
        $b = cellcosts[b_i]$;
        $f = 1$;
        while ($f \leq \mathcal{M}_c$)
            if ($f < b$)
                if ($c \leq f$)
                    $\mathcal{I}[c_i, b_i, f] = c\sqrt{2}$;
                else
                    $y = \min(\frac{f}{\sqrt{c^2 - f^2}}, 1)$;
                    $\mathcal{I}[c_i, b_i, f] = c\sqrt{1 + y^2} + f(1 - y)$;
            else
                if ($c \leq b$)
                    $\mathcal{I}[c_i, b_i, f] = c\sqrt{2}$;
                else
                    $x = 1 - \min(\frac{b}{\sqrt{c^2 - b^2}}, 1)$;
                    $\mathcal{I}[c_i, b_i, f] = c\sqrt{1 + (1 - x)^2} + bx$;
            $f = f + 1$;
        $b_i = b_i + 1$;
    $c_i = c_i + 1$;

**ComputeCost**($s, s_a, s_b$)
if ($s_a$ is a diagonal neighbor of $s$)
    $s_1 = s_b$; $s_2 = s_a$;
else
    $s_1 = s_a$; $s_2 = s_b$;
$c_i$ is traversal cost index of cell with corners $s, s_1, s_2$;
$b_i$ is traversal cost index of cell with corners $s, s_1$ but not $s_2$;
$c = cellcosts[c_i]$; $b = cellcosts[b_i]$;
if ($\min(c, b) = \infty$)
    $v_s = \infty$;
else if ($g(s_1) \leq g(s_2)$)
    $v_s = \min(c, b) + g(s_1)$;
else
    $f = g(s_1) - g(s_2)$;
    if ($f > \min(c, b)$)
        $v_s = \mathcal{I}[c_i, b_i, \mathcal{M}_c] + g(s_2)$;
    else
        $v_s = \mathcal{I}[c_i, b_i, f] + g(s_2)$;
return $v_s$;
```

Fig. 14. Using a Lookup Table to Store the Interpolation-based Cost Calculation

interpolation table was quite small (on the order of 25 KB) and the resulting implementation was very fast.

As a final note, as with all heuristic-based algorithms, the efficiency of Field D* depends heavily on the heuristic used to focus the search. For the results discussed in this paper, we used the standard Euclidean-cost metric and subtracted from this the maximum traversal cost of any (traversable) cell. This heuristic guarantees that the resulting solution will be optimal with respect to our linear interpolation assumption. However, in practice we have found that using the standard Euclidean-cost metric and dividing it by two is often much more efficient and produces solutions that are not noticeably different from the optimal solutions. The reason this latter approach tends to be more efficient, even though it often produces less informed heuristic values for each state, is because it reduces the number of times each state needs to be processed. With the former approach, it is not uncommon for a state to be popped off the queue and processed before one of its optimal successor states. This is rather inefficient, as the same state will have to be re-processed when its successor is finally processed and updates its path cost. With the latter approach, however, it is much less likely that this will occur. As a result, even though more states are often processed, these states are usually processed fewer times.

6 Path Extraction

As mentioned previously, once the cost of a path from the initial state to the goal has been computed, the path can be extracted by iteratively computing the next cell boundary point to move towards until the goal is reached. By using linear interpolation to approximate the path cost of any point on any edge, we can efficiently calculate the next point to transition to from any point anywhere in the grid.

However, because our linear interpolation-based approach provides only an approximation of the true path cost of any point in the grid, it pays to be a little bit careful about how we do this extraction, so as not to be unduly affected by cases in which the interpolation assumption breaks down.

In particular, we've found that we can avoid a couple approximation errors (see Section 8 for an example) by using a one-step lookahead when computing the next waypoint in the path. Basically, before transitioning to an edge point p for which we have computed an *interpolated* path cost, we calculate the path cost of p using the interpolated values of points on all other edges. Then, given this new path cost for p, we check if it is still the best point to use as the next waypoint in the path. This simple step greatly alleviates the most pathological instances of our interpolation assumption breaking down.

One other small modification we've found useful is to make sure that if one point in the path resides within some grid cell g, then the next point in the path can only be a corner state of g if the backpointer of this corner state points to at least one state not in g. Again, due to our use of interpolation this is not always naturally the case.

However, in practice it is most effective to use Field D* to compute the path *cost* of points in our grid, and use some local planner to compute the actual vehicle trajectory, taking into account vehicle dynamics and constraints [14]. For most of our systems, we couple an arc-based local planner with the global planning of Field D*.

7 Results

The true test of an algorithm is its practical effectiveness. We have found Field D* to be extremely useful for a wide range of robotic systems navigating through terrain of varying degrees of difficulty (see Fig. 4). Fig. 15 shows a simulated example of Field D* being used to navigate a robot through an initially unknown environment.

To provide a quantitative comparison of the performance of Field D* relative to D* Lite, we ran a number of replanning simulations in which we measured both the relative solution path costs and runtimes of the optimized versions of the two approaches. We generated 100 different 1000 $\times$ 1000 non-uniform cost grid environments in which each grid cell was assigned an integer traversal cost between 1 (free space) and 16 (obstacle). With probability 0.5 this cost was set to 1, otherwise it was randomly selected. For each environment, the initial task was to plan a path from the lower left corner to a randomly selected goal on the right edge. After this initial path was planned, we randomly altered the traversal costs of cells close to the agent (10% of the cells in the environment were changed) and had each approach repair its solution path. This represents a significant change in the information held by the agent and results in a large amount of replanning.

During initial planning, Field D* generated solutions that were on average 96% as costly as those generated by D* Lite, and took 1.7 times as long to generate these solutions. During replanning, the

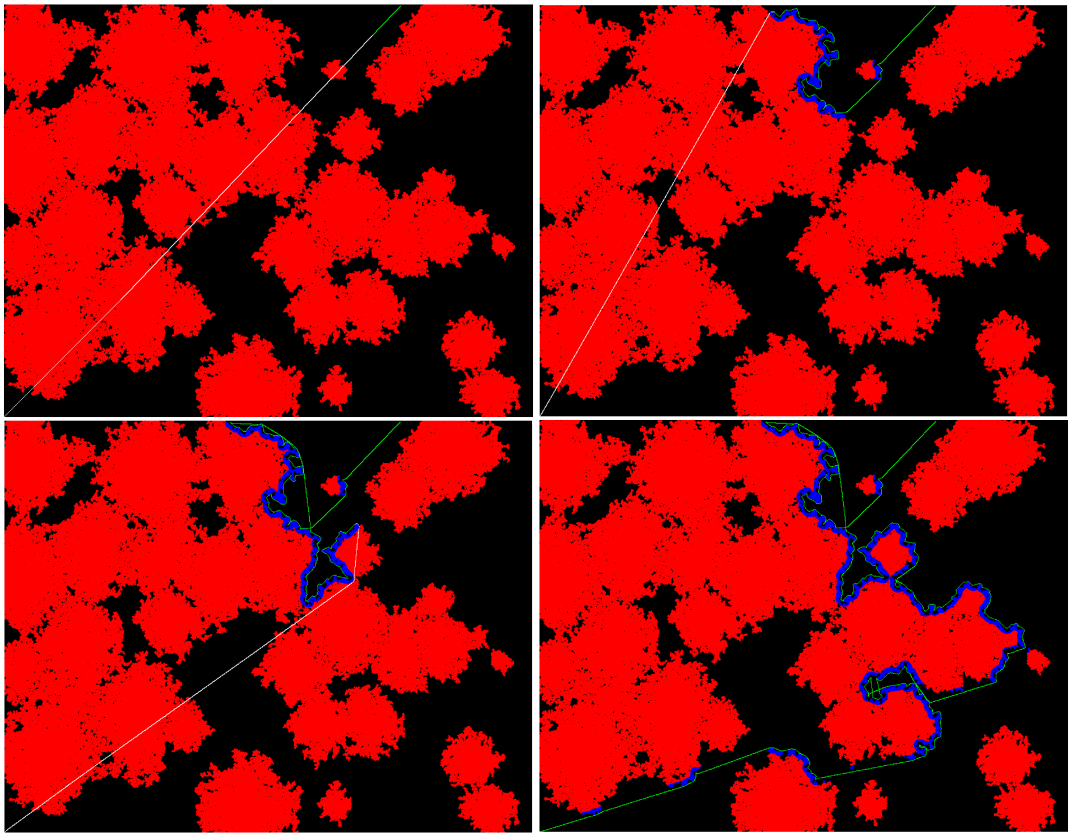

Fig. 15. A robot navigation example using Field D*. The robot starts at the top of the environment (about two-thirds of the way to the right edge), and plans a path (in white) to the bottom left corner, assuming the environment is empty. As it traverses its path (in green), it receives updated environmental information through an onboard sensor (observed obstacles shown in blue, actual obstacles shown in red). At each step it repairs its previous solution path based on this new information. Notice that the path segments are straight lines with widely-varying headings.

	D* Lite	Field D*
Initial Planning Time (s)	0.83	1.46
Initial Path Cost (relative)	1.00	0.96
Traversal Cost Update Time (s)	0.06	0.01
Replanning Time (s)	0.04	0.07
Replanned Path Cost (relative)	1.00	0.96

Table 1. The time and quality of solution associated with initial planning and replanning for Field D* and D* Lite. Also shown is the amount of time required to update the traversal cost of areas of the environment that have changed between replanning episodes. All values are averaged over 100 random environments with changes to the traversal cost of 10% of the environment. Reported run times are in seconds for a 1.5 GHz Powerbook G4 Processor.

results were similar: Field D* provided solutions on average 96% as costly and took 1.8 times as long. The average replanning runtime for Field D* on a 1.5 GHz Powerbook G4 was 0.07s. In practice, the algorithm is able to provide real-time performance on fielded systems.

8 Discussion and Conclusions

Although the results presented above show that Field D* generally produces less costly paths than regular grid-based planning, this is not guaranteed. It is possible to construct pathological scenarios where the linear interpolation assumption is grossly incorrect (for instance, if there is an obstacle in the cell to the right of the center cell in Fig. 5(i) and the optimal path for state s_2 travels above the obstacle and the optimal path for state s_1 travels below the obstacle). In such cases, the interpolated path cost of a point on an edge between two states may be either too low or too high. This in turn can affect the quality of the extracted solution path. However, such occurrences are very rare, and in none of our random test cases (nor any cases we have ever encountered in practice) was the path returned by Field D* more expensive than the grid-based path returned by D* Lite. In general, even in carefully-constructed pathological scenarios the path generated by Field D* is very close in cost to the optimal solution path.

Moreover, it is the difference in smoothness between the paths returned by Field D* and D* Lite rather than their relative costs that is the true advantage of Field D*. In both uniform and non-uniform cost environments, Field D* provides smooth, sensible paths for our agents to traverse. Consequently, as mentioned it is currently used as the path planner for a host of robotic platforms, from indoor systems to very rugged outdoor vehicles. Further, because its interpolation-based transition function is continuous, it has been used to successfully generate smooth, low cost paths in very low resolution grids when memory is limited.

In this paper we presented Field D*, an extension of classical grid-based planners that uses linear interpolation to efficiently produce less costly, more natural paths through grids. We have found Field D* to be extremely useful for mobile robot path planning in both uniform and non-uniform cost environments. We are currently looking at extending Field D* to interpolate over path headings, not just path costs, in order to produce solutions that minimize the amount of turning required over the path. We are also developing a multi-resolution version of the algorithm able to plan over very large distances and a 3D version for air vehicles.

9 Acknowledgments

This work was sponsored by the U.S. Army Research Laboratory, under contract "Robotics Collaborative Technology Alliance" (contract number DAAD19-01-2-0012). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements of the U.S. Government. Dave Ferguson is supported in part by a National Science Foundation Graduate Research Fellowship.

References

1. E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
2. P. Hart, N. Nilsson, and B. Rafael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
3. S. Koenig and M. Likhachev. D* Lite. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2002.

4. S. Koenig and M. Likhachev. Improved fast replanning for robot navigation in unknown terrain. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2002.
5. S. Koenig and M. Likhachev. Incremental A*. In *Advances in Neural Information Processing Systems*. MIT Press, 2002.
6. K. Konolige. A gradient method for realtime robot control. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*, 2000.
7. R. Larson. A survey of dynamic programming computational procedures. *IEEE Transactions on Automatic Control*, pages 767–774, 1967.
8. R. Larson and J. Casti. *Principles of Dynamic Programming, Part 2*. Marcel Dekker, New York, 1982.
9. S. LaValle. *Planning Algorithms*. Cambridge University Press (also available at http://msl.cs.uiuc.edu/planning/). To be published in 2006.
10. N. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.
11. R. Philippsen. *Motion Planning and Obstacle Avoidance for Mobile Robots in Highly Cluttered Dynamic Environments*. PhD thesis, EPFL, Lausanne, Switzerland, 2004.
12. R. Philippsen and R. Siegwart. An Interpolated Dynamic Navigation Function. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2005.
13. J. Sethian. A fast marching level set method for monotonically advancing fronts. *Applied Mathematics, Proceedings of the National Academy of Science*, 93:1591–1595, 1996.
14. A. Stentz and M. Hebert. A complete navigation system for goal acquisition in unknown environments. *Autonomous Robots*, 2(2):127–145, 1995.
15. Anthony Stentz. The Focussed D* Algorithm for Real-Time Replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.