Using Interpolation to Improve Path Planning: The Field D* Algorithm

Dave Ferguson and Anthony Stentz
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
e-mail: dif@cmu.edu, tony@cmu.edu

Received 1 September 2005; accepted 29 November 2005

We present an interpolation-based planning and replanning algorithm for generating low-cost paths through uniform and nonuniform resolution 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, $\pi/4$, $\pi/2$, etc.). As a result, even "optimal" grid-based planners produce unnatural, suboptimal paths. Our approach uses linear interpolation during planning to calculate accurate path cost estimates for arbitrary positions within each grid cell and produce paths with a range of continuous headings. Consequently, it is particularly well suited to planning low-cost trajectories for mobile robots. In this paper, we introduce a version of the algorithm for uniform resolution grids and a version for nonuniform resolution grids. Together, these approaches address two of the most significant shortcomings of grid-based path planning: the quality of the paths produced and the memory and computational requirements of planning over grids. We demonstrate our approaches on a number of example planning problems, compare them to related algorithms, and present several implementations on real robotic systems. © 2006 Wiley Periodicals, Inc.

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.

In robotics, it is common to improve efficiency by approximating this grid with a graph, where nodes are placed at the center of each grid cell and edges connect nodes within adjacent grid cells. Many algorithms exist for planning paths over such graphs. Dijkstra's algorithm computes paths from every node to a specified goal node (Dijkstra, 1959). 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 (Hart, Nilsson, & Rafael, 1968; Nilsson, 1980). D*, In-

Figure 1. Some robots that currently use Field D* for global path planning. These range from indoor planar robots (the Pioneers) to outdoor robots able to operate in harsh terrain (the XUV).

cremental A*, and D* Lite are extensions of A* that incrementally repair solution paths when changes occur in the underlying graph (Stentz, 1995; Koenig & Likhachev, 2002a, b, c). 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 from each node in the graph. For instance, given a graph extracted from a uniform resolution 2D grid, a path planned in the manner described above restricts the agent's heading to increments of $\pi/4$. This results in paths that are suboptimal in length and difficult to traverse in practice. Further, even when these paths are used in conjunction with a local arc-based planner [e.g., as in the RANGER system (Kelly, 1995; Stentz & Hebert, 1995)], they can still cause the vehicle to execute expensive trajectories involving unnecessary turning.

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 low-cost paths that eliminate unnecessary turning. The paths are optimal given a linear interpolation assumption and very effective in practice. This algorithm is currently being used by a wide range of fielded robotic systems (see Figure 1).

A second significant limitation of current planners (including even Field D*) arises from their use of uniform resolution grids to represent the environment. Often, it is unfeasible to use such grids because of the large amount of memory and computation required to store and plan over these structures. Instead, nonuniform resolution representations may be more appropriate, for instance when the environment is very large and sparsely populated.

In this paper, we thus also present Multi-resolution Field D*, an extension of Field D* able to plan over nonuniform resolution grids. This algorithm produces very cost-effective paths for a fraction of the memory and, often, for a fraction of the time required by uniform resolution grid-based approaches.

We begin by discussing the limitations of paths produced using classical uniform resolution grid-based planners and the 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. We provide several example illustrations and applications of Field D*, along with a comparison to competing approaches. In Sections 7 and 8 we describe Multi-resolution Field D*, an extension to the algorithm that is able to plan over nonuniform resolution grids. We conclude with discussion and extensions.

2. PLANNING OVER UNIFORM RESOLUTION GRIDS

Consider a robotic ground vehicle navigating an outdoor environment. We can represent this environment as a uniform resolution 2D traversability grid, in which cells are given a cost per unit of traverse (traversal cost) reflecting the difficulty of navigating the respective area of the environment. If this traversability grid encodes the configuration space costs (i.e., the traversal costs have been expanded to reflect the physical dimensions of the vehicle), then planning a path for the robot translates to generating a trajectory

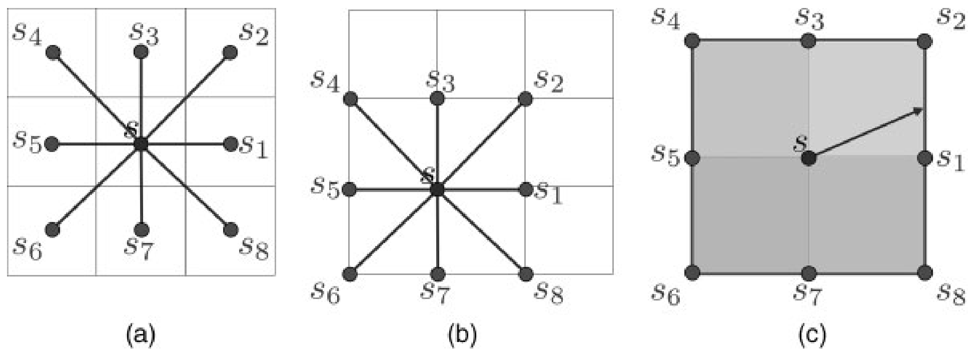


Figure 2. (a) A standard 2D grid used for global path planning in which nodes reside at the centers of the grid cells. The arcs emanating from the center node represent all the possible actions that can be taken from this node. (b) A modified representation used by Field D*, in which nodes reside at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

through this grid for a single point. A common approach used in robotics for performing this planning is to combine an approximate *global* planner with an accurate *local* planner (Kelly, 1995; Stentz and Hebert, 1995; Brock & Khatib, 1999; Singh *et al.*, 2000). The global planner computes paths through the grid that ignore the kinematic and dynamic constraints of the vehicle. Then, the local planner takes into account the constraints of the vehicle and generates a set of feasible local trajectories that can be taken from its current position. To decide which of these trajectories to execute, the robot evaluates both the cost of each local trajectory and the cost of a global path from the end of each trajectory to the robot's desired goal location.

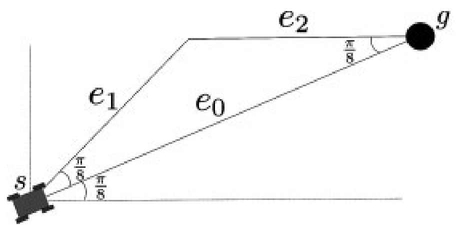
To formalize the global planning task, we need to define more precisely some concepts already introduced. First, each cell in the grid has assigned to it some real-valued traversal cost that is greater than zero. The cost of a line segment between two points within a cell is the Euclidean distance between the points multiplied by the traversal cost of the cell. The cost of any path within the grid is the sum of the costs of its line segments through each cell. Then, the global planning task (involving a uniform resolution grid) can be specified as follows.

The Global Planning Task: *Given a region in the plane partitioned into a uniform grid of square cells $\mathcal{T}$, an assignment of traversal costs $c:\mathcal{T}\rightarrow(0,+\infty]$ to each cell, and two points s_{start} and s_{goal} within the grid, find the path within the grid from s_{start} to s_{goal} with minimum cost.*

This task can be seen as a specific instance of the Weighted Region Problem (Mitchell & Papadimitriou, 1991), where the regions are uniform square tiles. A number of algorithms exist to solve this problem in the computational geometry literature [see Mitchell (2000) for a good survey]. In particular, Mitchell & Papadimitriou (1991) and Rowe & Richbourg (1990) present approaches based on Snell's law of refraction that compute optimal paths by simulating a series of light rays that propagate out from the start position and refract according to the different traversal costs of the regions encountered. These approaches are efficient for planning through environments containing a small number of homogenous-cost regions, but are computationally expensive when the number of such regions is very large, as in the case of a uniform grid with varying cell costs.

Because of the computational expense associated with planning optimal paths through grids, researchers in robotics have focused on basic approximation algorithms that are extremely fast. The most popular such approach is to approximate the traversability grid as a discrete graph, then generate paths over the graph. A common way to do this is to 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 transitions through and the length of the edge. Figure 2(a) shows this node and edge extraction process for one cell in a uniform resolution 2D grid.

We can then plan over this graph to generate paths from the robot's initial location to a desired goal location. As mentioned previously, a number of efficient algorithms exist for performing this planning,



$$|e_1| = |e_2|$$
$$\rightarrow |e_0|^2 = 2|e_1|^2 - 2|e_1|^2 \cos \tfrac{3\pi}{4}$$
$$\rightarrow |e_1| + |e_2| \approx 1.08|e_0|$$

Figure 3. A uniform resolution 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 $\pi/8$ and lies perfectly between the two nearest grid-based headings of 0 and $\pi/4$. This result is independent of the resolution of the grid.

such as A* for initial planning and D* and its variants for replanning (Hart *et al.*, 1968; Nilsson, 1980; Stentz, 1995; Koenig & Likhachev, 2002b). Unfortunately, paths produced using this graph are restricted to headings of $\pi/4$ 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 Figure 3). 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 $\pi/4$, traditional grid-based planners would return a path that has the robot first turn to attain the nearest grid heading, move some distance along this heading, and then turn $\pi/4$ 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. Such global paths, when coupled with the results of a local planner, cause the robot to behave suboptimally. Further, this limitation of traditional grid-based planners is not alleviated by increasing the resolution of the grid.

Sometimes it is possible to reduce the severity of this problem by postprocessing 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 Figure 4. Indeed, for nonuniform cost environments such postprocessing can often *increase* the cost of the path.

A more comprehensive postprocessing approach is to take the result of the global planner and use it to seed a higher dimensional planner that incorporates the kinematic or dynamic constraints of the robot. Stachniss & Burgard (2002) present an approach that takes the solution generated by the global planner and uses it to extract a local waypoint to use as the goal for a 5D trajectory planner. The search space of the 5D planner is limited to a small area surrounding the global solution path. Likhachev *et al.* (2003, 2005) present an approach that uses the cost-to-goal value function of the global planner to focus an anytime global 4D trajectory planner. Their approach improves the quality of the global trajectory while deliberation time allows. However, these higher dimensional approaches can be much more computationally expensive than standard grid-based planners and are still influenced by the results of the initial grid-based solution.

Recently, robotics researchers have looked at more sophisticated methods of obtaining better paths

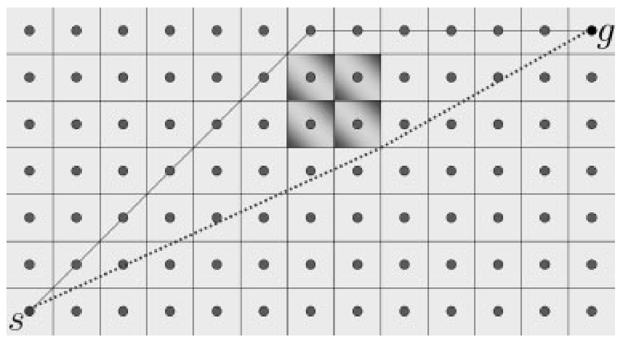


Figure 4. 2D grid-based paths cannot always be shortened in a postprocessing phase. Here, the grid-based path from s to g (top, in black) cannot be shortened because there are four obstacle cells (shaded). The optimal path is shown in blue/dashed.

through grids without sacrificing too much of the efficiency of the classic grid-based approach described above. Konolige (2000) presents an interpolated planner that first uses classic grid-based planning to construct a cost-to-goal value function over the grid and then interpolates this result to produce a shorter path from the initial position to the goal. This method results in shorter, less-costly paths for agents to traverse but does not incorporate the reduced path cost into the planning process. Consequently, the resulting path is not necessarily as good as the path the algorithm would produce if interpolated costs were calculated during planning. Further, if we are computing paths from several locations (which is common when combining the global planner with a local planner), then this postprocessing interpolation step can be expensive. Also, this approach provides no replanning functionality to update the solution when new information concerning the environment is received.

Philippsen & Siegwart (2005) present an algorithm based on Fast Marching Methods (Sethian, 1996) 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. This technique has been shown to generate nice paths in indoor environments (Philippsen, 2004; Philippsen & Siegwart, 2005). However, the search is not focused 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 grids with widely varying cell traversal costs.

The idea of using interpolation to produce better value functions for discrete samples drawn from 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 (Larson, 1967; Larson & Casti, 1982; LaValle, 2006). However, as LaValle (2006) points out, this becomes difficult when the action space is also continuous, as solving for the value of a state now requires minimizing over an 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 near-optimal paths which eliminate unnecessary turning. It relies upon an efficient, closed-form solution to the above minimization problem for 2D grids, which we introduce in the next section. This method produces much straighter, less-costly paths than classical grid-based planners without sacrificing real-time performance. As with D^* and D^* Lite, our approach focuses its search towards the most relevant areas of the state space during both initial planning and replanning. Further, it takes into account local variations in cell traversal costs and produces paths that are optimal given a linear interpolation assumption. As the resolution of the grid increases, the solutions returned by the algorithm improve, approaching true optimal paths.

3. IMPROVING COST ESTIMATION THROUGH INTERPOLATION

The key to our algorithm is a novel method for 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 the cheapest path from the node to the goal. In classical grid-based planning this value is computed as

$$g(s) = \min_{s' \in nbrs(s)} [c(s,s') + g(s')], \tag{1}$$

where $nbrs(s)$ is the set of all neighboring nodes of s (see Figure 2), $c(s,s')$ 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

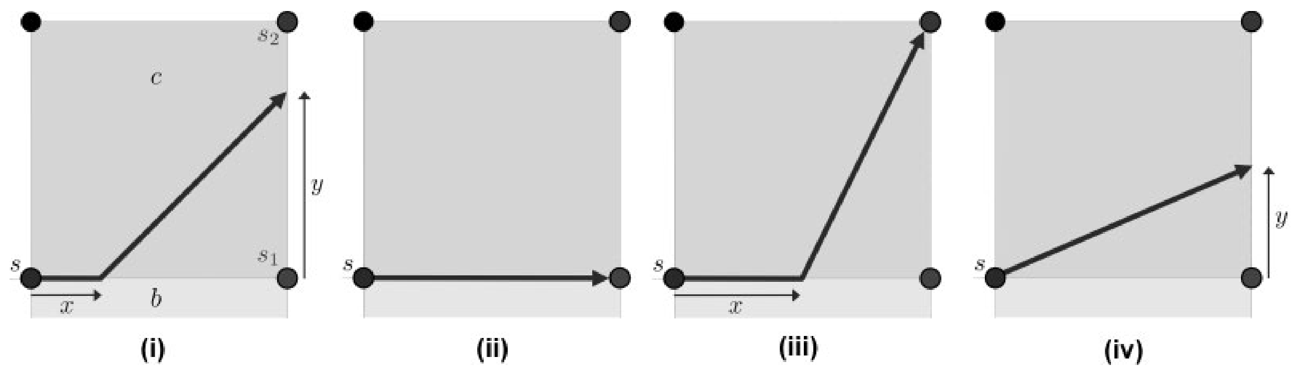


Figure 5. Computing the path cost of node s using the path cost of two of its neighbors, s_1 and s_2, and the traversal costs c of the center cell and b of the bottom cell. Illustrations (ii)–(iv) show the possible optimal paths from s to edge $\overrightarrow{s_1 s_2}$.

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 Figure 2(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 Figure 2(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 Figure 5).

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 $g(s_2)$:

$$g(s_y) = yg(s_2) + (1 - y)g(s_1), \tag{2}$$

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

$$\min_{x,y}[bx + c\sqrt{(1-x)^2 + y^2} + yg(s_2) + (1 - y)g(s_1)], \tag{3}$$

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 Figure 5(i)]. 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.

Let (x^*, y^*) be a pair of values for x and y that solve the above minimization. Because of our use of 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 $x^* = 0$ or $y^* = 1$. If there is no component of the path that cuts through the center cell, then $y^* = 0$. We prove this below.

Lemma 1: Let $\{x^*, y^*\} = \text{argmin}_{x,y}[bx + c\sqrt{(1-x)^2 + y^2} + yg(s_2) + (1-y)g(s_1)], x \in [0,1], y \in [0,1]$, *where* s_1, s_2, b, *and* c *are as defined in the text, and* $g(s_1)$ *and* $g(s_2)$ *are*

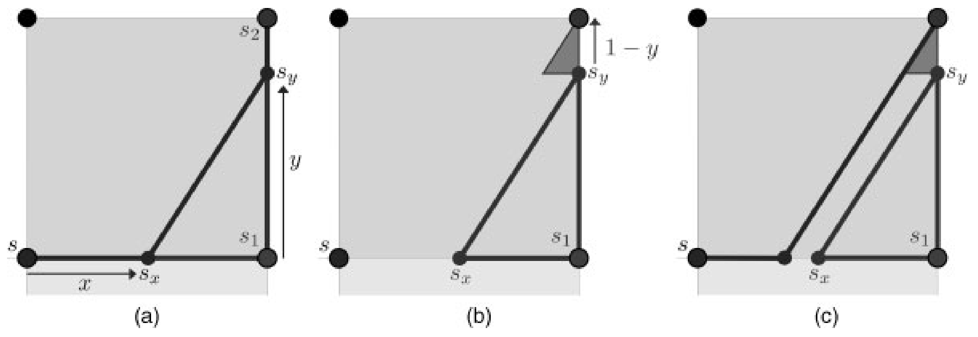


Figure 6. (a) Imagine the blue path is the optimal path, traveling along the bottom edge to s_x, then across the center cell to s_y, then up the right edge to s_2. Notice the triangle formed between vertices s_x, s_1, and s_y. (b) We can create a scaled version of this triangle with a vertical edge length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

optimal path costs for s_1 and s_2 given our linear interpolation assumption. Then $x^* \in \{0,1\}$ or $y^* \in \{0,1\}$.

Proof: To prove this, it is useful to transform our original path planning problem. If we pretend that the cost of traversing the edge $\overrightarrow{s_1 s_2}$ is the difference in cost between the two nodes s_1 and s_2, then our original problem can be solved by finding the cheapest path from s through s_2. To see this, let $f=g(s_1)-g(s_2)$. Then Eq. (3) can be written as

$$\min_{x,y}[bx + c\sqrt{(1-x)^2+y^2} + (1-y)f + g(s_2)]. \tag{4}$$

This is equivalent to computing the minimum-cost path from s through s_2, where portions of the path traveled along the edge $\overrightarrow{s_1 s_2}$ (e.g., between s_y and s_2) use a traversal cost of f.[1] If $f<0$, then it is always cheapest to take a direct route to s_1, then travel along the entirety of the edge $\overrightarrow{s_1 s_2}$.

Now, assume the optimal path involves traveling along parts of both the bottom and right edges *and* cutting across part of the center cell [i.e., $x^* \in (0,1)$ and $y^* \in (0,1)$]. 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 Figure 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:

$$c\sqrt{(1-x)^2+y^2} \leqslant (1-x)b + yf. \tag{5}$$

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).

Assume without loss of generality that $(1-y) < x$, so that s_y is closer to s_2 than s_x is to s. Consider a scaled version of this triangle with a vertical edge of length $(1-y)$ [see Figure 6(b)]. The horizontal edge of this triangle will have length $(1-y)[(1-x)/y]$. 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 $x-(1-y)[(1-x)/y]$ then straight to s_2, and the cost of

[1]If $f>r$, where r is the traversal cost of the right cell, then the path cost $g(s_1)$ is clearly suboptimal. This is a contradiction.

[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.

this path would be *less* than the cost of our original (optimal) path [see Figure 6(c)]. This is a contradiction. Thus, it is not possible that *both* x^* and y^* will be in the range (0, 1). □

From our proof, we know that 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] Instead, the path will either travel along the entire bottom edge to s_1 [Figure 5(ii)], or will travel a distance x along the bottom edge then take a straight-line path directly to s_2 [Figure 5(iii)], or will take a straight-line path from s to some point s_y on the right edge [Figure 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 travels straight to s_1 and will have a cost of $[\min(c,b) + g(s_1)]$ [Figure 5(ii)]. If $f = b$, then the cost of a path using some portion of the bottom edge [Figure 5(iii)] will be equivalent to the cost of a path using none of the bottom edge [Figure 5(iv)]. We can solve for the value of y that minimizes the cost of the latter path as follows.

First, let $k = f = b$. The cost of a path from s through edge $\overrightarrow{s_1 s_2}$ is

$$c\sqrt{1+y^2} + k(1-y) + g(s_2). \tag{6}$$

Taking the derivative of this cost with respect to y and setting it equal to zero yields

$$y^* = \sqrt{\frac{k^2}{c^2 - k^2}}. \tag{7}$$

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

[3]Note that it may be possible that the optimal path to s_2 involves traveling along the vertical edge from s for some distance, then cutting across to s_2. However, this possibility is examined when computing the path cost from s to neighbors s_2 and s_3 (see Figure 2). We can thus restrict our attention when computing 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.

```
ComputeCost($s, s_a, s_b$)
if ($s_a$ is a diagonal neighbor of $s$)
  $s_1 = s_b$; $s_2 = s_a$;
else
  $s_1 = s_a$; $s_2 = s_b$;
$c$ is traversal cost of cell with corners $s, s_1, s_2$;
$b$ is traversal cost of cell with corners $s, s_1$ but not $s_2$;
if ($\min(c, b) = \infty$)
  $v_s = \infty$;
else if ($g(s_1) \le g(s_2)$)
  $v_s = \min(c, b) + g(s_1)$;
else
  $f = g(s_1) - g(s_2)$;
  if ($f \le b$)
    if ($c \le f$)
      $v_s = c\sqrt{2} + g(s_2)$;
    else
      $y = \min(\frac{f}{\sqrt{c^2 - f^2}}, 1)$;
      $v_s = c\sqrt{1 + y^2} + f(1 - y) + g(s_2)$;
  else
    if ($c \le b$)
      $v_s = c\sqrt{2} + g(s_2)$;
    else
      $x = 1 - \min(\frac{b}{\sqrt{c^2 - b^2}}, 1)$;
      $v_s = c\sqrt{1 + (1 - x)^2} + bx + g(s_2)$;
return $v_s$;
```

Figure 7. The interpolation-based path cost calculation.

$y^* = 1 - x^*$ into the above equation. The resulting algorithm for computing the minimum-cost path from s through an edge between *any* two consecutive neighbors s_a and s_b is provided in Figure 7. Given the minimum-cost paths from s through each of its eight neighboring edges, we can compute the path cost for s to be the cost of the cheapest of these paths. The associated path is optimal given our linear interpolation assumption.

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 low-cost paths. Figure 8 presents our simplest formulation of (uniform resolution) Field D*, an incremental replanning

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

UpdateNode(s)
 2  if s was not visited before, $g(s) = \infty$;
 3  if ($s \neq s_{goal}$)
 4    $rhs(s) = \min_{(s', s'') \in connbrs(s)}$ ComputeCost($s, s', s''$);
 5  if ($s \in OPEN$) remove s from OPEN;
 6  if ($g(s) \neq rhs(s)$) insert s into OPEN with key(s);

ComputeShortestPath()
 7  while ($\min_{s \in OPEN}$(key(s)) $\dot{<}$ key($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
 8    remove node s with the minimum key from OPEN;
 9    if ($g(s) > rhs(s)$)
10      $g(s) = rhs(s)$;
11      for all $s' \in nbrs(s)$ UpdateNode($s'$);
12    else
13      $g(s) = \infty$;
14      for all $s' \in nbrs(s) \cup \{s\}$ UpdateNode($s'$);

Main()
15  $g(s_{start}) = rhs(s_{start}) = \infty; g(s_{goal}) = \infty$;
16  $rhs(s_{goal}) = 0; OPEN = \emptyset$;
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 node s on a corner of x
23        UpdateNode(s);
```

Figure 8. The Field D* algorithm (basic D* Lite version).

algorithm that incorporates these interpolated path costs. This version of Field D* is based on D* Lite.[4]

In this figure, *connbrs*(*s*) contains the set of consecutive neighbor pairs of node 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 Figure 2(c). Apart from this construction, notation follows the D* Lite algorithm: *g*(*s*) is the current path cost of node *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 nodes [i.e., nodes *s* for which $g(s) \neq rhs(s)$] in increasing order of *key* values (line 1), s_{start} is the initial agent node, and s_{goal} is the goal node. $h(s_{start}, s)$ is a heuristic estimate of the cost of a path

[4]Differences between Field D* and D* Lite appear on lines 4 and 20–23. 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, the heuristic value $h(s_{start}, s)$ must be small enough that when added to the cost of any edge incident on *s* it is still not greater than a minimum cost path from s_{start} to *s*.

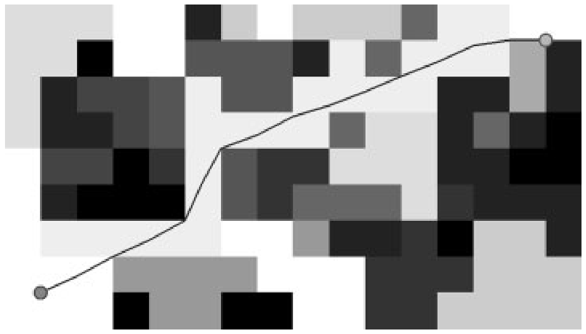


Figure 9. A close-up of a path planned using Field D* showing individual grid cells. Darker cells have larger traversal costs. Notice that the path is not limited to entering and exiting cells at corner points.

from s_{start} to *s*. Because the key value of each node contains two quantities a lexicographic ordering is used: key(*s*) < 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 Koenig & Likhachev (2002b, a). Also, the termination and correctness of the Field D* algorithm follow directly from D* Lite and the analysis of the cost calculation provided in Section 3.

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

Once the cost of a path from the initial node to the goal has been calculated, the path can be extracted by starting at the initial position and iteratively computing the cell boundary point to move to next. Because of our interpolation-based cost calculation, 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 entire path and calculating accurate path costs from noncorner points. See Section 5 for more details concerning path extraction.

Figures 9–12 illustrate paths produced by Field D* through three uniform resolution grids. In each of these figures, darker areas represent regions that are more costly to traverse. Notice that, unlike paths produced using classical grid-based planners, the paths produced using Field D* are not restricted to a small

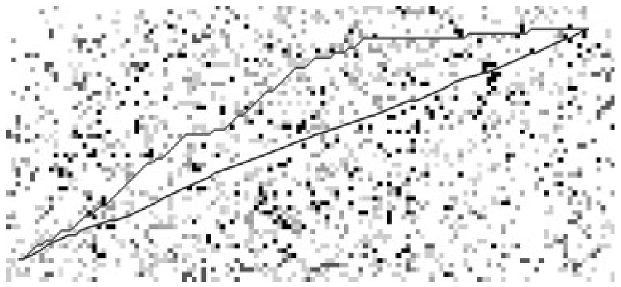


Figure 10. Paths produced by D* Lite (top) and Field D* (bottom) in a 150×60 uniform resolution grid. Again, darker cells have larger traversal costs.

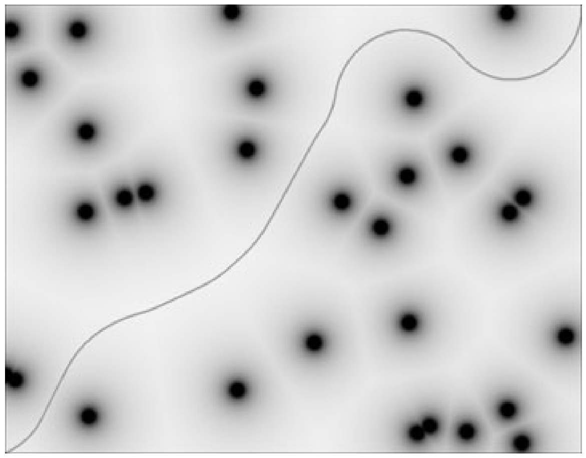


Figure 11. Field D* planning through a potential field of obstacles.

set of headings. As a result, Field D* provides lower-cost paths through both uniform and nonuniform cost environments.

5. PATH EXTRACTION

The linear interpolation assumption made by Field D* generally produces accurate path cost approximations. However, this assumption is clearly not perfect, and there are situations in which it is seriously violated. In order to avoid returning invalid or grossly suboptimal paths in these cases, the path extraction process must be performed carefully.

In particular, we have found that we can reduce errors due to our interpolation-based approximation 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 a simple interpolated path cost, we calculate a more accurate approximation of the path cost of p. We do this by looking to its neighboring edges and computing a locally optimal path from p given the path costs of the endpoint nodes of these edges and interpolated path costs for points along the edges (as in Figure 7). 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 reduces the effects of interpolation error on our path, particularly in pathological cases such as the one discussed in Section 6.

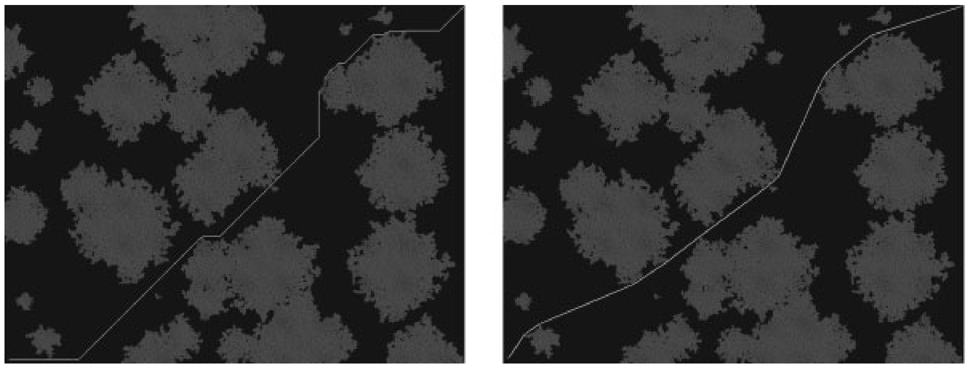


Figure 12. Paths produced by D* Lite (left) and Field D* (right) in a 900×700 binary cost grid. Here, obstacles are shown in dark gray and traversable area is shown in black.

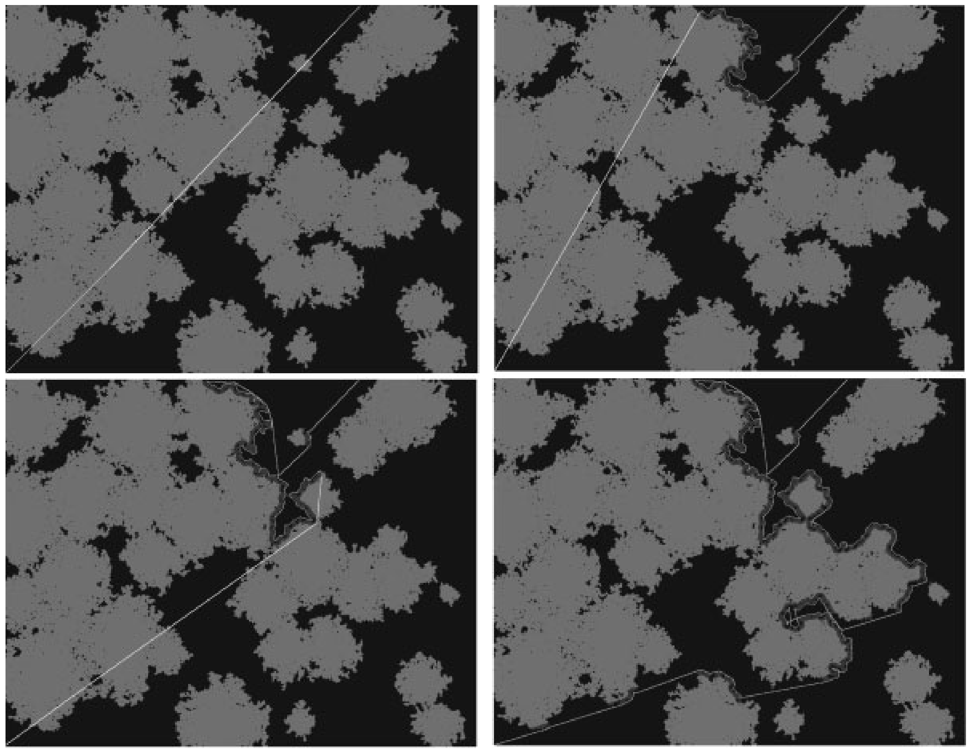


Figure 13. A robot navigation example using Field D*. The robot starts at the top of the environment (about two-thirds of the way to the right edge) and plans a path (in white) to the bottom left corner, assuming the environment is empty. As it traverses its path (shown in light gray), it receives updated environmental information through an onboard sensor (observed obstacles shown in dark gray, actual obstacles shown in gray, traversable area shown in black). 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.

One other small modification we have 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 of g if the computed path from the node at that corner does not transition back through g. Again, due to our use of interpolation this is not always naturally the case.

In practice it is most effective to use Field D* to compute the cost-to-goal value function over the grid, and use some local planner to compute the actual vehicle trajectory, as described in Section 2.

6. FIELD D* 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 Figure 1). Figure 13 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 1000×1000 nonuniform 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

Table I. 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.

	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

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.

The results from these experiments are shown in Table I. 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 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 0.07 s. In practice, the algorithm is able to provide real-time performance for fielded systems.

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 Figure 5(i) and the optimal path for node s_2 travels above the obstacle and the optimal path for node s_1 travels below the obstacle]. In such cases, the interpolated path cost of a point on an edge between two nodes 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 ability of Field D* to plan paths with a continuous range of headings, rather than simply its lower-cost solutions, that is its true advantage over regular grid-based planners. In both uniform and nonuniform cost environments, Field D* provides direct, sensible paths for our agents to traverse.

7. MULTI-RESOLUTION FIELD D*

Thus far, we have focused on algorithms appropriate for planning through uniform resolution grids. Although such grids are a common representation in mobile robotics, they can be very memory intensive. This is because the entire environment must be represented at the highest resolution for which information is available. For instance, consider a robot navigating a large outdoor environment with a prior overhead map. The initial information contained in this map may be coarse. However, the robot may be equipped with onboard sensors that provide very accurate information about the area within some field of view of the robot. Using a uniform resolution grid-based approach, if *any* of the high-resolution information obtained from the robot's onboard sensors is to be used for planning, then the *entire environment* needs to be represented at a high resolution, including the areas for which only the low-resolution prior map information is available. Storing and planning over this representation can require vast amounts of memory.

7.1. Multi-Resolution Grid Representations

A number of techniques have been devised to remedy this problem. One popular approach is to use quadtrees rather than uniform resolution grids (Samet, 1982; Kambhampati & Davis, 1986). Quadtrees offer a compact representation by allowing large constant-cost regions of the environment to be modeled as single cells. They thus represent the environment using grids containing cells of varying sizes, known as nonuniform resolution grids or *multi-resolution* grids.

However, paths produced using quadtrees and traditional quadtree planning algorithms are again constrained to transitioning between the centers of adjacent cells and can be grossly suboptimal. More recently, framed quadtrees have been used to alleviate this problem somewhat (Chen, Szczerba, & Uhran, 1995; Yahja, Singh, & Stentz, 2000). Framed quadtrees add cells of the highest resolution around the boundary of each quadtree region and allow transitions between these boundary cells. As a result, the paths produced can be much less costly, but the computation and memory required can be large due to the overhead of the representation (and in pathological cases can be significantly more than is required by a full high-resolution representation). Also, segments of the path between adjacent high-resolution cells suffer from the same limitations as classical uniform resolution grid approaches, since interpolation is not used.

As with uniform resolution grids, path planning through a multi-resolution grid is another special case of the Weighted Region Problem, and the general, purpose algorithms discussed in Section 2 are applicable. However, as the number of cells increases (e.g., as the agent observes information at a high resolution through its sensors and updates its representation), these algorithms become very computationally expensive. Ideally, we would like a planning algorithm that is as efficient as traditional multi-resolution grid-based algorithms but produces better paths.

In multi-resolution grids, interpolation has the potential to be of huge benefit, since it can eliminate the requirement that paths transition between the center points of adjacent grid cells. Further, it can be used without adding any extra cells or modifications to the grid.

In the following sections, we present an approach that combines the ideas of interpolation and nonuniform resolution grid representations to provide very cost-effective paths for a fraction of the memory and, often, for a fraction of the *time* required by uniform resolution grid approaches.

7.2. Combining Interpolation with Multi-Resolution Grids

We can use the same basic interpolation approach used by Field D* in uniform resolution grids to provide accurate path costs for nodes in multi-resolution grids. To begin with, we assign nodes to the corners of every grid cell, as in the uniform resolution case. We define the neighboring edges of a node s to be all edges that can be reached from s via a straight-line path for which s is *not* an endpoint [see Figure 14 (left)]. We allow each node to transition to any point on any of its neighboring edges. The rationale here is that the optimal path from s must pass through one of these neighboring edges, so if we knew the optimal path cost of every point on any of these edges we could compute the optimal path cost for s.

The main difference between the uniform resolution and nonuniform resolution grid scenarios is that, in the uniform resolution case, each node s has exactly eight neighboring edges of uniform length, while in the nonuniform case, a node may have many more neighboring edges with widely varying lengths. However, linear interpolation can still be used to approximate the path costs of points along these edges, exactly as in the uniform resolution case.

As a concrete example of how we compute the path cost of a node in a multi-resolution grid, we now focus our attention on a grid containing cells of two different resolutions: high and low resolution. This two-resolution case addresses the most common navigation scenario we are confronted with: a low-resolution prior map is available and the robot is equipped with high-resolution onboard sensors. Although we restrict our attention to this scenario, the approach is general and can be used with arbitrarily many different resolutions.

In a grid containing two different resolutions, each node can reside on the corner of a low-resolution cell, the corner of a high-resolution cell, and/or the edge of a low-resolution cell. Examples of each of these possibilities can be seen in Figure 14(b): the white node is the corner of a low-resolution cell and the gray node is the corner of a high-resolution cell *and* on the edge of a low-resolution cell. Let us look at each of these possibilities in turn.

First, imagine we have a node that resides on the corner of a low-resolution cell. We can calculate the least-cost path from the node through this cell by looking at all the points on the boundary of this cell and computing the minimum cost path from the node using any of these points. We can approximate the cost of this path by using linear interpolation to provide the path cost of arbitrary boundary points, exactly as in uniform resolution Field D*. However, some of the boundary may be comprised of high-resolution nodes. In such a case, we can either use interpolation between adjacent high-resolution

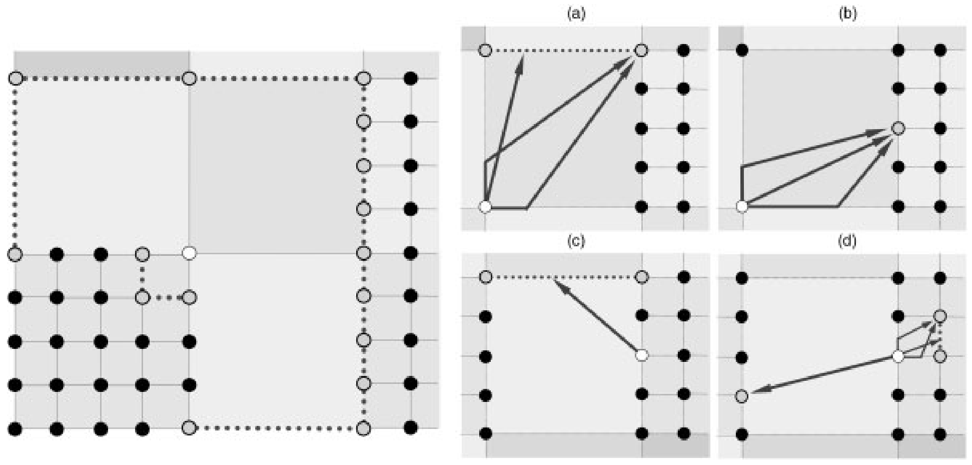


Figure 14. (Left) The neighboring edges (dashed in dark gray, along with their endpoints in gray) from a given node (white) in a grid containing cells with two different resolutions: low resolution and high resolution. (a, b) Some of the possible paths to a neighboring edge/node from a low-resolution node. On the left are the possible optimal path types (in dark gray) through the top low-resolution edge (dashed in dark gray) and its endpoint nodes (in gray). Linear interpolation is used to compute the path cost of any point along the top edge. On the right are the possible optimal path types (in dark gray) to one neighboring high-resolution corner node (in gray). (c, d) Some of the possible paths (in dark gray) from a high-resolution corner node (white) to a neighboring low-resolution edge (c) and to a high-resolution node (d, left) and edge (d, right).

nodes and allow the path to transition to any point on an adjacent high-resolution edge, or we can restrict the path to transitioning to one of the high-resolution nodes. The former method provides more accurate approximations, but it is slightly more complicated and less efficient. Depending on the relative sizes of the high-resolution cells and the low-resolution cells, either of these approaches may be appropriate. For instance, if the high-resolution cells are much smaller than the low-resolution cells, then interpolating across the adjacent high-resolution edges when computing the path from a low-resolution node is not that critical, as there will be a wide range of heading angles available just from direct paths to the adjacent high-resolution nodes. However, if the high-resolution cells are not significantly smaller than the low-resolution cells, then this interpolation becomes important, as it allows much more freedom in the range of headings available to low-resolution nodes adjacent to high-resolution nodes. In Figure 14 we illustrate the latter, simpler approach, where interpolation is used to compute the path cost of points on neighboring, strictly low-resolution edges [e.g., the top edge in (a)], and paths are computed to each neighboring high-resolution node [e.g., the gray node on the right edge in (b)].

For nodes that reside on the corner of a high-resolution cell, we can again use interpolation as presented in Section 3 to approximate the cost of the cheapest path through the high-resolution cell [see the paths to the right edge in (d)]. Finally, for nodes that reside on the edge of a low-resolution cell, we can use a similar approach as in our low-resolution corner case. Again, we look at the boundary of the low-resolution cell and use interpolation to compute the cost of points on strictly low-resolution edges [e.g., the top edge in (d)], and for each high-resolution edge we can choose between using interpolation to compute the cost of points along the edge, or restricting the path to travel through one of the endpoints of the edge. The latter approach is illustrated for computing a path through the left edge in (d).

Thus, for each node, we look at all the cells that it resides upon as either a corner or along an edge and compute the minimum path cost through each of these cells using the above approximation technique. We then take the minimum of all of these costs and use this as the path cost of the node.

Pseudocode of this technique is presented in Figure 15. In this figure, P_e is the (infinite) set of all

```
ComputePathCost($s$)
 1  $v_s = \infty$;
 2  for each cell $x$ upon which $s$ resides
 3    if $x$ is a high-resolution cell
 4      for each neighboring edge $e$ of $s$ that is on the boundary of $x$
 5        $v_s = \min(v_s, \min_{p \in P_e}(c(s,p) + g^i(p)))$;
 6    else
 7      for each neighboring edge $e$ of $s$ that is on the boundary of $x$
 8        if $e$ is a low-resolution edge
 9          $v_s = \min(v_s, \min_{p \in P_e}(c(s,p) + g^i(p)))$;
10        else
11          $v_s = \min(v_s, \min_{p \in EP_e}(c(s,p) + g(p)))$;
12  return $v_s$;
```

Figure 15. Computing the path cost of a node s in a grid with two resolutions.

points on edge e, EP_e is a set containing the two endpoints of edge e, $g^i(p)$ is an approximation of the path cost of point p (calculated through using linear interpolation between the endpoints of the edge p resides on), $c(s,p)$ is the cost of a minimum-cost path from s to p, and $g(p)$ is the current path cost of corner point p. We say an edge e is a "low-resolution edge" (line 8) if both the cells on either side of e are low resolution. An efficient solution to the minimizations in lines 5 and 9 was presented in Section 3.

7.3. The Multi-Resolution Field D* Algorithm

The path cost calculation discussed above enables us to plan direct, low-cost paths through nonuniform resolution grids. We can couple this with any standard path planning algorithm, such as Dijkstra's, A*, or D*. Because our motivation for this work is robotic path planning in unknown or partially known environments, we have used it to extend the Field D* algorithm to nonuniform resolution grids. To distinguish it from the uniform resolution version, we call the resulting algorithm *Multi-resolution Field D**. By coupling the low-cost paths generated by interpolation-based planning with the memory efficiency of nonuniform resolution grid representations, Multi-resolution Field D* is able to provide extremely effective paths for a fraction of the memory and computational requirements of current approaches.

A basic version of the algorithm is presented in Figure 16. Here, the ComputePathCost function (line

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

UpdateNode($s$)
 2  if $s$ was not visited before, $g(s) = \infty$;
 3  if ($s \neq s_{goal}$)
 4    $rhs(s)$ = ComputePathCost($s$);
 5  if ($s \in OPEN$) remove $s$ from $OPEN$;
 6  if ($g(s) \neq rhs(s)$) insert $s$ into $OPEN$ with key($s$);

ComputeShortestPath()
 7  while ($\min_{s \in OPEN}$(key($s$)) $\dot{<}$ key($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
 8    remove node $s$ with the minimum key from $OPEN$;
 9    if ($g(s) > rhs(s)$)
10      $g(s) = rhs(s)$;
11      for all $s' \in nbrs(s)$ UpdateNode($s'$);
12    else
13      $g(s) = \infty$;
14      for all $s' \in nbrs(s) \cup \{s\}$ UpdateNode($s'$);

Main()
15  $g(s_{start}) = rhs(s_{start}) = \infty$; $g(s_{goal}) = \infty$;
16  $rhs(s_{goal}) = 0$; $OPEN = \emptyset$;
17  insert $s_{goal}$ into $OPEN$ with key($s_{goal}$);
18  forever
19    ComputeShortestPath();
20    Wait for changes to grid or traversal costs;
21    for all new cells or cells with new traversal costs $x$
22      for each node $s$ on an edge or corner of $x$
23        UpdateNode($s$);
```

Figure 16. The multi-resolution Field D* algorithm (basic version).

4) takes a node s and computes the minimum path cost for s using the path costs of all of its neighboring nodes and interpolation across its neighboring edges, as discussed in Section 7 and presented in Figure 15. Other notation is consistent with the algorithm presented in Section 4: $g(s)$ is the current path cost of node s, $rhs(s)$ is the one-step lookahead path cost for s, *OPEN* is a priority queue containing inconsistent nodes [i.e., nodes s for which $g(s) \neq rhs(s)$] in increasing order of *key* values (line 1), s_{start} is the initial agent node, and s_{goal} is the goal node. $h(s_{start}, s)$ is a heuristic estimate of the cost of a path from s_{start} to s.

As with other members of the D* family of algorithms, significant optimizations can be made to this initial algorithm. In particular, several of the optimizations discussed in Appendix A are applicable and were used in our implementation.

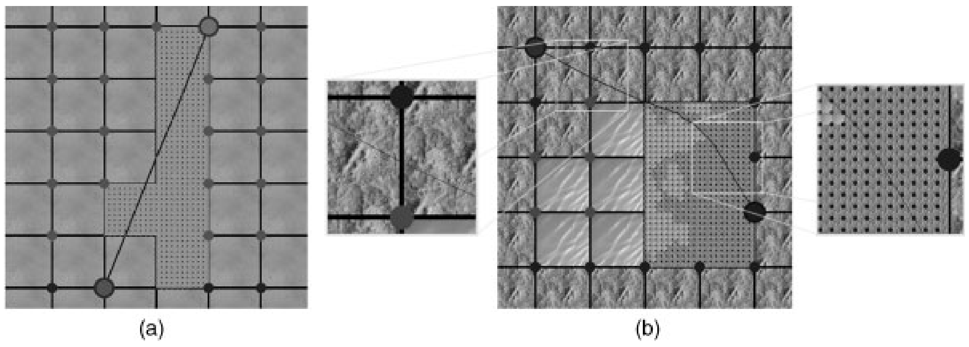


Figure 17. Multi-resolution Field D* produces direct, low-cost paths (in black) through both high-resolution and low-resolution areas. Each filled circle represents the lower-left corner node of a low-resolution cell (if the circle is large) or a high-resolution cell (if the circle is small). The detailed textures that appear in the right illustration are for visualization purposes only; the traversal cost is constant within each grid cell.

8. MULTI-RESOLUTION FIELD D* RESULTS

Multi-resolution Field D* was originally developed to extend the range over which unmanned ground vehicles (such as our outdoor vehicles in Figure 1) could operate by orders of magnitude. We have found the algorithm to be extremely effective at reducing both the memory and computational requirements of planning over large distances and at producing direct, low-cost paths. Figures 17 and 18 show example paths planned using the algorithm.

To quantify its performance, we ran experiments comparing Multi-resolution Field D* to uniform resolution Field D*. We used uniform resolution Field D* for comparison because it produces less costly paths than regular uniform grid-based planners and far better paths than regular nonuniform grid-based planners.

Our first set of experiments investigated both the quality of the solutions and the computation time required to produce these solutions as a function of the

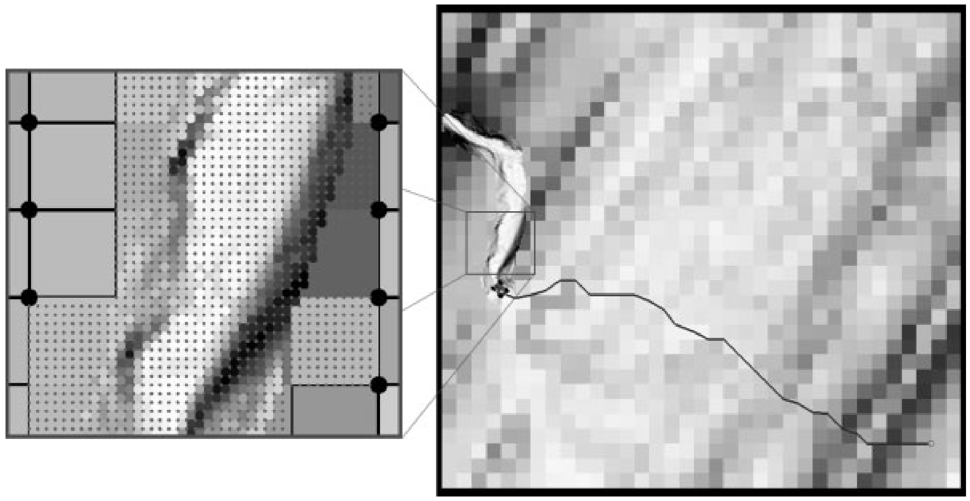


Figure 18. Multi-resolution Field D* used to guide an agent through a partially known environment. On the left is a section of the path already traversed showing the high-resolution cells. These data were taken from Fort Indiantown Gap, PA.

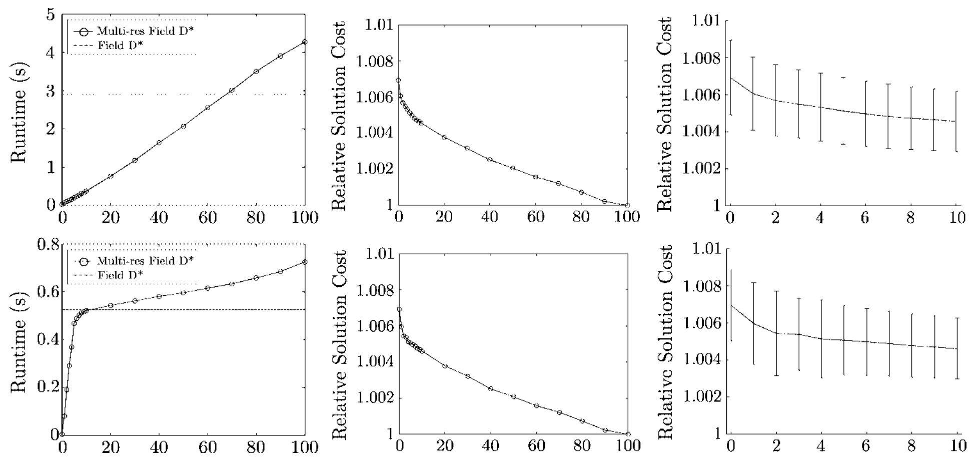


Figure 19. Computation time and solution cost as a function of how much of the environment is represented at a high resolution. The x axis of each graph depicts the percentage of the map modeled using high-resolution cells, ranging from 0 (all modeled at a low resolution) to 100 (all modeled at a high resolution). (Top) Initial planning. A path was planned from one side to the other of 100 randomly generated environments and the results were averaged. (Bottom) Replanning. 5% of each environment was randomly altered and the initial paths were repaired.

memory requirements of Multi-resolution Field D*. We began with a randomly generated 100×100 low-resolution environment with an agent at one side and a goal at the other. We then took some percent p of the low-resolution cells (centered around the agent) and split each into a 10×10 block of high-resolution cells. We varied the value of p from 0% up to 100%. We then planned an initial path to the goal. Next, we randomly changed 5% of the cells around the agent (at a low-resolution) and replanned a path to the goal. We focused the change around the robot to simulate new information being gathered in its vicinity. The results from these experiments are presented in Figure 19. The x axis of each graph represents how much of the environment was represented at a high resolution. The left graphs show how the time required for planning changes as the percent of high-resolution cells increases, while the middle and right graphs show how the path cost changes. The y values in the middle and right graphs are path costs, relative to the path cost computed when 100% of the environment is represented at a high resolution. The right graph shows the standard error associated with the relative path costs for smaller percentages of high-resolution cells. As can be seen from these results, modeling the environment as mostly low-resolution produces paths that are only trivially more expensive than those produced using a full high-resolution representation, for a small fraction of the memory and computational requirements.

This first experiment shows the advantage of Multi-resolution Field D* as we reduce the percentage of high-resolution cells in our representation. However, because there is some overhead in the multi-resolution implementation, we also ran uniform resolution Field D* over a uniform, high-resolution grid to compare the runtime of this algorithm with our Multi-resolution version. The results of uniform resolution Field D* have been overlaid on our runtime graphs. Although uniform resolution Field D* is more efficient than Multi-resolution Field D* when 100% of the grid is composed of high-resolution cells, it is far less efficient than Multi-resolution Field D* when less of the grid is made up of high-resolution cells.

Our second experiment simulated the most common case for outdoor mobile robot navigation, namely where we have an agent with some low-resolution prior map and high-resolution onboard sensors. For this experiment, we took real data collected from Fort Indiantown Gap, PA, and simulated a robotic traverse from one side of this 350×320 m environment to the other. We blurred the data to create a low-resolution prior map (at 10×10 m accuracy) that the robot updated with a simulated

Table II. Results for uniform resolution Field D* versus Multi-resolution Field D* on a simulated robot traverse through real data acquired at Fort Indiantown Gap. The robot began with a low-resolution map of the area and updated this map with a high-resolution onboard sensor as it traversed the environment. The map was 350×320 meters in size.

	Field D*	Multi-res Field D*
Total planning and replanning time (s)	0.493	0.271
Initial planning time (s)	0.336	0.005
Average replanning time (s)	0.0007	0.0012
Percent high-resolution cells	100	13

medium-resolution sensor (at 1×1 m accuracy with a 10 m range) as it traversed the environment.

The results from this experiment are shown in Table II. Again, Multi-resolution Field D* requires only a fraction of the memory of uniform resolution Field D*, and its runtime is very competitive. In fact, it is only in the replanning portion of this final experiment that Multi-resolution Field D* requires more computation time than uniform resolution Field D*, and this is only because the overhead of converting part of its map representation from low resolution to high resolution overshadows the trivial amount of processing required for replanning.

9. CONCLUSIONS

We have presented Field D* and Multi-resolution Field D*, two interpolation-based path planning algorithms that address two of the most significant shortcomings of grid-based path planning.

The first shortcoming they address concerns the quality of paths produced over grids. Almost all grid-based planners are limited to finding paths that transition only between adjacent grid points. This creates unnatural and often costly paths. Both Field D* and Multi-resolution Field D* use linear interpolation to approximate the path costs of points not sampled on the grid. This allows paths to transition between any two points on adjacent grid cell *edges*, rather than just between grid cell centers or corners. Thus, the paths produced are less costly and involve less unnecessary turning than those produced using current grid-based approaches.

The second shortcoming concerns the memory and computational requirements of grid-based path planning and is addressed by Multi-resolution Field D*. In large environments, planning over a uniform resolution grid requires significant amounts of memory and computation. Further, robots often have information concerning different parts of the environment at different resolutions, but many planners require that the entire environment be represented at the highest resolution for which any information is available. In contrast, Multi-resolution Field D* plans and replans over nonuniform resolution grids. The nonuniformity of the grids allows us to represent at a low resolution areas of the environment for which only low-resolution information exists. This significantly reduces the memory and often the runtime requirements of the planning task. Furthermore, because of its use of linear interpolation, the paths provided by Multi-resolution Field D* are comparable in quality to those of uniform resolution Field D*. Consequently, Multi-resolution Field D* is currently being used to extend the range of one of our outdoor mobile robots by one to two orders of magnitude. It is our belief that, by combining interpolation with nonuniform representations of the environment, we can “have our cake and eat it too,” with a planner that is extremely efficient in terms of both memory and computation while still producing very direct, low-cost paths.

We and others are currently extending these algorithms in a number of ways. First, a 3D version of the Field D* algorithm has been developed for vehicles operating in the air or underwater (Carsten, 2005). We are also developing a version that interpolates over headings, not just path costs, to produce smoother paths when turning is expensive. Finally, interpolation is currently being incorporated into the TEMPEST mission-level path planner, which takes into account time and energy constraints while generating paths (Tompkins, Stentz & Whittaker, 2004).

10. APPENDIX A: OPTIMIZATIONS

As with other members of the D* family of algorithms, there are a number of optimizations that can be made to the basic (uniform resolution) Field D* algorithm to significantly improve its efficiency. First, we can reduce the amount of computation required when updating the neighbors of a popped node (Figure 8, lines 9–14) by only considering those nodes actually affected by the new value of the popped node and how these nodes are affected.

To do this, we keep track of a backpointer for each node specifying from which nodes it currently derives its path cost. Since, in Field D*, the successor of each node is a point on an edge connecting two of its neighboring nodes, this backpointer needs to specify the two nodes that form the endpoints of this edge. We use *bptr*(*s*) to refer to the most *clockwise* of the two endpoint nodes relative to node *s*. For example, if the current best path from node *s* intersects edge $\overrightarrow{s_1 s_2}$ (see Figure 2), then $bptr(s)=s_1$. We also make use of two new operators, $cknbr(s,s')$ and $ccknbr(s,s')$, that, given a node *s* and some neighboring node s' of *s*, return the next neighboring node of *s* in the clockwise and counter-clockwise directions, respectively. Thus, using the node labels from Figure 2, $cknbr(s,s_1)=s_8$, $cknbr(s,s_8)=s_7$, 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 Figure 20. 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 only been presented in the current form 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 node that resides at one of the 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. First, when the traversal cost of a cell increases, we only need to update the *rhs* value of nodes that relied upon the old cell cost. Second, when the traversal cost of a cell decreases, we can avoid recomputing the *rhs* value for *any* node (at least initially) by making one small approximation to the algorithm. Basically, instead of recomputing *rhs* values for each of the corner nodes, we simply put the node with the minimum current *rhs* value onto the *OPEN* list. Since this node may have its *rhs* value equal to its *g* value, we need to modify the algorithm so that such nodes are processed as if their path costs have decreased (i.e., as if their *rhs* values were in fact lower than their *g* values).

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

UpdateNode(s)
 2  if (g(s) ≠ rhs(s)) insert s into OPEN with key(s);
 3  else if (s ∈ OPEN) remove s from OPEN;

ComputeShortestPath()
 4  while (min_{s∈OPEN}(key(s)) <˙ key(s_start) OR rhs(s_start) ≠ g(s_start))
 5    peek at state s with the minimum key on OPEN;
 6    if (g(s) > rhs(s))
 7      g(s) = rhs(s);
 8      remove s from OPEN;
 9      for all s' ∈ nbrs(s)
10        if s' was not visited before
11          g(s') = rhs(s') = ∞;
12        if (rhs(s') > ComputeCost(s', s, ccknbr(s', s)))
13          rhs(s') = ComputeCost(s', s, ccknbr(s', s));
14          bptr(s') = s;
15        if (rhs(s') > ComputeCost(s', s, cknbr(s', s)))
16          rhs(s') = ComputeCost(s', cknbr(s', s), s);
17          bptr(s') = cknbr(s', s);
18        UpdateNode(s');
19    else
20      g(s) = ∞;
21      for all s' ∈ nbrs(s)
22        if (bptr(s') = s OR bptr(s') = cknbr(s', s))
23          rhs(s') = min_{s''∈nbrs(s')} ComputeCost(s', s'', ccknbr(s', s''));
24          bptr(s') = argmin_{s''∈nbrs(s')} ComputeCost(s', s'', ccknbr(s', s''));
25          UpdateNode(s');
26      UpdateNode(s);

Main()
27  g(s_start) = rhs(s_start) = ∞; g(s_goal) = ∞;
28  rhs(s_goal) = 0; OPEN = ∅;
29  insert s_goal into OPEN with key(s_goal);
30  forever
31    ComputeShortestPath();
32    Wait for changes in cell traversal costs;
33    for all cells x with new traversal costs
34      for each node s on a corner of x
35        if s was not visited before, g(s) = ∞;
36        if (s ≠ s_goal)
37          rhs(s) = min_{s'∈nbrs(s)} ComputeCost(s, s', ccknbr(s, s'));
38          UpdateNode(s);
```

Figure 20. The Field D* algorithm (after initial optimizations).

Then, when this node is popped off the *OPEN* list, it will have a chance to update the *rhs* values of the other corner nodes based on the new traversal cost of the center cell.

Unfortunately, it turns out that this second method no longer guarantees optimal path costs given our linear interpolation assumption. This is because it is possible in theory that one of the corner nodes of the cell does not use the node 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 Figure 5(iii) and imagine the traversal cost of the bottom cell has decreased and the corner node 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 node 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 D* Lite and Incremental A* algorithms. In the optimized version of D* Lite, when a node 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 nodes may later 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 node to infinity, rather than recomputing its true value. Since the node 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 node 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 node. Nodes 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 Figures 21 and 22 and was used for generating the results presented in this paper. As with our earlier versions of the algorithm, some sections of this version are not as efficient as they could be (e.g.,

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

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

**ComputeShortestPath**$()$
 4  while $(\min_{s \in OPEN}(\text{key}(s)) \dot{<} \text{key}(s_{start})$ OR $rhs(s_{start}) \neq g(s_{start}))$
 5    peek at node $s$ with the minimum key on $OPEN$;
 6    if $(g(s) \geq rhs(s))$
 7      $g(s) = rhs(s)$;
 8      remove $s$ from $OPEN$;
 9      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 $(rhs(s') > \text{ComputeCost}(s', s, ccknbr(s', s)))$
14          $rhs(s') = \text{ComputeCost}(s', s, ccknbr(s', s))$;
15          $bptr(s') = s$;
16        if $(rhs(s') > \text{ComputeCost}(s', s, cknbr(s', s)))$
17          $rhs(s') = \text{ComputeCost}(s', cknbr(s', s), s)$;
18          $bptr(s') = cknbr(s', s)$;
19        if $(rhs(s) \neq rhs_{old})$
20          UpdateState$(s')$;
21    else
22      $rhs(s) = \min_{s' \in nbrs(s)} \text{ComputeCost}(s, s', ccknbr(s, s'))$;
23      $bptr(s) = \text{argmin}_{s' \in nbrs(s)} \text{ComputeCost}(s, s', ccknbr(s, s'))$;
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 $(rhs(s') \neq \text{ComputeCost}(s', bptr(s'), ccknbr(s', bptr(s'))))$
29              if $(g(s') < rhs(s')$ OR $s' \notin OPEN)$
30                $rhs(s') = \infty$;
31                UpdateNode$(s')$;
32              else
33                $rhs(s') = \min_{s'' \in nbrs(s')} \text{ComputeCost}(s', s'', ccknbr(s', s''))$;
34                $bptr(s') = \text{argmin}_{s'' \in nbrs(s')} \text{ComputeCost}(s', s'', ccknbr(s', s''))$;
35                UpdateNode$(s')$;
36      UpdateNode$(s)$;
```

Figure 21. The Field D* algorithm (optimized version): ComputeShortestPath function.

there are repeated path cost calculations in lines 13–14 and again in lines 16–17, 22–23, 33–34, and 45–46). This is solely for ease of presentation; these sources of inefficiency should not appear in any implementation.

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 (Figure 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

```
**UpdateCellCost**$(x, c)$
if ($c$ is greater than current traversal cost of $x$)
  for each node $s$ on a corner of $x$
    if either $bptr(s)$ or $ccknbr(s, bptr(s))$ is a corner of $x$
      if ($rhs(s) \neq$ ComputeCost$(s, bptr(s), ccknbr(s, bptr(s)))$)
        if ($g(s) < rhs(s)$ OR $s \notin OPEN$)
          $rhs(s) = \infty$;
          UpdateNode$(s)$;
        else
          $rhs(s) = \min_{s' \in nbrs(s)}$ ComputeCost$(s, s', ccknbr(s, s'))$;
          $bptr(s) = \mathrm{argmin}_{s' \in nbrs(s)}$ComputeCost$(s, s', ccknbr(s, s'))$;
          UpdateNode$(s)$;
else
  $rhs_{min} = \infty$;
  for each node $s$ on a corner of $x$
    if $s$ was not visited before, $g(s) = rhs(s) = \infty$;
    else if ($rhs(s) < rhs_{min}$)
      $rhs_{min} = rhs(s)$; $s^\star = s$;
  if ($rhs_{min} \neq \infty$)
    insert $s^\star$ into $OPEN$ with key$(s^\star)$;

**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 $c$
    UpdateCellCost$(x, c)$;
```

Figure 22. The Field D* algorithm (optimized version): main function.

$$\mathcal{N}_c^2 \times \mathcal{M}_c, \tag{8}$$

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., nonobstacle, cell.

The construction of this lookup table and the altered version of ComputeCost() are shown in Figure 23. 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 $\mathcal{I}$.

We have also used this approach to create an efficient implementation of Field D* for the Mars Exploration Rovers (see Figure 24). However, for this application we ignored the possibility of using the bottom cell b [Figure 5(iii)] and only considered paths involving the center cell, so that our lookup table was

```
**ConstructInterpolationTable**$(\mathcal{N}_c, \mathcal{M}_c, cellcosts)$
$c_i = 0$;
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 $v_s$;
```

Figure 23. Using a lookup table to store the interpolation-based cost calculation.

indexed by just c and f. We also used a small set of nonlinearly spaced cell traversal cost values (e.g., values of 255, 375, 510, 640, 1020, 1275, and 1785). The memory required for storing the corresponding interpolation table was quite small (on the order of 25 KB) and the resulting implementation was very fast.

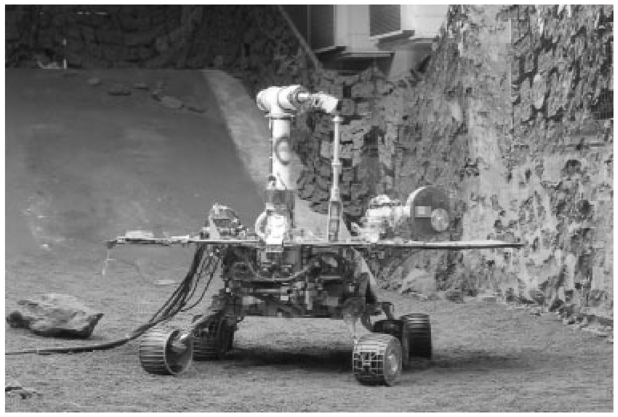


Figure 24. One of the Mars Exploration Rovers in the testing sandpit at NASA's Jet Propulsion Laboratory. An optimized implementation of Field D* has been created for these vehicles.

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 given 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 node, is because it reduces the number of times each node needs to be processed. With the former approach, it is not uncommon for a node to be popped off the queue and processed before one of its optimal successor nodes. This is rather inefficient, as the same node will have to be reprocessed 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 nodes are often processed, these nodes are usually processed fewer times.

11. APPENDIX B: MEDIA FILES

There are three media files accompanying this paper: D*-run, Field-D*-run, and PerceptOR-run.

D-run:* This animation illustrates regular D* being used to guide an agent from the top-right to the bottom-left of a binary-cost environment. The agent began with an empty map, represented as a uniform resolution grid. The obstacles in the environment appear in red, known obstacles appear in blue as they are observed by the agent. Notice that the path is restricted to heading increments of 45 degrees.

Field-D-run:* This animation illustrates Field D* being used to guide an agent through the same environment shown in the D*-run file. Notice that the path consists of straight-line segments with widely-varying headings.

PerceptOR-run: This animation shows part of a run conducted by an autonomous ATV at a site near the Pittsburgh Airport. The yellow ATV was given a single goal point on the opposite side of a wooded area. There were no initial map data. The vehicle followed a trail until it was obstructed by a fallen tree, at which point it left the trail and drove through the woods. Eventually, it rejoined the trail and achieved the goal point in a field on the other side of the woods. This animation shows the laser data, local map, and video of the vehicle for the first part of this traverse. The laser data and local map run at three times the speed of the video. This run was conducted as part of the DARPA-sponsored project "Perception for Off-Road Mobility (PerceptOR)" (Contract No. MDA972-01-9-0016).

ACKNOWLEDGMENTS

The authors would like to thank several people who were instrumental in the development of the fielded systems mentioned here. Bryan Nagy contributed significantly to the navigation code for several of the vehicles. Joseph Carsten, Arturo Rankin, and Mark Maimone incorporated Field D* into the onboard planning framework of the Mars Exploration Rovers. Marc Zinck, Boris Sofman, and Freddie Dias provided technical support for all aspects of the onboard autonomy of the E-Gators. We would also like to thank Nidhi Kalra and the anonymous reviewers for useful suggestions regarding the manuscript.

This work was sponsored in part by the U.S. Army Research Laboratory, under contract "Robotics Collaborative Technology Alliance" (Contract No. DAAD19-01-2-0012), and by DARPA, under contract "Perception for Off-Road Mobility (PerceptOR)" (Contract No. MDA972-01-9-0016).

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

Brock, O., & Khatib, O. (1999). High-speed navigation using the global dynamic window approach. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).

Carsten, J. (2005). 3D Field D*. Master's thesis, Carnegie Mellon University, Pittsburgh, PA.

Chen, D., Szczerba, R., & Uhran, J. (1995). Planning conditional shortest paths through an unknown environment: A framed-quadtree approach. In Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).

Dijkstra, E. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1, 269–271.

Hart, P., Nilsson, N., & Rafael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4, 100–107.

Kambhampati, S., & Davis, L. (1986). Multi-resolution path planning for mobile robots. IEEE Journal of Robotics and Automation, RA-2(3), 135–145.

Kelly, A. (1995). An intelligent predictive control approach to the high speed cross country autonomous navigation problem. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Koenig, S., & Likhachev, M. (2002a). D* Lite. In Proceedings of the National Conference on Artificial Intelligence (AAAI).

Koenig, S., & Likhachev, M. (2002b). Improved fast replanning for robot navigation in unknown terrain. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).

Koenig, S., & Likhachev, M. (2002c). Incremental A*. In Advances in neural information processing systems. Cambridge, MA: MIT Press.

Konolige, K. (2000). A gradient method for realtime robot control. In Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).

Larson, R. (1967). A survey of dynamic programming computational procedures. IEEE Transactions on Automatic Control, 12, 767–774.

Larson, R., & Casti, J. (1982). Principles of dynamic programming, part 2. New York: Marcel Dekker.

LaValle, S. (2006). Planning algorithms. New York: Cambridge University Press (also available at http://msl.cs.uiuc.edu/planning/).

Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., & Thrun, S. (2005). Anytime Dynamic A*: An anytime replanning algorithm. In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS).

Likhachev, M., Gordon, G., & Thrun, S. (2003). ARA*: Anytime A* with provable bounds on sub-optimality. In Advances in neural information processing systems. Cambridge, MA: MIT Press.

Mitchell, J. (2000). Geometric shortest parths and network optimization. In Handbook of computational geometry (pp. 633–701). Amsterdam: Elsevier Science.

Mitchell, J., & Papadimitriou, C. (1991). The weighted region problem: finding shortest paths through a weighted planar subdivision. Journal of the ACM, 38, 18–73.

Nilsson, N. (1980). Principles of artificial intelligence. Palo Alto, CA: Tioga Publishing Company.

Philippsen, R. (2004). Motion planning and obstacle avoidance for mobile robots in highly cluttered dynamic environments. Ph.D. thesis, EPFL, Lausanne, Switzerland.

Philippsen, R., & Siegwart, R. (2005). An interpolated dynamic navigation function. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).

Rowe, N., & Richbourg, R. (1990). An efficient Snell's-law method for optimal-path planning across two-dimensional irregular homogeneous-cost regions. International Journal of Robotics Research, 9(6), 48–66.

Samet, H. (1982). Neighbor finding techniques for images represented by quadtrees. Computer Graphics and Image Processing, 18, 37–57.

Sethian, J. (1996). A fast marching level set method for monotonically advancing fronts. Applied Mathematics, Proceedings of the National Academy of Science, 93, 1591–1595.

Singh, S., Simmons, R., Smith, T., Stentz, A., Verma, V., Yahja, A., & Schwehr, K. (2000). Recent progress in local and global traversability for planetary rovers. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).

Stachniss, C., & Burgard, W. (2002). An integrated approach to goal-directed obstacle avoidance under dynamic constraints for dynamic environments. In Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).

Stentz, A. (1995). The Focussed D* algorithm for real-time replanning. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI).

Stentz, A., & Hebert, M. (1995). A complete navigation system for goal acquisition in unknown environments. Autonomous Robots, 2(2), 127–145.

Tompkins, P., Stentz, A., & Whittaker, W. (2004). Mission-level path planning for rover exploration. In Proceedings of the International Conference on Intelligent Autonomous Systems (IAS).

Yahja, A., Singh, S., & Stentz, A. (2000). An efficient on-line path planner for outdoor mobile robots operating in vast environments. Robotics and Autonomous Systems, 33, 129–143.