Auton Agent Multi-Agent Syst (2013) 26:354–388
DOI 10.1007/s10458-012-9195-8

Field D* path-finding on weighted triangulated and tetrahedral meshes

Simon Perkins · Patrick Marais · James Gain · Mark Berman

Published online: 21 April 2012

Abstract Classic shortest path algorithms operate on graphs, which are suitable for problems that can be represented by weighted nodes or edges. Finding a shortest path through a set of weighted regions is more difficult and only approximate solutions tend to scale well. The Field D* algorithm efficiently calculates an approximate, interpolated shortest path through a set of weighted regions and was designed for navigating robots through terrains with varying characteristics. Field D* operates on unit grid or quad-tree data structures, which require high resolutions to accurately model the boundaries of irregular world structures. In this paper, we extend the Field D* cost functions to 2D triangulations and 3D tetrahedral meshes: structures which model polygonal world structures more accurately. Since robots typically have limited resources available for computation and storage, we pay particular attention to computation and storage overheads when detailing our extensions. We begin by providing analytic solutions to the minimum of each cost function for 2D triangles and 3D tetrahedra. Our triangle implementation provides a 50 % improvement in performance over an existing triangle implementation. While our 3D extension to tetrahedra is the first full analytic extension of Field D* to 3D, previous work only provided an approximate minimization for a single cost function on a 3D cube with unit lengths. Each cost function is expressed in terms of a general function whose characteristics can be exploited to reduce the calculations required to find a minimum. These characteristics can also be exploited to cache the majority of cost functions, producing a speedup of up to 28 % in the 3D tetrahedral case. We demonstrate that, in environments composed of non-grid aligned data, Multi-resolution quad-tree Field D* requires an order of magnitude more faces and between 15 and 20 times more node expansions, to produce a path of similar cost to one produced by a triangle implementation of Field D* on a lower resolution triangulation. We provide examples of 3D pathing through models of complex topology, including pathing through anatomical structures extracted from a medical data set.

S. Perkins (✉) · P. Marais · J. Gain
Computer Science Department, University of Cape Town, Cape Town, South Africa
e-mail: simon.perkins@gmail.com

M. Berman
Mathematics Department, University of Cape Town, Cape Town, South Africa

To summarise, this paper details a robust and efficient extension of Field D* pathing to data sets represented by weighted triangles and tetrahedra, and also provides empirical data which demonstrates the reduction in storage and computation costs that accrue when one chooses such a representation over the more commonly used quad-tree and grid-based alternatives.

Keywords Artificial intelligence · Problem solving · Control methods and Search · Graph and tree search strategies · Vision and scene understanding · Representations · Data structures and transforms · Perceptual reasoning

1 Introduction

Classic shortest path algorithms such as *Dijkstra's algorithm* [1] or *A** [2,3] operate on graphs where the edges or nodes are weighted. This weighting system provides flexibility in that any quantifiable characteristic can weight the graph, in addition to a typical distance metric. In a road network for example, edges representing congested roads can be weighted heavily so that a shortest path calculation will avoid them. The *Weighted Region Problem* [4] has been postulated with the intent of developing an algorithm to find the shortest path through a set of weighted regions. The ability to weight regions also provides flexibility in finding the shortest path through environments with varying characteristics. For example, a robot traversing terrain may weight rocky or muddy areas heavily, and grassed or sandy areas lightly, so that it will avoid difficult terrain.

Mitchell and Papadmitriou [4] introduced the first in a series of ϵ-approximation algorithms that find a shortest path through a set of weighted regions on a 2D plane for a certain tolerance ϵ. Their algorithm utilises *Snell's Law of Refraction* and runs in $O(n^8)$ time complexity. Mata and Mitchell [5] develop another ϵ-approximation algorithm based on a *pathnet graph*, which runs in $O(n^3)$ time complexity. Other ϵ-approximation methods [6–9] discretize weighted triangles by inserting *Steiner points* on boundary edges, creating an approximation graph on which a standard Dijkstra search can be applied. More Steiner points are inserted if greater accuracy is required. In general, this results in a time complexity of $mn\, O(\log mn)$ where $m \propto 1/\epsilon$ and relates to the number of Steiner points added. Later work [10,11] reduces this computational complexity at the cost of space by adding data structures that cache the results of shortest path queries between two points.

While the addition of Steiner points to achieve greater accuracy provides a mathematically rigorous approach to managing approximation error, this comes at the cost of greater resource utilisation, since many more vertices and edges may be required to achieve the desired accuracy. In our target application, robotics, resources are usually heavily constrained and the graph may not be allowed to grow beyond a prescribed size. The graph may also be updated in real time as additional information is received about the environment. These constraints motivated the development of the *Field D* algorithm* [12]—an efficient, but approximate, interpolated solution to the weighted region problem. This paper addresses extensions to the Field D* algorithm which allow it to operate on triangulated and tetrahedral graph structures.

Field D* adapts the underlying graph structure and cost functions of graph-based shortest path algorithms. The underlying representation is a unit grid whose squares or *cells* are assigned weights. The cost of traversing to a node is calculated by minimizing the cost of travelling through a weighted cell and interpolating the adjacent node costs along the cell edges. The resulting path can travel through weighted cells, and in practice produces paths of lesser cost than A*. Field D* also has replanning capability that allows sections of the path to be recalculated when changes in the environment occur.

Due to the interpolation error inherent in the Field D* algorithm, the resulting paths are not necessarily the shortest, but are reasonable approximations and provide an efficient alternative to analytic solutions. Extensions include Multi-resolution Field D* [13], which extends Field D* to quad-trees [14] to reduce the algorithm's computation time and space requirements and 3D Field D* [15], an approximate extension to 3D grids. Experimental evidence in [13] shows that Multi-resolution Field D* can improve performance over Field D* by a factor of 1.8 times when the resolution of the underlying quad-tree is 13 % of that of the grid.

Representing an environment with a grid or quad-tree can be expensive in terms of storage. In the field of *Geographic Information Systems* for example, terrain data can be represented with an image-based *Digital Elevation Model* (DEM) or a *Triangulated Irregular Network* (TIN) [16]. TINs are frequently based on *Delaunay Triangulations* [17] since this representation avoids narrow triangles. Since TINs use triangles to represents areas instead of the grid elements used by DEMs, less memory is required to accurately represent the terrain. This is related to the function approximation: Well-behaved functions can be approximated with *piecewise constant* elements and *piecewise linear* elements. A single piecewise linear element can more accurately fit a function segment than many piecewise constant elements, at the expense of a slightly more expensive element volume calculation. However, by reducing the number of elements, this increased expense becomes insignificant and the overall expense of computing the approximation is reduced.

Similarly, triangular subdivision of an irregular object is more accurate than a subdivision with grid or quad-tree cells, since triangles can represent the boundary of the object more accurately. This concept extends to 3D: approximating a polyhedral object with tetrahedra will be more accurate than using cubes. Since these structures can approximate objects and environments accurately, triangulated and tetrahedral meshes are common representations [18], especially in fields such as *Finite Element Methods* [19]. As these are important and useful representations, this paper presents an extension of Field D* to triangle and tetrahedral meshes, expressed in vector notation. Our results show that a triangle implementation of Field D* is faster than a quad-tree implementation of Field D* and requires fewer elements to represent the environment when it is not grid-aligned.

Other methods based on interpolation exist. Konolige [20] introduces a *Gradient Method* which uses classic grid-based planning to propagate costs over a grid and then uses a function to interpolate between grid values and calculate the shortest path from start to goal. While the resulting path is shorter than that on a grid, the initial node values are not as accurate as they could be since the costs are calculated from travelling along grid edges, but not through grid cells. The algorithm also has no replanning capability. *E** [21] uses *Fast Marching Methods* [22] to expand a surface outward from a goal to every part of the environment. This search is not focused towards the starting location and assigns a cost to travel through a node, assuming that the cost to transition between grid nodes is constant.

A number of other path-finding algorithms tackle the problem of finding paths across regions by partitioning environments into solid and empty space. *Near optimal hierarchical path-finding* [23] smooths the path produced by an A* search on a grid by iteratively examining a node and removing the node's parent from the path if the node has line-of-sight to the node's grandparent. This can still be suboptimal if a node has visibility of a much greater ancestor. *Basic Theta* * [24] improves this by allowing any vertex to be the parent of a node. Kallman [25] represents an environment with a *Constrained Delaunay Triangulation*, initially calculating the shortest path on the *adjacency graph* of the triangulation. This path is then refined using a *funnel algorithm* [26]. While these algorithms produce good paths, they lose the richness that a region weighting system provides.

This paper is structured as follows: We briefly mention some standard path finding literature in the Related Work section and describe how Ferguson et al. [27] extend the basic path finding cost function to a weighted grid. We describe the triangle and tetrahedral cost functions in Sect. 3, showing how the work for most of these functions can be precomputed and cached. In the Results section, we demonstrate that a triangulation implementation of Field D* is superior to a multi-resolution quad-tree implementation in environments where objects are not grid aligned, and detail the performance improvements that can be gained by function caching. We also provide a performance and space comparison on triangles between Generalized Field D* [28] and our implementation, as well as results for the 3D tetrahedral case.

2 Related work

Planning shortest paths over a graph is a common computer science problem. *Dijkstra's algorithm* [1] finds the shortest path between a particular node and every other node in a graph with non-negative edge costs. The algorithm is driven by a priority queue of nodes, ordered by their cost. When a node is popped off the queue, the costs of the node's neighbours are derived from the cost of the node and the weight of the connecting graph edge. The neighbours are then inserted onto the queue.

This graph-wide search can be unnecessary when the start and goal nodes are known. *A** [2,3] extends Dijkstra's algorithm via the use of a heuristic that focuses the search in the direction of the goal node. The use of a heuristic reduces the number of nodes that the algorithm searches. *D** [29], *Incremental A** [30] and *D* Lite* [31] extend A* by repairing calculated paths when dynamic cost changes occur in the underlying graph structure. Field D* [27] extends D* Lite to operate on a weighted grid by extending the standard cost function determining the cost of traversing a graph edge used in A* and D* Lite, to a set of cost functions determining the cost of traversing through a weighted cell. The paths produced by Field D* may travel through weighted cells and are not restricted to cell edges. Field D* also inherits the capacity to repair paths and the use of a heuristic from D* Lite.

*Theta** [24] is an extension of A* to OPEN/CLOSED grids that not only considers the neighbours of the node undergoing expansion as possible parents, but also all nodes that are visible from the expanded node. Thus, the authors note that a Basic Theta* node expansion is linear in the number of grid cells in the environment. The authors propose a variant called Angle-Propagation Theta* that reduces the complexity of a node expansion to constant time, but this is still computationally slower than Basic Theta*'s node expansion. This work also describes an extension of Basic Theta* to non-uniformly weighted grids based on accumulating grid cell costs along rays cast between the current node and all possible parents. Their work shows that in randomly weighted environments Field D* finds shorter paths in less time than Theta*, while in environments where 50 % of the cells are randomly weighted, the other 50 % are weighted with the cheapest cost and there are large regions of contiguous cost, Field D* produces equivalent path costs to Theta* in slightly shorter time. Unfortunately, the values provided are averaged over 100 random environments and paths, with no indication of variability, so it is difficult to make an informed comparison. It would be interesting to see Theta* extended to triangulations, along with more extensive benchmarking.

Choi and Yu [32] incrementally extends Theta*'s non-uniform weighted grid extension by calculating an arithmetic mean and a weighted mean over the ray cast by Theta*. The arithmetic mean averages the grid cell costs encountered by the ray, while the weighted mean accumulates the horizontal or vertical contribution—depending on a Bresenham-style

decision—of a cell's cost to the overall ray cost. Unfortunately, the authors do not perform a comparison with non-uniform Theta* mentioned above. The path costs produced by the weighted mean are equivalent to those produced by arithmetic mean and require 10 % more time to calculate.

2.1 Field D*

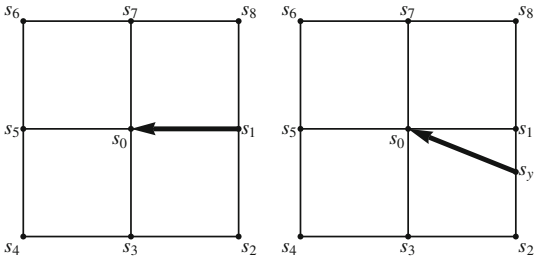
Here, we partially describe the Field D* Algorithm as published by Ferguson et al. [27], focusing chiefly on the derivation of the cost functions for the paths through a grid cell, since the extension of these cost functions to triangles and tetrahedra constitutes the main novelty in our work. In this sense, our work further extends the *ComputeCost* function that Ferguson et al. derive from traditional graph-based path planning. Readers interested in learning about the standard priority queue algorithm used in most A* derived path-finding algorithms should consult [2,3,27,31]. One difference in our figures compared to previous work, is that arrows point at the node for which the cost is being calculated (rather than away), as we consider this a better representation of the direction of cost derivation.

Classic Shortest Path Algorithms operate on a node and edge graph structure in which either the nodes or edges are weighted with some cost. Field D* also operates on a graph, with three notable differences. Firstly, the graph is restricted to a grid structure. Secondly, rather than weighting nodes or edges, the squares of the grid, or *cells*, are weighted. Thirdly, instead of being restricted to deriving the cost of a node from the weight of a neighbouring edge and the cost of the edge's source node, the cost of a node may be derived from a path travelling through a neighbouring cell.

Algorithm 1 Field D*

```
1: function KEY(s)
2:     return [min(g(s), rhs(s)) + h(s_start, s); min(g(s), rhs(s))]
3: function UPDATENODE(u)
4:     if s was not visited before then g(s) = ∞
5:     if u ≠ s_goal then
6:         rhs(u) = min_(s',s'')∈connbrs(u) ComputeCost(u, s', s'')
7:     if u ∈ U then U.Remove(u)
8:     if g(u) ≠ rhs(u) then U.Insert(u, Key(u))
9: function COMPUTESHORTESTPATH
10:    while U.TopKey() < Key(s_start) OR rhs(s_start) ≠ g(s_start) do
11:        u = U.Pop()
12:        if g(u) > rhs(u) then
13:            g(u) = rhs(u)
14:            for all s ∈ nbrs(u)  UpdateNode(s)
15:        else
16:            g(u) = ∞
17:            for all s ∈ nbrs(u) ∪ {u}  UpdateNode(s)
18: function MAIN
19:    g(s_start) = rhs(s_start) = ∞; g(s_goal) = ∞
20:    rhs(s_goal) = 0; U = ∅
21:    U.insert(s_goal, Key(s_goal))
22:    loop
23:        ComputeShortestPath()
24:        if any cell weights have changed then
25:            for all cells x with new weights do
26:                for all nodes s on x do
27:                    UpdateNode(s)
```

The basic Field D* algorithm as presented in [27] is shown in Algorithm 1. $g(s)$ and $rhs(s)$ are, respectively the path cost and lookahead path costs at node s. U is the priority queue containing inconsistent nodes ($g(s) \neq rhs(s)$). Nodes are lexicographically ordered



(a) Traditional **(b)** Field D*

Fig. 1 The layout and possible transitions in traditional graph-based pathing and in Field D* planning. In traditional pathing (**a**), it is only possible to transition to a node from another node, along a graph edge. Field D* (**b**) relaxes this assumption to allow transitions across grid cells, from edges between nodes

on U by the tuple $Key(s)$. This ordering induces a tie-breaking rule—if the first members of two tuples are equal, the second member will be used to break the tie. s_{start} and s_{goal} are the start and goal nodes and $h(s_{start}, s)$ is a heuristic estimate of the cost of a path from s_{start} to s. $nbrs(s)$ are the neighbours of s, while $connbrs(s)$ are the consecutive node pairs surrounding node s ($connbrs(s_0) = \{(s_1, s_2), (s_2, s_3), \ldots, (s_8, s_1)\}$ in Fig. 1b for example). $ComputeCost(u, s', s'')$ is the cost of travelling from edge $s's''$, across a cell to node u and is the distinguishing feature of Field D*. The authors state that Field D* differs from D* Lite in lines 5–6 and lines 24–27.

2.2 Field D* cost functions

In traditional graph-based path planning, the path cost of a node is determined by finding the neighbouring node that is cheapest to transition from, as illustrated in Fig. 1a. This is expressed as:

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

$g(s)$ is the accumulated path cost at node s, $nbrs(s)$ is the set of neighbouring nodes of s and $c(s, s')$ is the cost of travelling from s' to s. This formulation restricts path transition to graph edges. Field D* relaxes this assumption on a grid graph to allow transitions across grid cells from a point on an edge between two nodes, as illustrated in Fig. 1b.

For this reason, Field D* weights *cells*, rather than nodes or edges. Ferguson et al. modify Eq. 1 to accommodate this. Most notably, for some point s_y on an edge between two nodes, the path cost, $g(s_y)$, is estimated by linearly interpolating between the node g values. In Fig. 1b for example, $g(s_y) = yg(s_2) + (1 - y)g(s_1)$ for some parameter $0 \leq y \leq 1$. The general path across half of a square cell is shown in Fig. 2a. c is the weight of the cell, b is the weight of the adjacent cell and s_1 and s_2 are the neighbouring nodes of s that are adjacent to each other. The variables x and y parameterise vectors $\overrightarrow{ss_1}$ and $\overrightarrow{ss_2}$ respectively. Equation 1 then becomes:

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

Ferguson et al. show how a variable can be eliminated from Eq. 2 to produce three different cases.

$$g(s) = min(b, c) + g(s_1) \tag{3}$$

$$g(s) = c\sqrt{2} + g(s_2) \tag{4}$$

$$g(s) = c\sqrt{1 + (1 - x)^2} + bx + g(s_2) \tag{5}$$

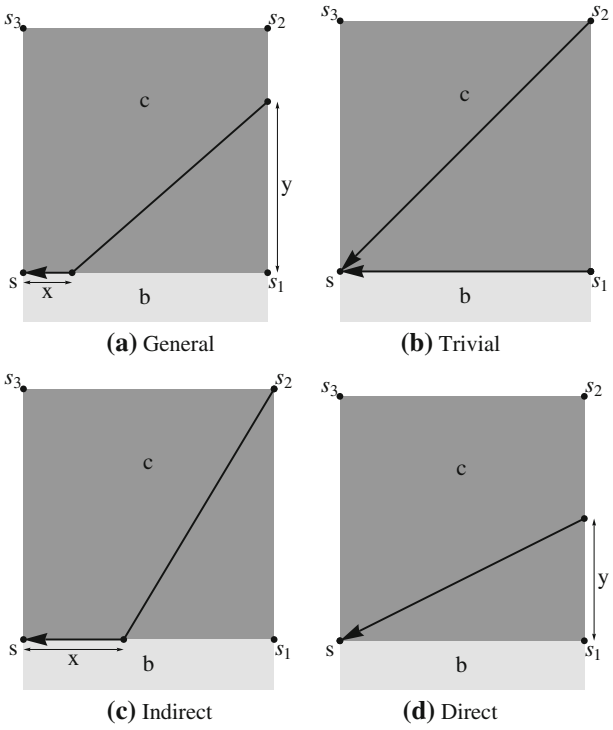


Fig. 2 The **a** General Field D* cost functions and three sub functions (**b–d**) derived by eliminating a variable

$$g(s) = c\sqrt{1+y^2} + (g(s_2) - g(s_1))y + g(s_1) \tag{6}$$

Figure 2b shows Eqs. 3 and 4, which are trivial in the sense that no minimisation over a variable is required to determine their optimal cost—it will always be constant. In the case of Eq. 3 the smallest value of b or c is chosen, as the path travels along the edge shared by two cells. Equations 5 and 6, illustrated by Fig. 2c and d respectively, are parameterised by a variable, which must be minimized to find the edge points that produce the cheapest cost.

This set of functions caters for the edge formed by nodes s_1 and s_2. Another set of cost functions must be solved for the edge formed by s_2 and s_3 to fully consider the optimal path across the cell. In this sense, the original Field D* algorithm finds the minimum cost path across each cell by subdividing it into two triangles. The minimum cost of the above cases is returned by the *ComputeCost* function in Algorithm 1.

Once Field D* has propagated costs to the appropriate nodes, the path between the start and the goal node must be extracted. Beginning with the start node,[1] the cost functions for a node are re-calculated to determine the point from which the node derived its cost. As the cost to travel from this point is the cheapest, it is the next point on the path. These points may lie on edges, the point between s_1 and s_2 in Fig. 2d for example. We elaborate on Field D*'s path extraction process in Sect. 4 for the sake of completeness.

[1] Field D* is derived from D* Lite and consequently propagates node costs from the goal node towards the start node. Path extraction is then performed in the opposite direction.

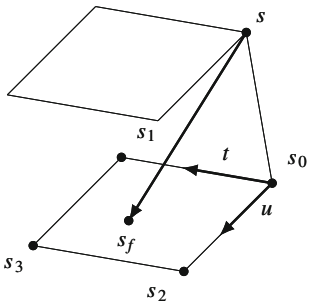


Fig. 3 3D Field D* parameterises s_f on face $s_0s_1s_2s_3$ with t and u

Two extensions to basic Field D*'s cost functions have been developed. *3D Field D** [15] extends Field D* to operate on a 3D grid by extending Field D*'s *Direct* cost function (corresponding to Fig. 2d and Eq. 6) to cubes. See Fig. 3 for the configuration. The authors state that no closed form minimization of this function exists in 3D and approximate the minimum to avoid the expense of numerical methods. For a cube face, this is accomplished by estimating the minima for the four edge cases, constructing two lines between the opposing estimated minima, and considering their intersection point as a possible minimum. Unfortunately, it not clearly explained how the edge minima are calculated. The authors state only that the process for finding the edge minima is similar to using the interpolation-based edge calculation $(yg(s_2) + (1 - y)g(s_1))$ for the 2D case, but do not elaborate further. This is an important point since if this is the basis of their technique, it is equivalent to using only the interpolation component $(g((s_2) - g(s_1))y + g(s_1))$ of Eq. 6 and neglecting the cost of travelling through the cell to a node $(c\sqrt{1 + y^2})$. This in turn implies that the estimation of these edge minima will only be accurate if the cell weight c is neglible compared to the interpolated cost between $g(s_1)$ and $g(s_2)$.

We note that the original Field D* cost functions could be used to estimate the minima along edges s_0s_1 and s_0s_2 in Fig. 3, since these form unit squares with node s. However, this is not the case for edges s_2s_3 and s_1s_3 since $|ss_2| = \sqrt{2}|s_2s_3|$, for example. Also, no *Indirect* cost function is presented for the case where the path travels partially along the side of a cube (2D equivalent is Fig. 2c, Eq. 5) and then cuts across it to an adjacent node.

*Generalized Field D** [28] modifies Field D*'s cost functions to operate on arbitrary triangles. These cost functions are expressed in terms of the edge lengths and angles of a triangle. This approach has a number of disadvantages. Firstly, if the angles and edge lengths are not precalculated, expensive trigonometric and square root operations are required to calculate these angles for each cost function. Alternatively, extra space would be required to store this data in a triangle. Secondly, an extension of this paradigm to 3D tetrahedra would be clumsy: Using 2D angles in a tetrahedron quadruples the number of angles and edge lengths, and true 3D angles (solid angles) are even more computationally expensive to maintain.

Our triangle cost functions express the mathematics in vector notation, reducing computational and space requirements, and allows an easier extension to 3D tetrahedra.

3 Cost functions

In this section we describe a general cost function of one variable, and how to efficiently minimize this function. It generalises the functions used to determine path costs across a triangle. (e.g. it could be applied to Eqs. 3–6). We show how to apply this minimization to

find paths through an arbitrary triangle on a 2D plane in Sect. 3.2. We also show how to find paths through arbitrary 3D tetrahedra by reducing to the 2D case in Sect. 3.3.

Three cases are presented for both triangles and tetrahedra, *Trivial*, *Indirect* and *Direct*. In the triangle case, two *Trivial*, two *Indirect* and one *Direct* cost functions must be evaluated. The least cost value produced by these functions is returned by the *ComputeCost* function. In the tetrahedral case, three *Trivial*, three *Indirect* and one *Direct* cost functions must be evaluated.

3.1 General cost function

The functions described later in this work require minimisation to find the cheapest cost across a triangle or tetrahedron. These problems can be reduced to solving a General Cost Function, whose solution and properties we will now describe. Let v_1, v_2 be non-zero, linearly independent vectors in $\mathbb{R}^n$ (for our purposes, we may assume $n = 2$ or 3). Let λ, μ, d be constants with $\lambda > 0$ and let x be a real variable. Let

$$G(x, \lambda, v_1, v_2, \mu, d) = \lambda \|v_1 + x v_2\| + \mu x + d \tag{7}$$

This is sometimes called the cost equation, but we will refer to it as the cost function, abbreviated as $G(x)$. In this section, we solve the problem of minimizing $G(x)$ for $x \in [0, 1]$. Let

$$l(x) = \|v_1 + x v_2\|$$

and note that

$$\begin{aligned} l(x) &= ((v_1 + x v_2).(v_1 + x v_2))^{1/2} \\ &= \left(\|v_1\|^2 + x^2 \|v_2\|^2 + 2x v_1.v_2\right)^{1/2}. \end{aligned}$$

For convenience, we let $a = \|v_1\|^2, b = \|v_2\|^2, c = v_1.v_2$ so that $l(x) = \left(bx^2 + 2cx + a\right)^{1/2}$.

Any local minimum of $G(x)$ must satisfy $0 = dG/dx = \lambda(bx+c)/\left(bx^2 + 2cx + a\right)^{1/2} + \mu$. Re-writing this as

$$\lambda(bx + c)/\left(bx^2 + 2cx + a\right)^{1/2} = -\mu \tag{8}$$

and squaring both sides yields the quadratic equation

$$b(\mu^2 - b\lambda^2)x^2 + 2c(\mu^2 - b\lambda^2)x + \mu^2 a - \lambda^2 c^2 = 0 \tag{9}$$

Note that in squaring, we may introduce extra solutions. In fact, in Eq. 8, we necessarily have $(bx + c)\mu < 0$ because $\mu > 0$ and v_1 and v_2 are linearly independent. Assuming this, the solutions to functions (8) and (9) are identical. If $\mu^2 - b\lambda^2 = 0$ then (9) has a solution if and only if $ab = c^2$, i.e. $\|v_1\|^2\|v_2\|^2 = (v_1.v_2)^2$, which is impossible by the Cauchy Schwartz inequality since v_1 and v_2 are linearly independent. If $\mu^2 - b\lambda^2 \neq 0$ then there are two solutions:

$$x = -\frac{c}{b} \pm \delta \tag{10}$$

where

$$\delta = \frac{\mu\sqrt{(\mu^2 - b\lambda^2)(c^2 - ab)}}{b(\mu^2 - b\lambda^2)}.$$

For these to be real, we require $(\mu^2 - b\lambda^2)(c^2 - ab) \geq 0$. By the Cauchy–Schwartz inequality, $c^2 - ab \leq 0$, so we require $\mu^2 < b\lambda^2$. Furthermore, as noted above we require $\mu(bx+c) < 0$, so that only the smaller root $(+\delta)$ satisfies (8) if $\mu > 0$, and only the larger $(-\delta)$ root does if $\mu < 0$. We have the following cases.

1. If $\mu = 0$, $G'(x)$ has a root at $x = -c/b$.
2. If $\mu^2 \geq b\lambda^2$, $G'(x)$ has no real root.
3. If $\mu^2 < b\lambda^2$, $G'(x)$ has a root at $x = -c/b + \delta$.

To determine whether a critical point is a local minimum, we consider the second derivative. We have

$$\begin{aligned}
\frac{d^2G}{dx^2} &= \lambda\left(\frac{l(x)b-(bx+c)l'(x)}{l(x)^2}\right) \\
&= \lambda\left(\frac{l(x)b-(bx+c)^2 l(x)^{-1}}{l(x)^2}\right) \\
&= \lambda\left(\frac{l(x)^2 b-(bx+c)^2}{l(x)^3}\right) \\
&= \lambda\left(\frac{ab-c^2}{l(x)^3}\right) \\
&= \lambda\left(\frac{\|v_1\|^2\|v_2\|^2-(v_1.v_2)^2}{l(x)^3}\right) \\
&> 0
\end{aligned}$$

again by the Cauchy–Schwartz Inequality. The fact that the second derivative is positive everywhere implies that the first derivative is strictly increasing on the whole of $\mathbb{R}$. There are three possibilities: If $G'(x)$ has a root α then $G(x)$ has a global minimum at α; if $G'(x)$ is positive everywhere, then $G(x)$ is strictly increasing; if $G'(x)$ is negative everywhere then $G(x)$ is strictly decreasing. The minimum value of $G(x)$ on the interval $[0, 1]$ therefore occurs at 0 and 1 in the second and third cases, respectively. Note that in the first case, the function $G(x)$ is strictly decreasing on $(-\infty, \alpha)$ and strictly increasing on (α, ∞).

Thus if $G(x)$ has a global minimum α that does not lie in the interval $[0, 1]$, the minimum on the interval $[0, 1]$ will occur at 0 if $\alpha < 0$ and at 1 if $\alpha > 1$.

3.2 Triangles

In this section we describe the cost functions for non-degenerate triangles. These can be thought of as embedded in $\mathbb{R}^2$ or in $\mathbb{R}^3$—the discussion will be the same, irrespective of the dimension of the ambient space.

Figure 4a shows the layout. Consider a triangle $\angle AB_1B_2$. We define the weight of the triangle as λ, the weight of the triangle opposite B_1 as λ_1 and the weight of the triangle opposite B_2 as λ_2.

Unless indicated otherwise, we will denote the cost at a point X by $g(X)$. Let the vectors corresponding to the vertices A, B_1, B_2 be w, v_1, v_2 respectively and let $u_1 = v_1 - w$, $u_2 = v_2 - w$ and $u_3 = v_2 - v_1$.

Trivial: Figure 4b illustrates trivial paths which travel along the edge of a triangle. In this case there is a unique path from B_1 to A and we have

$$g(A) = \min\{\lambda, \lambda_1\}|u_1| + g(B_1) \tag{11}$$

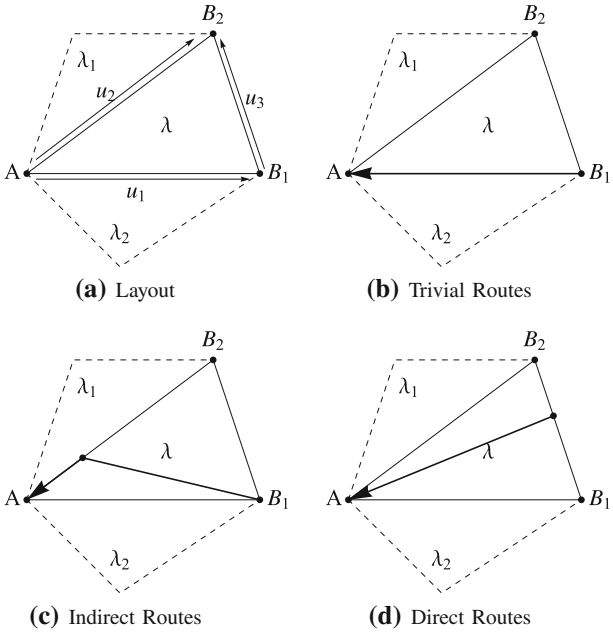


Fig. 4 The layout of a *triangle* is shown in (**a**). The *triangle* is defined by three vertices, A, B_1 and B_2. The *triangle* is weighted with value λ, while the triangles opposite B_1 and B_2 are weighted λ_1 and λ_2 respectively. **b–d** show the three types of path through a triangle

Indirect: Indirect paths originate at a node and cut across the main triangle to a point on the opposite edge, and then travel along this edge to the destination node, as shown in Fig. 4c. The intuition is that it is cheaper to travel some of the way through the adjacent triangle, rather than travelling the entire distance through the main triangle. We now express this problem in terms of the general cost function. We assume that the path originates at B_1, cuts across the triangle and travels along the edge opposite B_1 until it reaches A. The cost of this path can be expressed as

$$g(A) = \lambda \|u_1 - x u_2\| + \lambda_1 \|x u_2\| + g(B_1) \tag{12}$$

where x minimizes $g(A)$ and can be obtained by the method given above by noting that

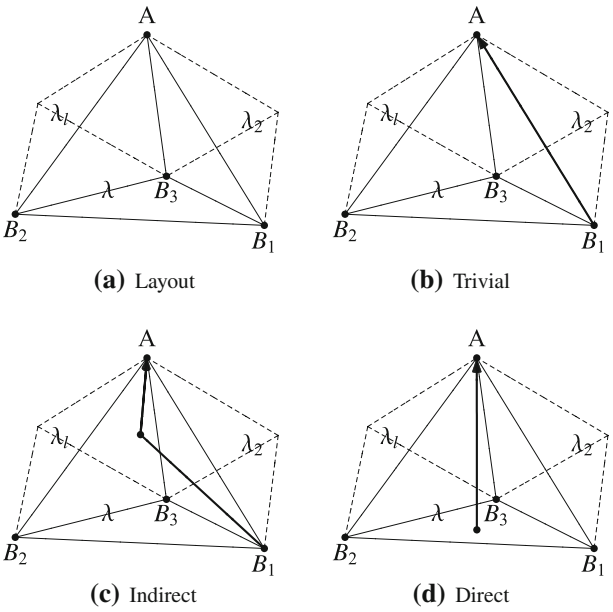
$$g(A) = G(x, \lambda, u_1, -u_2, \lambda_1 \|u_2\|, g(B_1)).$$

Direct: Figure 4d illustrates a direct path, which originates on an edge between two nodes B_1 and B_2 and travels straight through the main triangle to end at the destination node. It is on this path that the linear interpolation of Field D* is exercised. While the trivial and indirect paths both originate from a node B_1, adding $g(B_1)$ to their costs, the g value for a path originating on the edge $B_1 B_2$ must be estimated via interpolation. The cost function is formulated as:

$$g(A) = \lambda \|u_1 + x u_3\| + x g(B_2) + (1 - x) g(B_1) \tag{13}$$

This can be minimized by the method given above by noting that

$$g(A) = G(x, \lambda, u_1, u_3, g(B_2) - g(B_1), g(B_1)).$$



(a) Layout **(b)** Trivial

(c) Indirect **(d)** Direct

Fig. 5 The layout of a tetrahedron is shown in (**a**). The *tetrahedron* is defined by four vertices, A, B_1, B_2 and B_3. It is weighted with value λ, while the tetrahedra opposite B_1 and B_2 are weighted λ_1 and λ_2 respectively. (The tetrahedra opposite B_3, weighted with λ_3 is not shown.) **b–d** show the three types of path through a tetrahedron

3.3 Tetrahedra

In this section we describe the cost functions for non-degenerate tetrahedra in $\mathbb{R}^3$.

We define a tetrahedron by four vertices: A, B_1, B_2 and B_3. This layout is illustrated is Fig. 5a. We write the weight of the tetrahedron as λ, and the weights of the tetrahedra opposite B_1, B_2 and B_3 as λ_1, λ_2 and λ_3 respectively. We denote the vectors corresponding to A, B_1, B_2, B_3 by w, v_1, v_2, v_3 respectively. Set $u_2 = v_2 - v_1$ and $u_3 = v_3 - v_2$.

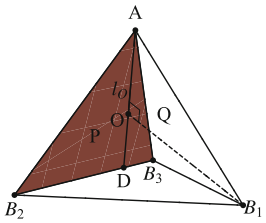
Trivial: Similar to the trivial triangle case, trivial tetrahedron paths originate at a node and travel along a tetrahedron's edge as show in Fig. 5b. The cost of the trivial path from B_1 to A is:

$$g(A) = \min\{\lambda, \lambda_2, \lambda_3\}|B_1A| + g(B_1) \tag{14}$$

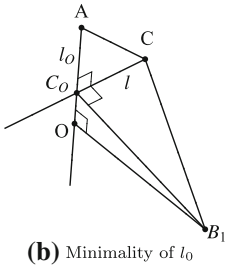
Indirect: Indirect routes originate at a node, cut across the tetrahedron and then travel across the opposing face to reach the destination node as shown in Fig. 5c. We assume that the node of origin is B_1. Since points on the face of a tetrahedron can be described by two parameters, a cursory formulation of the cost function might indicate that it is necessary to minimize a function of two variables. Specifically, if P is the plane passing through A, B_2 and B_3, and some point C lies on P, then

$$g(A) = \lambda|CB_1| + \lambda_1|CA| + g(B_1) \tag{15}$$

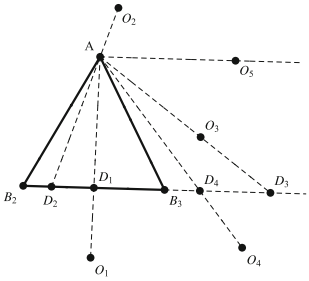
which would seem to require an iterative solution. However, we can show that there is a line in P on which the minimum occurs. Let O be the orthogonal projection of B_1 onto P and let Q be the plane containing A, B_1 and O. Note that $P \perp Q$. Let l_0 be the line containing AO; see Fig. 6a.



(a) Plane Q



(b) Minimality of l_0



(c) Possible positions of O relative to $\triangle AB_2B_3$

Fig. 6 **a** the plane Q resulting from the projection of B_1 onto plane P. **b** The fact that for any point $C \in P \setminus l_0$ its projection C_0 onto l_0 is closer to both B_1 and A than C

Let $\mathscr{L}$ be the set of all lines in P which are perpendicular to l_0. Consider a line $l \in \mathscr{L}$ and let C_0 be the point of intersection of l and l_0. Let C be a point on l distinct from C_0 and note that $\angle AC_0C$ is a right angle. It follows that $|AC| > |AC_0|$. Since l, l_0 and OB_1 are mutually orthogonal, $\angle CC_0B_1$ is a right angle. It follows that $|CB_1| > |C_0B_1|$. This is shown in Fig. 6b. Therefore, for C lying on the line l, the cost of the path $B_1 \to C \to A$ is minimized when $C = C_0$. If C_0 does not lie on $\triangle AB_2B_3$, the cost of the path is minimized by choosing C to lie on $\triangle AB_2B_3$ as close as possible to C_0, hence on the boundary of the triangle. Let D be the point of intersection of l_0 and line segment B_2B_3 (if it exists). We have shown that the minimum occurs either on AD or on the boundary.

Let u_4 be the vector for O. Let D be the point of intersection (if it exists) of the lines AO and B_2B_3. If O lies on $\triangle AB_2B_3$ then we need to minimize

$$g(A) = G(x, \lambda, w - v_1, u_4 - w, \lambda_1 \|u_4 - w\|, g(B_1)).$$

Alternatively, if O does not lie on $\triangle AB_2B_3$ then we need to consider only points on the triangle lying closest to the line l_0. The possible positions of O are shown in Fig. 6c with subscripts matching the cases laid out below. For the remainder of this subsection we will consider only line segments and not full lines. The cases are as follows.

1. If $D \in B_2B_3 \cap AO$ then we need only consider a point C lying on AD or on B_2B_3, the first of which we can solve as above. As for points on B_2B_3, this will be covered by the method indicated below.
2. If $D \in B_2B_3$ and $A \in OD$ then $g(A)$ is minimized by taking the trivial path from B_1 to A.
3. If $O \in AD$ then for $B_2 \in DB_3$, $g(A)$ is minimized by choosing a point C on AB_2, while for $B_3 \in DB_2$, $g(A)$ is minimized by choosing a point C on AB_3.
4. If $D \in AO$ then for $B_2 \in DB_3$, $g(A)$ is minimized by choosing a point C on AB_2 or B_2B_3, while for $B_3 \in DB_2$, $g(A)$ is minimized by choosing a point C on AB_3 or B_2B_3.
5. If AO and B_2B_3 are parallel, then for $\overrightarrow{AO}.\overrightarrow{B_2B_3} > 0$, $g(A)$ is minimized by choosing a point C on AB_3, while for $\overrightarrow{AO}.\overrightarrow{B_2B_3} < 0$, $g(A)$ is minimized by choosing a point C on AB_2.

We are reduced to minimizing the cost function for points lying on a single side of the triangle. For instance, for $C \in AB_2$ we have

$$g(A) = G(x, \lambda, w - v_1, v_2 - w, \min\{\lambda_1, \lambda_3\}\|v_2 - w\|, g(B_1)).$$

The cost functions for $C \in AB_3$ and for $C \in B_2B_3$ are similar.

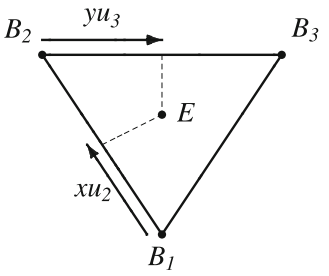
Direct: Direct routes originate on a face and cut across the tetrahedron to the destination node as shown in Fig. 5d. Here it is necessary to interpolate the g value from the three vertices of the face.

Let P be the plane containing B_1, B_2 and B_3. For a point E lying on $\triangle B_1B_2B_3$ we will consider the cost function of the direct path $E \to A$. Let v_4 denote the vector corresponding to the point E. Since u_2, u_3 generate P, we can write $v_4 = v_1 + xu_2 + yu_3$. The point E will lie on $\triangle B_1B_2B_3$ if and only if $0 \le y \le x \le 1$. This configuration is shown in Fig. 7a. We define our cost function h as follows:

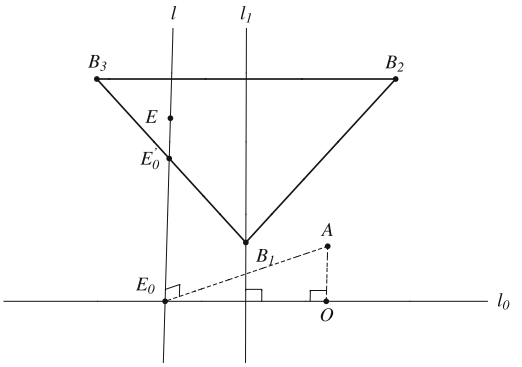
$$g(E) = \lambda|AE| + x(g_2 - g_1) + y(g_3 - g_2) + g_1 \tag{16}$$

where $g_i := g(B_i)$ for $i = 1, 2, 3$. The crux of our reduction to two dimensions will be to identify pencils of parallel lines in P such that the second part of this cost function, $\tilde{g}(E) = x(g_2 - g_1) + y(g_3 - g_2) + g_1$, is constant on each line in the pencil. Specifically, the linear functional $\tilde{g}$ on P is zero on the line $l_1 = v_1 + \{t(g_3 - g_2)u_2 + t(g_1 - g_2)u_3 \mid t \in \mathbb{R}\}$ and constant on any line in P parallel to l_1. Next we will identify a distinguished line in P. Let O be the orthogonal projection of A onto P. The vector corresponding to O may be written as $v_1 + x_0u_2 + y_0u_3$. Let l_0 be the line perpendicular to l_1 and passing through O. Let XY be the line segment consisting of the intersection of l_0 and $\triangle B_1B_2B_3$ (if it is non-empty). In contrast to the indirect case, the position of the candidate line l_0 for the minimum depends not only on the geometry of the tetrahedron but also on the weights g_1, g_2, g_3 at the nodes B_1, B_2, B_3.

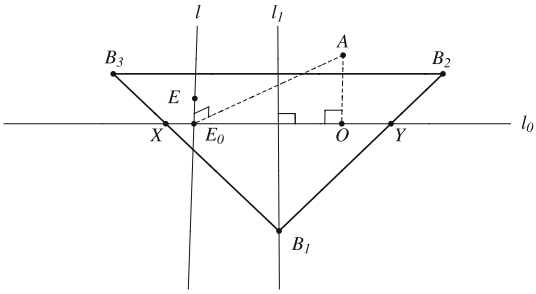
We next show that the minimum of g is achieved when $E \in \triangle B_1B_2B_3$ lies either on XY or on the boundary of $\triangle B_1B_2B_3$. (In fact one can do better than this, limiting to points on XY together with a subset of the perimeter of $\triangle B_1B_2B_3$ determined by a certain 'shadow' of the line l_0. However, for our purposes we shall be content to consider the entire perimeter.) The configuration is shown in Fig. 7b and c and goes as follows. Let l be a line perpendicular to l_0. We will consider how the cost function varies as E moves along l. As before let $v_1 + xu_2 + yu_3$ be the vector corresponding to E; then $l = v_1 + \{(x + t(g_3 - g_2))u_2 + (y + (g_1 - g_2))u_3 \mid t \in \mathbb{R}\}$. Let E_0 be the point of intersection of l and l_0, and let E_0' be the point on $EE_0 \cap \triangle B_1B_2B_3$ lying closest to E_0. Of course, if $E_0 \in \triangle B_1B_2B_3$ then $E_0' = E_0$ and if $E_0 \notin \triangle B_1B_2B_3$ then E_0' lies on the boundary of $\triangle B_1B_2B_3$. Since the lines l, l_0 and AO are mutually orthogonal, it



(a) Coordinate system of $\triangle B_1 B_2 B_3$



(b) Finding a minima on the boundary of $\triangle B_1 B_2 B_3$



(c) Finding a minima in $\triangle B_1 B_2 B_3$

Fig. 7 **a** How vectors v_1 and v_2 form a coordinate system in $\triangle B_1 B_2 B_3$. **b** Finding a minima on the boundary of $\triangle B_1 B_2 B_3$ and **c** finding a minima along the line XY within $\triangle B_1 B_2 B_3$

follows that $\angle AE_0E$ is a right angle. Thus the length of AE increases as E traces a path from E_0 along l. In particular, we have $|AE_0'| \leq |AE|$. Recall that $\tilde{g}$ is constant on l. It follows, therefore, that the cost function g, restricted to line segment EE_0', is minimal at E_0'. We have thus established our claim. Unlike the indirect case, here we will need to compute the cost functions for each side of $\triangle B_1 B_2 B_3$ as well as for the line segment XY (if it exists). The vector u_4 for O can be obtained in the same way as described in the indirect case. The line l_0 can be parametrised as $\{\gamma u_5 + u_4 \mid \gamma \in \mathbb{R}\}$ where u_5 is any non-zero solution to the equation $u_5 \cdot ((g_3 - g_2)u_2 + (g_1 - g_2)u_3) = 0$. The vectors for X, Y, say v_X, v_Y respectively, can be found in terms of the vectors for O, B_1, B_2, B_3 (this is easily done once a coordinate system is chosen for the plane P). Let g_X, g_Y denote $\tilde{g}(X)$, $\tilde{g}(Y)$ respectively. Then $g(A)$ is obtained as the minimum of the minima of the following functions:

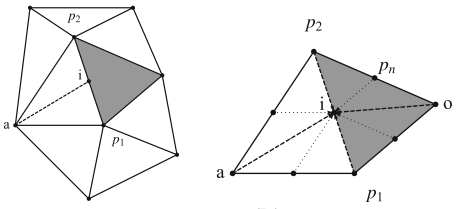
$$G(x, \lambda, w - v_X, v_Y - v_X, g_Y - g_X, g_X),$$
$$G(x, \lambda, w - v_1, v_2 - v_1, g_2 - g_1, g_1),$$
$$G(x, \lambda, w - v_2, v_3 - v_2, g_3 - g_2, g_2),$$
$$G(x, \lambda, w - v_3, v_1 - v_3, g_1 - g_3, g_3).$$

4 Path extraction

After propagating costs to the appropriate nodes, the path is extracted in an iterative process, beginning at the start node. Firstly, the start node is added to the path. Then, the cost functions of the last node on the path are re-evaluated to determine the point from which it derived its cost. As the cost to travel from this point is the cheapest, it is the next point on the path. This continues until the goal node is reached. The pseudocode for this process is shown in Algorithm 2. It is slightly more compact and general than the path extraction pseudocode provided for Generalized Field D* [28].

Algorithm 2 Path extraction. ComputeCost(s, a) computes the cost of travelling to node s across cell a. The cell subdivision process in InterpolatedChild is illustrated in Fig. 8b

```
1: function InterpolatedChild(p)
2:     Subdivide cells adjacent to p into temporary cells
3:     b_c ← ∞; b_p ← NULL
4:     for all temporary cells b do
5:         if ComputeCost(p, b) < b_c then
6:             b_c ← ComputeCost(p, b)
7:             b_p ← point associated with cost b_c.
8:     Return {b_c, b_p}
9: function ExtractPath
10:    s ← s_start; PATH={s_start}
11:    while s ≠ s_goal do
12:        if s is an interpolated point then
13:            {c, p} ← InterpolatedChild(s)
14:            s ← p; PATH=PATH∪{p}
15:        else
16:            a ← arg min_{c∈cellnbrs(s)} ComputeCost(s, c)
17:            A = {a1_c, ··· , ak_c, d_c} ← costs across a
18:            d_c ← cost of the Direct Path through a.
19:            d_p ← interpolated point associated with cost d_c.
20:            if d_c = ComputeCost(s, a) then                ▷ Direct Path is cheapest
21:                {b_c, b_p} ← InterpolatedChild(d_p)
22:                Update d_c ∈ A with b_c                    ▷ Check the estimate
23:            c ← min(A)
24:            s ← point(s) associated with c
25:            PATH=PATH∪{s}
```



(a) Incorrect Interpolation Assumption

(b) Lookahead

Fig. 8 **a** The interpolation cost at i may be a bad estimate since it is expensive to travel through the *grey triangle*. **b** The interpolation cost estimate is tested by subdividing the two *triangles* sharing the edge containing the interpolated point into four *subtriangles* and evaluating the cost functions originating at the surrounding nodes and edges

If the cheapest point to travel to is produced by a *Trivial* cost function then the next point is a node point. If produced by an *Indirect* cost function, then both an edge point and a node point are added to the path. In the *Direct* case, the interpolated point lying an edge or face is added to the path. Ferguson et. al. recommend a check of the interpolated cost at this point since it may, in fact, be incorrect.

To see why this may be the case, consider Fig. 8a. The grey triangle is expensively weighted, while the others are weighted cheaply. At node a, an evaluation of the cost functions suggests that the cheapest point to transition from is an interpolated point i, lying on the edge between p_1 and p_2. However, at i, the cheapest point to transition from would be p_1 or p_2 since it would be prohibitively expensive to travel through the grey triangle—the path from either p_1 or p_2 to a would be cheaper. The interpolation assumption is incorrect because the grey triangle is expensive and therefore the path must flow around instead of through the triangle. A better estimation of the cost at i would be derived as $g(i) = c\|i - p_1\| + g(p_1)$ for example, instead of interpolating between $g(p_1)$ and $g(p_2)$.

For this reason, it is necessary to perform a lookahead operation at interpolated points that checks the interpolated cost estimate. Firstly, the two triangles sharing the edge containing the interpolated point are subdivided into four triangles, with the interpolated point, i, at their apex. Then, the costs of travelling to i from the surrounding nodes and edges of the four sub-triangles are evaluated as illustrated in Fig. 8b. Both *Trivial* cost functions originating from nodes and *Direct* cost functions originating from edges [2] are evaluated and the cheapest of these costs replaces the interpolated cost.

Using this improved estimate, the extraction algorithm decides if the interpolated point is still the cheapest to transition from, compared to the original *Trivial* and *Indirect* cost functions, and if so it is added to the path. A useful side-effect of this operation is that if the lookahead confirms the interpolated cost, the point producing the cheapest lookahead cost can be used as the next point on the path.

An example referring to Fig. 8b: evaluating cost functions at node a indicates that the cost for a is derived from interpolated point i. The two triangles are subdivided and the cost functions of the four subtriangles triangles with apex i are evaluated. These costs are used to test the interpolated cost at point i. If all these costs are greater than the cost at i, it is rejected as the next point and p_1 or p_2 are considered. However, if there are costs that are equal to or less than that at i, the interpolated cost is confirmed, i is added as the next point on the path, and the point producing the least cost, p_n, for example, is evaluated next.

Note that a triangle and tetrahedral version of Field D* enables the subdivision of cells around an interpolated point into triangles and tetrahedra respectively. Consequently, the triangle and tetrahedral cost functions can be used to evaluate the cost of travelling across these temporary cells. In contrast, subdividing around interpolated points in Field D* and 3D Field D* will produce rectangles and cuboids, but the cost functions associated with these implementations only operate on squares and cubes respectively. It is not clear in Field D* [27] or 3D Field D* [15] whether these cost functions are employed during path extraction. In fact, [27] suggests using a local planner to perform path extraction instead.

[2] In the 3D case *Direct* cost functions originating from the surrounding tetrahedra faces are evaluated. Also, interpolated points may lie on tetrahedra edges or faces.

5 Caching

In this section, we describe how the path-finding algorithm can be made more efficient by caching calculations that remain constant regardless of the search parameters.

We have defined the cost functions for triangles and tetrahedra in terms of $G(x)$. A characteristic of $G(x)$ is that parameter d is not utilised in finding the roots in Eq. 10. Now, as long as parameters λ, v_1, v_2 and μ are calculated with constants, the roots of such functions can be cached.

If the mesh, and the weighting of the mesh remain constant, then the weights and vectors derived from the triangles and tetrahedra will also remain constant, regardless of the search parameters. The only values that change are the $g(p)$, representing the accumulated cost of the search at node p. Thus, if parameters λ, v_1, v_2 and μ of $G(x)$ do not contain $g(p)$ values, their roots can be cached. Additionally, since d is merely a scalar value added to the rest of the $G(x)$, the bulk of the cost calculation can also be cached.

On examining the cost functions, it can indeed be seen that the trivial and indirect cost functions for both triangles and tetrahedra only have $g(p)$ values in parameter d. Thus, their roots and the sections of $G(x)$ composed from λ, v_1, v_2 and μ can be also be cached.

We can further exploit the fact that a pair of trivial and indirect cost functions originate from the same node. For triangles, for example, one trivial and one indirect path originate from p_1. It is only necessary to store the root and cached cost for the least expensive path originating from p_1, since $g(p_1)$ will be added to the cost functions for both paths. The type of path can be indicated in the cached root via the use of ranges. For example, if the cached root and cost is for an indirect path, then $0 \leq \text{root} \leq 1$, but if they represent a trivial path, $\text{root} = 2$ for instance.

Thus for a triangle, two pairs of roots and costs need to be stored at each triangle vertex, resulting in 12 cached values. If each value is represented by a four byte floating point variable, 48 bytes of cache are required per triangle. For a tetrahedron, three pairs of roots and costs must be stored at each vertex, resulting in 24 cache values and 96 bytes of cache per tetrahedron.

To obtain performance gains from caching, the mesh and the triangle or tetrahedron weights should remain reasonably static, since changes to these values will require recalculating cached values for the modified triangle and its neighbours. In cases where the number of triangle weights changes are small, it may be feasible to recalculate cached values, but the performance gained from caching would be lost if the weighting and structure of large portions of the mesh change constantly.

6 Replanning

As stated earlier, Field D* is able to replan paths should grid cell weights change after a path has been computed. In lines 24–27 in Algorithm 1, if a grid cell weight is changed, then *UpdateNode* is invoked on the nodes on the corner of these cells, updating the RHS-values. Then, *ComputeShortestPath* is invoked to propagate the changed node values.

Similarly, if the weight of triangles or tetrahedra change, *UpdateCost* can be invoked on the nodes of these structures. Our extension to Field D*'s cost functions does not modify it's basic replanning capability and while we have not specifically investigated this part of the algorithm, this capability can be used as is to perform replanning on weighted triangulations and tetrahedralisations.

7 Results

In this section, we discuss results related to our Field D* implementation. Firstly, we compare the expense of our cost functions to those of Generalized Field D*. Secondly, we show how our Triangle implementation of Field D* provides superior performance to that of a Quadtree implementation, when the world data is not grid-aligned. Thirdly, we provide results for our 3D Tetrahedral implementation of Field D* and lastly, demonstrate the gains that can be obtained from caching.

We implemented Field D* using C++ and used a binary heap to represent the priority queue driving the algorithm. Random deletes of priority queue elements were optimised to bubble the element out of the queue, instead of deleting the element and shifting the array. Likewise, priority queue key updates were optimised to bubble the queue element to the new location. In terms of heuristics, we used the version suggested by Ferguson et. al. whereby the Euclidean distance is multiplied by half of the minimum weight in the triangulation/tetrahedralisation: $0.5 * minval * \sqrt{dx^2 + dy^2}$.

7.1 Performance comparison of Generalized Field D* and Triangulated Field D*

Generalized Field D* [28] evaluates the Field D* cost functions on a triangle using the inner angles and side lengths of that triangle. In contrast, our implementation of the Field D* cost functions for triangles is based on vector calculus operations on the points defining the triangle. Thus, Generalized Field D* must either calculate the angles and side lengths every time a triangle is processed, or store these values in addition to the triangle points. Additionally, both implementations produce two roots when minimising the indirect and direction cost functions, but our formulation of the general cost function presented in Sect. 3.1 allows our implementation to predict which root to use, which means that only the cost for one root must evaluated. Based on this reasoning, we expect that our vector calculus implementation of cost functions for triangles would be less expensive than those of Generalized Field D*.

To confirm this, we created a million random triangles and compared the time taken by our implementation and Generalized Field D* to evaluate their cost functions over 100 iterations, in addition to the space required for each implementation. Two versions of the Generalized Field D* cost function were implemented, one where the triangle edge lengths and trigonometric angles values are calculated for each cost function, and one in which they are cached. Note that for the same triangle, Generalized Field D* produces the same costs as our implementation, but uses a different formulation. For this reason, we compare the performance of the two techniques on the same triangle. Table 1 shows these results.

Our implementation based on vector calculus takes 13.12s to complete, requiring 28 bytes for the representation (six four bytes floats for the coordinates and one for the triangle weight). Our basic implementation of the Generalized Field D* cost functions takes 20.53s with the same representation, as the side lengths and trigonometric values must be calculated when

Table 1 Comparison of the time and space required by our triangle Field D* cost function implementation versus Generalized Field D*

	Vector Field D*	Generalized Field D*	Generalized Field D* (Cached)
Time (s)	13.12	20.53	14.83
Space (bytes)	28	28	64

evaluating a triangle's costs. Caching these values (three sines, three cosines and three edge lengths) results in a execution time of 14.83s, which is only slightly slower than our implementation. This is probably because three more cost functions must be evaluated to determine the correct root to use.

In summary, our vector implementation of the triangle cost functions is around 1.56 times faster than Generalized Field D*. Even if the various edge lengths and trigonometric values of Generalized Field D* are cached, our implementation is faster and requires less than half the space.

7.2 Comparison of Multi-resolution Field D* and Triangulated Field D*

We have extended Field D* to triangulations since triangulations represent general polygonal objects more accurately than grids and quad-trees. This is because triangles can represent polygonal objects exactly, since the interior of a polygonal object can always be subdivided into triangles. Grids or quad-trees, however, will always be subject to geometric error, unless that object's boundaries are grid-aligned. This implies that a grid or quad-tree requires high levels of subdivision to accurately represent polygonal objects. Additionally, since Field D* computes approximate paths across cells due to interpolation error, increasing the level of subdivision in either case should improve this approximation. In [28], the authors perform a single simple experiment showing that triangle-based Generalized Field D* is an improvement over the original Field D* in terms of node expansions. In the interests of generality, we perform several experiments contrasting our scheme with Multi-resolution Field D* as [13] shows that it provides time and space improvements over the original Field D*. In this section we demonstrate the *reduction in computational cost* that is afforded when one allows pathing through a triangulated, rather than grid-based, environment.

To this end, we compare the paths produced by Field D* implementations for quad-trees and triangulations at different levels of subdivision and demonstrate that, due to geometric error, a quad-tree requires a far higher level of subdivision than a triangulation to produce paths of similar cost. We also show that increasing the subdivision reduces interpolation error in both cases. We implemented two versions of Field D*, one based on the triangle cost functions described in this paper, and the other based on the quad-tree cost functions described by Ferguson et. al [13].

7.2.1 Triangulation construction

We construct *Constrained Triangulations*, which allow the specification of constraints in the form of edges that must be present in the triangulation. Therefore, if an environment is constructed out of a set of weighted, non-intersecting polygons, we derive a constrained triangulation by inserting polygon edges as constraints and weighting the triangles internal to the polygon with the polygon's weight. A *Constrained Triangulation* generates a relatively coarse mesh. We apply *Delaunay Refinement* [33] on the mesh to produce a finer *Constrained Delaunay Triangulation* that respects the original constraints. Triangles in a *Delaunay Triangulation* satisfy criteria that discourage thin triangles or slivers.

7.2.2 Quad-tree construction

Quad-trees [14] are restricted to representing polygonal data with squares or *cells*. To construct a quad-tree, we first subdivide the world into a square grid whose sides are a power of two. Then, we determine which polygons intersect each grid cell. If a polygon intersects a

cell, we store the area of intersection as well as the polygon's weight in a list of tuples within the cell as $\{\{a_1, w_1\}, \{a_2, w_2\}, \ldots, \{a_n, w_n\}\}$. The weight of the grid cell is then calculated as the sum of the products of each area-weight pair, divided by the total area of the cell a_c. Since the cell cannot represent the polygons intersecting it with complete accuracy, there is an error associated with the cell's weight which measures how accurately the quad-tree models the original polygonal representation. Given this cell weight, $\bar{w}$, the Root Mean Square Error, or L^2 error for the cell weight can also be calculated from the area-weight tuples.

$$\bar{w} = \frac{1}{a_c} \sum_{1}^{n} a_k w_k \tag{17}$$

$$L^2 = \sqrt{\sum_{1}^{n} [a_k (\bar{w} - w_k)]^2} \tag{18}$$

We then construct a quad-tree via the normal process of aggregating child cells with equal weights. Our quad-tree implementation trades space for time in that it stores references to neighbouring cells within a cell, rather than determining the neighbours at execution time.

7.2.3 Test environments

We constructed four environments, shown in Fig. 9, to contrast the paths produced by quad-tree and Triangulated Field D*. In each environment, we calculated the path between predefined start and goal points in the lower left and upper right corners, respectively, at differing levels of subdivision for both quad-tree and triangulation. Results for these paths are shown in Table 2, which details the path cost, number of faces, number of node expansions, path costs and L^2 error. In this table, quad-tree faces are square cells, whereas triangulation faces are triangles. However, it should be noted as in Sect. 2 that basic Field D* treats a cell as two triangles when computing cost functions. Quad-tree Field D* may treat a cell as a composition of even more triangles, if the cell has higher resolution neighbours, as shown in Fig. 12. Since this "decomposition" occurs during the runtime evaluation of cost functions it is difficult to directly compare quad-tree-generated triangles to pre-calculated triangles and we must instead compare squares to triangles. For this reason, we consider the number of faces to be prejudiced in Quad-tree Field D*'s favour. The number of node expansions refers to the numbers of nodes popped off the priority queue in order for the algorithm to complete.

It is an interesting exercise to compare the path costs produced by Field D* with those of an A* implementation. We created a directed graph from the edges of the various subdivisions of our Voronoi Diagram environment. The edges are weighted by their length multiplied by the minimum weight of the adjacent cells and we used a heuristic of the minimum triangle weight multiplied by the Euclidean distance. The results of A* searches on these constructed graphs are shown in Table 3.

The first environment is a grid maze (Fig. 9a) and we use it to show how quad-tree and triangulation implementations compare when data is grid-aligned and no geometric error is present. Since the data is aligned to a grid, a quad-tree cell represents a grid cell exactly and does not overlap with other grid cells. To subdivide in the quadtree case, we simply split the grid squares into four smaller squares at each level, rather than using normal quadtree decomposition. The triangulation is subdivided with the usual Delaunay Refinement, with the original grid squares as constraints. The polygons representing the other three environments are not grid-aligned in the sense that polygons may not necessarily fit exactly into a quad-tree

Table 2 The path cost, number of faces, number of node expansions, path lengths, time taken to find a path and normalised l^2 error for Field D* implemented on a quad-tree (Q) versus a triangulation (T)

Path cost		Faces		Node expansions		Path length		Time (s)		Normalised L^2 error
Q	T	Q	T	Q	T	Q	T	Q	T	Q
Grid maze										
16.69	17.03	1,682	3,362	977	964	165.87	166.31	0.02	0.01	N/A
16.65	16.93	6,725	12,984	2,929	2,682	165.79	166.68	0.06	0.01	N/A
16.63	16.90	14,885	13,858	5,840	2,943	165.76	166.56	0.11	0.02	N/A
16.62	16.83	26,897	25,866	9,685	5,492	165.74	166.49	0.16	0.04	N/A
16.61	16.78	41,617	41,316	14,464	7,710	165.73	166.17	0.25	0.05	N/A
16.61	16.75	60,517	61,006	20,155	10,919	165.72	166.13	0.35	0.07	N/A
16.60	16.74	81,797	82,366	26,762	13,726	165.72	166.17	0.48	0.10	N/A
Randomly weighted Voronoi Diagram										
12537.23		1,025		1,089		167.22		0.02		0.210
10480.93		4,076		4,220		181.54		0.06		0.141
9421.86	8439.44	14,405	16,789	15,673	9,511	182.33	184.31	0.24	0.05	0.079
8938.80	8414.64	40,046	41,865	47,669	21,727	184.04	182.82	0.76	0.12	0.041
8710.31	8404.57	95,780	92,344	120,254	47,238	183.76	182.77	2.03	0.26	0.021
8570.52	8390.41	211,697	219,242	273,470	111,659	182.55	182.42	4.87	0.65	0.011
8490.43	8386.65	447,932	461,450	587,532	234,162	182.01	182.31	10.82	1.49	0.005
Axis-aligned world										
1204.32		839		944		117.40		0.02		0.537
368.68	37.31	2,564	2,586	2,218	874	282.21	365.74	0.04	0.01	0.356
39.50	37.03	6,722	6,753	3,922	1,942	392.10	365.02	0.09	0.02	0.189

Table 2 Continued

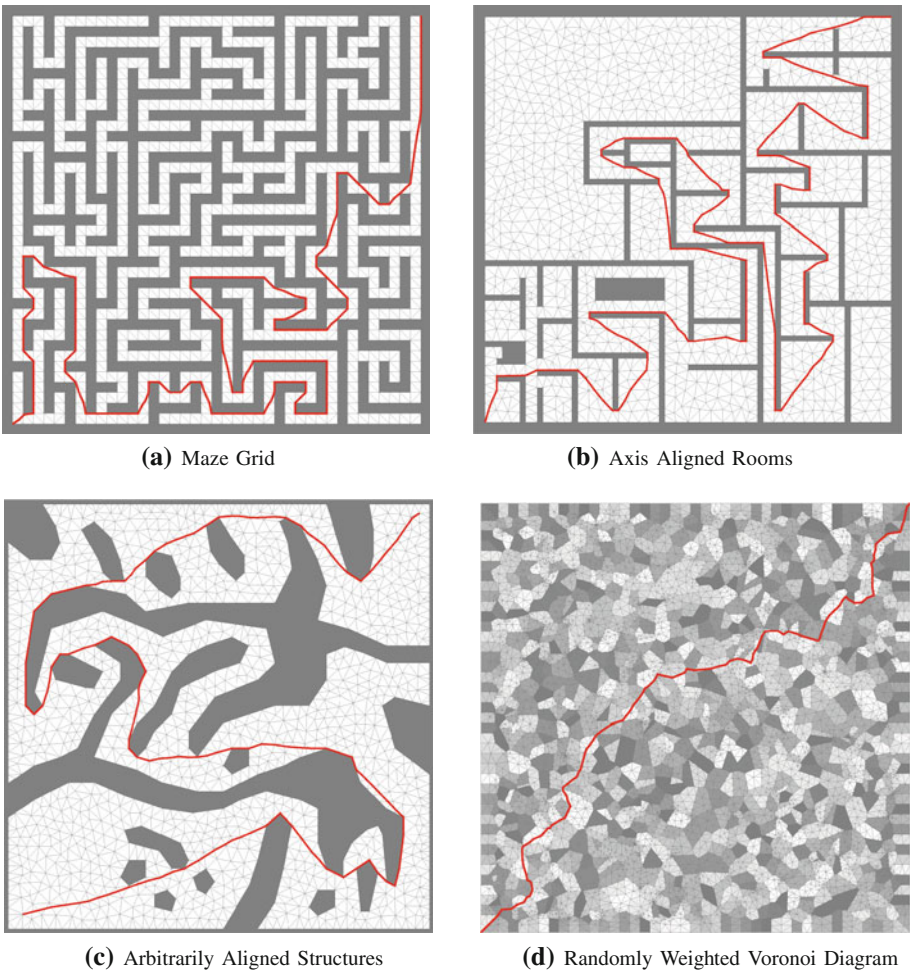
Path cost		Faces		Node expansions		Path length		Time (s)		Normalised L^2 error
Q	T	Q	T	Q	T	Q	T	Q	T	Q
37.98	36.87	15,389	15,366	9,417	4,396	377.06	364.56	0.21	0.03	0.081
37.47	36.74	32,738	32,479	20,884	8,480	371.96	364.13	0.57	0.06	0.044
37.16	36.63	67,412	68,643	46,424	17,443	368.79	363.77	1.29	0.11	0.027
37.00	36.55	136,694	135,177	104,728	33,471	367.32	363.51	1.89	0.22	0.015
Arbitrarily-aligned world										
1081.89	39.68	899	1676	806	633	210.86	388.31	0.01	0.01	0.321
42.45	39.33	2,588	2,722	1,848	1,056	422.03	387.02	0.04	0.01	0.176
40.76	39.18	6,239	6,315	4,360	2,399	404.82	386.15	0.09	0.02	0.088
39.80	38.97	13,535	13,340	9,959	4,910	395.20	385.38	0.19	0.03	0.045
39.21	38.84	28,178	28,620	21,904	10,258	389.52	385.11	0.39	0.06	0.018
38.94	38.73	57,479	57,857	49,487	20,120	387.09	384.74	0.87	0.13	0.010
38.82	38.66	116,177	115,644	112,041	39,652	386.13	384.67	2.04	0.26	0.006

The normalised L^2 error measures the geometric error in the quad-tree representation. Where possible, each row presents data for a similar number of quad-tree and triangulation faces, but this is not always possible when the two structures are at low resolution. In the case of the Voronoi Diagram for example, a minimum of around 16700 triangles is required to produce a Delaunay Triangulation. The italicized quad-tree path costs indicate instances where, due to geometric error in the representation, the path travels through expensive cells. These data points are not plotted in the following graphs since their magnitude is too great

Table 3 Comparison of path cost, node expansions and time taken between A* on a triangulation (TA*) and Field D* (TFD*)

Path cost		Faces	Node expansions		Time (s)		Path cost	Faces	Node Exp	Time (s)
TA*	TFD*		TA*	TFD*	TA*	TFD*	Grid A*			
Randomly weighted Voronoi Diagram										
8631.18	8439.44	16,789	9,760	9,511	0.0029	0.05	9620.49	16,384	16,640	0.0059
8655.48	8414.64	41,865	20,875	21,727	0.0068	0.12	9164.06	65,536	66,048	0.0287
8589.95	8404.57	92,344	50,007	47,238	0.0223	0.26	8924.06	262,144	263,138	0.1279
8563.51	8390.41	219,242	114,587	111,659	0.0593	0.65	8800.96	1,048,576	1,050,624	0.6090
8550.83	8386.65	461,450	231,625	234,162	0.1385	1.49	8744.71	4,194,304	4,198,400	2.9960
Axis-aligned world										
38.22	37.31	2,586	900	874	0.0003	0.01	369.32	4,096	3,334	0.0011
38.00	37.03	6,753	1,942	1,942	0.0006	0.02	40.06	16,384	7,620	0.0028
37.94	36.87	15,366	4,318	4,396	0.0015	0.03	38.77	65,536	30,633	0.0133
37.73	36.74	32,479	8,251	8,480	0.0030	0.06	38.29	262,144	121,843	0.0580
37.62	36.63	68,643	17,052	17,443	0.0074	0.11	38.01	1,048,576	487,144	0.2630
37.47	36.55	135,177	32,521	33,471	0.0152	0.22	37.9	4,194,304	1,947,273	1.1810
Arbitrarily-aligned world										
39.92	39.33	2722	965	1056	0.0003	0.01	43.17	4096	2513	0.0009
39.97	39.18	6315	2155	2399	0.0007	0.02	41.67	16384	9724	0.0033
39.8	38.97	13340	4482	4910	0.0016	0.03	40.90	65536	38874	0.0169
39.75	38.84	28620	9276	10258	0.0034	0.06	40.48	262144	154560	0.0769
39.74	38.73	57857	17994	20120	0.0076	0.13	40.28	1048576	615761	0.3250
39.55	38.66	115644	34768	39652	0.0198	0.26	40.20	4194304	2461271	1.5420

Table rows are ordered by the number of faces in the environment. In the Voronoi Diagram, TFD* provides a better path cost compared to TA* (8439.44 vs 8550.83) on a more coarsely triangulated graph (16,789 vs 461,450 faces) in a faster time (0.05 vs 0.1385 s). In the Axis-Aligned and Arbitrarily-Aligned Worlds, Field D* provides better path costs in equivalent time with fewer faces. The last four columns tabulate data for A* on a grid. Grid A*'s path costs converge much slower than TA* and TFD* and require many more faces. Consequently, the Grid A* data is not directly comparable to TA* and TFD* on the same row, but is provided for completeness



(a) Maze Grid **(b)** Axis Aligned Rooms

(c) Arbitrarily Aligned Structures **(d)** Randomly Weighted Voronoi Diagram

Fig. 9 Environments used in comparing Quadtree Field D* to Triangulated Field D*. **a** is a grid-aligned maze and is designed to contrast the two implementations to a case where no geometric error is present. **b** is a connected series of axis-aligned rooms, while **c** consists of arbitrarily aligned structures. **d** Randomly weighted Voronoi Diagram. *Darker regions* are weighted more heavily, while the *lighter regions* have lesser weightings

cell. This discrepancy in representation is quantified by the L^2 error metric we mentioned previously.

The second environment (Fig. 9b) is a series of interconnected, axis-aligned rooms. The third (Fig. 9c) consists of arbitrarily aligned structures, while the fourth (Fig. 9d) is a randomly weighted Voronoi Diagram. These last three environments are approximated by quadtree subdivision and are consequently subject to geometric error. The grid maze, axis-aligned and arbitrarily aligned world have their open space and obstacles weighted with 0.1 and 255 respectively. The Voronoi Diagram cells are randomly weighted with multiples of 16, clamped between 0.1 and 255. We have graphed the relationship between the normalised path cost and the number of faces in the environment in Fig. 10, and the normalised path cost

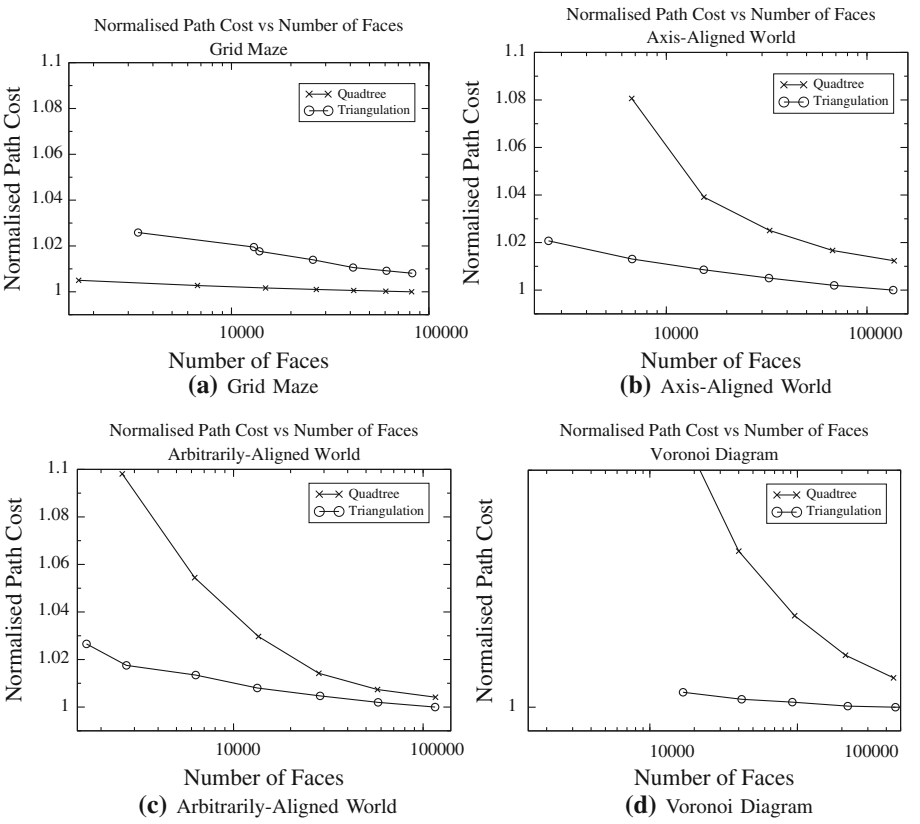


Fig. 10 Normalised path cost versus number of faces

and the number of node expansions required by the algorithm in Fig. 11. We normalise the path costs for a particular environment by dividing path costs by the minimum path cost.

7.2.4 Discussion

The path costs for the grid maze decrease slowly as environmental subdivision increases for both the quadtree and the triangulation, with the path costs for the quadtree case being slightly lower than those of the triangulated case. This is because the environment is a grid, which ensures that cell edges will largely be parallel with the direction of the path. This provides superior interpolation results since, when edges are not parallel to the path direction, one of the nodes of the edge being interpolated is favoured, causing the path to "hug" or travel directly along an edge connected to the node. While both quadtree and triangulated variants are subject to this edge-hugging behaviour, the subdivision of the grid environment favours the quadtree slightly in this regard. In Fig. 9a for example, the grid cells in the upper right corner are mostly subdivided from the top left to the bottom right corner of the cell. The bottom right corner is favoured, causing the algorithm to "hug" the right wall.

However, in the other three environments, the quadtree requires an order of magnitude more faces to produce a path cost similar to that of the triangulation at the lowest subdivision level. In the axis-aligned world for example, 2,586 triangles produce a path cost of 37.31,

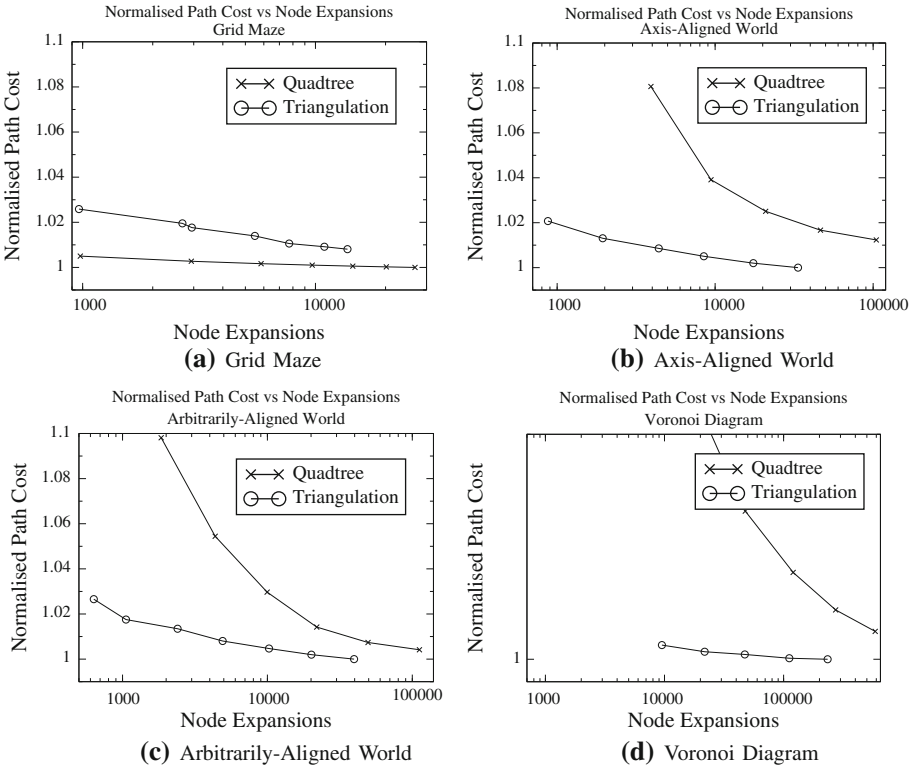


Fig. 11 Normalised path cost versus node expansions

while 12.7 times (32,738) more quadtree faces are required to produce a slightly higher path cost of 37.47. In the arbitrarily-aligned world, 1,676 triangles produce a path cost of 39.68, while 8 times (13,535) more quadtree faces are required for a higher path cost of 39.8. As the number of faces used to represent the environment grows and geometric error decreases, the quadtree begins to produce improved path cost estimates as can be seen in Fig. 10.

Since the Delaunay Triangulation requires a minimum of around 16,500 triangles to represent the Voronoi Diagram, it was not possible to compare path costs at 1,024 and 4,076 quad-tree faces respectively. The path cost of 8439.44 for 16,789 triangles beats the quadtree path costs at all levels of subdivision so there are no comparable data points, but Fig. 10d shows that the Voronoi Diagram exhibits a similar graph profile to the axis-aligned and arbitrarily-aligned world for the relationship between path cost and number of faces.

The quadtree representation of the Voronoi Diagram starts with a normalised L^2 of 0.210 at the lowest level of subdivision, while the quadtree representations of the axis-aligned and arbitrarily-aligned worlds start with much higher Normalised L^2's of 0.537 and 0.321, respectively. This indicates that the quadtree has difficulty in accurately representing these structures at low resolutions. Regions of high and low cost may be aggregated into a single cell, creating obstacles not necessarily present in the polygonal representation and causing Field D* to underestimate the path length by travelling through regions of high cost. Extreme cases of this, indicated in grey in Table 2, are not used as graph data points due to issues of scale. Note how the quad-tree first underestimates path length, then reaches a point where it

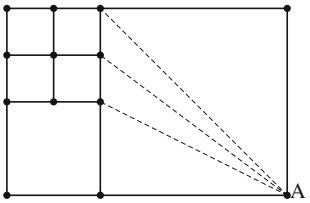


Fig. 12 Quadtree subdivision: when calculating the cost of node A, the *upper left* quadtree cell must be further subdivided into four *triangles*, since this cell has high resolution neighbours

overestimates the path length before tending once again to lower path lengths. This suggests that a certain level of quad-tree subdivision is required before pathing through high cost regions is avoided, around 40,000 faces in the case of the Voronoi Diagram for example. It is also interesting to note that the axis-aligned world suffers the most from geometric error. This is because the walls in this world are relatively thin and require high subdivision for accurate representation.

The number of node expansions required for the quadtree implementation to complete is consistently greater than that of the triangulated implementation. Between two and three times as many expansions are required on the quadtree for a similar number of faces, since a node in the triangulation has fewer neighbours compared with the quadtree. The Delaunay Refinement we used produces vertices with an average of six neighbours. A node in basic Field D* has eight neighbours and a quadtree representation will increase this if the node is on the border of a low-resolution cell with high-resolution neighbours (see Fig. 12). If we consider the node expansions required to produce a similar path cost, the quadtree requires 23 times more node expansions to produce a path cost of 37.47 in the axis-aligned world, compared to the triangulation path cost of 37.31. For the arbitrarily-aligned world, 15 times more expansions are required for a quadtree path cost of 39.8, compared to a triangulation path cost of 39.68.

In terms of running time, our implementation of the Field D* on a triangulation is between seven and 10 times faster than the quadtree implementation for a similar number of faces. A number of factors favour the triangulation implementation. Firstly, as noted above, the average valence of a node in the quadtree is greater compared to a quadtree node, increasing the number of node expansions. Also, more faces are adjacent and consequently more cost functions are evaluated. Secondly, a quadtree face requires further subdivision into triangles, again increasing the number of cost functions evaluated. Thirdly, we implemented Field D* optimisations for the triangulated case, described in [27], that are not applicable to the quad-tree's multi-resolution structure. Lastly, the triangulated implementation utilises caching while the quadtree implementation does not, since it does not make sense to cache data for triangles that are temporarily constructed during the calculation of a node's cost. Dividing the time taken in seconds by the number of nodes expanded, a value of around 18 μs is required for a quadtree node expansion as compared to about 6 μs for a node expansion on the triangulation.

Since these differences in structure and implementation exist, we use the number of faces in the environment as a measure of the space required and the number of nodes expanded as a measure of the time taken to find a path when comparing the two implementations. A triangulated version of Field D* requires an order of magnitude less space and between 10 and 20 times less running time to produce paths of similar costs within an environment, compared to a quadtree. As the geometric error in quadtree representation decreases, the differences in time and space measures decrease. Our results show that a triangulation implementation performs slightly worse than a quadtree implementation when the data is grid-aligned, but is far

superior for non grid-aligned environments. It can also be seen that increasing the subdivision level of the environment decreases the path cost at a slow linear rate for all triangulations in Fig. 10 and also for the quadtree in the grid maze case.

In comparison to A* on the triangulation edges, Field D* on the triangulation returns shorter paths in equivalent or less time, and requires fewer faces for the representation. For example, with respect to the Voronoi Diagram in Table 3, Field D* produces a cost of 8439.44 in 0.05s on 16789 faces compared to 8550.83 in 0.14s on 461450 faces. In the case of the Axis-Aligned and Arbitrarily-Aligned Worlds, Field D* produces a better path cost on fewer triangles, in equivalent time. A*'s path cost on grid edges is relatively expensive and does not converge as quickly as A* on triangulation edges. A node expansion of our A* implementation takes about 0.6 μs, 10 times faster than a node expansion of our Field D* implementation.

The original Field D* algorithm was designed for use on the Mars Rovers and so, as a practical example of the environments in which Field D* can be applied, we show (see Fig. 13) how paths can be plotted across the surface of Mars. In this figure, triangles have been weighted according to their steepness, encouraging the algorithm to plot paths avoiding difficult features. This would be useful as the battery life of these vehicles is limited and maximising their lifespan involves conserving energy.

7.3 Pathing through 3D models

We obtained a number of 3D surface models and tetrahedralized their interiors. The tetrahedra used to generate the path in Fig. 14a were uniformly weighted. Additionally, we obtained a 3D Medical DICOM data set in which the structures of the abdomen were segmented and labelled. We tetrahedralized this data set using the Computational Geometry and Algorithms Library (CGAL) [34] to produce paths through anatomical structures. Such path information could be used in angiographic (vascular) surgical planning and training, or in applications like virtual endoscopy, where a path needs to be traced through a 3D model of the winding, tubular structure of the intestinal tract without piercing the wall.

Figure 14a illustrates 3D pathing through an object with high genus. Figure 14b shows a 3D path through the human skeletal structure, starting at the *sternum*, travelling along a *true rib*, down the spine and across the *pelvis* to a *femoral head*. The tetrahedra in this structure were weighted uniformly. Figure 14c shows a path starting at the leg vein and travelling up the *inferior vena cava* to the *hepatic* vein within the liver. Inexpensive weighting of the vein and expensive weighting of the liver tetrahedra encourage the algorithm to avoid pathing directly through the liver when tetrahedra from the two structures are adjacent.

Finally, we simulated the velocity of fluid over an underwater terrain model using the Palabos Lattice Boltzmann Method package. We tetrahedralized a timestep of the simulation using CGAL and plotted paths through the fluid, as shown in Fig. 15. The black path favours fluid represented by high velocity tetrahedra and follows the general fluid flow, while the red path favours low velocity tetrahedra and descends into the crevices of the model, where the fluid moves more slowly. This demonstrates how our technique could be applied to plotting a safe course for submersible robot in a 3D underwater environment.

We note that the same geometric error that arises from representing arbitrarily-aligned shapes with a 2D grid also applies to the 3D grids employed by 3D Field D* [15].

Table 4 shows data for paths across two 3D models as the number of tetrahedra used to represent the object increases. We chose these objects since they have large amounts of space in their interiors, allowing us to vary the number of tetrahedra used to represent them, as opposed to the medical data sets which require high levels of subdivision to produce

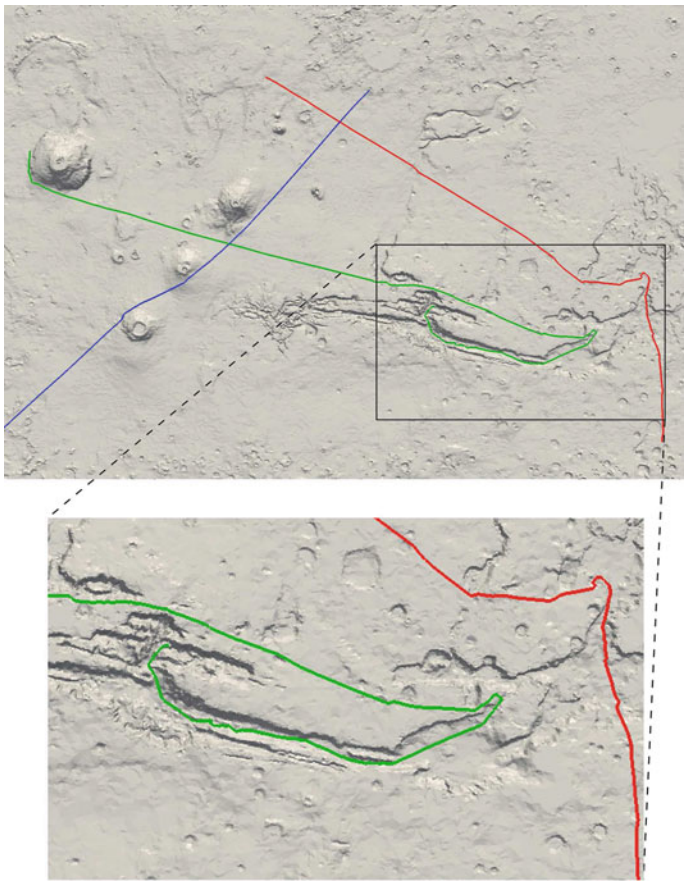
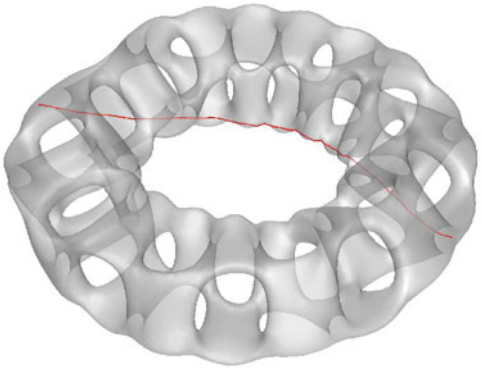


Fig. 13 Three paths plotted across a triangulation of the Mars landscape. The *triangles* are weighted according to the difference in angle between the z-axis and their normal. The *red* and *green* paths illustrate how steep sections of the *Valles Marineris* are avoided, with the *green* path showing how the flatter end of the valley is favoured when leaving it. Similarly, the *blue* path avoids pathing over the steep volcanoes of the *Tharsis Plateau* (Color figure online)

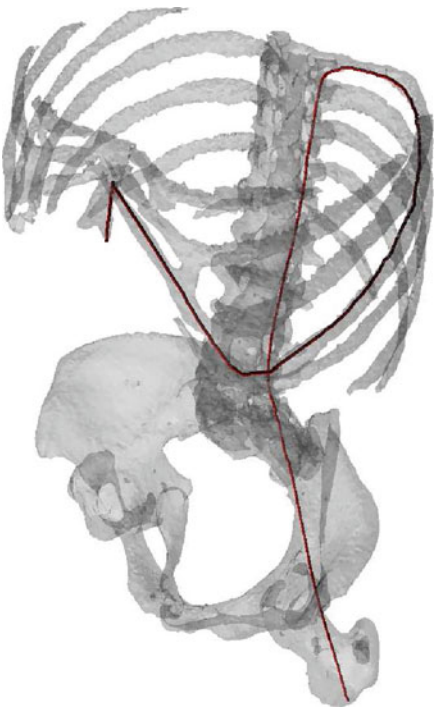
tetrahedra representing veins and ribs. In both cases, increasing the number of tetrahedra, decreases the path cost and path length somewhat, but at the cost of more node expansions and a greater running time. The path lengths can fluctuate slightly upwards as the number of tetrahedra increases, but the trend for the path lengths is downwards. The time taken for the algorithm to complete increases linearly as the number of faces and node expansions required increases.

7.4 Timings

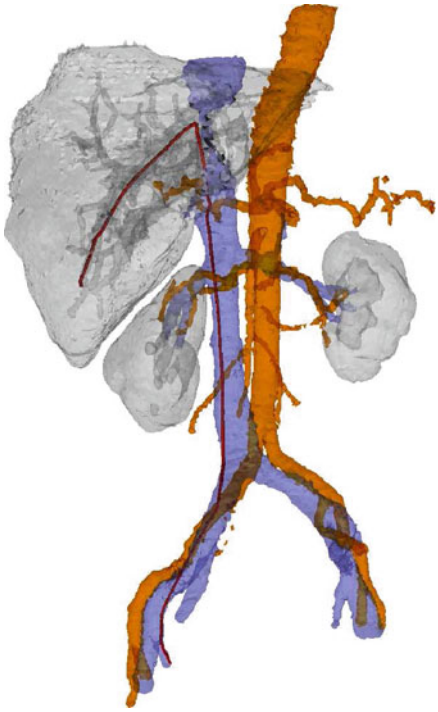
We tested the running time of Field D* on a single core of a Intel Quad Core Q9550 2.83 Ghz CPU with 4 GB RAM. For the triangulated mesh case, we constructed a random Delaunay Triangulation within a square, while for the tetrahedral mesh case, we constructed a uniform tetrahedralisation within a cube grid. Half of the triangles or tetrahedra were weighted with 0.1 (open space), while the other half were weighted with a random multiple of 16 between 16 and 256.



(a) High Genus Object



(b) Sternum to Femoral Head



(c) Leg Vein to Hepatic Vein

Fig. 14 3D path-finding. **a** and **b** show paths through objects composed of uniformly weighted tetrahedra. In **c**, tetrahedra representing the veins were weighted inexpensively, and other anatomical structures weighted expensively, resulting in the path following the veins (Color figure online)

We generated 100 random environments for both the triangulated and tetrahedral mesh case and measured the time it took for the algorithm to find a path from one corner of the square (or cube) to the opposite corner. For each case, we measured the time for the algorithm to complete with caching turned both on and off. Table 5 shows the average of these times for both the normal and cached cases and for a varying number of elements.

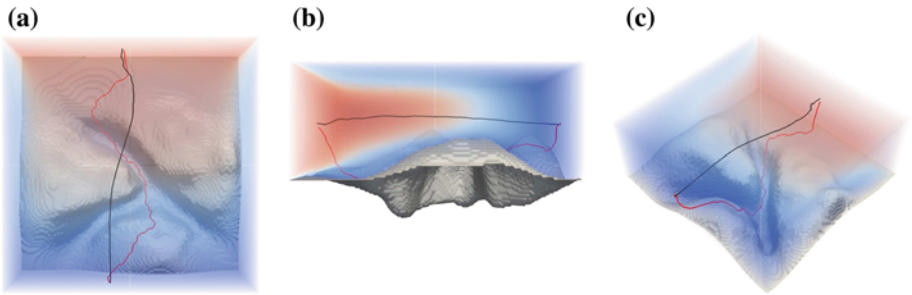


Fig. 15 3D path-finding through a fluid simulation. **a** Top-down, **b** side and **c** three-quarter views. The *red shading* indicates areas of high fluid velocity, while *blue* indicates low velocity. The *black* path results from a tetrahedral weighting favouring the high velocity, while the *red* path favours low velocity, diving into the terrain crevices

Table 4 This table shows how the number of node expansions, time to find a path, path cost and path length vary as the number of tetrahedra in the object increases

Number of tetrahedra	Node expansions	Time (s)	Path cost	Path length
Cow model				
45,684	7,923	0.64	1.0923	10.6936
86,939	14,126	1.40	1.0754	10.6360
146,774	23,773	2.60	1.0735	10.6258
217,889	35,365	4.18	1.0722	10.6282
High genus model				
83,919	13,129	1.12	0.9000	8.75
100,088	14,438	1.29	0.8975	8.7821
121,232	16,725	1.64	0.8939	8.7441
164,971	22,226	2.27	0.8901	8.7161
273,943	36,338	4.06	0.8876	8.7213

Table 5 Algorithm run-times for non-cached and cached cases

Number of elements	Normal time (s)	Cached time (s)	% Speedup
Triangulation			
52,600	0.18	0.15	16.6
80,700	0.28	0.24	14.2
102,000	0.36	0.32	11.1
Tetrahedral mesh			
52,800	1.56	1.11	28.8
79,350	2.41	1.72	28.6
101,250	3.18	2.29	27.9

Caching proved to be of greater benefit in the tetrahedral case, which is to be expected considering the greater number of calculations involved in the tetrahedral cost functions and the increased opportunity for caching.

In terms of space, we define a triangle as having three indices to vertices, three indices to neighbouring triangles and a floating point value defining the triangle weight. If each variable takes up four bytes, then 28 bytes is required to represent a basic triangle. To cache function values in the triangle an additional 48 bytes are needed, resulting in a total size of 76 bytes. Similarly, a tetrahedron will have four vertex indices, four neighbour indices and a weight, requiring 36 bytes of space. 96 bytes of cache is required to cache tetrahedron functions for a total of 132 bytes per tetrahedron. Therefore, to cache triangle functions, approximately 2.71 times more space is required per triangle to produce an average speedup of between 11 and 16 %, while 3.66 times more space is required per tetrahedron to produce an average speedup of 28 %.

8 Conclusion

This paper describes an extension of Field D*'s cost functions to triangles and tetrahedra. The analytic solutions for finding the minimum of these functions are provided for both triangles and tetrahedra, expressed in terms of vectors. Experimental results indicate a 50 % increase in performance over a previous extension of the cost functions to triangles, which relied on the expensive calculation of trigonometric values and triangle edge lengths. Also, by providing the complete set of cost functions and their minimizations to tetrahedra, a full analytic extension of Field D* to 3D has been achieved, improving upon previous work where only an approximate minimization to one function was provided for a cube.

These functions allow Field D* to operate on triangulated and tetrahedral meshes, which approximate irregular environments more accurately. Since the triangle functions can be applied to triangles situated in 3D, the triangle cost functions can be applied to triangulated surfaces in 3D and not just triangular subdivisions of a 2D plane.

We have demonstrated that, for non grid-aligned data, a quadtree requires an order of magnitude more faces compared to a low resolution triangulation for Field D* to find a path of similar cost. Due to this, and also because a node in a triangulation has fewer neighbours, Field D* has to expand between 10 and 20 times fewer nodes when calculating a shortest path on a triangulation.

We have also analysed the triangle and tetrahedra cost functions for values that can be pre-calculated and cached. For the tetrahedral case, this results in a 28 % improvement in the algorithm's running time at the cost of using 3.66 times more space.

In terms of future work, Graphics Processing Units (GPUs) implement vector operations very efficiently and also support a high degree of parallelism. Thus, a GPU implementation of Triangulated and Tetrahedral Field D* could potentially offer significant speed increases over a CPU implementation, as well as allowing many path queries to be performed simultaneously.

The original Field D* cost functions operates on unit squares. In practice, each square is temporarily subdivided into two triangles during evaluation and the minimum cost across both is calculated. Similarly, it may be useful to consider subdividing a world into convex polytopes. The polytopes can then be temporarily subdivided into general triangles or tetrahedrons when calculating the path cost of a node on the polytope.

Finally, high levels of subdivision can reduce the interpolation error inherent in Field D*, but the path cost decreases too slowly to justify the increase in space and time requirements. A better solution might be to calculate a shortest path on a coarse triangulation and then use *Adaptive Mesh Refinement* [35] to subdivide the relevant sections of the triangulation during path extraction to improve the solution. This approach would also have the advantage of

being able to place nodes and edges during the refinement process in such a way as to avoid the kind of interpolation problems that occur with the grid maze test case.

Acknowledgments We acknowledge the use of the Computational Geometry and Algorithms Library [36] in this work, which we used extensively in the generation of triangulated and tetrahedral meshes. Thanks also to Keegan Carruthers-Smith for the use of his A* code. We used the Cow model with permission from the AIM Shape Laboratory, the High Genus Object with permission from the INRIA Gamma Shape Laboratory and the 3D Medical Image data with permission from 3DIrcadb library of the IRCAD/EITS Laparoscopic Center. The Parallel Lattice Boltzmann Solver (PALABOS) was used in our fluid simulation example.

References

1. Djikstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik, 1*, 269–271.
2. Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics, 4*(2), 100–107.
3. Nilsson, N.J. (1993). *Principles of artificial intelligence*. San Francisco: Morgan Kaufmann Publishers.
4. Mitchell, J. S. B., & Papadimitriou, C. H. (1991). The weighted region problem: Finding shortest paths through a weighted planar subdivision. *Journal of the ACM, 38*(1), 18–73. http://doi.acm.org/10.1145/102782.102784.
5. Mata, C. S., & Mitchell, J. S. B. (1997). A new algorithm for computing shortest paths in weighted planar subdivisions (extended abstract). In *Proceedings of the thirteenth annual symposium on computational geometry, SCG '97* (pp. 264–273). New York: ACM. http://doi.acm.org/10.1145/262839.262983.
6. Lanthier, M., Maheshwari, A., & Sack, J. R. (1997). Approximating weighted shortest paths on polyhedral surfaces. In *Proceedings of the thirteenth annual symposium on computational geometry, SCG '97* (pp. 485–486). New York: ACM. http://doi.acm.org/10.1145/262839.263101
7. Aleksandrov, L., Maheshwari, A., & Sack, J. R. (2000). Approximation algorithms for geometric shortest path problems. In *Proceedings of the thirty-second annual ACM symposium on theory of computing, STOC '00* (pp. 286–295). New York: ACM. http://doi.acm.org/10.1145/335305.335339
8. Sun, Z., & Reif, J. (2001). Bushwhack: An approximation algorithm for minimal paths through pseudo-euclidean spaces. In P. Eades, T. Takaoka (Eds.), *Algorithms and computation*. Lecture notes in computer science (Vol. 2223, pp. 160–171). Berlin: Springer.
9. Sun, Z. S., & Reif, J. (2003). Adaptive and compact discretization for weighted regions optimal path finding. In *Proceedings of 14th Symposium on fundamentals of computation theory* LNCS (pp. 258–270). New York: Springer.
10. Aleksandrov, L., Djidjev, H., Guo, H., Maheshwari, A., Nussbaum, D., & Sack, J. R. (2006). Approximate shortest path queries on weighted polyhedral surfaces. In R. Krlovic & P. Urzyczyn (Eds.), *Mathematical foundations of computer science 2006*. Lecture notes in computer science (Vol. 4162, pp. 98–109). Berlin: Springer.
11. Cheng, S. W., Na, H. S., Vigneron, S., & Wang, Y. (2010). Querying approximate shortest paths in anisotropic regions. *SIAM Journal on Computing, 39*, 1888–1918. http://dx.doi.org/10.1137/080742166.
12. Ferguson, D., & Stentz, A. T. (2005). The Field D* algorithm for improved path planning and replanning in uniform and non-uniform cost environments. Tech. Rep. CMU-RI-TR-05-19, Robotics Institute, Pittsburgh.
13. Ferguson, D., & Stentz, A. (2006). Multi-resolution Field D*. In *Proceedings of the international conference on intelligent autonomous systems (IAS)*.
14. Samet, H. (1984). The quadtree and related hierarchical data structures. *ACM Computing Surveys, 16*(2), 187–260. http://doi.acm.org/10.1145/356924.356930
15. Carsten, J., Ferguson, D., & Stentz, A. (2006). 3D Field D*: Improved Path Planning and Replanning in Three Dimensions. In *IEEE/RSJ international conference on intelligent robots and systems* (pp. 3381–3386). doi:10.1109/IROS.2006.282516.
16. Peucker, T. K., Fowler, R. J., Little, J. J., & Mark, D. M. (1978). The triangulated irregular network. In *Proceedings, American Society for photogrammetry, digital terrain models symposium* (Vol. 516, pp. 96–103). St.Louis, Missouri.
17. Fowler, R. J., & Little, J. J. (1979). Automatic extraction of irregular network digital terrain models. In *Proceedings of the 6th annual conference on computer graphics and interactive techniques, SIGGRAPH '79* (pp. 199–207). New York: ACM.

18. Edelsbrunner, H., Ablowitz, M. J., Davis, S. H., Hinch, E. J., Iserles, A., Ockendon, J., & Olver, P. J. (2006). *Geometry and topology for mesh generation*. Cambridge monographs on applied and computational mathematics. New York: Cambridge University Press.
19. Shewchuk, J. R. (2002). What is a good linear element? Interpolation, conditioning, and quality measures. In *Proceedings, 11th international meshing roundtable* (pp. 115–126).
20. Konolige, K. (2000). A gradient method for realtime robot control. In *International conference on intelligent robots and systems, 2000 (IROS 2000), Proceedings*. IEEE/RSJ (Vol. 1, pp. 639–646). doi:10.1109/IROS.2000.894676.
21. Philippsen, R., & Siegwart, R. (2005). An interpolated dynamic navigation function. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)*.
22. Sethian, J. A. (1995). A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences of the United States of America, 93* 1591–1595.
23. Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of Game Development, 1*, 7–28.
24. Daniel, K., Nash, A., Koenig, S., & Felner, A. (2010). Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence (JAIR), 39*, 533–579.
25. Kallman, M. (2005). Path planning in triangulations. In *Proceedings of the IJACI workshop on reasoning, representation and learning in computer games*.
26. Chazelle, B. (1982). A Theorem on polygon cutting with applications. In *SFCS '82: Proceedings of the 23rd annual symposium on foundations of computer science* (pp. 339–349). Washington, DC: IEEE Computer Society. http://dx.doi.org/10.1109/SFCS.1982.58.
27. Ferguson, D., & Stentz, A. T. (2006). Using interpolation to improve path planning: the Field D* algorithm. *Journal of Field Robotics, 23*(2), 79–101.
28. Sapronov, L., & Lacaze, A. (2008). Path planning for robotic vehicles using Generalized Field D*. *Proceedings of SPIE, 6962*, 69621C. doi:10.1117/12.780650.
29. Stentz, A. (1995). The focussed d* algorithm for real-time replanning. In *Proceedings of the 14th international joint conference on artificial intelligence* (Vol. 2, pp. 1652–1659). San Francisco: Morgan Kaufmann Publishers Inc.
30. Koenig, S., & Likhachev, M. (2002a). Incremental A*. In *Proceedings of the neural information processing systems*. Cambridge, MA: MIT Press.
31. Koenig, S., & Likhachev, M. (2002b). D* Lite. In *Eighteenth national conference on artificial intelligence* (pp. 476–483). Menlo Park, CA: American Association for Artificial Intelligence.
32. Choi, S., & Yu, W. (2011). Any-angle path planning on non-uniform costmaps. In *IEEE international conference on robotics and automation (ICRA)* (pp. 5615–5621). doi:10.1109/ICRA.2011.5979769.
33. Shewchuk, J. R. (2000). Mesh Generation for domains with small angles. In *SCG '00: proceedings of the sixteenth annual symposium on computational geometry* (pp. 1–10). New York: ACM. http://doi.acm.org/10.1145/336154.336163.
34. Alliez, P., Rineau, L., Tayeb, S., Tournois, J., & Yvinec, M. (2011). 3D mesh generation. CGAL user and reference manual (3.9th ed.). CGAL Editorial Board. http://www.cgal.org/Manual/3.9/doc_html/cgal_manual/packages.html#Pkg:Mesh_3.
35. Golias, N. A., & Dutton, R. W. (1997). Delaunay Triangulation and 3D Adaptive Mesh Generation. *Finite Elements in Analysis and Design, 25*(3–4), 331–341. doi:10.1016/S0168-874X(96)00054-6.
36. CGAL Editorial Board. (2011). *The CGAL Project: CGAL user and reference manual* (3.9th ed.). http://www.cgal.org/Manual/3.9/doc_html/cgal_manual/packages.html.