Field D* Pathfinding in Weighted Simplicial Complexes

Simon Perkins

A thesis presented for the degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

UNIVERSITY OF CAPE TOWN

October 2013

Supervised by

Associate Professor P. C. Marais

Associate Professor J. E. Gain

Abstract

The development of algorithms to efficiently determine an optimal path through a complex environment is a continuing area of research within Computer Science. When such environments can be represented as a *graph*, established graph search algorithms, such as Dijkstra's shortest path and A*, can be used. However, many environments are constructed from a set of *regions* that do not conform to a discrete graph. The *Weighted Region Problem* was proposed to address the problem of finding the shortest path through a set of such regions, weighted with values representing the cost of traversing the region.

Robust solutions to this problem are computationally expensive since finding shortest paths across a region requires expensive minimisation. Sampling approaches construct graphs by introducing extra points on region edges and connecting them with edges criss-crossing the region. Dijkstra or A* are then applied to compute shortest paths. The connectivity of these graphs is high and such techniques are thus not particularly well suited to environments where the weights and representation frequently change.

The *Field D** algorithm, by contrast, computes the shortest path across a grid of weighted square cells and has replanning capabilites that cater for environmental changes. However, representing an environment as a weighted grid (an image) is not space-efficient since high resolution is required to produce accurate paths through areas containing features sensitive to noise.

In this work, we extend Field D* to *weighted simplicial complexes* – specifically – triangulations in 2D and tetrahedral meshes in 3D.

Such representations offer benefits in terms of space over a weighted grid, since fewer triangles can represent polygonal objects with greater accuracy than a large number of grid cells. By exploiting these savings, we show that Triangulated Field D* can produce an equivalent path cost to grid-based Multi-resolution Field D*, using up to an order of magnitude fewer triangles over grid cells and visiting an order of magnitude fewer nodes.

Finally, as a practical demonstration of the utility of our formulation, we show how Field D* can be used to approximate a distance field on the nodes of a simplicial complex, and how this distance field can be used to weight the simplicial complex to produce contour-following behaviour by shortest paths computed with Field D*.

Plagiarism Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the standard BibTeX convention for citation and referencing. Each contribution to, and quotation in, this thesis from the work(s) of other people has been attributed, and has been cited and referenced.
3. This thesis is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

SIGNATURE: ____________________________________
DATE: ____

Acknowledgements

The undertaking of a PhD is a long and challenging journey. It goes without saying that such an undertaking would not have been possible without the support and encouragement of many people.

I wish to thank my supervisors, Patrick Marais and James Gain, for their support and encouragement throughout the process. Their constant availability and advice undermines the common perception of absentee advisors! Thanks you for introducing me to lecturing opportunities at UCT.

Also, thanks to my parents, Richard and Margaret, for their love and support throughout the PhD.

I've had the opportunity to know many labmates during this time, too many to mention! Thanks for your friendship during this time. Steve, Chris, David, Hilton, Andrew (x2!), Carl, Ian, Keegan, Julian, Jason, Rickert, Craig and Justin to name a few.

I wish to extend thanks to Mark Berman for his assistance during the development of the 3D cost functions. Bruce Merry suggested the the use of Linear Algebra for the general solution and helped me back into the topic.

I acknowledge the use of the excellent Computational Geometry and Algorithms Library [137] 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. The Cow model, High Genus Object and 3D Medical data were was used with permission from the AIM Shape Laboratory, INRIA Gamma Shape Laboratory and the 3DIrcadb library of the IRCAD/EITS Laparoscopic Center respectively. The Parallel Lattice Boltzmann Solver (PALABOS) was used in the fluid simulation example.

Part of this research was funded by the National Research Foundation of South Africa under grants GUN2053402 and GUN66423.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Navigation through an environment is a skill that human beings learn as they grow and explore the world. Experienced travellers, who know the terrain they are navigating, will avoid dangerous and difficult terrain to make their journey safer and more predictable. With the advent of increasingly sophisticated navigation systems, there is a growing need to codify these human intuitions in software systems that can adapt to changing environments and circumstances. A navigator's behaviour is influenced by their objectives while traversing the terrain and the characteristics of the terrain. Consequently, an understanding of the space is important to facilitate travel through safe and advantageous regions and to avoid dangerous or difficult terrain.

Navigation routes can be represented as a *mathematical graph*, a representation consisting of a set of nodes and edges connecting these nodes. Graph theory is a fundamental area of mathematics, and powerful algorithms exist that operate on graphs. Specifically, the *Shortest Path Problem* is an area of graph theory where algorithms are developed to find the shortest path through a graph of weighted edges, such that the summed weight of the edges on the resulting path is minimal. By weighting different edges expensively or inexpensively, these algorithms can be made to select different routes based on characteristics within an environment. A common example is that of a road network: edges representing roads with congested traffic can be expensively weighted, while roads with free-flowing traffic can be weighted inexpensively. Such a formulation results in shortest path algorithms optimising the final route to avoid roads with congested traffic. Prominent examples include the *Floyd-Warshall* [50, 147, 114], *Bellman-Ford* [10], *Dijkstra*, [37] and *A** [61, 101] algorithms.

The above-mentioned algorithms assume a complete understanding of the weighting and structure of the environment. In practice, *a priori* conditions are not always known and can only be discovered and updated during navigation of the environment. Such requirements motivated the development of *dynamic replanning* algorithms that replan routes when new information about the environment is discovered during navigation. D* Lite [86] is an algorithm that can handle replanning requirements efficiently.

In the road example above, the decision of which route to take is based on a changing representation of physical conditions within the environment. Thus, behaviour *emerges* in a bottom-up manner from the weighting system, rather than being decided upon in a top-down manner by a controlling intelligence. Due to this bottom-up approach, shortest path planning can be said to fall within the area of *Nouvelle AI* [21] where the *Physical Grounding Hypothesis* states that to produce intelligence, systems must be grounded in the physical world, able to sense and respond to changing conditions.

These algorithms, in combination with graphs, provide elegant solutions to the shortest path problem, when the routes in the environment can logically be reduced to nodes and edges. While structures such as roads map very well to this requirement, finding shortest paths through weighted regions is a more difficult problem. The *Weighted Region Problem* has been posed as the task of finding the shortest path through a set of weighted polygons within a plane, where each polygon is weighted with the cost of travelling through it.

Finding shortest paths through weighted regions is important because representing an environment as a collection of regions is both convenient and compact. For example, in the field of *Geographic Information Systems*, terrain data is frequently represented as a *Triangulated Irregular Network* (TIN), since the representation of a set of triangles is far more compact than an image-based *Digital Elevation Model* (DEM) represented by height values on a regular grid. A compact terrain representation is not only desirable but critical in scenarios where computer memory is limited. Autonomous Robots for example, are required to operate with both limited power supply and memory resources that must be shared amongst other critical components. As a concrete example, the Mars Opportunity Rover has a main memory of 128MB of DRAM and 256 MB of secondary flash memory, which is also used for taking high-resolution photographs and operating scientific instruments.

Exact solutions to the *Weighted Region Problem* are computationally expensive. An alternative, inexact approach introduces extra points on region boundaries and links these points with edges through the region interior. This creates a searchable graph which graph algorithms can operate on, and also allows the specification of an error bound related to the degree of sampling. However, this approach is expensive in terms of memory due to the region sampling strategy.

The need for more efficient solutions motivated the introduction of approximate Weighted Region algorithms most notably the *Field D* algorithm*. Field D* [49] operates on a grid of weighted cells, essentially an image. While graph-based algorithms compute the cost of travelling across a *weighted edge* to a node, Field D*, by contrast, computes the cost of travelling across a *weighted cell* to a node. This formulation requires minimising a cost function expressing the costs of a continuous range of paths across a cell. The basic Field D* algorithm suffers from the storage issues related to resolution, since the data that it operates on is an image. Further work on Field D* [47] extends the algorithm to multi-resolution grids to increase computational efficiency and decrease space requirements, with minor reductions in accuracy. Experiments show that this representation can halve the time taken by Field D* to find a path, while only at 13% of the resolution of a uniform grid.

The core contribution of this thesis is an extension of Field D* to *weighted simplicial complexes*. A *simplex* is the general term for a triangle in 2D and a tetrahedron in 3D, and a *simplicial complex* is a collection of these structures linked together at their vertices. They may also be called triangle or tetrahedral *meshes*. 2D Polygons can be exactly decomposed into triangles and their generalisation in higher dimensions, the polytope, can be exactly subdivided into simplices. By contrast, an exact subdivision of polytopes by finite numbers of hypercubes is not generally possible. Therefore, in practice, a grid representing a planar polygonal subdivision will always contain a degree of geometric error due to its approximation of these structures. However, extending Field D*'s cost functions to simplices elimates geometric error from Field D*'s approximation of the Weighted Region Problem.

Additionally, simplicial complexes offer advantages over grid-based, and even multi-resolution grid-based representations since simplices can represent irregular features with greater accuracy and fewer elements compared to a grid. These advantages in representation lead to time and space improvements over multi-resolution Field D* in environments where features are irregular and not grid-aligned by reducing the number of elements considered by the algorithm.

Our development of Field D* arose from an exploration of a novel *Spatial Awareness Framework* codifying information about the width, curvature and logical connections within polygonal regions along either side of a *skeleton*, or *medial axis*. As we were interested in providing this information to autonomous agents in a bottom-up manner, the requirement for a pathfinding algorithm to navigate an environment of weighted regions naturally arose from the need for agents to favour or avoid specific areas.

To demonstrate the utility of our extensions, we show how Field D* can be used to compute an approximate *distance field* on the vertices of a simplicial complex. A distance field describes the distance at a particular point from important features in an environment and was first represented with 2D pixel or 3D voxel grids. These structures typically require high resolution to represent an environment and a simplicial complex offers a space efficient alternative. We show how this distance field can be used to induce contour-following, obstacle avoidance behaviour when finding paths with Field D*.

1.1 Contributions

The contributions of this thesis are four-fold:

- An extension of Field D* to 2D *weighted triangulations* based on a linear algebra formulation. We show how our extension improves upon a previous triangle implementation and how using a simplicial complex as a representation offers significant benefits over a multi-resolution grid in terms of time and space.

- An extension of Field D* to *weighted simplicial complexes* and 3D *weighted tetrahedral meshes* in particular. Our extension is expressed using linear algebra. As examples, we show how Field D* can find the shortest path through the vascular system of a patient, as an aid to endovascular surgical planning, as well as establishing the best path through a complex oceanic environment. We also perform experiments showing how many of Field D*'s side cases do not contribute significantly to the final path cost.

- A novel technique for calculating an approximate *distance field* on simplicial complexes using Field D*. This offers space benefits over voxel grid or octree techniques.

- A novel Spatial Awareness Framework for providing autonomous agents with information about the intrinsic qualities of the space that they navigate in.

1.2 Thesis Structure

This thesis is structured as follows:

- Chapter 2 surveys pathfinding literature, describing basic pathfinding on graphs using algorithms such as Dijkstra and A*, and the use of dynamic replanning on these structures to accommodate changes within the environment as a robot or agent travels along a path. A discussion of the *Weighted Region Problem* follows, describing the various approaches and algorithms proposed to solve it. We contrast these approaches and algorithms, showing how they are useful in different circumstances, before motivating our reasons for extending the Field D* algorithm.

- Chapter 3 documents the *foundations* of the Field D* algorithm, describing how it evolved from basic path-planning algorithms such as Dijkstra, A* and D* Lite. The Field D* algorithm itself is described in detail, showing how the Field D* cost functions can be developed from a simple summation of edge weights in algorithms such as A*, to finding a path that minimises the cost of travelling across a weighted grid cell. Algorithms that build on Field D*, such as *Multi-resolution Field D** and an approximate extension to 3D grids, *3D Field D** are also described.

- Chapter 4 describes our extension of Field D* to 2D triangulations using linear algebra and shows how all the Field D* cost functions can be expressed in terms of a general cost function, which can be efficiently minimised. Our approach is compared with an existing extension of Field D* to triangulations, *Generalized Field D** [119], and we show that our formulation provides performance benefits. We demonstrate how a triangulated version of Field D* requires an order of magnitude fewer triangles compared to the number of grid cells required by Multi-resolution Field D* to produce a similar path cost, documenting the time and space improvements. Some of the cases of the Field D* cost functions can be cached and we show how these offer modest performance gains.

- Chapter 5 describes our extension of Field D* to simplices in 3D and higher dimensions using linear algebra. We show how Field D*'s cost functions separate into different cases. Similarly to the 2D case, we present a general cost function, as well as more expressive version that can represent more complex cases. An analyic solution is provided for the general cost function and we show how Field D*'s various cost functions in 3D and higher dimensions can be expressed in terms of this general cost function. We perform experiments in 3D, running pathfinding queries through 3D medical data and a simulated ocean environment. Findings are presented which show that many of the Field D* cost functions do not signicantly contribute to the final path cost in 3D, which is significant to those seeking performance benefits.

- Chapter 6 describes how Field D* can be modified to create a distance field on the nodes of a simplicial complex. We identify features for which a distance field is to be computed and extract an initial set of nodes. Field D* is adapted to perform a Dijkstra's shortest path expansion on this set. Path extraction is then performed to connect each node in the simplicial complex with a point on the feature boundary. This distance field is used to weight the simplicial complex to enable contour-following behaviour by Field D*.

- Chapter 7 concludes this work as well as listing areas of *Future Work*.

- Appendix A describes our exploratory *Spatial Awareness Framework*, detailing its construction and describing experiments that show how autonomous agents using this framework can improve their behaviour. The need to develop a pathfinding component motivated our extension of the Field D* algorithm.

- Appendix B develops the mathematics and analysis surrounding the cost functions presented in Chapter 5.

Chapter 2

Pathfinding Literature Survey

Pathfinding is an important application of graph theory which aims to provide computers with representations for comprehending an environment and algorithms for navigating within it. Computers are by their very nature designed to perform logical and arithmetic operations, and thus knowledge of the environment must be distilled into a numerical representation, and the task of navigating that environment into a sequence of logical operations.

Graphs are often used to formally model such environments. A graph is a mathematical abstraction modelling the relations between a set of *nodes*, or *vertices*. Relations are modelled as *edges* connecting two vertices together. A numeric value may be associated with edges, expressing the cost of travelling on them. Similar values may also be associated with nodes, to represent the cost of travelling through them. A typical example is a road network, where graph edges represent roads, and graph nodes represent road intersections.

Once this graph has been constructed, an algorithm can be designed to find paths within it. While it is possible for an arbitrary path to be chosen through the graph, it is usually more advantageous to select a path that optimises some metric. The *shortest path* is frequently chosen since it is generally desirable to save both time and energy when travelling. More formally, the shortest path between two nodes, consisting of linked edges, must minimise the summed edge costs. The shortest path problem is often divided into four separate cases:

- single-pair shortest path problem (SPSP). Find the shortest path between a vertex v and one other vertex v'.
- single-source shortest path problem (SSSP). Find the shortest path between a vertex v and all other vertices. Dijkstra's algorithm [37] solves this problem for graphs with positive edge-weights, while the Bellman-Ford algorithm [10] caters for graphs which contain negative edge-weights.

- single-destination shortest path problem (SDSP). Find the shortest paths to one vertex v from all other vertices. This is a reversal of the single-source shortest path problem.
- all-pairs shortest path problem (APSP). Find the shortest path between every pair of vertices in a graph.

We only consider the single-pair shortest path problem. Graphs are appropriate for representing structures that map easily onto the node and edge concept. Edges can be weighted with some cost associated with travelling along the road. However, when the environment consists of weighted regions, the mapping to nodes and edges is no longer self-evident. Moreover, algorithms designed to find shortest paths on graphs may need to be adapted to operate on weighted regions.

In this chapter we begin by briefly describe mathematical graphs, followed by a summary of search strategies outlined in [116]. Next, we describe basic pathfinding algorithms that operate on graphs: Dijkstra's shortest path and A*. Additionally we describe dynamic replanning algorithms – Lifelong Planning A* and D* Lite – which replan paths when dynamic changes to the underlying graph occur.

The *Weighted Region Problem* [93] specifically poses the challenge of finding the shortest path across a weighted planar subdivision. We proceed to describe algorithms which solve this problem:

- The *Continuous Dijkstra Method* [92], a Dijkstra-like algorithm that considers distance functions between edges of the planar subdivision. Edges are placed on a priority queue and distance functions on an edge are propagated to other edges.
- *Steiner point techniques* discretise the planar subdivision into triangles. Extra vertices – *Steiner points* – are introduced along the triangle edges and connected to produce a graph on which graph-based path-finding algorithms can be applied.
- The *Field D* algorithm* [49] discretises the planar subdivision into a weighted grid. The path cost at a grid point is evaluated by minimising cost functions incorporating the cost of cell traversal and the interpolated path costs of neighbouring grid points.

Next, we describe *Finite Element Methods* (FEM) a powerful technique for solving Partial Differential Equations (PDE) over domains discretised into triangle and tetrahedral elements. Numeric techniques approximate the PDE's within elements, building a solution to a PDE over the whole domain. We also briefly discuss algorithms and pathfinding techniques that rely on interpolation as well as algorithms that solve the special case where weighted regions are either traversable or non-traversable.

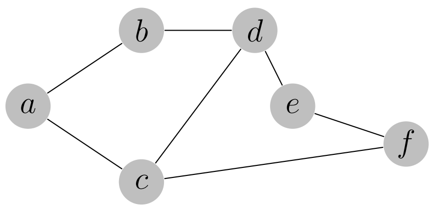
Finally, we compare and contrast the three main techniques for solving the Weighted Region Problem. Note that when discussing algorithmic computation complexity within this chapter, we use n to refer to the number of vertices considered by the algorithm and k to refer to the degree of subdivision introduced by Steiner point techniques.

2.1 Graph-based pathfinding algorithms

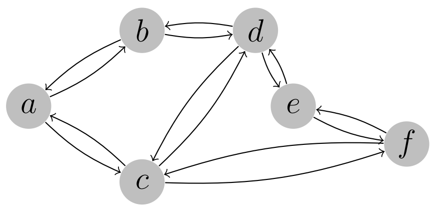
This section describes pathfinding algorithms that operate on graphs with weighted edges. A brief overview of mathematical graphs is followed by descriptions of Dijkstra's shortest path, A*, Lifelong Planning A* and D* Lite.

2.1.1 Graphs

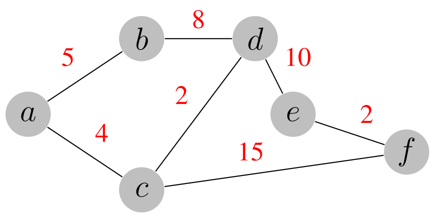
A *graph* is a mathematical abstraction modelling the relationships between vertices or nodes. Practically, graphs can be used to model structures and problems in diverse domains such as Linguistics, Chemistry, Physics, Biology, Mathematics and Computer Science. In this section we describe some basic types and some concepts associated with them. For a more in-depth look at graphs, readers may wish to consult [6].



(a) Undirected Graph



(b) Directed, Symmetric Graph



(c) Weighted, Undirected Graph

Figure 2.1: Three graph types.

A graph, G is composed of a set of *vertices* V, and a set of *edges* E, connecting the vertices together. The graph may be referred to as an ordered pair $G = (V, E)$.

If the edges of a graph have no orientation, connecting vertices in both directions, the graph is said to be *undirected*. Figure 2.1a is an undirected graph. It consists of a set of six vertices $V = (a, b, c, d, e, f)$ and six edges $E = (\{a, b\}, \{a, c\}, \{b, d\}, \{c, d\}, \{d, e\}, \{c, f\}, \{e, f\})$.

A *directed graph*, also called a *digraph*, is an ordered pair $D = (V, A)$ of a set of vertices V, and A, a set of directed edges or *arcs*. An arc $a = (x, y)$ connects x to y, but not y to x. A directed graph can be *symmetric* if, for every arc $a = (x, y)$ in D, a corresponding, inverted arc $a' = (y, x)$ also exists in D. An example of a symmetric digraph is shown in Figure 2.1b.

A *weighted graph* assigns numeric values to each edge. These weights describe the values of a problem represented by the graph. For example, weights may describe the cost of travelling along an edge, the length of an edge, or an edge's capacity to handle traffic.

Two vertices $x, y \in V$ are said to be *connected* if G contains a path from x to y. If this holds for every pair of distinct vertices, then G is said to be *connected*.

Graphs can be *embedded* onto surfaces in such as a way that edges do not cross nodes or other edges. The intuition is that edges can only intersect at their endpoint nodes. A graph embedded on a plane is called a *planar graph*. Euler's formula describes a relation between the number of nodes V, the number of edges E and the number of regions bounded by edges F as follows:

$$V - E + F = 2$$

A *maximal planar graph* is a graph that is planar, but to which it is impossible to add further edges without violating its planar property. This is also referred to as a *plane triangulation* since each region is bounded by three edges. Then, the number of edges and regions can be expressed in terms of V by $E = 3V - 6$ and $F = 2V - 4$, respectively.

The graphs that we refer to in this work are undirected, weighted and connected.

2.2 Search Strategies

The *single-pair shortest path* problem finds the least cost path between two nodes on a weighted, undirected graph. Finding this path requires an algorithm, or search strategy. In this section, we summarise search strategies described by Russell and Norvig [116] that relate to pathfinding algorithms we describe later. In particular we focus on breadth-first search, uniform-cost search and A*.

The authors discuss search strategies in terms of a *complete search tree*, a tree of depth m where each tree node has exactly b children (the branching factor). They are evaluated in terms of four criteria:

- **Completeness**: Does the strategy find a solution if it exists?
- **Time Complexity**: How long does it take to find a solution, in terms of the parameters of the search space.
- **Space Complexity**: How much memory is required to find a solution, in terms of the parameters of the search space.
- **Optimality**: Will this strategy find the best solution if several different solutions exist?

The search strategies are further classified into two categories:

- **Uninformed**: These strategies have no clear conception of the location of goal nodes relative to the current node. As such, they blindly visit various branches of the search space until a goal is reached. Examples include *breadth-first search* (BFS), *uniform cost search* (UCS) and *depth-first search.*
- **Informed**. By contrast, informed strategies are provided with information about relative goal positioning and use this information to move towards it. Examples include *greedy search* and *A**.

Breadth-first search visits nodes in order of their depth from the root, breaking ties arbitrarily, until a goal node is found at depth d. Each node s is assigned a path cost $g(s)$, and $c(s, s')$ is the assigned cost of travelling to child node s'. If the goal node exists at the deepest level of the tree, BFS will consider all possible paths. BFS is complete because it is guaranteed to find a goal. However, this strategy has an expensive worst-case time and space complexity of $O(b^d)$. BFS is optimal if $c(s, s') = a \ \forall s, s'$ and $a \geq 1$, because $g(s)$ becomes a non-decreasing function of node depth.

Uniform-cost search is a variant of BFS that allows the search problem to be specified as finding a goal with the least-cost path from the root. UCS maintains a set of nodes from which further explorations of the tree will be mounted. If the set's least cost node is a goal, the search terminates, otherwise its children are added to the set and the search continues. When $c(s, s')$ is positive, UCS is guaranteed to find the cheapest solution first, and is optimal. As a variant of BFS its time and space complexity is still $O(b^d)$.

Greedy search seeks to minimise the estimated cost of reaching a goal node. While UCS considers the node with the least cost, greedy search always considers the node whose cost-to-goal estimate is minimal. The function estimating this cost is called a *heuristic function*, usually denoted by $h(s)$. Greedy searches are useful because they often find goal states quickly, although the route taken to the goal may not be optimal in terms of path cost. They also tend to search down to the maximum depth of the tree m and therefore their time and space complexity is $O(b^m)$. They may also recurse down infinite search trees and are therefore not complete.

*A** [61] combines both the optimality and completeness provided by UCS, with the potential efficiency of a greedy search. This is accomplished by combining the path cost and heuristic functions into a directed function $f(s) = g(s) + h(s)$. By combining the path cost with an estimate, $f(s)$ becomes an estimate of the *cost of a path to the goal travelling through* s. Then, A* always searches the node with the lowest total path estimate, $f(s)$, in a manner similar to UCS. For A* to be complete and optimal, $h(s)$ must be *admissable*. This means that $h(s)$ is restricted so that it never *overestimates* the cost to the goal. An admissable $h(s)$ confers admissability to $f(s)$, so that $f(s)$ in turn never overestimates the total path cost.

A*'s time and space complexity depend on how well the heuristic, $h(s)$, estimates the cost to the goal from s. It can be shown that they are exponential unless the error in the heuristic function grows no faster than the logarithm of the actual path cost to the goal, $h^*(s)$. Mathematically, this is expressed as:

$$|h(s) - h^*(s)| \leq O(\log h^*(s))$$

In practice, most heuristics are proportional to $h^*(s)$ and therefore A*'s time and space complexity is usually exponential $O(b^d)$. However, the use of a good heuristic with A* offers far superior performance over an uninformed search.

The discussion so far has dealt with search strategies on a complete tree, but they also can be applied to undirected, weighted graphs. Similarly to the root tree node, a graph node can be selected as the starting node of the search. The *degree*, or number of edges incident to a graph node is similar to a tree node's number of branches b, while the depth of the solution d is similar to the number of nodes between the starting node and the goal. Then a BFS, for example, expands outward visiting nodes until the goal is discovered. Since such a search may have to visit all nodes in the graph, the time and space complexity is $O(n)$ where n is the number of nodes or vertices in the graph.

2.2.1 Dijkstra's Shortest Path Algorithm

The *single-pair shortest path problem* (SPSP) problem of graph theory poses the challenge of finding the least cost path between two nodes in a graph of weighted edges. The resulting path, a consecutive list of edges proceding from the start node to the goal node, should minimise their summed weights. *Dijkstra's Algorithm* [37] solves the SPSP problem using a uniform cost-search strategy on an undirected, connected graph with positive edge weights.

For each node s, the algorithm maintains the cost of travelling to that node from the start node, $g(s)$. Additionally, two sets of nodes are maintained, the CLOSED and the OPEN set. The cost of nodes in the CLOSED set are considered to be final, while those in the OPEN set are still subject to change. All nodes are initially placed in the OPEN set, with a cost of ∞, except for the start node s_{start} which is assigned a path cost $g(s_{\text{start}}) = 0$.

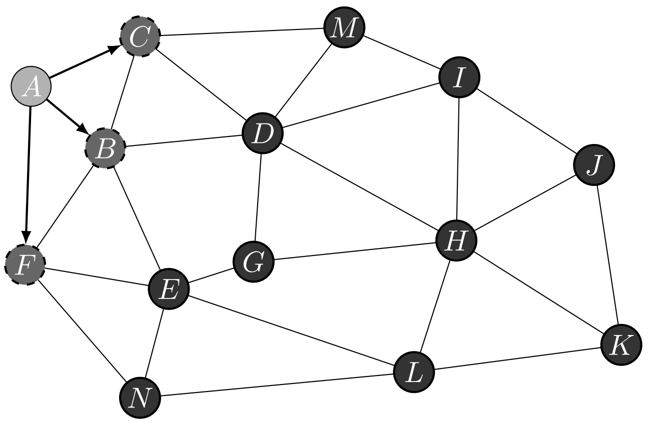
During the *node expansion* phase of the algorithm, the node s with the lowest path cost, $g(s)$ is removed from the OPEN set and its cost is propagated to its neighbours. In Algorithm 1, this takes place in the UpdateNode function. The cost, $g(s')$, of each node s' neighbouring s, is calculated by adding $g(s)$ to the cost $c(s, s')$ of travelling along their connecting edge. This new cost, $g(s) + c(s, s')$ replaces $g(s')$ if it is smaller, and s is set to be the predecessor $\text{pred}(s') = s$ of s'. One iteration of this process is referred to as a *node expansion*.

Once node expansion has occurred, s is transferred to the CLOSED set and will never again be placed in OPEN. Then, the node in the OPEN set with the least path cost is removed and chosen as s. If $s = s_{\text{goal}}$, or there are no more nodes in the CLOSED set, this phase of the algorithm terminates. Figure 2.2 illustrates three iterations of the algorithm.

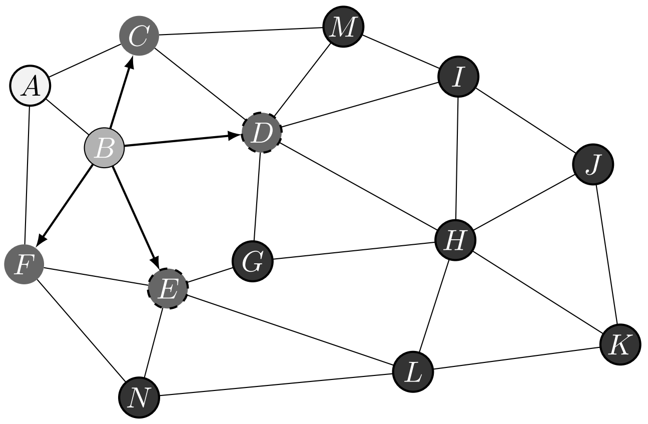
Algorithm 1 Dijkstra's Algorithm. $g(s)$ is the path cost at node s, while $c(s, s')$ is the cost of travelling the edge between nodes s and s'. pred(s) returns the node from which s derives its cost, while nbrs(s) is the set of nodes neighbouring s. OPEN is the priority queue of non-finalised nodes, ordered by path cost, while CLOSED is the set of nodes whose path cost is finalised.

```
 1: function UPDATENODE($s, s'$)                      ▷ Update the $s'$ cost and insert on the priority queue
 2:     if $g(s) + c(s, s') < g(s')$ then             ▷ Can the current path cost $g(s')$ be improved?
 3:         $g(s') \leftarrow g(s) + c(s, s')$        ▷ Yes, assign the new path cost.
 4:         pred$(s') \leftarrow s$                   ▷ Set $s$ to be the predecessor of $s'$
 5:         OPEN.$Insert(s', g(s'))$                  ▷ Insert $s'$ into OPEN with priority $g(s)$
 6:     end if
 7: end function
 8: function DIJKSTRA($s_{\text{start}}, s_{\text{goal}}$)
 9:     $g(s_{\text{start}}) \leftarrow 0$            ▷ Path cost of the start node is zero
10:     pred$(s_{\text{start}}) \leftarrow s_{\text{start}}$   ▷ Predecessor of start node is itself
11:     OPEN $\leftarrow \emptyset$                   ▷ Initialise the OPEN set
12:     OPEN.$Insert(s_{\text{start}}, g(s_{\text{start}}))$   ▷ Insert $s_{\text{start}}$ with priority $g(s_{\text{start}})$.
13:     CLOSED $\leftarrow \emptyset$                 ▷ Initialised the CLOSED set
14:     while OPEN $\neq \emptyset$ do                ▷ Node Expansion Phase
15:         $s \leftarrow$ OPEN.$Pop()$               ▷ Pop node with lowest path cost
16:         CLOSED $\leftarrow$ CLOSED $\cup \{s\}$
17:         if $s = s_{\text{goal}}$ then
18:             break                                 ▷ The goal has been found
19:         end if
20:         for each $s' \in$ nbrs$(s)$ do            ▷ Iterate over the neighbours of $s$
21:             if $s' \notin$ CLOSED then            ▷ Ignore nodes in the CLOSED set
22:                 if $s' \notin$ OPEN then          ▷ Initialise any unexplored nodes
23:                     $g(s') \leftarrow \infty$
24:                     parent$(s') \leftarrow NULL$
25:                 end if
26:                 UpdateNode$(s, s')$
27:             end if
28:         end for
29:     end while
30:     PATH = $\{s_{\text{goal}}\}$
31:     $s \leftarrow s_{\text{goal}}$
32:     while $s \neq s_{\text{start}}$ do            ▷ Path Extraction Phase
33:         PATH = PATH $\cup \{s\}$
34:         $s \leftarrow$ pred$(s)$
35:     end while
36: end function
```

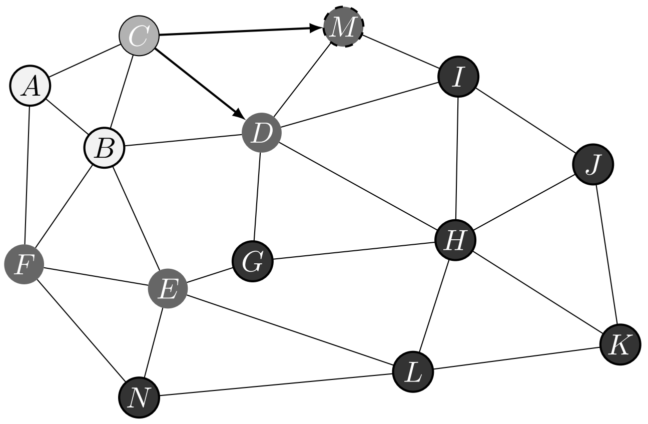
If the goal has been found, the path extraction phase can take place. Starting at the goal node, the path back to the start node is found by selecting the predecessor of the current node. Space-efficient implementations may wish to avoid the space used to store a predecessor. In this case, the path may be extracted by, starting with the start node, recursively selecting the cheapest neighbouring node as



(a) First Iteration: A popped, moved to CLOSED. B, C, F added to Priority Queue.



(b) Second Iteration, B popped, moved to CLOSED. D, E added to Priority Queue.



(c) Third Iteration, C popped, moved to CLOSED. M added to Priority Queue.

Figure 2.2: Three Iterations of Dijkstra's algorithm. The lowest priority node on the queue is popped and moved to the CLOSED set. Neighbours of the popped node have their path costs/priorities updated and are moved from the OPEN set to the priority queue.

the next node on the path until the goal node has been reached. If multiple neighbours share the same cost, implying that there are multiple shortest paths to the goal, the *tie* may be broken arbitrarily.

The original algorithm did not order OPEN, yielding $O(V^2)$ computational complexity, where V is the number of nodes in the graph. Fredman and Tarjan [51] introduced a priority queue to sort nodes on OPEN, yielding a running time of $O(E + V \log V)$, where E is the number of edges in the graph.

On a *planar graph*, we can re-express this complexity in terms of the number faces F bounded by the edges, through the use of Euler's formula. Re-arranging so that we have $E = V + F - 2$, and substituting yields $O(V + F - 2 + V \log V)$ which simplifies to $O(F + V \log V)$. If the graph is a maximal planar graph, or a grid, E and F are linear in terms of V and the complexity expressed as $O(V \log V)$, or $O(n \log n)$.

2.2.2 A*

A* [61] is a best-first search algorithm that combines Dijkstra's algorithm with a heuristic function. The heuristic, $h(s)$, estimates the costs of travelling from s to the goal node. Then, a directed cost function $f(s) = g(s) + h(s)$, illustrated in Figure 2.3, sums the path cost $g(s)$ and heuristic $h(s)$ at node s. As mentioned earlier, $f(s)$ estimates the cost of a path to the goal through node s.

Nodes are ordered in a priority queue in a manner similar to Dijkstra's algorithm. However, they are ordered by $f(s)$ rather than their path cost $g(s)$. By incorporating the heuristic into the priority ordering, nodes with a cheaper heuristic gravitate towards the front of the queue and are considered earlier than other nodes. Therefore, the heuristic focuses the direction of the search towards the goal. The A* algorithm documented in Algorithm 2 differs from Dijkstra in the ordering of the priority queue by $f(s)$.

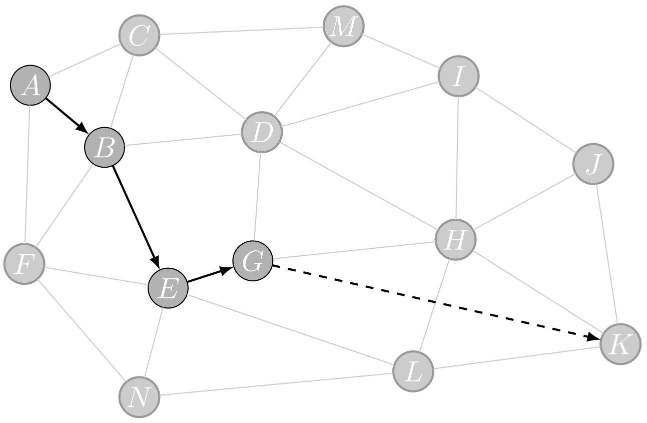


Figure 2.3: In this example, the A* algorithm finds a shortest path between A and K. The priority of node G, $f(G)$ is the sum of its path cost along $ABEG$, $g(G)$ and a heuristic estimate of the path cost between G and K, $h(G)$. The heuristic estimate must be shorter than the actual path cost from G to K, for instance the path cost of travelling along GHK.

A*'s behaviour depends on the heuristic that it uses. A* is optimal if the heuristic is *admissable*. This means that the heuristic must underestimate the distance to the goal. Thus, if $c^*(s, g_{\text{goal}})$ is the actual cost of travelling from s to the goal, then the following must hold for all s: $h(s) \leq c^*(s, g_{\text{goal}})$. By contrast, an inadmissable heuristic *overestimates* the $h(s)$ term and, as the $g(s)$ term is exact, this has the effect of encouraging the algorithm to consider nodes with low $h(s)$ values. This forces the algorithm towards the goal, ignoring side paths that may contain the shortest path. Consequently, an inadmissable heuristic may result in a sub-optimal path, but this situation may be useful when a fast, instead of an optimal solution, is required.

An important class of heuristic function are the *consistent* heuristic functions. These satisfy the following criterion: $h(s) \leq c(s, s') + h(s')$ where s is closer to s_{goal} than neighbour s'. This implies that the estimated cost of reaching s_{goal} from s is not greater than the estimate cost from s', added to the cost of travelling between s and s'. Consistent functions are also called *monotonic*. Consistency implies admissability, but the reverse does not necessarily hold.

If A* uses a consistent heuristic, it will never consider a node more than once. This means that, once a node has been placed in the CLOSED set, it need never be considered again. This property can be used to optimise the implementation. Line 24 of Algorithm 2 discards neighbours that are in the CLOSED set for example, but should be removed if the heuristic is not consistent. Additionally, A* can be transformed to a Dijkstra's shortest path algorithm with reduced cost function $c'(s, s') = c(s, s') - h(s) + h(s')$.

Thus, the time complexity of A* depends on the heuristic. A pathalogical heuristic can result in an exponential worst-case complexity. By contrast, a heuristic which estimates the distance to goal perfectly, can produce $O(V)$ worst-case complexity, although is is not usually possible to construct a perfect heuristic. A zero-heuristic $h(s) = 0 \ \forall s$ reduces A* to Dijkstra's algorithm with $O(E + V \log V)$ worst-case complexity. Additionally, a consistent heuristic allows A* to be transformed to a Dijkstra's shortest path, also with $O(E + V \log V)$ worst-case complexity.

A popular heuristic is the *Euclidean distance* between the current and goal node: $h(s) = \alpha \| s - goal \|$ where α is a scaling constant, set to the cost of the most minimally weighted edge, divided by its length. This heuristic results in the algorithm favouring nodes that are physically closer to the goal during node expansion. When s is closer to the start node, the heuristic dominates $f(s)$ since $g(s)$ is smaller and $h(s)$ larger, but as the search gets closer to the goal, $g(s)$ becomes dominant.

Other distance heuristics may be more appropriate to the underlying graph. If the graph is a grid, the Manhattan distance, which sums distances between vertical and horizontal coordinate components is better, as it is only possible to travel along vertically and horizontally oriented edges. The Manhattan distance is formulated as $h(s) = |s_x - goal_x| + |s_y - goal_y|$. Both Euclidean and Manhattan heuristics are consistent.

A particularly strong property of A* is that it is *optimally efficient*. This means that, given a heuristic $h(s)$, there is no other algorithm that expands fewer nodes. For this reason, A* is a popular, widely

Algorithm 2 The A* algorithm

```
function UPDATENODE($s, s'$)
    if $g(s) + c(s, s') < g(s')$ then
        $g(s') \leftarrow g(s) + c(s, s')$
        pred$(s') \leftarrow s$
        if $s' \in$ OPEN then
            OPEN.$Remove(s')$
        end if
        OPEN.$Insert(s', g(s') + h(s'))$
    end if
end function
function A*($s_{\text{start}}, s_{\text{goal}}$)
    $g(s_{\text{start}}) \leftarrow 0$
    pred$(s_{\text{start}}) \leftarrow s_{\text{start}}$
    OPEN $\leftarrow \emptyset$                                    ▷ Initialise the OPEN set
    OPEN.$Insert(s_{\text{start}}, g(s_{\text{start}}) + h(s_{\text{start}}))$
    CLOSED $\leftarrow \emptyset$                                  ▷ Initialised the CLOSED set
    while OPEN $\neq \emptyset$ do
        $s \leftarrow$ OPEN.$Pop()$
        CLOSED $\leftarrow$ CLOSED $\cup \{s\}$
        if $s = s_{\text{goal}}$ then
            break                                                  ▷ The goal has been found
        end if
        for each $s' \in$ nbrs$(s)$ do
            if $s' \notin$ CLOSED then
                if $s' \notin$ OPEN then
                    $g(s') \leftarrow \infty$
                    parent$(s') \leftarrow NULL$
                end if
                UpdateNode$(s, s')$
            end if
        end for
    end while
end function
```

used algorithm.

The disadvantage of A* is that, as an adaption of BFS it can use large amounts of space, exhausting available memory long before an actual solution is found. Iterative Deepening A* (IDA*) [75] and Simplified Memory Bounded A* (SMA*) [115] both attempt to bound A*'s memory use, at the cost of computational complexity.

Algorithm 3 Lifelong Planning A* supporting functions

```
 1: function CalculateKey($s$)
 2:     return $[\min(g(s), \text{rhs}(s)) + h(s); \min(g(s), \text{rhs}(s))]$
 3: end function
 4: function Initialise
 5:     $U = \emptyset$
 6:     for all $s \in S$   $\text{rhs}(s) = g(s) = \infty$
 7:     $\text{rhs}(s_{\text{start}}) = 0$
 8:     U.insert($s_{\text{start}}, [h(s_{\text{start}}); 0]$)
 9: end function
10: function UpdateNode($u$)
11:     if $u \neq s_{\text{start}}$ then
12:         $\text{rhs}(u) = \min_{s' \in \text{pred}(u)}(g(s') + c(s', u))$
13:     end if
14:     if $u \in U$ then
15:         U.Remove($u$)
16:     end if
17:     if $g(u) \neq \text{rhs}(u)$ then
18:         U.Insert($u$, CalculateKey($u$))
19:     end if
20: end function
21: function ComputeShortestPath($s_{\text{start}}, s_{\text{goal}}$)
22:     while U.TopKey() $<$ CalculateKey($s_{\text{goal}}$) OR $\text{rhs}(s_{\text{goal}}) \neq g(s_{\text{goal}})$ do
23:         u = U.Pop()
24:         if $g(u) > \text{rhs}(u)$ then
25:             $g(u) = \text{rhs}(u)$
26:             for all $s \in \text{succ}(u)$   UpdateNode($s$)
27:         else
28:             $g(u) = \infty$
29:             for all $s \in \text{succ}(u) \cup \{u\}$   UpdateNode($s$)
30:         end if
31:     end while
32: end function
```

2.2.3 Lifelong Planning A*

*Lifelong Planning A** (LPA*) [73] is an incremental version of A* that allows A* to replan the shortest path when changes in graph topology and cost occur. This is important because, in real-life scenarios, environments often do not remain static. While it is possible to replan a path when the environment changes, this is expensive if the environment changes continually. Also, if only parts of the environment are changing, it would be more efficient to recalculate only those parts of the shortest path that have been affected by the environmental changes. Lifelong Planning A* (LPA*) solves this problem.

LPA* differs from A* in a number of ways and new notation must be introduced to explain it. S is the set of nodes in the graph and $s_{\text{start}} \in S$ and $s_{\text{goal}} \in S$ are the start and goal nodes of an LPA* search.

Algorithm 4 Lifelong Planning A* main function

```
1: function LPA*($s_{\text{start}}, s_{\text{goal}}$)
2:     Initialise()
3:     while true do
4:         ComputeShortestPath($s_{\text{start}}, s_{\text{goal}}$)
5:         Wait for changes in edge costs
6:         for all directed edges $(u, v)$ with changed edge costs do
7:             Update the edge cost $c(u, v)$
8:             UpdateNode($v$)
9:         end for
10:     end while
11: end function
```

succ$(s) \subseteq S$ is a set denoting the successors of $s \in S$, while pred$(s) \subseteq S$ denotes the predecessors of $s \in S$. The terminology of successors and predecessors is utilised to distinguish the incoming and outgoing edges on a directed graph, but for the undirected graphs, they can be considered the same. $c(s, s')$ is the cost of moving from node s to $s' \in$ succ(s). $g^*(s)$ denotes the start distance of node $s \in S$, which is the cost of the shortest path from s_{start} to s. $g^*(s)$ satisfies the following relationship:

$$g^*(s) = \begin{cases} 0 & \text{if } s = s_{\text{start}} \\ \min_{s' \in \text{pred}(S)}(g^*(s') + c(s', s)) & \text{otherwise} \end{cases}$$

LPA* maintains two estimates of the start distance of a node s, $g(s)$ and rhs(s). When LPA* first calculates the shortest path, the g-values of nodes are calculated in precisely the same order as those of an A* search. However LPA* utilises the relation between these values in subsequent replanning searches. rhs(s) is a lookahead value that is used to determine whether $g(s)$ must be updated. All rhs(s) values must satisfy the following relationship:

$$\text{rhs}(s) = \begin{cases} 0 & \text{if } s = s_{\text{start}} \\ \min_{s' \in \text{pred}(S)}(g(s') + c(s', s)) & \text{otherwise} \end{cases}$$

The purpose of the lookahead is to detect if structural changes have been made in the surrounding topology that invalidate the $g(s)$ estimate. If $g(s) =$ rhs(s), node s is said to be *locally consistent*, if $g(s) >$ rhs(s) it is *overconsistent* and if $g(s) <$ rhs(s) it is *underconsistent*. When nodes are inconsistent, it means that the lookahead has found an updated path to node s. If overconsistent, it implies that the path to s has become cheaper, while underconsistency implies that the path to s has become more expensive.

In a similar fashion to A*, LPA* maintains a priority queue of nodes that must be expanded and their path costs calculated. The set of nodes that are locally consistent are similar to the nodes in A*'s CLOSED set in the sense that their values are considered to be final and require no further processing. Thus, the OPEN priority queue contains all overconsistent and underconsistent nodes.

Similarly to A*, LPA* recalculates the path cost $g(s)$ of the priority queue node s with the minimum key. The keys that order nodes in the LPA* priority queue are a tuple that bear similarity to the $f(s)$ used to order A*'s priority queue. The key of a node s is

$$\begin{aligned} k(s) &= [k_1(s), k_2(s)] \\ \text{where } k_1(s) &= \min(g(s), \text{rhs}(s)) + h(s) \\ k_2(s) &= \min(g(s), \text{rhs}(s)) \end{aligned} \tag{2.1}$$

Keys on the priority queue are ordered *lexicographically*. Thus $k(s) \leq k(s')$ *iff* either $k_1(s) < k_1(s')$ or $(k_1(s) = k_1(s')$ and $k_2(s) \leq k_2(s'))$. The first key component, $k_1(s)$ directly mirrors A*'s $f(s)$ values, since both $g(s)$ and rhs(s) estimates correspond with A*'s $g(s)$. The second tuple component serves to break ties when the first tuple components are equal in favour of the tuple with the lowest estimate.

The LPA* algorithm is shown in Algorithm 3 and 4. The Initialise function empties the priority queue U and initialises the $g(s)$ and $rhs(s)$ of all nodes to infinity. It also inserts the start node s_{start} onto the priority queue in an inconsistent state.

After initialisation, LPA* waits for changes in the underlying graph costs. The nodes affected by these changes are modified by the UpdateNode function, which updates their rhs-values and key-values. It also removes the nodes from the priority queue, and reinserts them if they are inconsistent. The shortest path is then recalculated by calling ComputeShortestPath(), which expands nodes on the priority queue according to the ordering of their keys.

If ComputeShortestPath() encounters a locally overconsistent node, the implication is that a new shortest path to this node has been discovered and thus $g(s)$ is set to be equal to the lookahead value rhs(s). The neighbouring nodes then update their rhs(s) and key values, via the UpdateNode function, since their consistency may be affected by the new value of $g(s)$.

If ComputeShortestPath() encounters a locally underconsistent node, the implication is that the shortest path to this node has somehow become more expensive since the lookahead is less expensive. In this case, $g(s)$ is set to ∞ to make it either consistent or overconsistent. UpdateNode instructions are issued to the surrounding nodes, as well as the previously underconsistent node to restore their consistency.

The LPA* algorithm continues until two conditions hold: s_{goal} is locally consistent and the key of the next node for expansion is greater than the key of s_{goal}. Similar to A* one can trace the shortest path, starting with $s = s_{\text{goal}}$ back to s_{start} by moving from current vertex s to the predecessor s' that minimises $g(s') + c(s', s)$. Ties may also be broken arbitrarily.

2.2.4 D* Lite

D* Lite [86] extends Lifelong Planning A* to perform dynamic path replanning in environments where costs for large portions of the environment are unknown. As the agent or robot explores the environment, these costs are discovered and require replanning of existing paths. This problem was originally solved by *D** or *Dynamic A** [130], but D* Lite's solution is regarded as simpler to understand and implement.

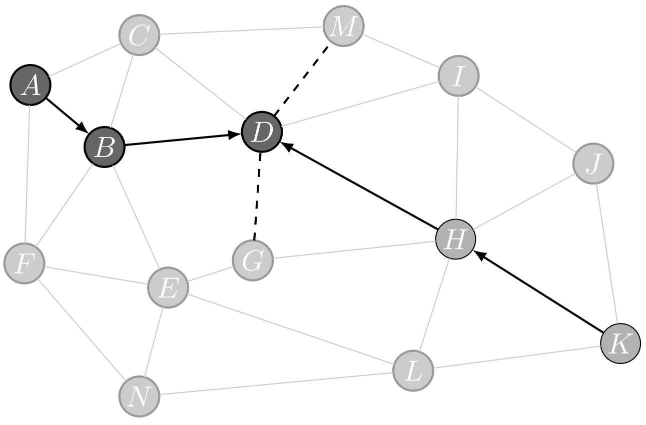


Figure 2.4: D* Lite initially finds a path $KHDBA$ from goal K to start node A. In this example, an agent then travels along the path ABD, examining the immediate environment for changes in cost, in this case edges DG and DM. If node inconsistency results, D, G or M may be placed on the priority queue after which D* Lite replans the path from K to D. As only a few nodes towards the end of the path are inconsistent, replanning is a cheap operation.

The first major difference between D* Lite and LPA* is that D* Lite replans the path from the goal to the start node, rather than from the start to the goal node in LPA*. The reason is that an initial shortest path is calculated, the agent moves along that path and in doing so encounter changes in the environment, as shown in Figure 2.4. When this occurs, a replan is initiated and the current node becomes the start node.

Therefore the reversal of seach direction avoids replanning of sections of the path that have already been travelled. Additionally, since only local changes to the environment near the start node are applied, the algorithm need only recalculate the priority of nodes towards the end of the path. Consequently, only a small section of the path needs replanning and replans become efficient.

Due to this change, successor nodes become predecessor nodes and vice versa. As the agent may start with minimal information about the environment, the cost of travelling along unknown edges can be set to some constant. These initial estimates of the edge costs may be naïve, but the agent assigns more realistic estimates as it travels to and encounters previously unknown parts of the environment.

The change in direction requires the heuristic to estimate the cost of travelling between a node and the start node, instead of the goal node. Also, since the start node may change when replans occur,

Algorithm 5 D* Lite supporting functions

```
1: function CALCULATEKEY($s$)
2:     return $[\min(g(s), \text{rhs}(s)) + h(s_{\text{start}}, s) + k_m; \min(g(s), \text{rhs}(s))]$
3: end function
4: function INITIALISE
5:     $U = \emptyset$
6:     $k_m = 0$
7:     for all $s \in S$    $\text{rhs}(s) = g(s) = \infty$
8:     $\text{rhs}(s_{\text{start}}) = 0$
9:     U.insert($s_{\text{goal}}$, $[h(s_{\text{goal}}); 0]$)
10: end function
11: function UPDATENODE($u$)
12:     if $u \neq s_{\text{goal}}$ then
13:         $\text{rhs}(u) = \min_{s' \in \text{succ}(u)}(g(s') + c(s', u))$
14:     end if
15:     if $u \in U$ then
16:         U.Remove($u$)
17:     end if
18:     if $g(u) \neq \text{rhs}(u)$ then
19:         U.Insert($u$, CalculateKey($u$)
20:     end if
21: end function
22: function COMPUTESHORTESTPATH(start, goal)
23:     while U.TopKey() < CalculateKey($s_{\text{start}}$) OR $\text{rhs}(s_{\text{start}}) \neq g(s_{\text{start}})$ do
24:         $k_{old}$ = U.TopKey()
25:         u = U.Pop()
26:         if $k_{old}$ < CalculateKey($u$) then
27:             U.Insert($u$, CalculateKey($u$))
28:         end if
29:         if $g(u) > \text{rhs}(u)$ then
30:             $g(u) = \text{rhs}(u)$
31:             for all $s \in \text{pred}(u)$    UpdateNode($s$)
32:         else
33:             $g(u) = \infty$
34:             for all $s \in \text{pred}(u) \cup \{u\}$    UpdateNode($s$)
35:         end if
36:     end while
37: end function
```

the priority of nodes on the priority queue may be incorrect as the heuristic values involved in their calculation are no longer valid.

To solve this, D* Lite recognises that when an agent has moved along a path from node s to s', the priorities of existing elements on the queue can be decreased by $h(s, s')$ to ensure that they will be consistent with the new heuristic values used to calculate queue priorities. To allow this, a consistent

Algorithm 6 D* Lite Main Function

```
1:  function D*LITE(start, goal)
2:      $s_{\text{last}} = s_{\text{start}}$
3:      Initialise()
4:      while $s_{\text{start}} \neq s_{\text{goal}}$ do
5:          $s_{\text{start}} = \text{arg min}_{s' \in \text{succ}(s_{\text{start}})} c(s_{\text{start}}, s') + g(s')$
6:          Move to $s_{\text{start}}$
7:          Scan graph for changed edge costs
8:          if Any edge costs have change then
9:              $k_m = k_m + h(s_{\text{last}}, s_{\text{start}})$
10:             $s_{\text{last}} = s_{\text{start}}$
11:             for all directed edges $(u, v)$ with changed edge costs do
12:                 Update the edge cost $c(u, v)$
13:                 UpdateNode$(v)$
14:             end for
15:             ComputeShortestPath($s_{\text{start}}, s_{\text{goal}}$)
16:         end if
17:     end while
18: end function
```

heuristic is required. $h(s, s')$ must now always be positive and satisfy the following

$$h(s, s') \leq c^*(s, s')$$
$$h(s, s') \leq h(s, s') + h(s, s'')$$

where $s, s', s'' \in S$ and $c^*(s, s')$ is the shortest path between nodes s and s'. However, rather than subtracting this value from existing queue elements, it is added to new elements when they are placed on the queue via the use of variable k_m (Algorithm 5 Line 2, Algorithm 6 Line 9), which contains the accumulated $h(s, s')$ values from replans as the agent moves towards the goal.

D* Lite therefore plots an initial naïve estimate of the path from the start node to the goal node. As the algorithm moves away from the start node and detects the actual costs within the environment, the path is replanned and the current position is set to be the start node. This process continues until the goal node is reached or another goal is selected. Alternatively, if D* Lite is provided with correct environment costs, it performs a path planning operation equivalent to A*.

2.3 Region-based pathfinding algorithms

Here, we describe pathfinding algorithms designed to find shortest paths between two points in a collection of weighted regions. We begin by presenting the Weighted Region Problem which describes this challenge formally, followed by a discussion of *Steiner point techniques* which discretise regions into graphs and apply standard graph-based pathfinding algorithms. Lastly, we discuss the Field D* algorithm, which uses interpolation to specify cost functions over weighted grid cells.

2.3.1 The Weighted Region Problem

The *Weighted Region Problem* (WRP) [93] poses the problem of finding a least cost path between two points in a plane, subdivided into *weighted* polygons. In a general sense, a path through this environment consists of a number of line segments. The cost of travelling along these line segments is their length multiplied by the weight of the polygon that they are travelling through. To find the least cost path, one must find the set of line segments connecting the start and goal whose summed cost is minimal.

One of the earliest solutions for the WRP was the *Continuous Dijkstra Method* [92]. The Continuous Dijkstra Method is conceptually similar to a Dijkstra search on a graph. However, instead of maintaining the path cost at nodes in a graph, Continuous Dijkstra maintains *distance functions* over edges within a graph which model how the distance to other edges changes. The algorithm also maintains an "event" queue, equivalent to the priority queue in Dijkstra, in which optimal edge intervals are placed for consideration. When "events" are removed from the queue, the optimal intervals are propagated to other edges. Thus, it conceptually behaves as a wavefront that propagates continuous fields of cost outwards from a source node.

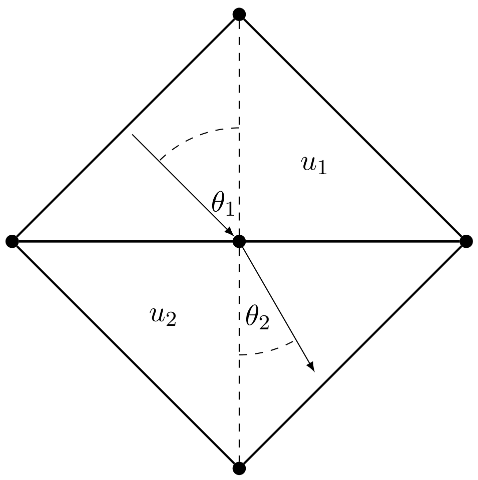


Figure 2.5: Snells law states that $u_1 \cdot sin(\theta_1) = u_2 \cdot cos(\theta_2)$ where θ_1 and θ_2 are the angles the rays make with the edge, and u_1 and u_2 are the different medium densities.

This algorithm is guaranteed to find a path whose cost is within $(1 + \epsilon)$ of the actual least cost path where ϵ is the level of the algorithm's precision. The authors show that the time complexity of the algorithm is $O(E \cdot S)$, where E is the number of "events" in the Continuous Dijkstra Method and S is the complexity of a numeric search to find a $(1 + \epsilon)$ shortest path from the start to the goal through a sequence of edges within the triangulation.

The authors exploit the fact that an optimal path will bend according to *Snell's Law of Refraction* when the path exits one polygon and enters another. This law states that the direction of a ray of light travelling through a medium of some density will change direction in a predictable manner when it enters another medium of differing density (see Figure 2.5). The above-mentioned numerical search utilises a form of binary search in conjunction with Snell's Law of Refraction to find a shortest path through the triangulation. The authors evaluate the time complexity of E and S and show that the total time complexity of the algorithm is $O(n^8 L)$, where L is the related to ϵ and defines the precision required of the solution.

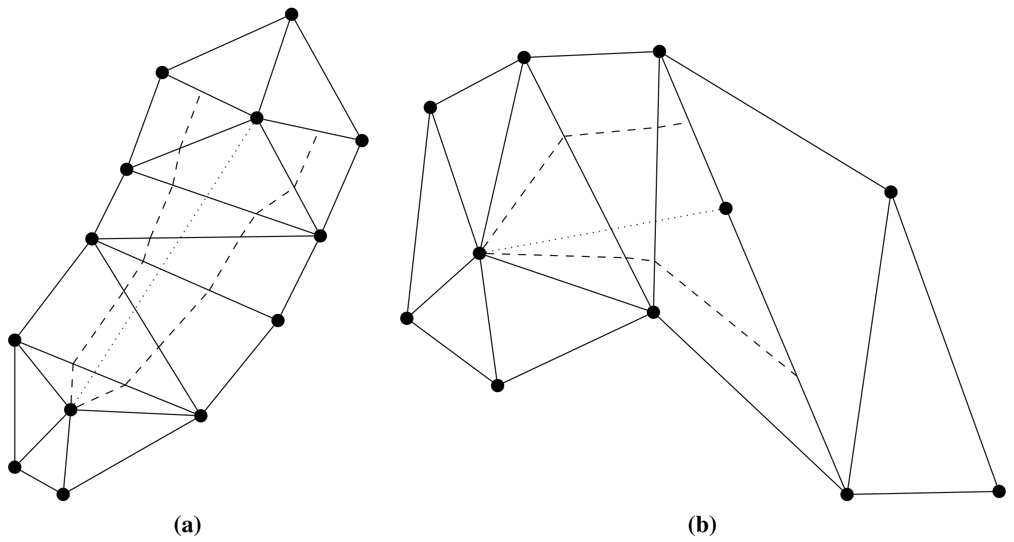


Figure 2.6: The propagation of pathnet rays, represented by dashed edges result in node connections represented by dotted edges. (a) Pathnet rays that pass on either side of a node result in that node being connected to the node the rays emanated from. (b) If both rays intersect the same edge, but the angle of one ray is too steep, a node is created between the ray intersections and connected to the emanating node.

The *Pathnet* [90] algorithm adapts The Weighted Region Problem [93] by constructing a graph $G = (V, E)$ on the vertices V of a triangular subdivision of a plane S. The angle range around a node $v \in V$ is discretized into k evenly-spaced directional cones. Ray pairs, corresponding to the cone boundaries are projected outwards from a node, using Snells's Law to modify their direction whenever a triangle boundary is crossed. This projection continues until one of three conditions is met:

- The rays flow around a graph node, in which case an edge is created between this node and the node from which the rays emanated (Figure 2.6a).
- The rays intersect the same edge, but one of them intersect the edge at too steep an angle. A node is created between the ray edge intersections and connected to the original node with an edge (Figure 2.6b).
- The rays encounter the environment's boundary. No action take place in this case.

A Dijkstra or A* search is then constructed on the resulting *pathnet*. $O(kn^3)$ time is required to construct the pathnet with $O(kn)$ space needed for the representation. An A* search on the resulting pathnet would take $O(kn \log kn)$ time. The pathnet must also store the refracting paths in the edge connecting two nodes.

2.3.2 Steiner Point Techniques

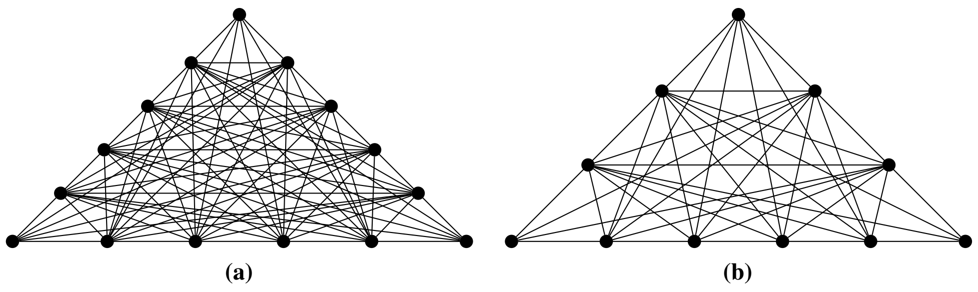


Figure 2.7: Basic Steiner point subdivision schemes: (a) four points are distributed along each edge. (b) Shorter edges are assigned fewer points compared to longer edges.

A simple and useful approach to the Weighted Region Problem is to subdivide the environment to a particular resolution and use existing search algorithms on the subdivided representation. In this section, we discuss techniques that introduce *Steiner points*, vertices not present in the original triangulation, along triangle edges. These Steiner points are connected to adjacent nodes with edges and pathfinding algorithms are invoked on the new graph. The Weighted Region Problem [93] permits the specification of an error tolerance ϵ in the solution. Steiner point techniques also allow the specification of an error tolerance ϵ that governs the number of points introduced along each edge.

Approximating weighted shortest paths on polyhedral surfaces [80] detail a number of strategies for distributing points along triangle edges. The simplest strategy, called a *fixed* strategy, involves distributing k evenly spaced points along the edge (Figure 2.7a). This may provide too much resolution in the case of small triangles. An interval scheme can improve upon the simple distribution by spacing points along regular intervals. The authors suggests that the interval be $|l_e|/(k+1)$, where l_e is the longest edge in the triangulation. In the case of small triangles (Figure 2.7b), this can reduce the number of points, thus reducing the time complexity, with minimal impact on accuracy. The third scheme they propose is to use β-spanners to reduce the number of edge connections to individual Steiner point in a bid to reduce the time complexity at the cost of accuracy. A β-spanner subdivides the space around a Steiner point into cones. The Steiner points along adjacent triangle edges that lie within the cone are considered and only the one that produces the least-cost path has an edge connected to the original Steiner point.

Once the subdivision is complete, a Dijkstra can be run on the resulting graph. Depending on k, the number of the Steiner points introduced, the running time will then be $O(kn \log kn)$. The error, ϵ, is related to k, $\epsilon \propto \frac{1}{k}$ and thus the complexity can also be expressed as $O(\frac{n}{\epsilon} \log \frac{n}{\epsilon})$.

Approximation algorithms for geometric shortest path problems [3] suggest a scheme for spacing Steiner points along an edge based on the maximum distance from the edge to edges of incident faces. The Steiner points are also prevented from being placed within a weighted radius of each node. This radius is dependent on the maximum and minimum weights within the triangulation. They show that this placement of Steiner points produces time complexity of $O(\frac{n}{\epsilon}\text{log}\frac{1}{\epsilon}(\frac{1}{\sqrt{\epsilon}} + \text{log}n))$.

The BUSHWHACK algorithm [134] uses a similar idea to the β-spanner to reduce the number of edges in a graph. It uses a property of shortest paths within a triangular region, namely that two shortest paths travelling through different faces will not intersect each other within the face interior. Due to this it is possible, for any two consecutive Steiner points along an edge for which the path distance from the start node is known, to constrain the possible edge connections to an opposing edge to a defined range. BUSHWHACK maintains a dynamic interval $I_{v,e,e'}$ which is defined by a node v on edge e and another edge e' within a triangle. For every Steiner point $v \in e$ that has been discovered (assigned a path cost representing the distance from the starting node to v), $I_{v,e,e'}$ contains an *interval* of Steiner points $v* \in e'$ that *produce the least-cost path from the start node through* v. Steiner points that are not in this interval need not be considered and this reduces the overall connectivity of the graph. The resulting time complexity is $O(\frac{n}{\epsilon}(\text{log}\frac{1}{\epsilon} + \text{log}n)\text{log}\frac{1}{\epsilon})$, improving upon the time complexity of $O(\frac{n}{\epsilon}\text{log}\frac{1}{\epsilon}(\frac{1}{\sqrt{\epsilon}} + \text{log}n))$ in [3].

The BUSHWHACK algorithm is further refined here by Sun et. al [135] where the authors note that the algorithmic time complexities presented in [93], [90], [3] and [134] all depend on constants related to the geometric configuration of the problem. They present an adaptive discretisation scheme that places Steiner points in such a way that the resultant time complexity is independent of the ratio between the maximum and minimum triangle weights and also improve BUSHWHACK's heuristics.

The techniques presented up until this point have focused on improving the running time by reducing the $\log kn$ term of the $O(kn \log kn)$ time complexity. *Approximate shortest path queries on weighted polyhedral surfaces* [2] improves these time complexities by introducing data structures that cache path queries for both single-source queries (SSQ) and all-path queries (APQ). If it is known that only paths from a single node will be calculated, an SSQ structure is created that can return the shortest path in $O(\text{log}\frac{1}{\epsilon})$ time by using $O(\frac{n}{\sqrt{\epsilon}}\text{log}\frac{1}{\epsilon})$ space. An APQ structure can also be created from embedded SSQ structures, returning the shortest path in $O(q)$ time for $O(\frac{(g+1)n^2}{\epsilon^{3/2}q}\text{log}^4\frac{q}{\epsilon})$ space, where g is the genus of the graph and q is some parameter acting as an upper bound on the query time.

Querying approximate shortest paths in anisotropic regions [25] note that the data structures presented in [2] depend on geometric parameters. They present a data structure independent of geometric parameters that can return a shortest path from a fixed source in $O(\text{log}\frac{\rho n}{\epsilon})$ time in return for $O(\frac{\rho^2 n^4}{\epsilon^2}(\text{log}\frac{\rho n}{\epsilon})^2)$ space, where ρ is used to parameterise a distance function.

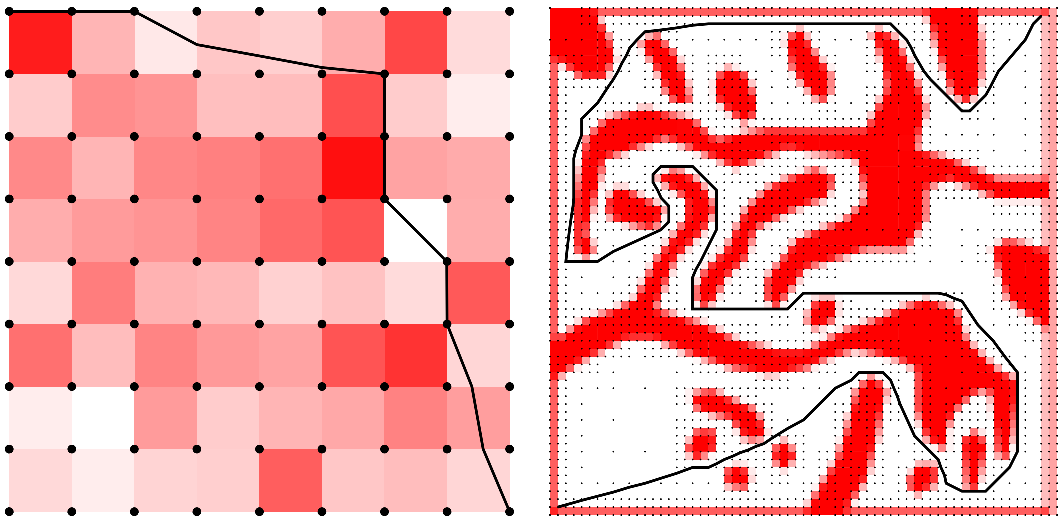

(a) Field D*　**(b)** Multi-resolution Field D*

Figure 2.8: Examples of (a) Field D* and (b) Multi-resolution Field D*

2.3.3 Field D*

Field D* [48] is presented as an approximate solution to the Weighted Region Problem which finds shortest paths on weighted grids. While graph-based algorithms only consider paths that traverse across weighted graph edges, Field D* is able to consider paths that travel through grid cells.

Algorithms such as Dijkstra and A* operate on graphs consisting of edges and nodes. Node path costs are calculated by adding adjacent node path costs to the traversal costs of the connecting edge. As Field D* operates on a weighted grid, the traditional graph cost function is adapted to calculate the cost of travelling to a node through a weighted grid cell or square. The Field D* cost function splits into three cases, but the case providing the least cost can always be selected, based on the relative size of cell surrounding weights.

Since the path can now originate from a continuous range of headings, the origin of such paths may not lie on grid nodes, but rather a continuous range of points along a grid edge. Since path costs are only stored at grid nodes, the path cost along a grid edge must be estimated, and Field D* accomplishes this by *interpolating* the path costs of the nodes adjacent to an edge. Field D*'s operation is still based on the traditional pathfinding techniques developed from Dikjstra and A*, but provides a way for these algorithms to operate on a different underlying structure. Paths produced by Field D* can travel through cells,as shown in Figure 2.8a. Therefore they are smoother and shorter than paths produced by A* on grid edges, for example.

Field D* is an approximate solution to the WRP for two reasons. Firstly, the underlying weighted grid representation can, in general, only approximate polygon regions, since an infinite number of grid cells may be required to represent a polygon exactly. Therefore, Field D*'s solution to the WRP is subject to geometric error since grid cells will lie across polygon boundaries. An extended discussion of this error is presented in Chapter 4. Secondly, Field D* linearly interpolates path costs along grid cell edges and is therefore subject to interpolation error, the difference between the actual and interpolated path cost at a point on a grid edge.

Field D* inherits from D* Lite [86], which in turn is derived from A* [61]. In Section 2.2.2 we explained that A* with a consistent heuristic can be transformed into a Dijkstra's shortest path algorithm, with $O(E + V \log V)$ worst-case time complexity where V and E are the number of nodes and edges in a graph. As Field D* is built on A*, it also has this complexity, which can be re-expressed as $O(F + V \log V)$ where F are the regions in a *planar graph*. The planar graph in this case is a grid and, since the number of cells F are linear in the number of nodes V, the complexity can be more compactly expressed as $O(n \log n)$ where $n = V$. Field D* requires $O(F + V)$, or $O(n)$, space to represent the grid.

Large, uniform regions take up many grid cells, requiring space to represent and computational time to evaluate their traversal costs. Field D* as a variant of A*, is inherently subject to the A*'s space problems and highly sampled grids exacerbate this. Multi-resolution Field D* [47] aims to reduce these time and space requirements by adapting Field D* to multi-resolution grids. Instead of a uniform grid, the environment is represented with a region quadtree [118]. Large uniform regions can be represented with large quadtree cells, while regions containing much variability can be represented with the finer resolution of smaller quadtree cells, as shown in Figure 2.8b. Algorithms are presented for propagating path costs between nodes that lie on cells of different resolution. In their experiments, the authors show cases where Multi-resolution Field D* can improve performance over basic Field D* by 1.8 times when the resolution of the underlying quadtree is 13% of that of the grid.

3D Field D* [23] extends Field D* to 3D weighted grids. One of Field D*'s' cases is adapted to operate on cube faces. The minimum of the function representing this case is approximated from boundary conditions on face edges. The authors also integrate global scale costs into this cost function.

Generalized Field D* [119] extends Field D*'s cost function to arbitrary triangles. The triangles and cost functions are defined in terms of their edge lengths and internal angles. The size of each triangle can be arbitrary, so it is no longer possible to select which of the three cases to use beforehand. This means that each case must be minimised, its cost evaluated and compared to the other cases before a least-cost path to a node from across a triangle can be selected.

The extension to triangles is useful as the authors show that large areas in a grid can be represented with a small number of triangles, as opposed to a large number of grid cells, with comparable accuracy in path cost. However, this is only shown in one experiment which subdivided a uniform grid into triangles and no comparison against Multi-resolution Field D* was performed. Each minimisation produces two roots and the only presented technique for determining which root is the minima is to evaluate the cost function with both roots. A naïve implementation of the cost functions requires trigonometric functions due to the use of internal triangle angles, therefore finding the least-cost path across a triangle requires much computation.

2.4 The Finite Element Method

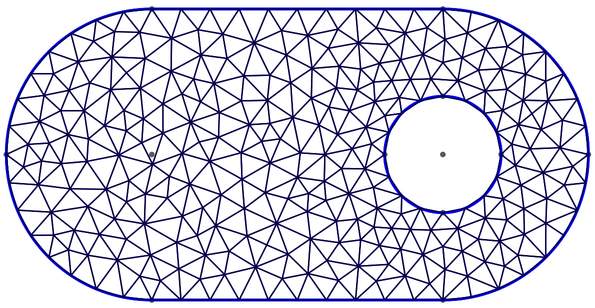


Figure 2.9: A finite element domain Ω, consisting of the space between the outer boundary and a circular hole, is partitioned into triangles by a Delaunay Triangulation. Functions defined over Ω are approximated over the triangular subdomains with polynomial functions.

The Finite Element Method [149] (FEM) is a powerful technique for numerically solving boundary and initial-value problems defined by partial differential equations (PDE). A distinctive feature of this method is that the bounded domain $\Omega \in \mathbb{R}^n$ over which the problem is solved is partitioned into many smaller, non-overlapping subdomains, called *finite elements*. An example of a partitioned domain is shown in Figure 2.9. Functions defined over Ω are locally approximated within these elements by interpolation, typically using polynomial functions. These local approximations can also be said to be *piecewise approximations*.

PDEs to which the method is applied are *weakly formulated*, by which it is meant that linear algebra techniques are used to approximate the PDEs. A weak formulation does not require the PDE to hold absolutely and this allows *weak solutions* to be obtained in cases where the PDE itself does not admit a sufficiently smooth, differentiable solution. Additionally, a weak formulation allows the contribution of each element to be summed to produce an integral representing the problem over the entire domain Ω.

FEM has interesting parallels with the work in this thesis. Firstly, our extension of Field D* requires partitioning environments into triangles and tetrahedra, similar to the way FEM partitions a domain into elements. This is an important topic in FEM since "good" finite elements are required to generate accurate solutions, especially when piecewise linear interpolation is performed over an element [122]. *Delaunay* triangulations and tetrahedral meshes are considered desirable [122, 124] since they ensure that no point in the mesh is inside the *circumcircle* of any simplex. In practice, this means that a Delaunay triangulation is unlikely to contain near degenerate (skinny) triangles.

Secondly, FEM and our implementation of Field D* both focus on solving local problems inside a triangle or tetrahedron, while ultimately solving a problem in the original domain, or environment.

In the case of FEM, a PDE is sampled at element nodes and these values are interpolated over the element using polynomial functions. In this way, properties such as the stress or strain are evaluated over the element via *integration*. While Field D* does interpolate path costs over an edge, it seeks to find a least-cost path from this edge, over a uniformly weighted region, to a node: the cost function attempts to find the minimum path that accomplishes this.

Furthermore, at the original domain level, FEM aims to evaluate a function over the domain. Field D*, however, aims to optimise a path across a subset of the domain. Indeed, since it is a best-first search algorith, Field D* attempts to visit as little of the domain as possible in order to obtain good performance.

The mature mesh generation techniques FEM uses to partition domains are relevent to this work. However, FEM element interpolation techniques are not applicable to Field D*, since their purpose is to approximate a function over an element, rather than find a least cost path through it. Indeed, since Field D* solves the Weighted Region Problem, a uniform cost is associated with a triangle region, for instance. Also, since FEM evaluates the entire domain, it would require adaptation to function as a single-pair shortest path algorithm, whereas Field D* already contains this capability.

2.5 Other algorithms of interest

In this section, we describe some other algorithms of interest. We begin by describing the Fast Marching Method class of algorithms, which approximate the Eikonal equation's wavefront model over a grid. Next, we describe algorithms that aim to solve the Weighted Region Problem over regions that are either open or occupied.

Fast Marching Methods: *Fast Marching Methods* (FMM) are a group of algorithms that propagate costs in a wavelike fashion over a discretised domain of points. They are derived from Classical Hamilton Mechanics, and the *Hamilton-Jacobi equation* (HJE) [54], a partial differential equation whose solution describes a trajectory requiring the least of amount of energy through a mechanical system. The *Hamilton-Jacobi-Bellman equation* (HJB) [9] of *optimal control theory* [70] is a variant which seeks to find a set of control laws, or actions changing the state of a system, satisfying some optimality criterion.

Exact solutions to these equations are usually impossible. Therefore, they are discretised and solved numerically, specifically with finite-difference methods, and more generally with finite-element methods [8, 38, 39, 45, 56, 79, 129].

Tsitsiklis [141] presents a technique for approximating the HJB, whereby the domain of the problem is sampled as a grid. A value $V(\mathbf{x})$ is associated with each grid node $\mathbf{x}$. Values for $V(\mathbf{x})$ are obtained by evaluating an *approximation function* which interpolates the values of neighbouring grid nodes and, incorporates the cost of travelling *through* $\mathbf{x}$. Since $V(\mathbf{x})$ is defined in terms of $V(\mathbf{y})$ of neighbouring grid nodes $\mathbf{y}$, an ordering exists between grid nodes and using this property, a Dijkstra-like algorithm can be constructed. Numerical minimisation of the approximation function produces optimal values for $V(\mathbf{x})$.

The *Fast Marching Method* [121], a special case of *Level Set Methods* [105], is a numeric technique approximating the *Eikonal equation* which describes how the boundary of a shape expands over time. Tracking a moving boundary is complex, especially in higher dimensions. Therefore, the problem is converted into a stationary problem by discretising the domain with a grid storing the time $T(\mathbf{x})$ at which the moving boundary intersects grid nodes $\mathbf{x}$, and $F(\mathbf{x})$, the speed of the boundary at $\mathbf{x}$. $T(\mathbf{x})$ is calculated using *upwind difference operators* which interpolate the $T(\mathbf{y})$ of neighbouring points y and incorporate the speed $F(\mathbf{x})$ of travelling through x. FMM [121] and Tsitsiklis' approach [141] are considered to be equivalent methods.

*E** [109, 108] is a path planning algorithm based on FMM that is capable of finding smooth interpolated paths over a time-crossing grid. A wavefront is propagated outwards from a goal node to the entire grid using an upwind operator, thereby forming a *navigation function* [81]. Gradient descent is used on the navigation function to follow the steepest, and therefore optimal, path to the goal. Similarly to [141, 121], E* associates an effort or cost with travelling through a grid node, rather than a grid cell or region. Consequently, the cost of travelling through a particular region or grid cell is dependent on the node under consideration, rather than properties of the cell. As such, FMM formulations of the path planning problem are related, but do not correspond to, The Weighted Region Problem. [108] notes the existence of similarities and differences between E* and Field D* for example.

Konolige's *Gradient Method* [74] 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.

Path planning through open/closed space: Other algorithms exist that plot paths through environments composed of open space and obstacles. An environment consisting of polygonal obstacles can be represented with a *Visibility Graph* [88], where edges are constructed between vertices unobstructed by polygons. A Dijkstra or A* search can then be executed on the resultant graph. However, the number of edges in the environment can grow quadratically with the number of vertices [98].

Path planning in Triangulations [68] represents an environment with a *Constrained Delaunay Triangulation*, initially calculating the shortest path on the *adjacency graph* of the triangulation. The connected triangles of this path form a *channel* which is employed in a *funnel algorithm* [24] to find the actual shortest path within the channel.

Near optimal hierarchical pathfinding [19] 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

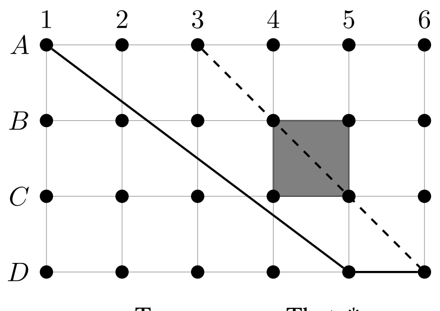


Figure 2.10: In the Theta* algorithm, a parent node does not need to be connected to the child node, it must merely be visible. For example $A1$ can be a parent of $D5$, while $A3$ cannot be a parent of $D6$ because there is a CLOSED cell between them.

to the node's grandparent. This can still be suboptimal if a node has visibility of another ancestor that is further away.

*Theta** [98] is an extension of A* that provides any-angle path-planning on grids consisting of OPEN and CLOSED cells. Previous extensions of A* use *visibility graphs* on environments consisting of open space and obstacles. In these algorithms, heading changes of the path occur on the corners of obstacles. Visibility graphs can be expensive as they can grow quadratically with the number of cells in the environment. Theta* combines the visibility graph ideas with implementations of A* in a grid environment.

Theta* allows any visible node to be the parent of a node (there are no CLOSED cells between a node and the parent). For example, in Figure 2.10, the parent of $D5$ can be $A1$ because the nodes have visibility of each other, while $A3$ and $D6$ do not. Consequently, each node expansion performed by Theta* is linear in the number of grid cells due to line-of-sight checks, yielding $O(n^2)$ time complexity. The resulting path segments can span many grid cells. *Any-angle Theta** [98] reduces this to a constant factor by propagating the visible angle range from a node's parent, but the actual time and path cost are slightly higher than basic Theta*.

Nash et. al also compare Basic Theta* to Field D* on uniform grids. In their experiments on environments composed of OPEN and CLOSED cells in a 500x500 grid, Field D*'s runtime is between three and 1.7 times greater than Basic Theta* when the environments are 0% and 30% blocked respectively. Also, Field D* requires between ten and 1.4 times as many node expansions when environments are 0% and 30% blocked, but it must firstly be noted that in the 0% case, many cells are considered by a single Theta* node expansion, and secondly the authors do not optimise their implementations. Theta* also manages to avoid the "jitter" that Field D*'s path direction's experience as they transition between cells, an artifact produced by Field D*'s linear interpolation assumption.

These techniques find smooth shortest paths, but the environments in which they operate lack the richness of a weighted region representation.

A subsequent development of *Theta** [35] 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 – once again this implies that a node expansion may be linear in the number of cells. Their work shows that in randomly weighted environments Field D* finds shorter paths in 78% of Theta*'s time, 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.

Any-angle path planning on non-uniform costmaps [27] incrementally extends Theta*'s non-uniform weighted grid extension by calculating both an arithmetic mean and weighted 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.

2.6 Discussion

The time and space requirements of the algorithms discussed in this section are tabulated in Table 2.1.

Traditional graph-based pathfinding algorithms such as Dijkstra and A* are well understood and efficiently find shortest paths. However, they are designed to operate on graphs and the paths that they produce are constrained to graph edges. If the underlying environment is not fully represented by the graph then the resultant path will, firstly, be longer than it should be and, secondly, the path will make sharp heading changes when following graph edges.

The algorithm for solving the *Weighted Region Problem* presented in [93] is a theoretically rigorous contribution and lays the foundation for later work, especially in recognising the usefulness of Snell's Law of Refraction in solving the problem. This solution also allows the specification of an error tolerance, ϵ, that the solution must adhere to. Unfortunately, the algorithm's time complexity of $O(n^8 L)$ precludes a practical implementation.

The strategy employed by the *Steiner point* techniques [80] presented in Section 2.3.2 is to discretise a triangle face by introducing extra graph nodes along edges and graph edges across the face as a pre-process. These techniques also allow the specification of an error tolerance ϵ which is related to the number of Steiner points introduced along each edge. In general the time complexity of an A* search on the resultant graph is $O(kn \log kn)$ where $k \propto \frac{1}{\epsilon}$

The advantage of the Steiner point techniques in their original presentation [80], is the simplicity in creating and searching on the graph, and the fact that the resultant path will conform to an error tol-

Algorithm	Time Complexity	Space Requirements
Weighted Region Problem [93]	$O(n^8 L)$	$O(n^4)$
Pathnet [90]	$O\left(\frac{n}{\epsilon}\log\frac{n}{\epsilon}\right)$	$O\left(\frac{n^3}{\epsilon}\right)$
Approximating weighted shortest paths on [80] polyhedral surfaces	$O\left(\frac{n}{\epsilon}\log\frac{n}{\epsilon}\right).$	$O\left(\frac{n^2}{\epsilon}\right)$
Approximation algorithms for geometric shortest [3] path problems	$O\left(\frac{n}{\epsilon}\log\frac{1}{\epsilon}(\frac{1}{\sqrt{\epsilon}}+\log n)\right).$	$O\left(\mathcal{C}\frac{n}{\epsilon}\log_2\frac{2}{\epsilon}\right)$
Bushwhack [134]	$O\left(\frac{n}{\epsilon}(\log\frac{1}{\epsilon}+\log n)\log\frac{1}{\epsilon}\right)$	
Approximate shortest path queries on weighted [2] polyhedral surfaces	$O\left(\log\frac{1}{\epsilon}\right)$ $O(q)$	$O\left(\frac{n}{\sqrt{\epsilon}}\log\frac{1}{\epsilon}\right)$ $O\left(\frac{(g+1)n^2}{\epsilon^{3/2}q}\log^4\frac{q}{\epsilon}\right)$
Querying approximate shortest paths in [25] anisotropic regions	$O(\log\frac{\rho n}{\epsilon})$	$O(\frac{(g+1)n^2}{\epsilon^{3/2}q}\log^4\frac{q}{\epsilon})$
Weighted Theta* [35]	$O\left(n^2\right)$	$O\left(n\right)$
Field D* [49]	$O\left(n\log n\right)$	$O\left(n\right)$

Table 2.1: Comparison of the time and space requirements of the various algorithms that solve The Weighted Region Problem.

erance. One disadvantage is that decreasing the error tolerance, and by implication k, increases both the time and space complexity of the algorithm. Thus, [80, 3] present point placement techniques that reduce the number of introduced points while still maintaining the error bounds. [80] also identifies ways in which the number of edge connections to a particular node can be reduced without affecting the accuracy of the resultant solution. [134, 135] exploit this concept of edge reduction in their BUSHWHACK algorithm, which modifies Dijkstra's algorithm to limit the number of edge connection choices across a triangle face. [2] and [25] present data structures that can answer an all-path query in constant time, but these require substantial increases in space requirements.

Thus, much of the subsequent work on Steiner point techniques has been involved in reducing the computational expense of reducing the allowed error. This often increases either the complexity of the underlying discretisation [80, 3], the complexity of implementation [134, 135] or the space requirements [2, 25].

The *Pathnet* algorithm [90] shares some similarities with the Steiner point techniques in that it constructs a graph by sampling paths through a weighted triangulation and then performs a Dijkstra or A* search upon the resultant graph. Interestingly, [80] claim that *Pathnet* has $O(kn^3)$ time complexity, where k is the number of sampling cones around a point when constructing the graph. This is not correct: The graph is *constructed* in $O(kn^3)$ time, but an A* search would take $O(kn \log kn)$ time complexity, and consequently, would have a similar time complexity to the Steiner point techniques.

The majority of Steiner point techniques operate on a weighted triangulation operating on a 2D plane, or weighted polyhedral surfaces. [3] also presents a placement algorithm for weighted tetrahedra. This placement yields a running time of $O(\frac{n}{\epsilon^3}\log\frac{1}{\epsilon}(\frac{1}{\sqrt{\epsilon}}+\log n))$. The $\frac{n}{\epsilon^3}$ term makes this discretisation expensive in terms of time and space.

By contrast, Field D* [49] operates on a weighted grid and provides new cost functions to calculate the cost of travelling across the cells of this grid. An A* or Dijkstra algorithm can then use these new cost functions to perform a shortest path search in $O(n \log n)$ time. Field D* interpolates the path costs of adjacent nodes to approximate the path costs of points on the connecting edge. Field D* is therefore subject to interpolation error and may not always find the shortest path, but in practice finds paths that are at least as short as those of an A* search on the weighted cell edges. Multi-resolution Field D* [47] aggregates grid cells with the same weight together into one cell, reducing the computation required when evaluating a shortest path. 3D Field D* [23] provides a partial extension of the Field D* cost equations to 3D, while Generalized Field D* [119] extends Field D* to arbitrary triangles.

The main disadvantage of the Steiner point techniques is their pre-processing and space requirements, since, to reach a specified error tolerance, an environment must be sampled as a pre-process before a search can be performed. In contrast, Field D* can perform a search on the underlying weighted grid without the need for pre-processing. The path produced by Field D* is approximate, as is the path produced by Steiner point techniques, but Steiner point techniques have the advantage of being able to place an upper bound on the error in the shortest path. Field D* also finds points on the boundaries of triangles to travel from, but this is achieved by minimising a cost function that interpolates the path costs between nodes. Thus, Field D* achieves by computation what the Steiner point techniques achieve by node placement, or extra space. This is desirable, given that memory access in modern CPU's is expensive compared to computation.

The fact that Field D* is built on D* Lite is another advantage in that the algorithm is specifically designed to cater for scenarios where replanning occurs due to changes in the environment. Multi-resolution Field D* was, in fact, created to cater for the limited memory available on robots by maintaining a high-resolution environment in the robot's immediate vicinity, and a low-resolution representation further away. Therefore, Field D* is useful in cases where the underlying representation is changing. The difficulty that Steiner point techniques face in these cases is that the graph on which the search is performed must be resampled. It may be possible to only resample the parts of the environment that change, but these techniques depend on geometric parameters such as the smallest angle between adjacent boundary edges or the maximum integer coordinate of a vertex. To our knowledge, no work has yet been performed on adapting Steiner point techniques to dynamic environments.

Field D* thus provides a useful alternative to Steiner point techniques in cases where memory is constrained, precise error bounds are not required and the underlying representation is dynamically changing.

2.7 Conclusion

This chapter describes the literature for finding shortest paths through graphs and weighted regions. Graph-based search algorithms, while suitable for searches on graph structures, do not adapt directly to weighted regions. The Weighted Region Problem [93] was posed specifically for the purposes of finding shortest paths across a weighted planar polygonal subdivision. We classified the approaches to solving this problem into two different categories: *Steiner point* and *Field D** techniques. *Steiner point* techniques discretise weighted regions by introducing extra nodes and edges. Field D*, in contrast, introduces cost functions for finding the shortest paths across the cells of a weighted grid.

Steiner point techniques are able to place a bound on the error in their shortest path approximation, but this requires a pre-process and varying degrees of extra space. While Field D* places no such theoretical bounds on its shortest path, the resultant path is at least as short as that produced by an A* search on the weighted cell edges and is able to pass through these cells. Field D* has also been developed to operate on dynamically changing and multi-resolution environments. This makes it an appropriate solution to the Weighted Region Problem in cases where space is at a premium, a theoretical bound is not required, and the environment may be changing dynamically.

In the following chapter, we describe the foundations of the Field D*, the algorithm itself, and some variants.

Chapter 3

Field D* Foundations and Variants

In this chapter we present the foundations of the Field D* algorithm and also describe relevent techniques that build on Field D*. As is the case with most path-planning algorithms, Field D* has its original basis in Dijkstra's algorithm [37]. It incorporates the heuristic developed for A* [61], as well as the replanning capabilities of Lifelong Planning A* [73] and D* Lite [86].

The Field D* algorithm [48] is described by showing how Field D* adapts traditional edge-based cost functions to those operating across the faces of a weighted grid. Next, Field D*'s path-extraction process is explained in detail, expanding the condensed description given here [48] and here [49].

A number of algorithms derived from Field D* are also covered. Multi-resolution Field D* [47] extends Field D* to multi-resolution grids while 3D Field D* [23] is an extension to 3D unit grids. We note some of the problems that Field D*, Multi-resolution Field D* and 3D Field D* may encounter through their use of interpolation in Section 3.5. Finally, we describe Generalized Field D*, an extensions of Field D* to arbitrary triangles.

3.1 Field D*

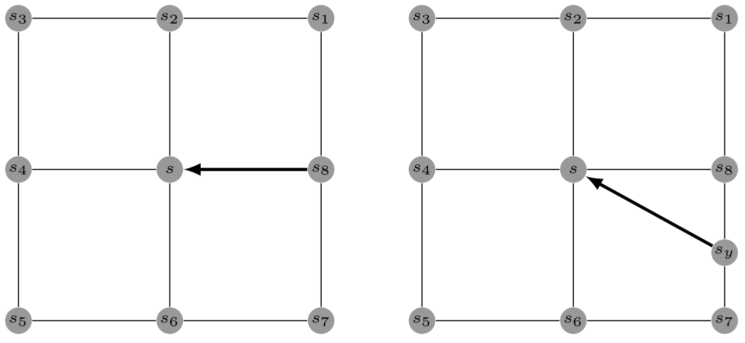
Graph search algorithms are appropriate when the search problem can be mapped to a graph structure. This becomes more difficult when the problem is extended to finding the shortest path through a set of weighted regions. *Steiner point techniques*, described in Section 2.3.2, address this problem by *sampling*: Extra nodes are introduced within the weighted regions and new edges are introduced to connect the additional nodes. This raises the computational complexity of the problem from $O(n\,log\,n)$ to $O(k\,n\,log\,k\,n)$, where n is the number of original nodes and k relates to the degree of sampling. The spatial complexity also increases due to the additional nodes and edges.

Field D* [49] adopts a different approach to the weighted region problem. Firstly, the problem is restricted to finding a shortest path through a weighted uniform grid. The underlying structure, instead of being a mathematical graph is a *weighted grid* consisting of grid nodes and cells. Thus, instead of being connected to each other through edges, nodes are connected to each other via adjacent, weighted grid cells. Field D* calculates a shortest path through this grid structure, and while information about the cost of the path is propagated and stored within the grid nodes, the path may travel through the edges and interior of a grid cell.

3.1.1 Field D* Cell Cost Function

Field D* adapts the search algorithm in D* Lite [86] to operate on a weighted grid. In D* Lite, itself derived from A* and Dijkstra's algorithm, the path cost $g(s)$ for a node s is derived from the path cost $g(s')$ of its surrounding nodes $s' \in \text{nbrs}(s)$ and the cost of the connecting edge $c(s, s')$. This type of configuration is show in Figure 3.1a and is expressed as:

$$g(s) = \min[c(s, s') + g(s')] \text{ where } s' \in \text{nbrs}(s) \tag{3.1}$$



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

Figure 3.1: (a) In graph-based pathfinding, paths only travel along grid edges, while for (b) Field D*, the path may originate on an edge and travel through a neighbouring cell

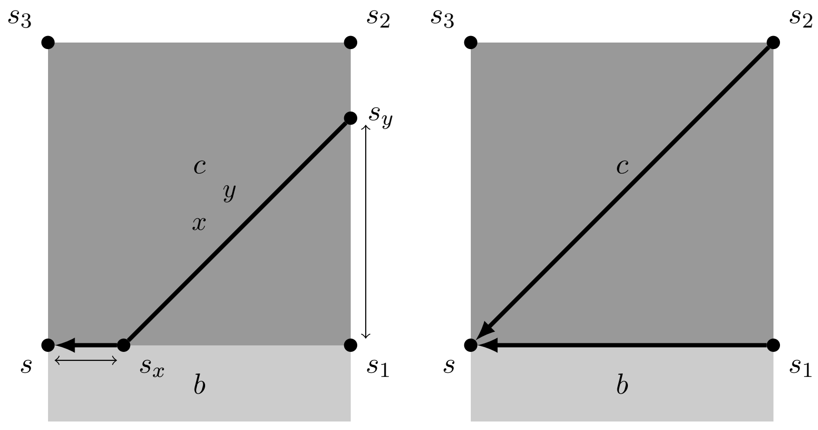
Field D* modifies the derivation of the cost. Instead of calculating the cost from a weighted edge, it must calculate the cost of travelling through a weighted cell and thus the cost function becomes more complicated. Two factors come into play here. Firstly, instead of travelling to the node across a discrete edge, it is now possible to travel to the node from any point on the grid cell boundary, as shown in Figure 3.1b. Secondly, since it is possible to travel from any point on the grid cell boundary, the accumulated path cost $g(s_y)$ for some boundary point s_y must be estimated from the values of the surrounding nodes. Field D* accomplishes this by interpolating the path cost values of the two nodes on either side of the boundary. So if s_1 and s_2 are the nodes on either side of the boundary, the path cost for some point $g(s_y)$, $0 \leq y \leq 1$ is estimated by linear interpolation as:

$$g(s_y) = (1 - y)g(s_1) + yg(s_2) \tag{3.2}$$

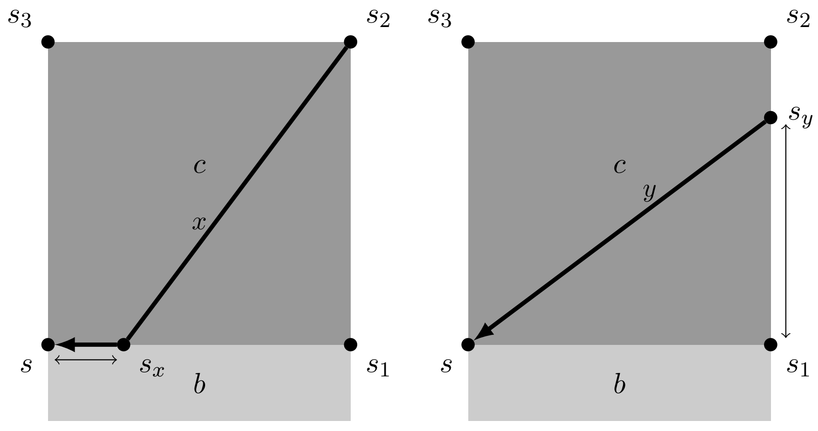
Thus, instead of iterating over graph edges adjacent to a node and choosing the edge that produces the minimum cost, Field D* now needs to iterate over grid cells adjacent to a node, evaluating the least cost path within each cell and the minimum cost overall. Function 3.1 then becomes:

$$g(s) = \min_{x,y}[bx + c\sqrt{(1-x)^2 + y^2} + (1-y)g(s_1) + yg(s_2)] \tag{3.3}$$

where c is the weight of the cell, b is the weight of the adjacent cell and s_1 and s_2 are neighbouring nodes of s that are adjacent to each other. A visual configuration of this function is shown in Figure 3.2a. Variables x and y parameterise vectors $\overrightarrow{ss_1}$ and $\overrightarrow{s_1s_2}$ respectively. Note that the cell is a unit square and thus $|s - s_1| = |s_1 - s_2| = 1$ and $|s - s_2| = \sqrt{2}$. This function generalises the cost of travelling across one half of the cell. The same function must be applied to the other half of the cell to fully consider all least cost paths across the cell.



(a) General **(b)** Trivial



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

Figure 3.2: The (a) Field D* cost equation and its three sub-cases (b) Trivial (c) Indirect and (d) Direct

The authors show in [49] that it is possible to eliminate a variable from Function 3.3 to produce three sub-cases of the function. The first is the *Trivial* case, so named because the path travels directly between nodes, does not require any minimisation of a variable to calculate the least cost path, and produces paths similar to pathfinding algorithms on edge-based graphs. From the configuration in Figure 3.2b these functions can be expressed as:

$$g(s) = \min(b, c) + g(s_1) \tag{3.4}$$

$$g(s) = c\sqrt{2} + g(s_2) \tag{3.5}$$

In the above functions, the first trivial path travels either from node s_1, along the edge shared by the cells with weight b and c to node s. The second travels from node s_2 through the cell with weight c.

The second sub-case is the *Indirect* case, so named because the path starts at a node, then cuts across the cell to the cell boundary before travelling to the destination node. They are expressed as:

$$g(s) = c\sqrt{1 + (1 - x)^2} + bx + g(s_2) \tag{3.6}$$

In the above function the indirect path travels from node s_2 to some point on vector $\overrightarrow{ss_1}$, parameterised by variable x. This configuration is shown in Figure 3.2c. The parameterisation of $\overrightarrow{ss_1}$ that produces shortest cost path across the cell for this case depends on the differences between the weight c of the cell being considered, and the weight b of travelling along the edge ss_1 shared with the adjacent cell. Minimising equation 3.6 in terms of x yields:

$$x = 1 - \sqrt{\frac{b^2}{c^2 - b^2}} \tag{3.7}$$

Note that Function 3.7 requires $b < c$ to produce a real value for x. If $b \geq c$, it would be cheaper to travel directly throught the cell with weight c to node s and would devolve into the following case.

The third sub-case is called the *Direct* case, because the path travels from a point on the cell edge directly across the cell in question. It is also the case in which the interpolation assumption of Field D* is exercised since the cost of the point on the cell edge must be estimated. Setting $f = g(s_1) - g(s_2)$ to be the relative difference in node costs between s_1 and s_2, the function for this path is expressed as:

$$g(s) = c\sqrt{1 + y^2} + f(1 - y) + g(s_2) \tag{3.8}$$

The path cost of a point s_y on vector $\overrightarrow{s_1 s_2}$ is estimated by linearly interpolating the values of $g(s_1)$ and $g(s_2)$ along the vector with variable y. Thus, the function must be minimised in terms of both this linear interpolation and the cost of travelling through the cell with weight c. Minimising Function 3.8

in terms of y yields

$$y = \sqrt{\frac{f^2}{c^2 - f^2}} \tag{3.9}$$

Algorithm 7 Field D*'s cost function. It can be used in any node-based planning and replanning algorithms as long as the underlying data structure is a weighted grid.

```
 1: function ComputeCost($s, s_a, s_b$)
 2:     if $s_a$ is a diagonal neighbour of $s$ then
 3:         $s_1 = s_b; s_2 = s_a;$
 4:     else
 5:         $s_1 = s_a; s_2 = s_b;$
 6:     end if
 7:     $c$ is traversal cost of cell with corners $s$,$s_1$,$s_2$
 8:     $b$ is traversal cost of cell with corners $s$, $s_1$ but not $s_2$
 9:     if $\min(c, b) = \infty$ then
10:         $v_s = \infty$
11:     else if $g(s_1) < g(s_2)$ then
12:         $v_s = \min(c, b) + g(s_1)$
13:     else
14:         $f = g(s_1) - g(s_2)$
15:         if $f \leq b$ then
16:             if $c \leq f$ then
17:                 $v_s = c\sqrt{2} + g(s_2)$
18:             else
19:                 $y = \min(\sqrt{\frac{f^2}{c^2-f^2}}, 1)$
20:                 $v_s = c\sqrt{1 + y^2} + f(1 - y) + g(s_2)$
21:             end if
22:         else
23:             if $c \leq b$ then
24:                 $v_s = c\sqrt{2} + g(s_2)$
25:             else
26:                 $x = 1 - \min(\sqrt{\frac{b^2}{c^2-b^2}}, 1)$
27:                 $v_s = c\sqrt{1 + (1 - x)^2} + bx + g(s_2)$
28:             end if
29:         end if
30:     end if
31:     return $v_s$
32: end function
```

Similar to the indirect case, $f < c$ must hold to produce a real value for y. Thus, Function 3.3 has been reduced to three cases: *Trivial*, *Indirect* and *Direct*. The appropriate case depends on the relative sizes of c, b and f and the pseudo-code for this is shown in Algorithm 7.

It is simple enough to know when to use the trivial case. If $f \leq 0$ then the optimal path always travels from s straight to s_1, because $g(s_1) \leq g(s_2)$ and $|s - s_1| < |s - s_2|$ by Pythagorus. The cost of this path will be $\min(c, b) + g(s_1)$.

It requires more effort to determine when to choose the other two cases. In the direct case when $f > 0$, $g(s_1) > g(s_2)$. Thus, as f increases $g(s_1)$ increases relative to $g(s_2)$ and it becomes cheaper for the path to travel from a point on the vector $\overrightarrow{s_1 s_2}$ that is closer to s_2. Indeed, if one examines Equation 3.9 it can be see that y, the value parameterising this vector increases as f increases since the denominator decreases with increasing f.

In a similar manner for the indirect case, as b increases the value of x produced by Equation 3.7 *decreases*, since it becomes more expensive to travel along the bottom edge before cutting across the cell to s_2.

To choose between the indirect and direct cases, the authors begin by generalising Functions 3.6 and 3.8 to the form:

$$g(s) = c\sqrt{1 + y^2} + k(1 - y) + g(s_2) \tag{3.10}$$

When attempting to solve Function 3.6, b is substituted for k and $1 - x$ for y and for Function 3.8, f is substituted for k. Note that when $k = f = b$, the costs produced by the two sub-cases are equivalent. Thus, since the same equation is being solved for both cases, it is only necessary to solve for the edge with the cheapest cost. If $f < b$ then Function 3.8 will produce the cheapest cost, and vice versa. Figure 3.3 illustrates this relation.

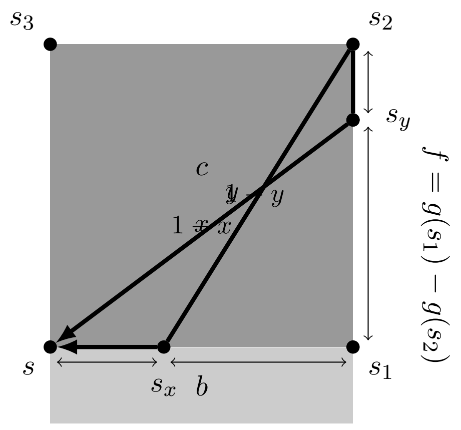


Figure 3.3: Choosing between the Indirect and Direct sub-case depends on the relative sizes of b and f. If $b < f$ then it is cheaper to choose the bottom edge and the reverse holds if $b > f$.

In the case where $b > c$, Function 3.7 produces a complex root. Physically this means that there is no point on the bottom edge that will produce a minimum cost, since it will always be cheaper to directly travel through the cell and thus the cost for this case can be set to $g(s) = c\sqrt{2} + g(s_2)$. The same applies when $f > c$ for Function 3.9.

3.1.2 Main Algorithm

The authors use D* Lite, described in Section 2.2.4 as their basic planning algorithm. The major difference between the two algorithms is that the ComputeCost Function is used in D* Lite's UpdateNode Function when iterating around a node and computing the least cost path to it. Field D* is shown in 8.

Thus, Field D* is also based on a priority queue containing nodes. The node with the least cost is popped from the queue and the cost of its children calculated via the use of the ComputeCost Function. The children are then placed on the priority queue. This continues until the start node [1] is reached.

3.1.3 Path Extraction

Once the start node has been reached, *Path Extraction* must be performed. In traditional pathfinding algorithms based on node and edge graph structures, this can be accomplished by storing the predecessor of a node within the node itself and iterating backwards over the predecessors until the entire path has been extracted. In another sense, the predecessor node represents the node from which the current node derived its cost. One can also begin with the start node, and transition to the cheapest adjacent node until the goal node is reached.

Field D*'s path extraction process is more complicated since it is now possible for path points to occur on cell boundaries. The authors explain their path extraction process using the concept of predecessors, but a predecessor now refers to a point on an edge between two nodes. This predecessor edge for a node s, bptr$(s) = s'$ is stored as most *clockwise* node s' on the edge $s's''$ as shown in Figure 3.4a.

The exact predecessor point p for a node s can then be determined firstly by retrieving s' from bptr(s) and then determining the counter-clockwise node s'', relative to s and s' in the *weighted grid*. ComputeCost(s, s', s'') can then be calculated and the resulting point p used as the next point in the path. cknbr(s, s') and ccknbr(s, s') are the names of the functions used to retrieve the clockwise and counter-clockwise nodes, respectively.

If the predecessor point p for s lies on either s' or s'', then the same process can be used to find the predecessor point for these nodes. However, if p is an interpolated point on the edge between s' or s'' then there is no predecessor or connectivity information stored for this point. Unfortunately, it is not made entirely clear in previous work how the next point on the path from an interpolated point p, is computed. Section 4 of [49], which describes the main Field D* algorithm states:

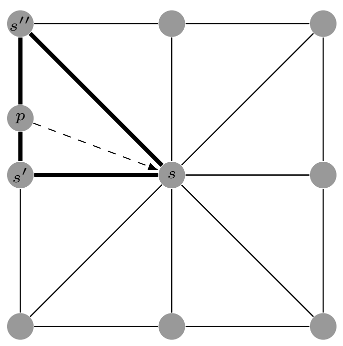
> Once the cost of a path from the initial node to the goal has been calculated, the path can be extracted by starting at the initial position and iteratively computing the cell boundary

[1]Recall that Field D* is based on D* Lite and searches from the goal to the start node.

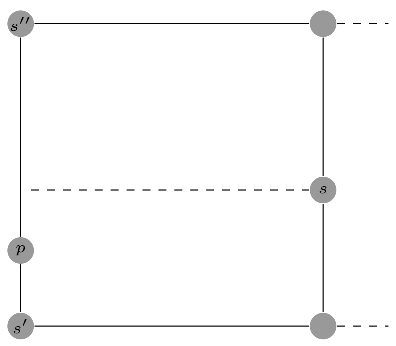
Algorithm 8 The Field D* algorithm. nbrs(s) denotes the neighbouring nodes of u, while connbrs(s) denotes the set of neighbouring node *pairs* surrounding u, $\{(s_1, s_2), (s_2, s_3), \cdots, (s_8, s_1)\}$.

```
1:  function KEY($s$)
2:      return $[\min(g(s), \text{rhs}(s)) + h(s_{\text{start}}, s); \min(g(s), \text{rhs}(s))]$
3:  end function
4:  function UPDATENODE($u$)
5:      if $u$ was not visited before then $g(u) = \infty$
6:      end if
7:      if $u \neq s_{\text{goal}}$ then
8:          $\text{rhs}(u) = \min_{(s',s'') \in \text{connbrs}(u)} \text{ComputeCost}(u, s', s'')$
9:      end if
10:     if $u \in U$ then U.Remove($u$)
11:     end if
12:     if $g(u) \neq \text{rhs}(u)$ then U.Insert($u$, Key($u$))
13:     end if
14: end function
15: function COMPUTESHORTESTPATH
16:     while U.TopKey() $<$ Key($s_{\text{start}}$) OR $\text{rhs}(s_{\text{start}}) \neq g(s_{\text{start}})$ do
17:         u = U.Pop()
18:         if $g(u) > \text{rhs}(u)$ then
19:             $g(u) = \text{rhs}(u)$
20:             for all $s \in \text{nbrs}(u)$    UpdateNode($s$)
21:         else
22:             $g(u) = \infty$
23:             for all $s \in \text{nbrs}(u) \cup \{u\}$    UpdateNode($s$)
24:         end if
25:     end while
26: end function
27: function MAIN
28:     $g(s_{\text{start}}) = \text{rhs}(s_{\text{start}}) = \infty; g(s_{\text{goal}}) = \infty$
29:     $\text{rhs}(s_{\text{goal}}) = 0; U = \emptyset$
30:     U.insert($s_{\text{goal}}$, Key($s_{\text{goal}}$))
31:     loop
32:         ComputeShortestPath()
33:         if any cell weights have changed then
34:             for all cells $x$ with new weights do
35:                 for all nodes $s$ on $x$ do
36:                     UpdateNode($s$)
37:                 end for
38:             end for
39:         end if
40:     end loop
41: end function
```

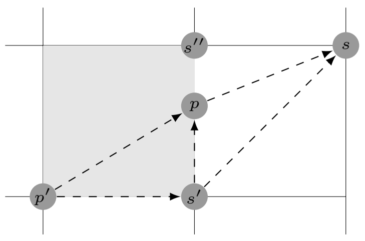
point to move to next. Because of our interpolation-based cost calculation, it is possible to



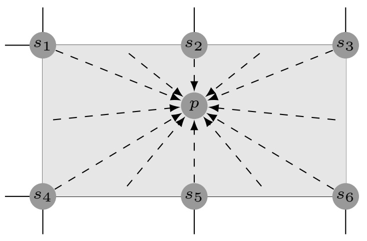
(a) Predecessor of a Node



(b) Predecessor of an Interpolated Point



(c) Breakdown of the Interpolation Assumption



(d) Testing the Interpolation Assumption

Figure 3.4: (a) The predecessor p to node s is determined firstly by retrieving s', the clockwise node relative to s, of the edge containing the point, from bptr(s). The counter-clockwise node on the edge s'' is determined from the relative positions of s and s' in the *weighted grid*. p is then calculated from ComputeCost(s, s', s''). (b) If s is an interpolated point on the path, the ComputeCost function would need to operate on rectangles in addition to unit-squares to calculate the next point on the path from s. (c) The interpolation assumption can break down. ComputeCost(s, s', s'') may produce p as the next path to travel to. If the gray cell is very expensive to travel through, it would make sense to travel around the cell to p' via s', in which case, one should transition to s' in the first place. (d) Field D* tests the accuracy of the interpolation assumption by calculating the cost of travelling to p from all the surrounding nodes and interpolated edges.

> compute the path cost of any point inside a grid cell, not just the corners, which is useful for both extracting the entire path and calculating accurate path costs from noncorner points.

It is possible that the authors use the ComputeCost function on the cells around an interpolated point p in order to find the next point on the path. However, the cells adjacent to the interpolated point would now be shaped as rectangles as shown in Figure 3.4b and the ComputeCost function operates on a unit-square. Alternatively, the quoted statement could also mean that Function 3.2 is minimised to find the point on edge $s's''$ where the interpolation $g(p)$ of path costs $g(s')$ and $g(s'')$ is minimal. We discuss the latter option in Section 3.5.

Later in Section 5 of [49], where the actual Path Extraction process is described, the authors explain that there are cases where Field D*'s interpolation assumption can be incorrect. Consider Figure 3.4c, which contains a very expensively weighted dark gray cell surrounded by inexpensively weighted white cells. Assume $g(s') < g(s'')$ and that when ComputeCost(s, s', s'') is invoked a *Direct* case is produced, and the linear interpolation assumption results in p as the least cost point to travel to. Now consider the subsequent least cost point to travel to from p. Clearly the path to this point cannot travel through the expensive cell. It would be cheaper to follow the edges of the expensive cell and travel to s' and then to p', for example. This means it would be cheaper to simply travel from s to s' in the first place.

Thus, the linear interpolation assumption can be incorrect and Field D* checks this interpolation assumption via the use of a one-step lookahead. From Section 5 of [49]:

> ...we calculate a more accurate approximation of the path cost of p. We do this by looking to its neighboring edges and computing a locally optimal path from p given the path cost of the endpoint nodes of these edges and interpolated path costs for points along the edges ...

The cost of travelling to p from the surrounding nodes is calculated as shown in Figure 3.4d. For example the cost of travelling from s_1 to p would be $c|s_1 - p|$ where c is the weight of the cell. Once again, it is not clear from the text whether some modified version of ComputeCost is used when calculating the optimal path to p from interpolated points along the edges.

Given this new improved estimate of the path cost at p, Field D* decides whether p is still the cheapest point to transition to. If not, it will transition to the point indicated by the *Trivial* or *Indirect* sub-cases. For instance, in our previous example, the algorithm transitioned to s' instead of p, since it was easier to travel to p' around the dark cell. Curiously, having introduced the lookahead, the following is stated:

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

A path produced by Field D* is shown in Figure 3.5. The path through this environment avoids the expensively weighted squares, represented with darker shading, and interpolates through the inexpensive, lightly shaded squares.

3.2 Multi-resolution Field D*

In order to accurately represent paths in a complex, detailed environment Field D* may require a high-resolution grid. Increasing the grid resolution increases the number of cells and consequently the number of cost equations Field D* needs to evaluate to find a shortest path. Thus, the time and space complexity of the algorithm increases.

Since Field D* was developed to operate on the Mars Rovers [49], a high-resolution grid was not practical, since the robot's processing power and onboard memory were limited.

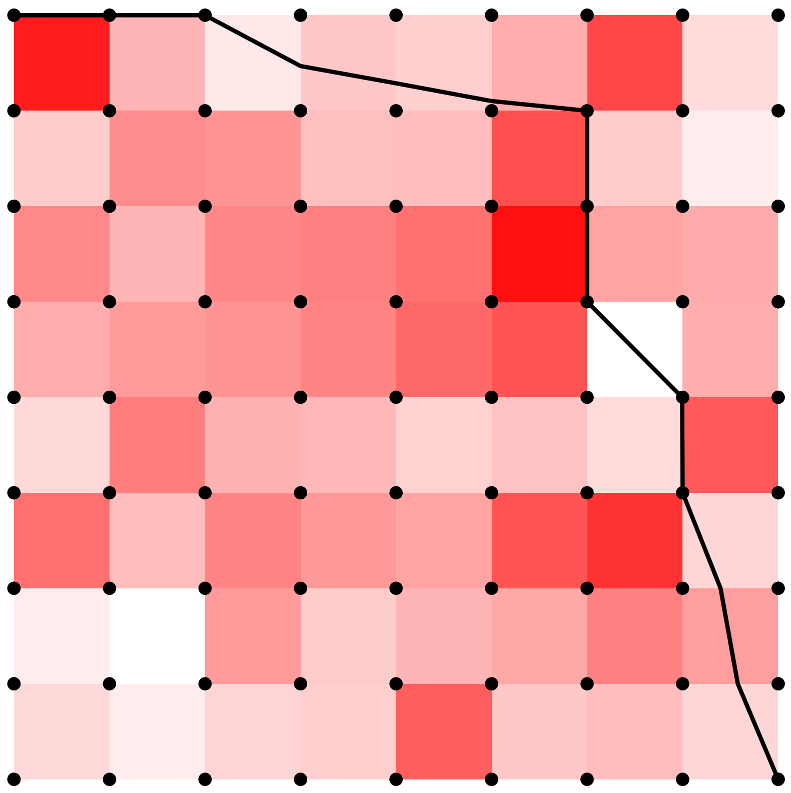


Figure 3.5: Field D* path through a grid. Cells weighted with a darker shade of red are more expensive, while ligher shades are correspondingly cheaper to travel through.

Multi-resolution Field D* [47] was developed to reduce these resource requirements. It adapts Field D* from operating on a weighted grid to operating on a weighted region *quadtree* [118]. A quadtree is a tree data structure that recursively divides square regions into quarters. Each tree node is associated with a square region and has four children associated with the subdivision of the parent.

Quadtrees recursively subdivide a two-dimensional space in progressively finer quarters. Large regions of constant cost can be represented as a single cell, while other regions containing smaller areas of differing cost can be recursively subdivided. This represention is known as a nonuniform resolution grid or *multi-resolution* grid.

A cell within this multi-resolution grid may be either lower or higher in resolution relative to its neighbouring cells. In Figure 3.6a for example, the white node is adjacent to three low-resolution and one high-resolution cell. Multi-resolution Field D* proposes using interpolation to propagate path costs from nodes surrounding these cells.

In Field D* on a uniform grid, a node only has eight neighbours, but in the multi-resolution case, this number may be vary, depending on the connectivity and level of subdivision within the grid. The authors identify three separate cases that should be dealt with when propagating costs to a node.

Node is a corner of a low-resolution cell: If it is necessary to propagate costs from an edge with a

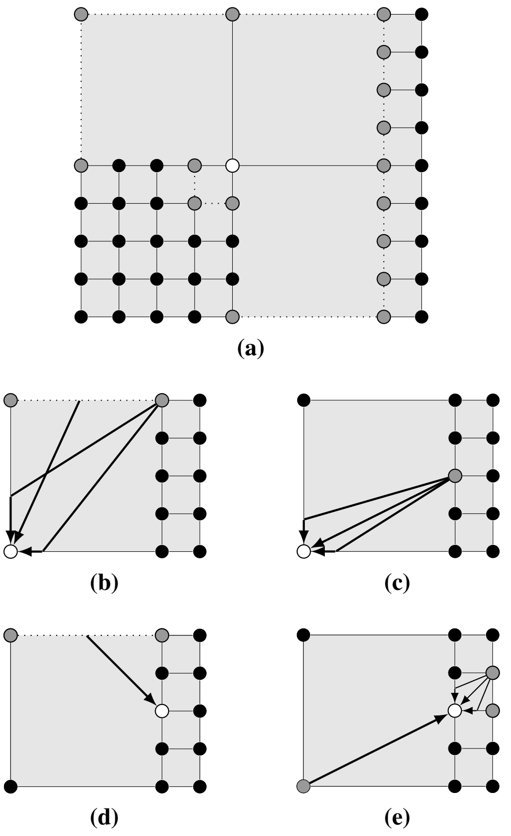


Figure 3.6: (a) Three low-resolution and a greater number of high-resolution cells. The white node is the node for which the cost is being calculated. The grey nodes are the neighbouring nodes of the white node. (b) and (c) show the possible least cost paths to a node on a corner of a low-resolution cell. (d) and (e) show possible least cost paths to a node on a corner of a high-resolution cell.

cell of low resolution, then the ComputeCost function developed for Field D* can be used. In Figure 3.6b for example, a *Direct* and two *Indirect* case for the top edge are illustrated. Some subtlety is required when the edge under consideration is high-resolution, as shown in Figure 3.6c. The authors suggest two approaches.

Firstly, they suggest that it is possible to use the interpolation techniques developed for Field D* to compute the least cost path from a node on the high-resolution edge to the node whose cost is being calculated. As noted in Section 3.1.3, the ComputeCost function operates on a unit square, and would presumably need to be extended to rectangles to be applicable here. An alternative interpretation is that Function 3.2 is utilised to estimate the point on the edge which has the minimal interpolated cost. This approach has accuracy problems which we discuss further in Section 3.5.

Secondly, the use of interpolation is discarded and only the cost of travelling from the high-resolution node to the low-resolution node is computed. The authors note that this sacrifices some accuracy for simplicity of implementation. In Figure 3.6c for example, only the *Direct* path would be considered.

Node is a corner of a high-resolution cell: ComputeCost can once again be used to calculate the

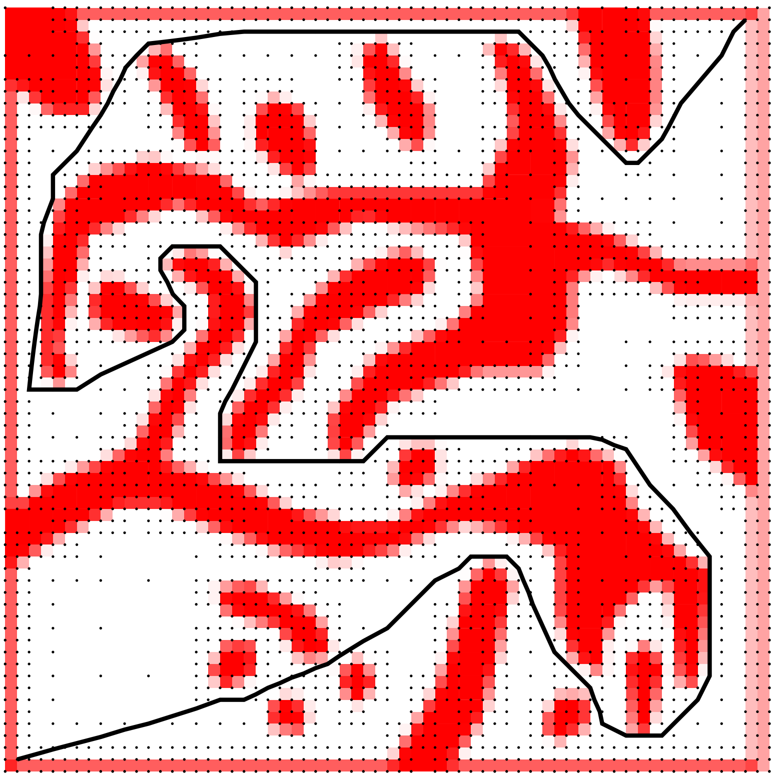


Figure 3.7: Multiresolution Field D* path through a weighted quadtree

costs of a path through a high-resolution cell to one of its corner nodes. This is illustrated for the paths from the right-hand edge in Figure 3.6e.

Node is on the edge of a low-resolution cell: In this case, the authors suggest that when deriving the cost from a high-resolution edge, then, as in the low-resolution cell corner case, either interpolation or a direct path can be used. In Figure 3.6e, a direct path is used on the left edge for example. Otherwise, if the cost is derived from an edge of low resolution (Figure 3.6d), the authors state that interpolation can once again be used. Once again we note that a ComputeCost for rectangles would be required for this to be possible.

Thus, when the algorithm attempts to calculate the path cost for a node, the cells adjacent to the node are examined and, using the above cases, the minimum cost through the cell is calculated. The least cost path through all the cells is then selected and used as the cost of the node. The pseudo-code for this is shown in Algorithm 9. P_e is the infinite set of all points on the edge e, while EP_e is the set consisting of the two endpoints of edge e. $g^i(p)$ approximates the path cost at point p by interpolating between the path cost of the endpoints for the edge containing p, while $c(s, p)$ is the least cost path travelling from p to s through a cell of some weight. $g(p)$ is the path cost of an edge's endpoint p. Edge e is considered to be low-resolution if the cells to either side of e are low-resolution. The authors refer the reader to the original cost functions when solving the least cost path from points $p \in P_e$.

In their experiments comparing Multi-resolution Field D* to the original Field *, the authors examine

Algorithm 9 Multi-resolution Field D* ComputePathCost Function

```
1: function COMPUTEPATHCOST($s$)
2:     $v_s = \infty$
3:     for each cell $x$ adjacent to $s$ do
4:         if $x$ is a high-resolution cell then
5:             for each neighbouring edge $e$ of $s$ on the boundary of $x$ do
6:                 $v_s = \min(v_s, \min_{p \in P_e}(c(s,p) + g^i(p)))$
7:             end for
8:         else
9:             for each neighbouring edge $e$ of $s$ on the boundary of $x$ do
10:                if $e$ is a low-resolution edge then
11:                    $v_s = \min(v_s, \min_{p \in P_e}(c(s,p) + g^i(p)))$
12:                else
13:                    $v_s = \min(v_s, \min_{p \in EP_e}(c(s,p) + g(p)))$
14:                end if
15:            end for
16:        end if
17:    end for
18:    return $v_s$
19: end function
```

the performance and path costs of Multi-resolution Field D* operating at different levels of resolution. They randomly generate a 100x100 environment and randomly subdivide a percentage of cells into 10x10 high-resolution cells. Thus, if the environment only consists of low-resolution cells, it is considered to be at 0% resolution, while if it consists of only high-resolution cells, it is considered to be at 100% resolution. The authors find that in initial planning Multi-resolution Field D* offers superior performance to Field D* when 80% or less of the environment is represented as high-resolution. Even when the resolution is at 0% the difference in path cost is only between 1.006 and 1.008 times that of the path cost produced by standard Field D*.

In a second experiment involving a robot traversing a terrain, Multi-resolution Field D* computes a path through an environment at 13% of the resolution of a uniform grid, in half the time of standard Field D*.

3.3 Optimisations

The authors also present several optimisations to the main Field D* algorithm. The main optimisation involves reducing the number of cost equations evaluated when calling UpdateNode. In the main Field D* algorithm, when a node s is popped off the stack, its g-values are updated and the surrounding nodes, $s' \in \text{nbrs}(s)$ are updated so that their path costs reflect this change in relation to s's cost. To accomplish this, the standard UpdateNode function evaluates all the neighbouring cells of s', even though only the path cost of s has changed. The authors modify Field D* to only evaluate the cells on either side of edge ss', as shown in Figure 3.8. Evaluating the path costs of the other cells is unnecessary since the path costs of their nodes have not changed during this iteration of the algorithm.

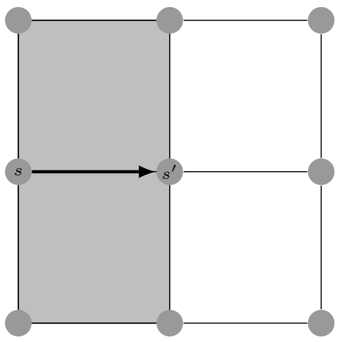


Figure 3.8: Optimisation of *UpdateCost*, Node s was updated, and thus its neighbours, $s' \in \text{nbrs}(s)$ must also be updated to incorporate the new path cost $g(s)$. However, only the path costs through the cells on either side of edge ss' need be evaluated, since only the path cost of s has changed.

3.4 3D Field D*

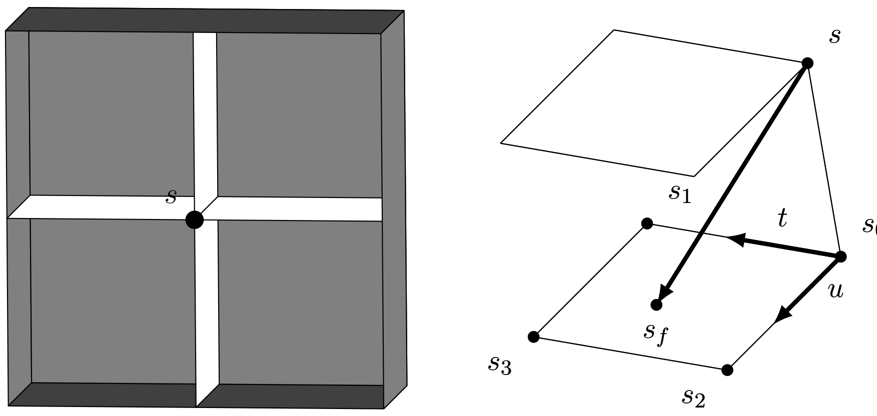
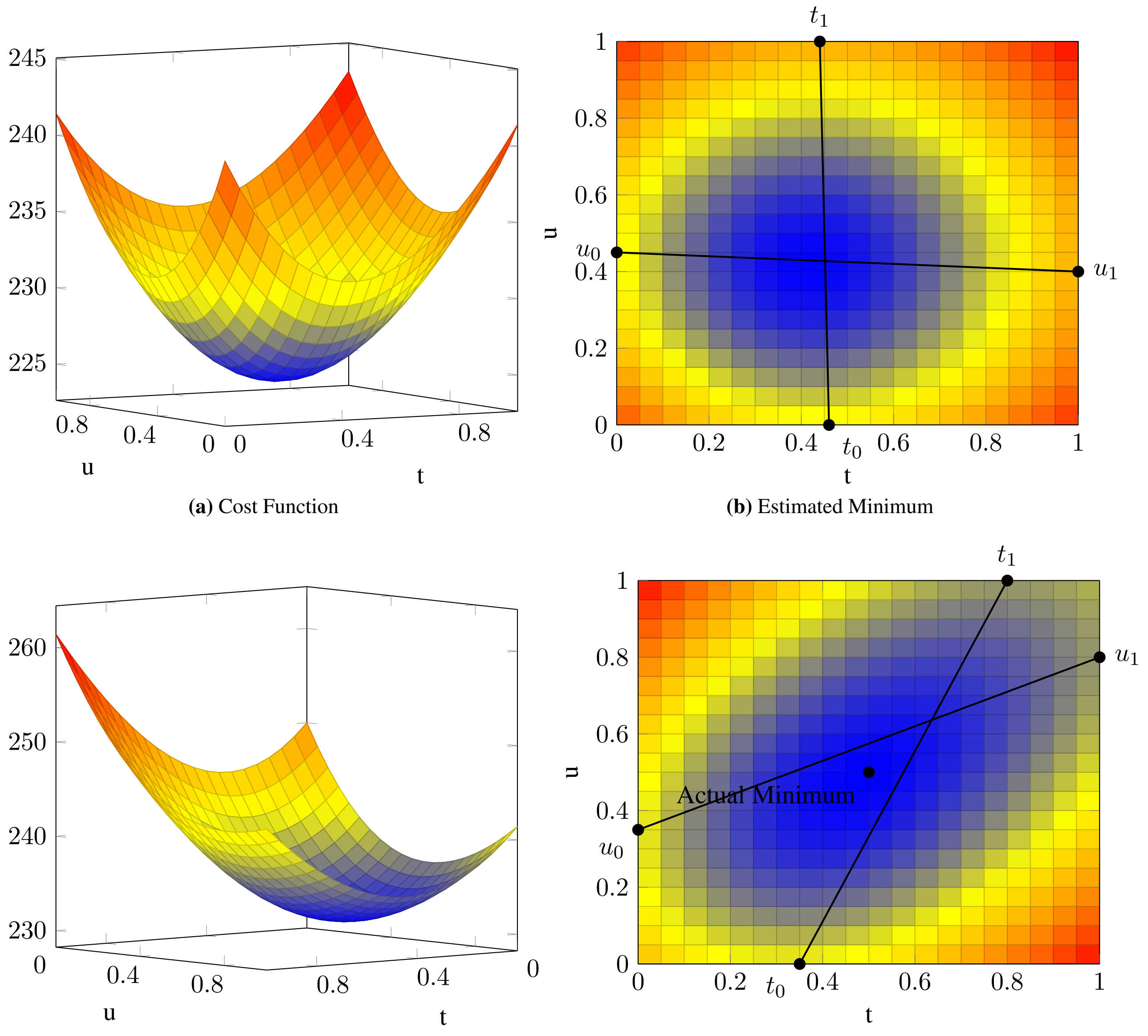


Figure 3.9: (a) Four of the eight octants adjacent to node s. (b) To calculate the least cost path from face f to node s, the path cost values of the nodes on the face corners are bilinearly interpolated. The interpolation is parameterised by variables t and u.

3D Field D* [23], extends the standard Field D* algorithm to operate on a three-dimensional (3D) grid. Whereas the 2D version of the algorithm uses interpolation to estimate path costs on grid cell edges, 3D Field D* interpolates path costs over 3D cell *faces*.

In 3D Field D*, each node s, is a corner node of eight neighbouring octants. Each octant contains faces, as shown in Figure 3.9a. As each face has four nodes, the path costs of these nodes must be interpolated over the face. Thus, Function 3.2 is modified to become:



(a) Cost Function **(b)** Estimated Minimum

(c) Cost Function **(d)** Innacurate Estimation

Figure 3.10: (a) Visualisation of Equation 3.8, parameterised by variables v and u. (b) 3D Field D* estimates the minima along each edge, connects the opposing minima using lines, and estimates that the minimum for this equation will be at the line intersection. This may sometimes be innacurate as in (c) and (d).

$$\begin{aligned} g(s_f) = {} & [g(s_1) + (g(s_0) - g(s_1)) \cdot t] \cdot (1 - u) \\ & + [g(s_2) + (g(s_3) - g(s_2)) \cdot t] \cdot u \end{aligned} \tag{3.11}$$

where t and u parametrise the position of s_f in the face f defined by the four vertices s_0, s_1, s_2, s_3 with path costs $g(s_0), g(s_1), g(s_2), g(s_3)$. This configuration is shown in Figure 3.9b. The authors define the path cost of travelling to a node s from the face of a node with weight c as:

$$\begin{aligned} g(s) = {} & c \cdot \sqrt{1 + t^2 + u^2} + \\ & + [g(s_1) + (g(s_0) - g(s_1)) \cdot t] \cdot (1 - u) \\ & + [g(s_2) + (g(s_3) - g(s_2)) \cdot t] \cdot u \end{aligned} \tag{3.12}$$

This function is visualised in Figure 3.10a. To calculate the shortest path from face f to node s, Equation 3.12 must be minimised with respect to t and u. The authors claim that there is no closed form solution to this Equation. [2] To avoid the expense of using numerical methods, the authors use an approximation technique to estimate the minimum.

Their approximation initially finds the minima for the boundary conditions of Equation 3.12. These boundary conditions are equivalent to the four edges forming face f. The paper text states:

> Finding the minimum along each edge is straightforward. In fact, it is nearly identical to the interpolation-based edge calculation for the two dimensional case ....

but does not elaborate further. As the interpolation-based edge calculation that the authors are referring to is Function 3.2, it is reasonable to assume that the authors estimate the boundary minima by minimising Function 3.2. However, this would imply that the weight c of travelling through the cube is not utilised, but this does not take into account the cost of travelling through the cell and would thus not be accurate in all circumstances. We discuss the implications of this further in Section 3.5.

After calculating the boundary minima, the minima of opposing boundaries are connected by lines as shown in Figure 3.10b, and the intersection point of these lines is examined to see if it is cheaper than the edge minima. If so, the value at the intersection point is chosen as the path cost, otherwise the least cost edge minimum is selected.

The values t_{int} and u_{int} of this intersection point are calculated as follows:

$$\begin{aligned} t_{int} = {} & \frac{(t_1 - t_0) \cdot u_0 + t_0}{1 - (t_1 - t_0) \cdot (u_1 - u_0)} \\ u_{int} = {} & (u_1 - u_0) \cdot t_{int} + u_0 \end{aligned}$$

where t_0, t_1 and u_0, u_1 correspond to the minima pairs for the t-axis and u-axis, respectively. The accuracy of this approximation technique may vary and no comparison with an exact solution is performed. The intersection point in Figure 3.10b is a good estimate, but the estimate in Figure 3.10d does not match the function's (Figure 3.10c) actual minimum.

[2] In Chapter 5 we show how equations of a similar form can be solved for tetrahedra.

Thus, 3D Field D*'s contribution extends Field D*'s *Direct* equation to cubes via the use of an estimation technique. No extension of the *Indirect* case is presented, which would involve travelling from a node, s_3 for example, then some of the way across an adjacent face before cutting across the cell to s.

The path extraction process is also not described, but we note that 3D Field D* would encounter the same difficulties as Field D* in deciding the predecessor of an interpolated point. Since an interpolated point on an octant face would not necessarily subdivide surrounding octants into smaller octants, but rather rectangular cuboids, a cost function that operated on rectangular cuboids would be required to determine the predecessor point of least cost. Alternatively, as may be the case with Field D*, the authors may minimise 3.11 to estimate the predecessor point on surrounding faces.

The authors also expand Function 3.12 to incorporate global scaling factors c_x, c_y and c_z, to represent the expense of travelling in a particular direction:

$$\begin{aligned} g(s) = c \cdot & \sqrt{c_z^2 + (c_x \cdot t)^2 + (c_y \cdot u)^2} + \\ & + [g(s_1) + (g(s_0) - g(s_1)) \cdot t] \cdot (1-u) \\ & + [g(s_2) + (g(s_3) - g(s_2)) \cdot t] \cdot u \end{aligned} \tag{3.13}$$

In path-planning involving aircraft, c_z could be weighted expensively to represent cost in terms of time and fuel.

Experiments on a 1.9GHz Pentium P4 with 512MB RAM show that initial planning with 3D Field D* takes around 20 seconds to expand all 894000 nodes in the test environment. The environment consisted of either free or obstacle cells and does not exercise 3D Field D*'s use of a cell weight c.

3.5 Criticism of Field D*'s use of Interpolation

As we have previously noted, the authors of Field D* and its derived works suggest using interpolation Function 3.2 to estimate the next point to travel to during Field D*'s path extraction process, to calculate path costs for Multi-resolution Field D* and to estimate the minima for boundary conditions in 3D Field D*. In all three cases, the requirement is to find the cheapest path from an edge to a node. To solve this, a point on the edge must be selected as the point to travel from and this point must minimise both an interpolation component (the interpolation of the adjacent path costs along the edge) *and*, a distance component (the cost of travelling from that point to the node).

The *Direct* case of the ComputeCost function presented in standard Field D* solves this type of problem as it includes a distance component ($c\sqrt{1+y^2}$), and an interpolation component ($(g(s_2) - g(s_1))y + g(s_2)$) as shown in Figure 3.12. However, ComputeCost only operates on perfect squares and in the three presented cases, Field D* needs to solve the requirement for non-square cases.

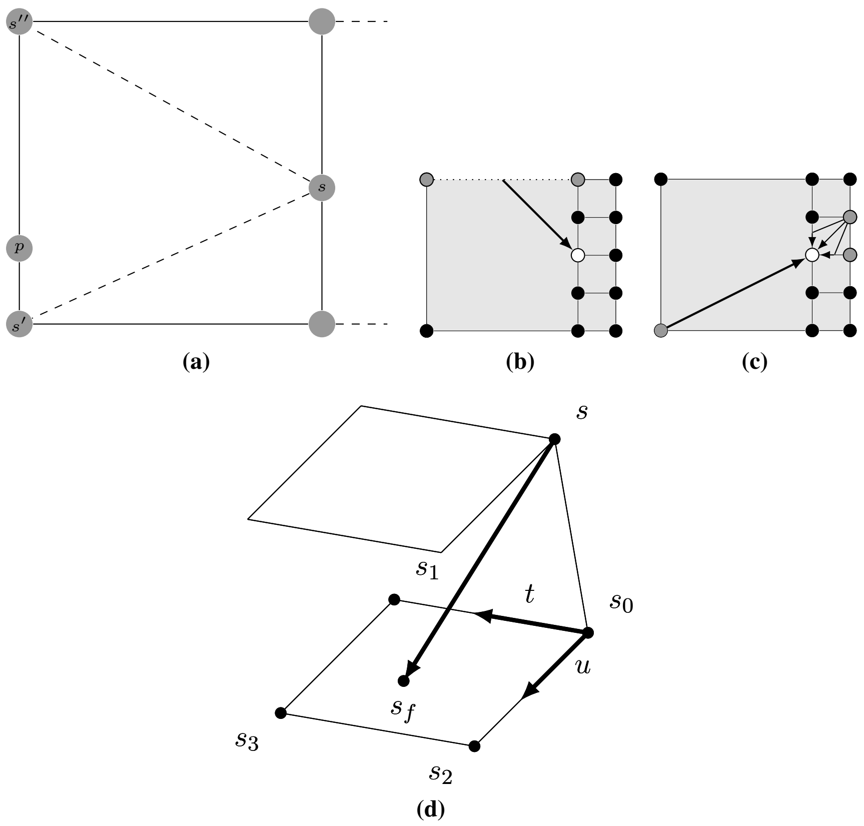


Figure 3.11: Interpolation assumption cases

When the current point in the Field D* path extraction process is an interpolated point, the algorithm needs to decide which point on the boundary of the surrounding cells to travel to. Since the current point is interpolated, the shapes that this point makes with surrounding edges are not squares but triangles, as shown in Figure 3.11a. In [49], the authors suggest that using Function 3.2 is useful to the path extraction process, but admit that it may be innaccurate without clearly explaining their use of it, or an alternative ComputeCost that operates on a rectangle. Indeed, they suggest using a local path planner for path extraction.

Similarly, when propagating node costs (and performing path extraction) in Multi-resolution Field D* [47], it is both necessary to calculate the path costs of nodes that are on the edge of a low-resolution cell (Figure 3.11b) , as well as the path costs *derived* from nodes on the edge of a low-resolution cell. (Figure 3.11c). In the first case, the authors suggest that interpolation can be used to estimate the point to travel from. In the second case, only the cost of the direct path from the grey node to the white node is evaluated.

In the case of 3D Field D* [23], the authors suggest the use of interpolation to estimate the minima for the four boundary conditions corresponding to the face edges (Figure 3.11d). We note that it may be possible to use ComputeCost to estimate the minima for triangles ss_0s_1 and ss_0s_2, since they lie on a unit face, but triangles ss_1s_3 and ss_2s_3 do not.

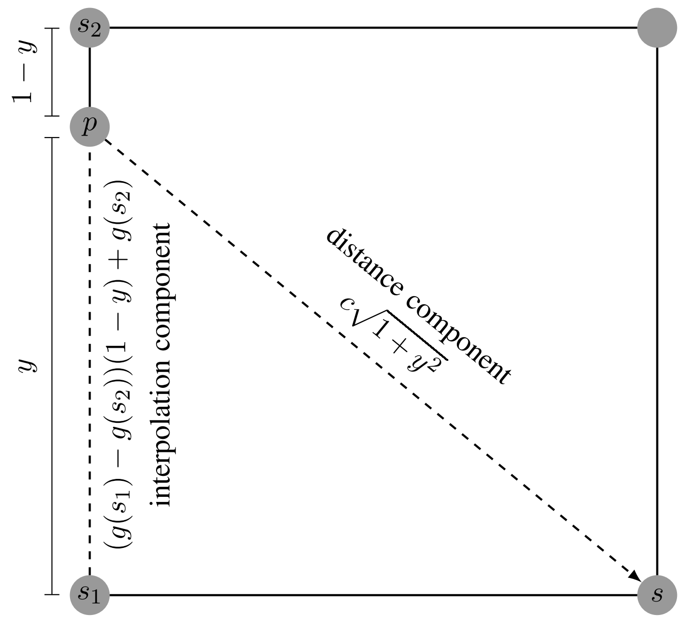


Figure 3.12: The *Direct* Field D* cost function is composed of an interpolation component and a distance component. If one were to only use the interpolation component to make estimates of the least cost path from edge $s_1 s_2$ to s, the estimate may be innacurate if the cell weight c is large relative to $g(s_1)$ and $g(s_2)$.

The problem with using Function 3.2 to estimate the point to derive cost from is that it only represents the interpolation component, and does not take into account the distance component. In Figure 3.12, for example, let p be the point that minimises Function 3.2 with respect to $g(s_1)$ and $g(s_2)$. If the weighting c of the distance from s to p, $c\sqrt{1+y^2}$ is small, relative to the path costs at $g(s_1)$ and $g(s_2)$, then this interpolation assumption may be reasonably accurate, since this distance component contributes a small portion of the total path cost.

However, as c increases, the distance component of the path cost increases. If this distance component is large relative to $g(s_1)$ and $g(s_2)$, it would actually be cheaper to travel from s_1 to s because the distance component is now such a dominant contributor to the path cost. This is where the interpolation assumption breaks down and may produce innacurate results.

In the case of 3D Field D*, this point is especially important to note, since the boundary minima are first estimated using the interpolation component. These estimated minima are used to make a further estimation of the cost function's minimum. While Field D* inherently contains interpolation error, only using the interpolation component and ignoring the weighted cell travel cost introduces futher error. When this use of interpolation is used for further estimation techniques, the resultant error may be compounded. Also, the use of scaling factors in 3D Field D* may also increase the importance of the distance component, if the scaling factors are significant.

The problem is that the described cost functions only apply to squares or cubes, but are required to calculate least cost paths across rectangles or rectangular cuboids. Rather than simply using the interpolation component to make an estimate, it would be more accurate to develop cost functions that operate on these objects.

3.6 Generalized Field D*

Sapronov and Lacaze present *Generalized Field D** [119], which extends the Field D* cost equations to arbitrary triangles. The authors define their triangles using point coordinates (s, s_1, s_2), interior triangle angles $(\theta_1, \theta_2, \theta_3)$, edge lengths (l_1, l_2, l_3), the weight of adjacent triangles (c_1, c_2, c_3) and triangle weight c_f. This configuration is shown in Figure 3.13.

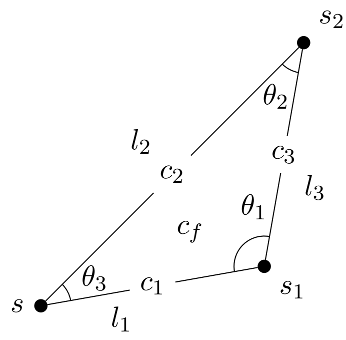


Figure 3.13: Configuration of a Generalized Field D* triangle. A triangle is characterised by points (s, s_1, s_2), interior angles $(\theta_1, \theta_2, \theta_3)$, edge lengths (l_1, l_2, l_3), weight of adjacent triangles (c_1, c_2, c_3) and interior triangle weight c_f.

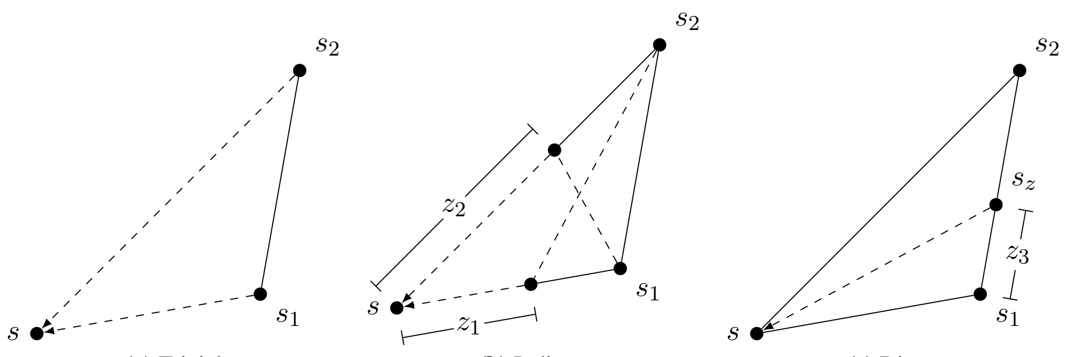


Figure 3.14: Generalized Field D*'s three cases: trivial, indirect and direct

Similarly to original Field D*, the authors break down the general case for a path across a triangle into the three sub-cases *Trivial*, *Indirect* and *Direct*. These cases are shown in Figure 3.14.

$$g(s) = \begin{cases} \min\limits_{z_1} \left[c_1 z_1 + c_f \sqrt{(l_1 - z_1)^2 + l_3^2 - 2(l_1 - z_1) l_3 \cos\theta_1} + g(s_z) \right] \\ \min\limits_{z_2} \left[c_2 z_2 + c_f \sqrt{(l_2 - z_2)^2 + l_3^2 - 2(l_2 - z_2) l_3 \cos\theta_2} + g(s_z) \right] \\ \min\limits_{z_3} \left[c_f \sqrt{l_1^2 + z_3^2 - 2 l_1 z_3 \cos\theta_1} + g(s_z) \right] \\ c_1 l_1 + g(s_1) \\ c_2 l_2 + g(s_2) \end{cases} \tag{3.14}$$

These cost functions have some similarity to the original Field D* cost functions, but now cater for arbitrary triangles. Thus, while distances in Field D*'s cost functions are expressed in terms of Pythagorus on a unit square, Generalized Field D* uses the *Law of Cosines* to express distances within the triangle. Thus, in the first case, $c_f \sqrt{(l_1 - z_1)^2 - 2(l_1 - z_1) l_3 \cos\theta_1} + g(s_2)$ represents the cost of travelling from s_2 to a point of distance z_1 from s (Figure 3.14b).

In the first two cases, $g(s_2)$ and $g(s_1)$ can be substituted for $g(s_z)$ respectively, since the paths for these cases start at these points (Figure 3.14b). In the third case, $(g(s_1) - g(s_2))(l_3 - z_3) + g(s_2)$ should be subsituted for $g(s_z)$ (Figure 3.14c). The authors present the minimisations for z_1, z_2 and z_3:

$$z_1 = (l_1 - l_3 \cos\theta_1) \pm \frac{c_1 l_3 \sin\theta_1}{\sqrt{c_1^2 - c_f^2}} \tag{3.15}$$

$$z_2 = (l_2 - l_3 \cos\theta_1) \pm \frac{c_2 l_3 \sin\theta_2}{\sqrt{c_2^2 - c_f^2}} \tag{3.16}$$

$$z_3 = l_1 \cos\theta_1 \pm \frac{l_1 sin\theta_1 \left(g(s_1) - g(s_2)\right)}{\sqrt{\left(g(s_1) - g(s_2)\right)^2 - c_f^2 l_3^2}} \tag{3.17}$$

We note some mistakes in these minimisations: In the square root under the denominator, the component containing c_f^2 is always negative compared to the other positive component. Thus, the first minimisation for example, would only return real values if $c_f < c_1$. However if $c_f < c_1$ then it would always be cheaper to simply travel straight through the triangle with weight c_f, rather than some of the way through the adjacent triangle of weight c_1 and the rest through the triangle weighted c_f. Also, the l_3 variable in the third case should not be in the denominator as it would be removed when the derivative is calculated. The corrected minimisations are:

$$z_1 = (l_1 - l_3 \cos\theta_1) \pm \frac{c_1 l_3 \sin\theta_1}{\sqrt{c_f^2 - c_1^2}} \tag{3.18}$$

$$z_2 = (l_2 - l_3 \cos\theta_1) \pm \frac{c_2 l_3 \sin\theta_2}{\sqrt{c_f^2 - c_2^2}} \tag{3.19}$$

$$z_3 = l_1 \cos\theta_1 \pm \frac{l_1 sin\theta_1 \left(g(s_1) - g(s_2)\right)}{\sqrt{c_f^2 - \left(g(s_1) - g(s_2)\right)^2}} \tag{3.20}$$

Unlike Field D*, it is not possible a priori to choose the case in Function 3.14 that will produce the cheapest path cost. Therefore, the minima and costs for each case must be evaluated and the cheapest case selected at the end of this. Also, we note that both of the roots produced by each minimisation need to be evaluated by the corresponding cost function to differentiate the point of inflection from the actual minimum.

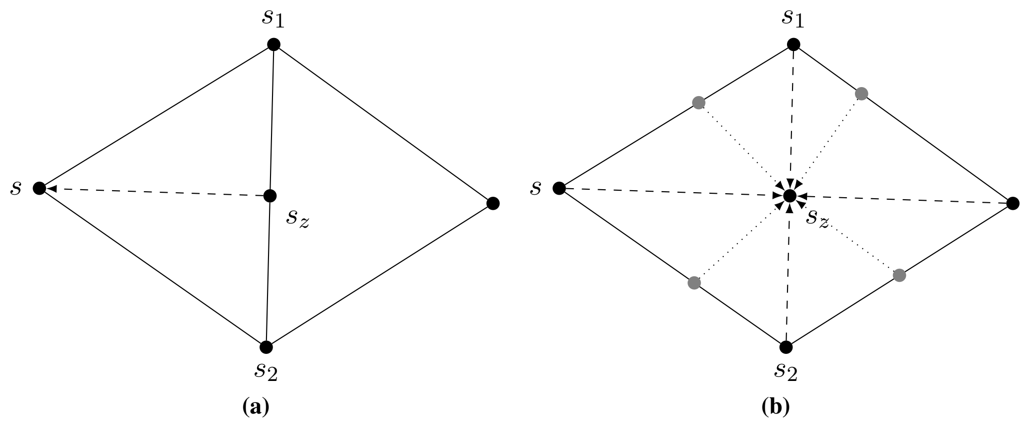


Figure 3.15: (a) Evaluation of the cost functions around s determines that the interpolated point on edge $s_1 s_2$ is s's predecessor. (b) To find s_z's predecessor, the two surrounding triangles are subdivided into four subtriangles, with s_z at the head of each triangle. *Trivial* and *Direct* cost functions are evaluated to find s_z's predecessor.

Generalized Field D*'s extension to triangles is useful during path extraction and solves the issues presented in Section 3.5. Consider Figure 3.15. In this diagram, evaluating cost functions at node s indicates that s_z is the cheapest predecessor point to s. s_z is an interpolated point on an edge and the algorithm must now select its predecessor point.

At a conceptual level, Field D* subdivides the two triangles adjacent to the edge containing s_z into four triangles, with s_z as their base node. Thus in Figure 3.15b for example, s is connected with s_z to subdivide ss_1s_2 into triangles ss_1s_z and ss_zs_2 and similarly for the other triangle.

Once this has been accomplished, the *Direct* and *Trivial* cost functions presented above are applied to these sub-triangles. Each function is evaluated and the one that produces the least cost path determines the predecessor point. Note that while s_z is an interpolated point, the other two points in the triangle, s and s_1 for example are actual graph nodes.

Thus, because (a) the Generalized Field D* cost functions operate on arbitrary triangles and (b) it is possible to subdivide the triangles around an interpolated point into smaller triangles, Generalized Field D*'s path extraction process is more accurate than that of Field D* since it uses both the linear interpolation and distance components of its cost function during path extraction, as opposed to Field D*, which can only use the linear interpolation component of its cost functions (Refer back to 3.12).

Sapronov and Lacaze perform experiments on a square grid divided into four regions. One of these regions is composed of cells with random weights, while the other three are composed of inexpensive, uniformly weighted cells. They use this basic grid to evaluate Field D* path costs from the random region to a uniform region and compare this to Generalized Field D*'s path costs on a triangle representation of the same environment. Their evaluation shows that Generalized Field D* can produce path costs within 1% of Field D* while using fewer triangles compared to grid cells to represent the uniform regions.

As an example of one of their specific cases, their Field D* representation consists of 625 nodes and 2304 edges while their Generalized Field D* representation consists of 252 nodes and 970 edges, yet the path costs produced by the two algorithms are within 1% of each other.

3.7 Field D* Heuristics

Field D* uses a relatively poor heuristic function to focus the search towards a goal. [49] suggests using:

$$h(x) = 0.5\alpha \|s - s_{\text{goal}}\| \tag{3.21}$$

where s is the current node, s_{goal} the goal node and α is the minimum weight in the entire environment. This is the heuristic that we have used in our implementation. It can be good if the range of weights exhibited within the environment is small, but most environments will not fit this criteria.

Very recent work [22] focuses on improving Field D*'s heuristic in static environments by using standard graph-based algorithms on the graph formed by the weighted edges of a triangulation. Since the distances are computed on a graph, these heuristics over-estimate, but the authors provides empirical evidence that the path cost is no more than 2% of the true path cost.

In particular, the authors uses a *differential heuristic* [133] to improve Field D*'s running time. A differential heuristic is part of a class of heuristics termed *true distance heuristics* [133] and is a database storing a subset of the all-pairs shortest path matrix of a graph. Nodes within this subset are termed *canonical* because they store the true distances to all other nodes in the environment, which can be retrieved in constant time.

This network of canonical nodes allows a pathfinding algorithm to find good heuristic values between any two nodes because canonical nodes can be used to *estimate* the distance between any two nodes in environment. For any two nodes, $\mathbf{v}$ and $\mathbf{u}$, and a canonical node $\mathbf{w}$, if $\mathbf{u}$ lies on the shortest path between $\mathbf{v}$ and $\mathbf{w}$, then $|\|\mathbf{v}-\mathbf{w}\| - \|\mathbf{w}-\mathbf{u}\||$ is the exact shortest distance between $\mathbf{v}$ and $\mathbf{u}$. The further $\mathbf{u}$ is from the shortest path, the less accurate this expression will be. Using a differential heuristic, the authors reduce the runtime of Field D* to between 60% to 80% of Field D* with a naïve heuristic.

The differential heuristic requires pre-processing of the graph in order to create the canonical nodes. We note that during this pre-processing, it should be possible to calculate a true heuristic by performing a Field D* search from canonical nodes, rather than a Dijkstra on graph edges.

Discarding the need for pre-processing, the authors also use an A* search between the start and goal nodes, using the propagated path costs as heuristic estimates. This approach leads to a reduction in runtime of between 65% and 75%.

3.8 Conclusion

In this chapter we described the algorithms that Field D* is based on, Field D* itself and the work that derives from Field D*. We also covered some of the issues associated with Field D*, most notably the use of the basic interpolation function in Field D*'s path extraction, one of Multi-resolution Field D*'s cases and 3D Field D*'s minimisation estimate.

Field D* was posed as a specific formulation of the Weighted Region Problem: finding the least cost path between two points on a weighted grid. This formulation is less general than the original: finding the least cost path between two points on a weighted planar polygonal subdivision. In the following chapter, we describe our extension of Field D* to triangulations. This provides the algorithm with the capability to solve the WRP on a structure conforming to the original specification of the problem.

Chapter 4

Extending Field D* to Weighted Triangulations

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 Weighted Region Problem (WRP), described in Section 2.3.1, poses the challenge of finding the shortest path between two points in a weighted planar polygonal subdivision. Field D* [49] is presented as an approximate solution to a specific formulation of the WRP: Finding the least cost path between two nodes in a weighted grid or quadtree [47].

Field D*'s solution to the WRP is approximate for two reasons. Firstly, it uses interpolation to approximate path costs along grid edges. Secondly, while grid representations are convenient, they are, by their nature, only capable of approximating simple polygons. Therefore, to increase the accuracy of Field D*'s solution to the WRP, high levels of grid subdivision are required for a weighted planar polygonal subdivision.

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* [47], which extends Field D* to quadtrees [118] to reduce the algorithm's computation time and space requirements and 3D Field D* [23], an approximate extension to 3D grids. Experimental evidence in [47] shows that Multi-resolution Field D* can improve performance over Field D* up to a factor of 1.8 times when the resolution of the underlying quadtree is 13% of that of the grid.

Partitioning a polygon requires an infinite number of grid cells if the polygon edges are not grid-aligned. By contrast, a simple polygon of n vertices can always be *exactly* partitioned into $n - 2$ triangles by the triangulation theorem [95]. Indeed, the initial solution to the WRP posed in [93] used triangles, without loss of generality. In this chapter, we extend the Field D* cost functions to triangles, thereby giving the algorithm the capability of solving the WRP on a triangulation. Consequently, a source of error is removed from Field D*'s solution to the WRP.

This extension also has important practical implications which should be emphasised. Representing an environment with a grid or quadtree is comparatively expensive in terms of storage, compared to a triangulation. In the field of *Geographic Information Systems* for example, the *Triangulated Irregular Network* (TIN) [107] is frequently chosen over image-based *Digital Elevation Models* (DEM) because fewer triangles are required to represent regional information, compared to the grid elements of a DEM. Consequently, less space or memory is required to accurately represent the terrain.

Similarly, triangular subdivision of an irregular object is more accurate than a subdivision with grid or quadtree cells, since triangles can represent the boundary of the object more accurately, as shown in Figure 4.1. 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 [41], especially in fields such as *Finite Element Methods* [122].

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 also reduced.

These practical savings in space and time further motivate the extension of Field D* to triangulations in this chapter. Our results show that a triangle implementation of Field D* is faster than a quadtree implementation of Field D*'s, requiring fewer elements to represent the environment when it is not grid-aligned.

This chapter is structured as follows. We present a brief overview of some standard path finding literature in Section 4.1. We then describe a general cost function in Section 4.2, conveniently expressed in vector mathematics. The characteristics of this function can be exploited to reduce the cost of finding a minimum and provides a basis for solving the Field D* cost functions on triangles. A description of the actual cost functions and their solution follows, as well as details on path extraction, the capability to cache the results of certain cost functions and a brief discussion of Field D*'s replanning ability.

Next, we present results. Section 4.6 presents results which detail a performance and space comparison on triangles between Generalized Field D* [119] and our implementation, showing a 50% improvement in our implementation. We then show that, in environments composed of non-grid aligned data, Multi-resolution quadtree 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. Finally, we show how the work for most of these functions can be precomputed and cached, producing a speedup of up to 16%.

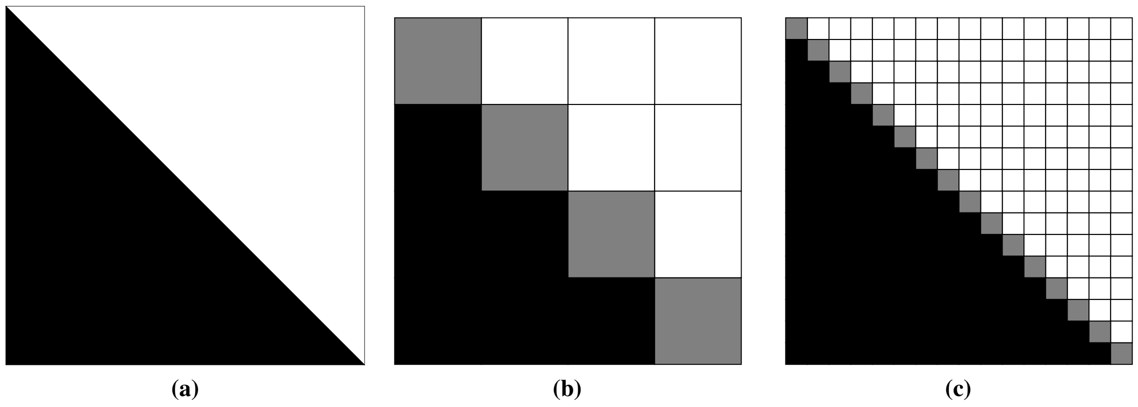


Figure 4.1: Grid subdivision of (a) a triangle at (b) and (c) different resolutions. As the triangle is not axis-aligned, high levels of subdivision is required for a grid to represent the triangle accurately.

4.1 Related Work

*Generalized Field D** [119] also 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 and general simplices 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 calculate and maintain.

Our triangle cost functions express the mathematics in vector notation, reducing computational and space requirements, and allows an easier extension to 3D tetrahedra. Minimising Generalized Field D*'s cost functions requires evaluating the cost of two local minima, whereas our implementation can decide which root to use without evaluating costs by inspecting the sign of a cost function term.

4.2 Cost Functions

In this section we describe a general cost function of one variable, and how to efficiently minimize this function. It serves as a basis for solving triangle cost functions, since each reduce to this general case. We show how to apply this minimization to find paths through an arbitrary triangle in section 4.2.2. Three cases are presented for triangles, *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.

4.2.1 General Cost Function

The functions described later in this work require minimisation to find the cheapest cost across a triangle. These problems can be reduced to solving a General Cost Function, whose solution and properties we will now describe. Let $\mathbf{v}_1, \mathbf{v}_2$ 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

$$G(x, \lambda, \mathbf{v}_1, \mathbf{v}_2, \mu, d) = \lambda\|\mathbf{v}_1 + x\mathbf{v}_2\| + \mu x + d \tag{4.1}$$

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

$$l(x) = \|\mathbf{v}_1 + x\mathbf{v}_2\|$$

and note that

$$\begin{aligned} l(x) &= ((\mathbf{v}_1 + x\mathbf{v}_2).(\mathbf{v}_1 + x\mathbf{v}_2))^{1/2} \\ &= \left(\|\mathbf{v}_1\|^2 + x^2\|\mathbf{v}_2\|^2 + 2x\mathbf{v}_1.\mathbf{v}_2\right)^{1/2}. \end{aligned}$$

For convenience, we let $a = \|\mathbf{v}_1\|^2$, $b = \|\mathbf{v}_2\|^2$, $c = \mathbf{v}_1.\mathbf{v}_2$ so that $l(x) = \left(bx^2 + 2cx + a\right)^{1/2}$.

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

$$\lambda(bx + c)/\left(bx^2 + 2cx + a\right)^{1/2} = -\mu \tag{4.2}$$

and squaring both sides yields the quadratic equation

$$b(\mu^2 - b\lambda^2)x^2 + 2c(\mu^2 - b\lambda^2)x + \mu^2 a - \lambda^2 c^2 = 0 \tag{4.3}$$

Note that in squaring, we may introduce extra solutions. In fact, in Equation 4.2, we necessarily have $(bx + c)\mu < 0$ because $\mu > 0$ and $\mathbf{v}_1$ and $\mathbf{v}_2$ are linearly independent. Assuming this, the solutions to functions (4.2) and (4.3) are identical. If $\mu^2 - b\lambda^2 = 0$ then (4.3) has a solution if and only if $ab = c^2$, i.e. $\|\mathbf{v}_1\|^2\|\mathbf{v}_2\|^2 = (\mathbf{v}_1.\mathbf{v}_2)^2$, which is impossible by the Cauchy-Schwartz inequality since $\mathbf{v}_1$ and $\mathbf{v}_2$ are linearly independent. If $\mu^2 - b\lambda^2 \neq 0$ then there are two solutions:

$$x = -\frac{c}{b} \pm \delta \tag{4.4}$$

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 (4.2) 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

$$\begin{aligned}
\frac{d^2G}{dx^2} &= \lambda\left(\frac{l(x)b-(bx+c)l'(x)}{l(x)^2}\right) \\
&= \lambda\left(\frac{l(x)b-(bx+c)^2 l(x)^{-1}}{l(x)^2}\right) \\
&= \lambda\left(\frac{l(x)^2 b-(bx+c)^2}{l(x)^3}\right) \\
&= \lambda\left(\frac{ab-c^2}{l(x)^3}\right) \\
&= \lambda\left(\frac{\|\mathbf{v}_1\|^2\|\mathbf{v}_2\|^2-(\mathbf{v}_1.\mathbf{v}_2)^2}{l(x)^3}\right) \\
&> 0
\end{aligned}$$

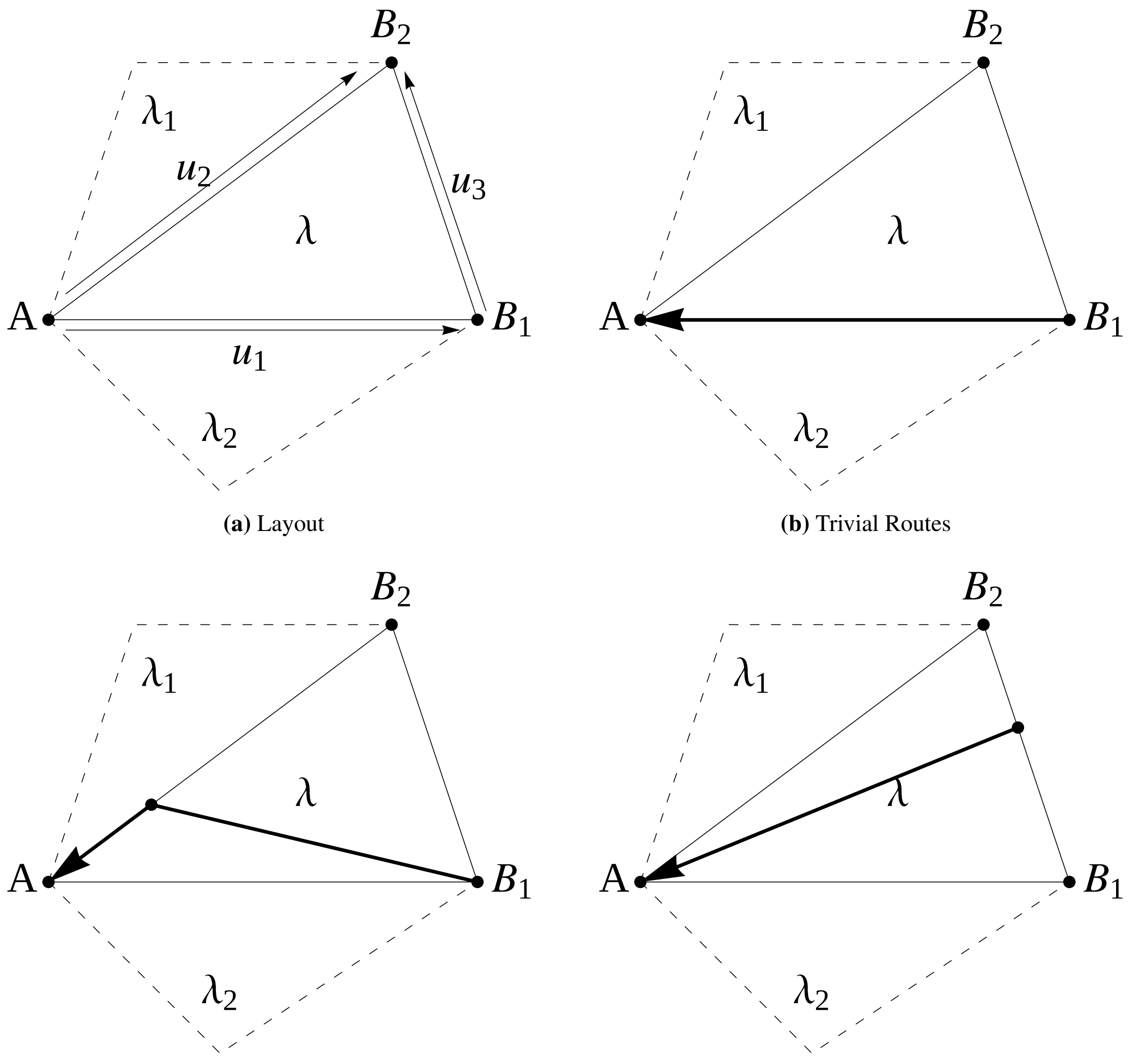
again by the Cauchy-Schwartz Inequality. The fact that the second derivative is positive everywhere implies that the first derivative is strictly increasing on the whole of $\mathbb{R}$. There are three possibilities: If $G'(x)$ has a root α then $G(x)$ has a global minimum at α; if $G'(x)$ is positive 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 $\alpha > 1$.

4.2.2 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 exposition is the same in both cases.

Figure 4.2a 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.



(a) Layout **(b)** Trivial Routes **(c)** Indirect Routes **(d)** Direct Routes

Figure 4.2: 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) (c) and (d) show the three types of path through a triangle

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 $\mathbf{w}, \mathbf{v}_1, \mathbf{v}_2$ respectively and let $x\mathbf{u}_1 = \mathbf{v}_1 - \mathbf{w}$, $x\mathbf{u}_2 = \mathbf{v}_2 - \mathbf{w}$ and $x\mathbf{u}_3 = \mathbf{v}_2 - \mathbf{v}_1$.

Trivial: Figure 4.2b 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

$$g(A) = \min\{\lambda, \lambda_2\}|\mathbf{u}_1| + g(B_1) \tag{4.5}$$

Indirect: Indirect paths originate at a node and cut across the main triangle to a point on the opposite

Algorithm 10 The *UpdateNode* function now iterates over the triangle neighbours of node u, represented by the set trinbrs(u) below.

```
function UPDATENODE($u$)
    if $u$ was not visited before then $g(u) = \infty$
    end if
    if $u \neq s_{goal}$ then
        $\text{rhs}(u) = \min_{s \in \text{trinbrs}(u)} \text{ComputeCost}(u, s)$
    end if
    if $u \in U$ then U.Remove($u$)
    end if
    if $g(u) \neq rhs(u)$ then U.Insert($u$, Key($u$))
    end if
end function
```

edge, and then travel along this edge to the destination node, as shown in Figure 4.2c. 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

$$g(A) = \lambda\|\mathbf{u}_1 - x\mathbf{u}_2\| + \lambda_1\|x\mathbf{u}_2\| + g(B_1) \tag{4.6}$$

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

$$g(A) = G(x, \lambda, \mathbf{u}_1, -\mathbf{u}_2, \lambda_1\|\mathbf{u}_2\|, g(B_1)).$$

Direct: Figure 4.2d 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_1B_2 must be estimated via interpolation. The cost function is formulated as:

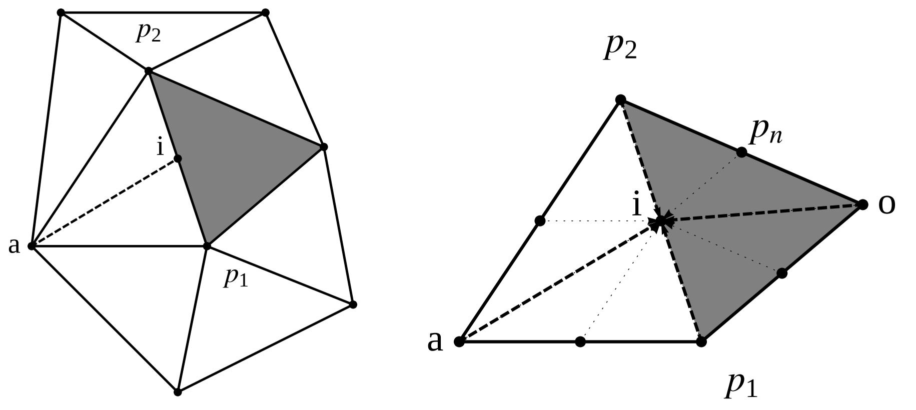
$$g(A) = \lambda\|\mathbf{u}_1 + x\mathbf{u}_3\| + xg(B_2) + (1 - x)g(B_1) \tag{4.7}$$

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

$$g(A) = G(x, \lambda, \mathbf{u}_1, \mathbf{u}_3, g(B_2) - g(B_1), g(B_1)).$$

Implementation Details:. The cases described above are evaluated separately and the case producing the least cost is returned by the *ComputeCost* function. Our implementation of *UpdateNode*, shown in Algorithm 10, differs slightly from the original Field D* in that, instead of iterating over the neighbouring edges, we iterate over the neighbouring triangles. We also find it convenient to store a back pointer to the triangle, instead of a node.

4.3 Path Extraction



(a) Incorrect Interpolation Assumption

(b) Lookahead

Figure 4.3: (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 subdiving 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.

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 11. It is slightly more compact and general than the path extraction pseudocode provided for Generalized Field D* [119].

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. [49] 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 Figure 4.3a. 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

Algorithm 11 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 Figure 4.3b.

```
function INTERPOLATEDCHILD($p$)
    Subdivide cells adjacent to $p$ into temporary cells
    $b_c \leftarrow \infty$; $b_p \leftarrow NULL$
    for all temporary cells $b$ do
        if ComputeCost$(p, b) < b_c$ then
            $b_c \leftarrow$ ComputeCost$(p, b))$
            $b_p \leftarrow$ point associated with cost $b_c$.
        end if
    end for
    Return $\{b_c, b_p\}$
end function
function EXTRACTPATH
    $s \leftarrow s_{start}$; PATH=$\{s_{start}\}$
    while $s \neq s_{goal}$ do
        if $s$ is an interpolated point then
            $\{c, p\} \leftarrow$ InterpolatedChild$(s)$
            $s \leftarrow p$; PATH=PATH$\cup\{p\}$
        else
            $a \leftarrow$ arg min$_{c \in \text{cellnbrs}(s)} ComputeCost(s, c)$
            $A = \{a1_c, \cdots, ak_c, d_c\} \leftarrow$ costs across $a$
            $d_c \leftarrow$ cost of the Direct Path through $a$.
            $d_p \leftarrow$ interpolated point associated with cost $d_c$.
            if $d_c =$ ComputeCost$(s, a)$ then                ▷ Direct Path is cheapest
                $\{b_c, b_p\} \leftarrow$ InterpolatedChild$(d_p)$
                Update $d_c \in A$ with $b_c$                    ▷ Check the estimate
            end if
            $c \leftarrow$ min$(A)$
            $s \leftarrow$ point(s) associated with $c$
            PATH=PATH$\cup\{s\}$
        end if
    end while
end function
```

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 Figure 4.3b. Both *Trivial* cost functions originating from nodes and *Direct* cost functions originating from edges [1] are evaluated and the cheapest of these costs replaces the interpolated cost.

[1]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.

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 Figure 4.3b: 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, 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* [49] or 3D Field D* [23] whether these cost functions are employed during path extraction. In fact, [49] suggests using a local planner to perform path extraction instead.

4.4 Caching

In this section, we describe how the pathfinding 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 in terms of $G(x)$. A characteristic of $G(x)$ is that parameter d is not utilised in finding the roots in Equation 4.4. Now, as long as parameters λ, $\mathbf{v}_1$, $\mathbf{v}_2$ 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 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, $\mathbf{v}_1$, $\mathbf{v}_2$ 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 only have $g(p)$ values in parameter d. Thus, their roots and the sections of $G(x)$ composed from λ, $\mathbf{v}_1$, $\mathbf{v}_2$ 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.

To obtain performance gains from caching, the mesh and the triangle 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.

4.5 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 change, *UpdateCost* can be invoked on the nodes of these structures. Our extension to Field D*'s cost functions does not modify its 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.

4.6 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: $0.5 * minval * \sqrt{dx^2 + dy^2}$.

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

	Vector Field D*	Generalized Field D*	Generalized Field D* *(Cached)*
Time	13.12s	20.53s	14.83s
10^6 Triangles per second	7.621	4.871	6.743
Space	28 bytes	28 bytes	64 bytes

Table 4.1: Comparison of the time and space required by our triangle Field D* cost function implementation vs Generalized Field D*.

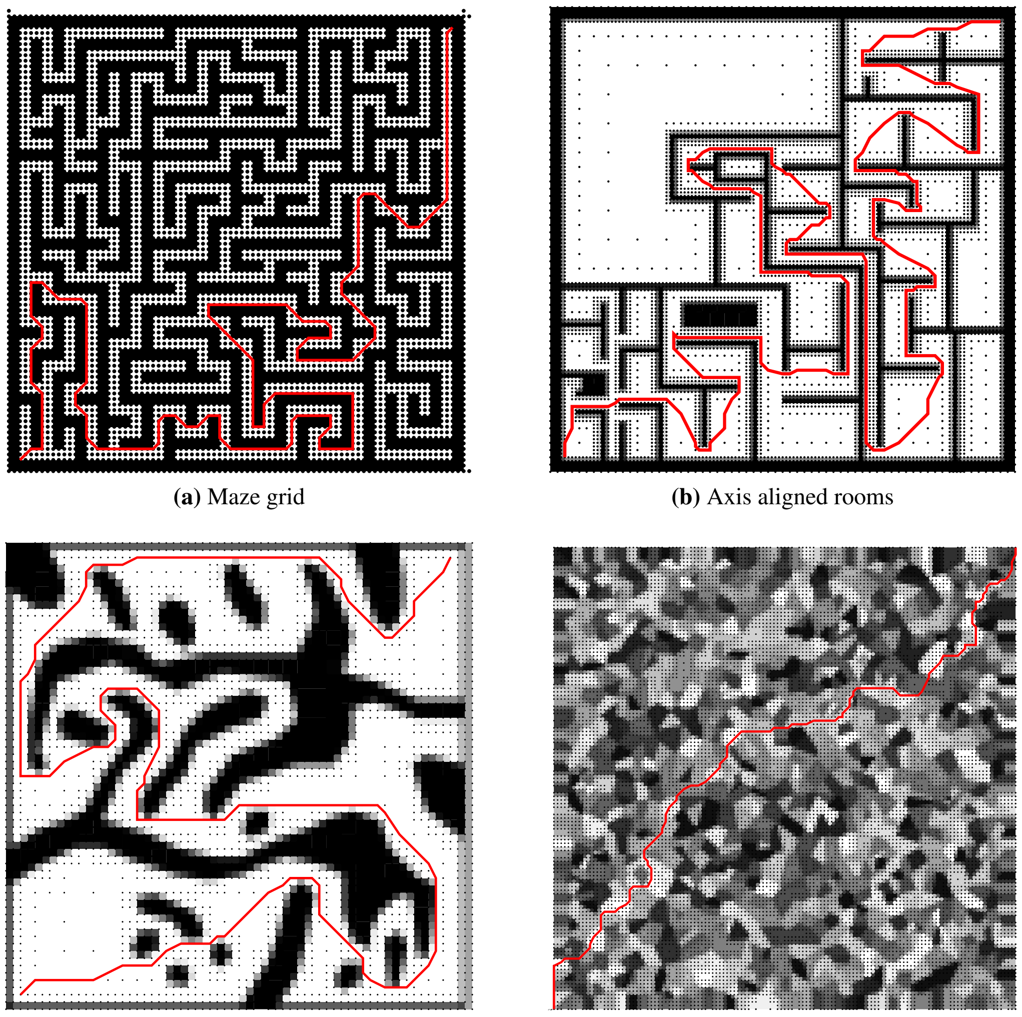
Generalized Field D* [119] evaluates the Field D* cost functions on a triangle using the inner angles and side lengths of that triangle. In contrast, our implementation of the Field D* cost functions for triangles uses vector operations on the triangle points. Thus, Generalized Field D* must either calculate 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 Section 4.2.1 allows our implementation to predict which root to use, meaning that only the cost for one root must evaluated. Based on this reasoning, we expect that our vector 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 4.1 shows these results.

Our vector-based implementation takes 13 seconds to evaluate the cost functions of a million triangles 100 times, requiring 28 bytes for the representation (six four bytes floats for the coordinates and one for the triangle weight). By comparison our implementation using Generalized Field D* cost functions takes 20.5 seconds to evaluate the cost functions, as the side lengths and trigonomentric values must be calculated when evaluating a triangle's costs. Caching these values (three sines, three cosines and three edge lengths) results in a execution time of 15 seconds, 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-based implementation of the triangle cost functions is around 56% 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.

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



(a) Maze grid

(b) Axis aligned rooms

(c) Arbitrarily aligned structures

(d) Randomly weighted Voronoi diagram

Figure 4.4: Quadtree environments used to compare Quadtree Field D* and 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) is a randomly weighted Voronoi Diagram. Darker regions are weighted more heavily, while the lighter regions have lesser weightings.

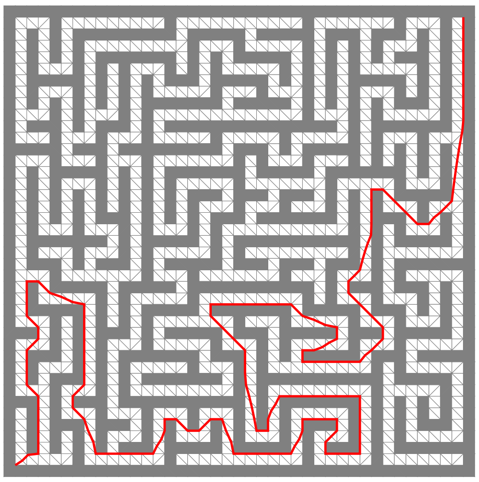
We have extended Field D* to triangulations since triangulations represent general polygonal objects

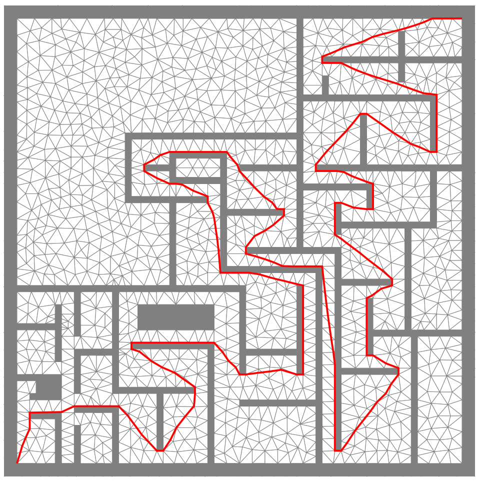
Path Cost		Faces		Node Expansions		Path Length		Time (s)		Normalised L_2 Error
Q	T	Q	T	Q	T	Q	T	Q	T	Q
Grid Maze										
16.69	17.03	1682	3362	977	964	165.87	166.31	0.02	0.01	N/A
16.65	16.93	6725	12984	2929	2682	165.79	166.68	0.06	0.01	N/A
16.63	16.90	14885	13858	5840	2943	165.76	166.56	0.11	0.02	N/A
16.62	16.83	26897	25866	9685	5492	165.74	166.49	0.16	0.04	N/A
16.61	16.78	41617	41316	14464	7710	165.73	166.17	0.25	0.05	N/A
16.61	16.75	60517	61006	20155	10919	165.72	166.13	0.35	0.07	N/A
16.60	16.74	81797	82366	26762	13726	165.72	166.17	0.48	0.10	N/A
Randomly Weighted Voronoi Diagram										
12537.23		1025		1089		167.22		0.02		0.210
10480.93		4076		4220		181.54		0.06		0.141
9421.86	8439.44	14405	16789	15673	9511	182.33	184.31	0.24	0.05	0.079
8938.80	8414.64	40046	41865	47669	21727	184.04	182.82	0.76	0.12	0.041
8710.31	8404.57	95780	92344	120254	47238	183.76	182.77	2.03	0.26	0.021
8570.52	8390.41	211697	219242	273470	111659	182.55	182.42	4.87	0.65	0.011
8490.43	8386.65	447932	461450	587532	234162	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	2564	2586	2218	874	282.21	365.74	0.04	0.01	0.356
39.50	37.03	6722	6753	3922	1942	392.10	365.02	0.09	0.02	0.189
37.98	36.87	15389	15366	9417	4396	377.06	364.56	0.21	0.03	0.081
37.47	36.74	32738	32479	20884	8480	371.96	364.13	0.57	0.06	0.044
37.16	36.63	67412	68643	46424	17443	368.79	363.77	1.29	0.11	0.027
37.00	36.55	136694	135177	104728	33471	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	2588	2722	1848	1056	422.03	387.02	0.04	0.01	0.176
40.76	39.18	6239	6315	4360	2399	404.82	386.15	0.09	0.02	0.088
39.80	38.97	13535	13340	9959	4910	395.20	385.38	0.19	0.03	0.045
39.21	38.84	28178	28620	21904	10258	389.52	385.11	0.39	0.06	0.018
38.94	38.73	57479	57857	49487	20120	387.09	384.74	0.87	0.13	0.010
38.82	38.66	116177	115644	112041	39652	386.13	384.67	2.04	0.26	0.006

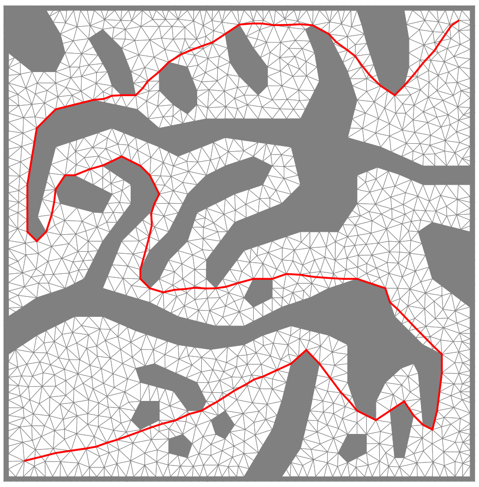
Table 4.2: The path cost, number of faces, number of node expansions, path lengths, time taken to find a path and Normalised L_2 Error for Field D* implemented on a quadtree (Q) versus a triangulation (T). The normalised L_2 Error measures the geometric error in the quadtree representation. Where possible, each row presents data for a similar number of quadtree 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 highlighted quadtree 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.

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	16789	9760	9511	0.0029	0.05	9620.49	16384	16640	0.0059
8655.48	8414.64	41865	20875	21727	0.0068	0.12	9164.06	65536	66048	0.0287
8589.95	8404.57	92344	50007	47238	0.0223	0.26	8924.06	262144	263138	0.1279
8563.51	8390.41	219242	114587	111659	0.0593	0.65	8800.96	1048576	1050624	0.6090
8550.83	8386.65	461450	231625	234162	0.1385	1.49	8744.71	4194304	4198400	2.9960
Axis-Aligned World										
38.22	37.31	2586	900	874	0.0003	0.01	369.32	4096	3334	0.0011
38.00	37.03	6753	1942	1942	0.0006	0.02	40.06	16384	7620	0.0028
37.94	36.87	15366	4318	4396	0.0015	0.03	38.77	65536	30633	0.0133
37.73	36.74	32479	8251	8480	0.0030	0.06	38.29	262144	121843	0.0580
37.62	36.63	68643	17052	17443	0.0074	0.11	38.01	1048576	487144	0.2630
37.47	36.55	135177	32521	33471	0.0152	0.22	37.9	4194304	1947273	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 4.3: Comparison of path cost, node expansions and time taken between A* on a triangulation (TA*) and Field D* (TFD*). 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 (16789 vs 461450 faces) in a faster time (0.05 vs 0.1385 seconds). 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

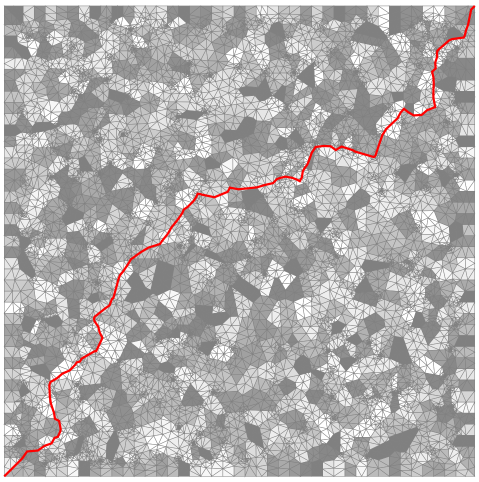
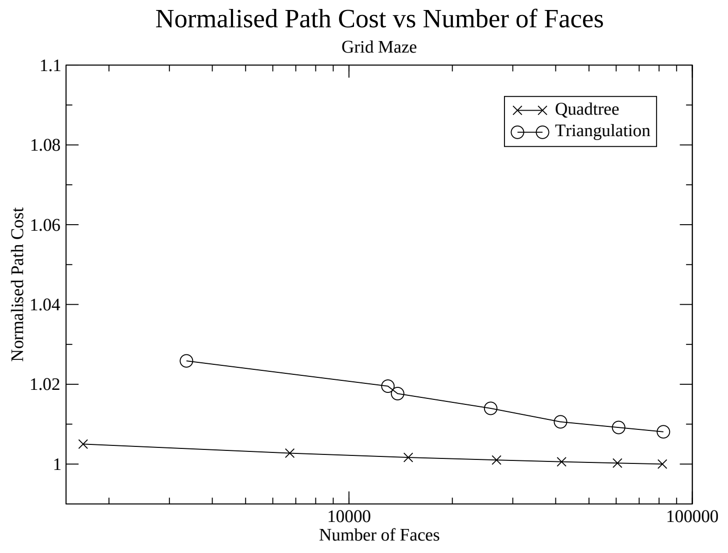
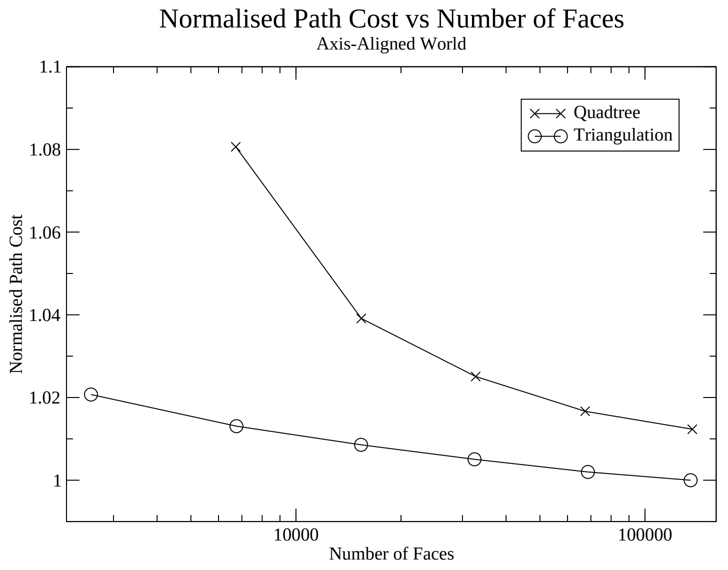

(d) Randomly weighted Voronoi diagram

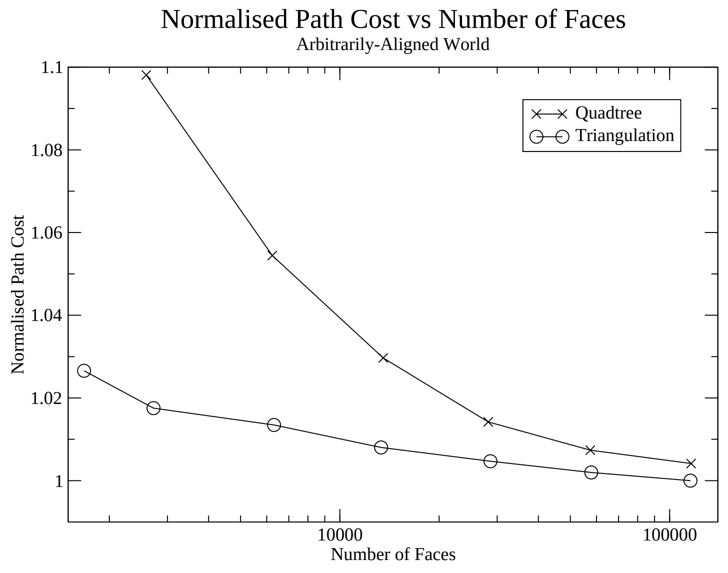
Figure 4.5: Triangulated environments used to compare Quadtree Field D* and Triangulated Field D*. (a) is a grid-aligned maze and is designed to contrast the two implementations to a case where no geometric error is present. (b) is a connected series of axis-aligned rooms, while (c) consists of arbitrarily aligned structures. (d) is a randomly weighted Voronoi Diagram. Darker regions are weighted more heavily, while the lighter regions have lesser weightings.

more accurately than grids and quadtrees. This is because triangles can represent polygonal objects exactly, since the interior of a polygonal object can always be subdivided into triangles. Grids or quadtrees, however, will always be subject to geometric error, unless that object's boundaries are grid-aligned. This implies that a grid or quadtree 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 [119], 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

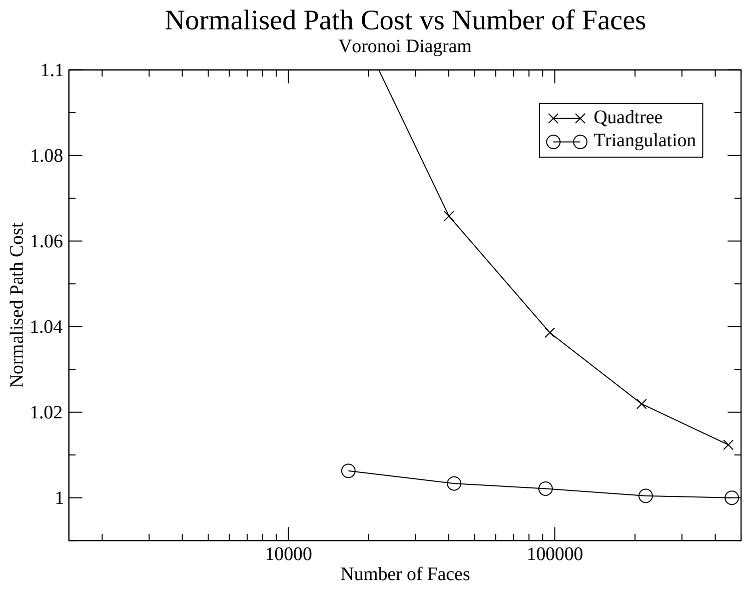


(a) Grid Maze



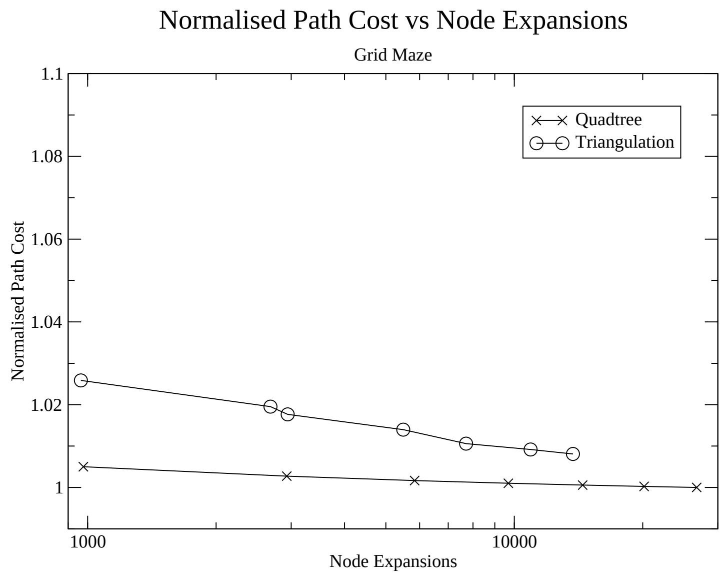


(c) Arbitrarily-Aligned World

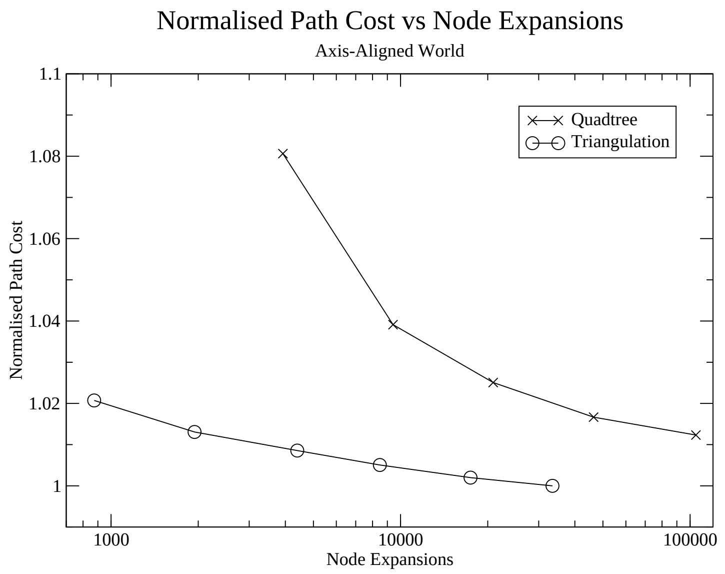


(d) Voronoi Diagram

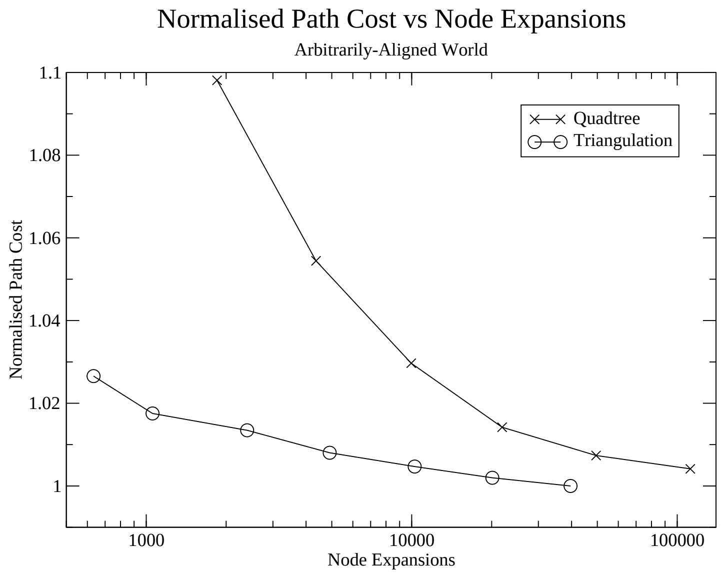
Figure 4.6: Normalised path cost vs number of faces



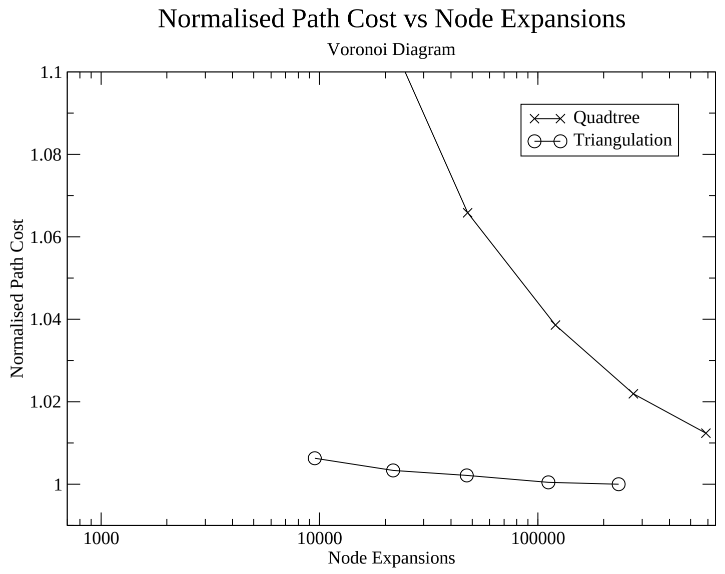
(a) Grid Maze



(b) Axis-Aligned World



(c) Arbitrarily-Aligned World



(d) Voronoi Diagram

Figure 4.7: Normalised path cost vs node expansions

[47] 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 quadtrees and triangulations at different levels of subdivision and demonstrate that, due to geometric error, a quadtree 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 chapter, and the other based on the quadtree cost functions described by Ferguson et. al [47].

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* [123] 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.

Quadtree Construction: Quadtrees [118] are restricted to representing polygonal data with squares or *cells*. To construct a quadtree, 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 quadtree 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.

$$\bar{w} = \frac{1}{a_c} \sum_{1}^{n} a_k w_k \tag{4.8}$$

$$L_2 = \sqrt{\sum_{1}^{n} \left[a_k \left(\bar{w} - w_k\right)\right]^2} \tag{4.9}$$

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

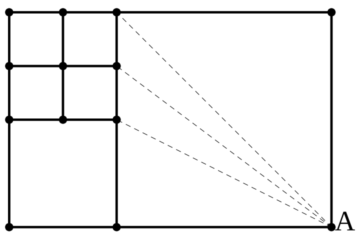


Figure 4.8: Quadtree subdivision: When calculating the cost of node A, the costs of travelling from four neighbouring edges must be considered, since this cell has high resolution neighbours.

Test Environments: We constructed four environments, shown in Figure 4.4 and Figure 4.5, to contrast the paths produced by quadtree 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 quadtree and triangulation. Results for these paths are shown in Table 4.2, which details the path cost, number of faces, number of node expansions, path costs and L_2 error. In this table, quadtree faces are square cells, whereas triangulation faces are triangles. However, it should be noted as in Section 4.1 that basic Field D* treats a cell as two triangles when computing cost functions since it must find calculate the shortest path from two edges. Quadtree Field D* may consider even more edges, if the cell has a number of higher resolution neighbours, as shown in Figure 4.8. Since this "decomposition" occurs during the runtime evaluation of cost functions it is difficult to directly compare quadtree-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 Quadtree 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 4.3.

The first environment is a grid maze (Figure 4.5a) and we use it to show how quadtree and triangulation implementations compare when data is grid-aligned and no geometric error is present. Since the data is aligned to a grid, a quadtree 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 quadtree cell. This discrepancy in representation is quantified by the L_2 error metric we mentioned previously.

The second environment (Figure 4.5b) is a series of interconnected, axis-aligned rooms. The third

(Figure 4.5c) consists of arbitrarily aligned structures, while the fourth (Figure 4.5d) 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 Figure 4.6, and the normalised path cost and the number of node expansions required by the algorithm in Figure 4.7. We normalise the path costs for a particular environment by dividing path costs by the minimum path cost.

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 Figure 4.5a 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, 2586 triangles produce a path cost of 37.31, while 12.7 times (32738) more quadtree faces are required to produce a slightly higher path cost of 37.47. In the arbitrarily-aligned world, 1676 triangles produce a path cost of 39.68, while 8 times (13535) 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 Figure 4.6.

Since the Delaunay triangulation requires a minimum of around 16500 triangles to represent the Voronoi diagram, it was not possible to compare path costs at 1024 and 4076 quadtree faces respectively. The path cost of 8439.44 for 16789 triangles beats the quadtree path costs at all levels of subdivision so there are no comparable data points, but Figure 4.6d 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 4.2, are not used as graph data points due to issues of scale. Note how the quadtree first underestimates path length, then reaches a point where it overestimates the path length before tending once again to lower path lengths. This suggests that a certain level of quadtree subdivision is required before pathing through high cost regions is avoided, around 40000 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 algorithm 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 Figure 4.8). 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 Field D* on a triangulation is between seven and ten 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 [49], that are not applicable to the quadtree'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 microseconds is required for a quadtree node expansion as compared to about 6 microseconds for a node expansion on the triangulation.

As these differences in structure and implementation exist, it is useful to refer to the worst-case time complexity when performing comparisons. As explained in Section 2.3.3, Field D* exhibits a worst-case time complexity of $O(F+V \log V)$ and requires $O(F+V)$ to represent the environment. In order to increase path accuracy, the geometric error present in the quadtree representations must be reduced

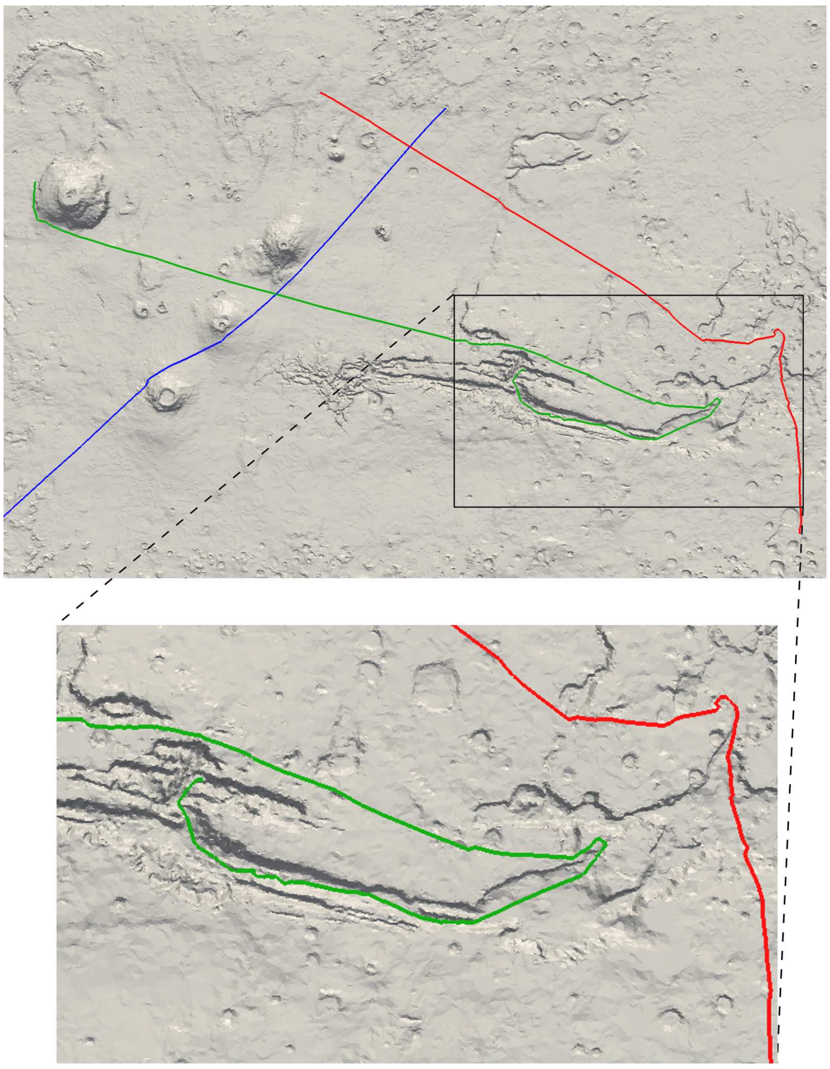

(a) Tharsis Plateau and Valles Marineris

Figure 4.9: 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*.

by increasing the environment subdivision. To reduce it to the point where it no longer significantly effects path costs requiress increasing the F and V factors by an order of magnitude. These increases in space directly increase the time complexity of Multi-resolution Field D*, compared to our triangle implementation of Field D*. Additionally, since the valence of the triangle implementation is lower than the quadtrees, the branching factor of the algorithm is lower, which reduces the number of nodes placed on the priority queue.

Therefore, 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 the quadtree representation decreases, the differences in accuracy also 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 Figure 4.6 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 4.3, 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 microseconds, 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 Figure 4.9) how paths can be plotted across the surface of Mars. In this figure, triangles have been weighted according to their steepeness, 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.

4.6.3 Timings

Number of Elements	Normal Time	Cached Time	% Speedup
Triangulation			
52600	0.18s	0.15s	16.6%
80700	0.28s	0.24s	14.2%
102000	0.36s	0.32s	11.1%

Table 4.4: Algorithm run-times for non-cached and cached cases.

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. To accomplish this, we constructed a random Delaunay Triangulation within a square. Half of the triangles were weighted with 0.1 (open space), while the other half were weighted with a random multiple of 16 between 16 and 256.

We generated 100 random triangle environments and measured the time it took for the algorithm to find a path from one corner of the square to the opposite corner. For each case, we measured the time for the algorithm to complete with caching turned both on and off. Table 4.4 shows the average of these times for both the normal and cached cases and for a varying number of elements.

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. Therefore, to cache triangle functions, approximately 2.71 times more space is required per triangle to produce an improvement in running time of between 11% and 16%.

4.7 Conclusion

This chapter describes an extension of Field D*'s cost functions to triangles. We provided analytic solutions for the minima of these functions expressed using vectors. Experimental results show a 56% 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. Expressing the cost functions using vectors also allows us to provide a more general extension to higher dimension, presented in Chapter 5.

These functions allow Field D* to operate on triangle meshes, thereby providing the algorithm with the ability to solve the Weighted Region Problem on representations free from geometric error. This has practical benefits, since triangles can always decompose a polygon exactly, compared to grid squares which, in general, require an infinite level of subdivision to achieve the same. Thus, the space required by the algorithm, $O(F + V)$ to obtain a solution free from geometric error is significantly reduced. As the worst-case time complexity of the algorithm is $O(F + V \log V)$, reducing the number of faces F and nodes V also running time of the algorithm.

In this chapter, we have demonstrated this experimentally: For non grid-aligned data, a quadtree requires an order of magnitude more faces compared to a low resolution triangulation in order for Field D* to find a path of similar cost. As fewer faces are used to represent the environment and because triangulation nodes have fewer neighbours compared to a multi-resolution grid, Field D* operating on a triangulation has to expand between 10 and 20 times fewer nodes when calculating a shortest path. While the computational expense of triangle cost functions on triangles may be greater than those of a grid cell, the reduction in time complexity of the algorithm dwarfs this expense. In our Voronoi diagram example for instance, 0.05 seconds is required to find a shortest path in 16789 triangles, compared to 10.82 seconds in 461450 quadtree cells.

We have also analysed the triangle cost functions for values that can be pre-calculated and cached. This can produce up to a modest 16.6% improvement in the algorithm's running time, at the cost of using 2.77 times more space.

Additionally, since the triangle cost functions can be applied to weighted triangles embedded in 3D, they can also be applied to 3D triangulated surfaces and not just triangular subdivisions of a 2D plane.

Chapter 5

Extending Field D* to N-Dimensions

The Field D* algorithm finds the least cost path between two nodes on a weighted grid. As such, it is an approximate solution to the Weighted Region Problem (WRP) which poses the challenge of finding the least cost path between two points in a weighted planar polygonal subdivision [93]. In the previous chapter we extended Field D* to triangles, allowing the algorithm to operate on structures that subdivide polygons exactly, and thereby removing a significant source of error in Field D*'s solution to the WRP. This extension also offers benefits in terms of computational and space complexity for the algorithm.

The WRP was originally specified in terms of a weighted planar polygonal subdivision, or, a *weighted polygonal mesh*. However, the WRP can be solved without loss of generality by conversion to a weighted triangle subdivision [93]. More generally, the WRP can be posed in terms of a weighted polytope subdivision in higher dimensions, but solved without loss of generality by decomposition to a weighted simplicial subdivision [67]. Then, we can define the WRP more generally as finding the least cost path between two points in a *simplicial complex*.

In the previous chapter, we described an extension of the Field D* cost functions to triangles, or 2D simplices, allowing the algorithm to operate on 2D simplicial complexes. Here, we describe the extension of Field D*'s cost functions to arbitrary simplices, thereby allowing the algorithm to operate on general simplicial complexes, and providing an approximate solution to the WRP in higher dimensions.

We start by discussing quadratic functions and polynomials, as well as convex functions and nonlinear optimisation. In particular quadratic functions represent squared distances which arise in cost functions developed in this chapter, while quadratic polynomials represent the contours, or, iso-surfaces of these functions. Convex functions, which include quadratic functions, have desirable properties and, in particular, are guaranteed to have an optimum. Consequently, many techniques, some analytical, but also numeric nonlinear optimisation and quadratic programming have been developed to find these optima.

3D Field D*'s solution to the WRP is discussed, noting that the underlying representation of a 3D grid that it operates on will always introduce geometric error into its solution of the WRP.

We proceed to introduce some notation defining these simplices and prove how shortest paths within a simplex separate into either the *direct* or *indirect* cases. The general cost function, defined in Chapter 4, is then extended to arbitrary dimensions using linear algebra, and an analytic minimisation of this function is presented. This minimum must be solved subject to optimisation constraints defined by the boundary of the simplex, and we present an efficient method for constraining the minimum to these boundaries.

This function does not express the higher dimensional indirect case as effectively as possible since it does not express the distance between two *vector subspaces*. To solve this, we present an extended version of the general cost function. A full analytic solution for the extended version is not obtainable, but it can be reduced to the general cost function in certain cases.

We proceed to show how the direct and indirect cases in higher dimensions can be solved using the general cost function. Finally, we present results for pathfinding through 3D environments, demonstrating how the 3D version of Field D* can be used to find paths through a fluid simulation and through blood vessels in 3D medical data. We also present experimental evidence suggesting that the indirect cases contribute minimally to the final path cost in 3D.

5.1 Quadratic Functions and nonlinear optimisation

In the following section, we discuss quadratic functions and some of their properties. In particular, they are *convex functions* and therefore have an optimum. They are relevent to our work since they can express the squared distance between a parameterised point in a vector subspace and another point that is not in the vector subspace. These distances are components of the cost functions that we will need to solve. We also discuss nonlinear optimisation for quadratic programming, which involves finding optima to quadratic formulae, subject to linear equality and inequality constraints.

5.1.1 Quadratic Polynomials and Functions

A *quadratic polynomial* is a polynomial composed of variables whose exponent is no greater than two. For scalar variables, the quadratic polynomial can be expressed as:

$$ax^2 + bx + c \tag{5.1}$$

where a,b and c are scalar constants and x is a scalar variable. *Quadratic functions* evaluate the polynomial for a particular value of x when $a \neq 0$:

$$f(x) = ax^2 + bx + c$$

Quadratic functions have well-known properties:

- If $a > 0$, then the graph of $f(x)$ forms a downward parabola, with a minimum at $\frac{-b}{2a}$.
- If $a < 0$, then the graph of $f(x)$ forms a upward parabola, with a maximum at $\frac{-b}{2a}$.

Quadratic functions can be generalised in higher dimensions to *multivariate quadratic functions* of the form:

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c \tag{5.2}$$

where $\mathbf{A}$ is a square matrix of dimension $n \times n$, $\mathbf{b}$ and $\mathbf{x}$ are row vectors of dimension n and c is a scalar. Distance functions in higher dimensions can be expressed in terms of a quadratic since, for some $n \times m$ matrix $\mathbf{M}$ with $m < n$ expressing a linearly independent basis of m vectors , and vectors $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$, with sizes m and n respectively, we can derive:

$$\begin{aligned}
||\mathbf{M}\mathbf{x} + \mathbf{v}|| &= \left((\mathbf{M}\mathbf{x} + \mathbf{v})^T (\mathbf{M}\mathbf{x} + \mathbf{v})\right)^{\frac{1}{2}} \\
&= \left(\left(\mathbf{x}^T\mathbf{M}^T + \mathbf{v}^T\right)(\mathbf{M}\mathbf{x} + \mathbf{v})\right)^{\frac{1}{2}} \\
&= \left(\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x} + \mathbf{v}^T\mathbf{M}\mathbf{x} + \mathbf{x}^T\mathbf{M}^T\mathbf{v} + \mathbf{v}^T\mathbf{v}\right)^{\frac{1}{2}} \\
&= \left(\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x} + 2\mathbf{v}^T\mathbf{M}\mathbf{x} + \mathbf{v}^T\mathbf{v}\right)^{\frac{1}{2}}
\end{aligned}$$

Setting $\mathbf{A} = \mathbf{M}^T\mathbf{M}$, an $m \times m$ square matrix, $\mathbf{b}^T = 2\mathbf{v}^T\mathbf{M}$, a vector of size m and $c = \mathbf{v}^T\mathbf{v}$, a scalar, it can be seen that the term in the square root is a multivariate quadratic function:

$$\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x} + 2\mathbf{v}^T\mathbf{M}\mathbf{x} + \mathbf{v}^T\mathbf{v} = \mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c$$

Given that $m < n$, $\mathbf{M}$ has a left inverse, $(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T$, and the analytic optimum to the quadratic above is well known [20]:

$$x_{\text{opt}} = \left(\mathbf{M}^T\mathbf{M}\right)^{-1}\mathbf{M}^T\mathbf{v}$$

Also, as long as $\mathbf{v}$ is non-zero, $||\mathbf{M}\mathbf{x} + \mathbf{v}||$ is differentiable everywhere and has the same minimum as the quadratic.

5.1.2 Quadric Surfaces

Quadrics are quadratic polynomial equations of the following form:

$$\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c = 0$$

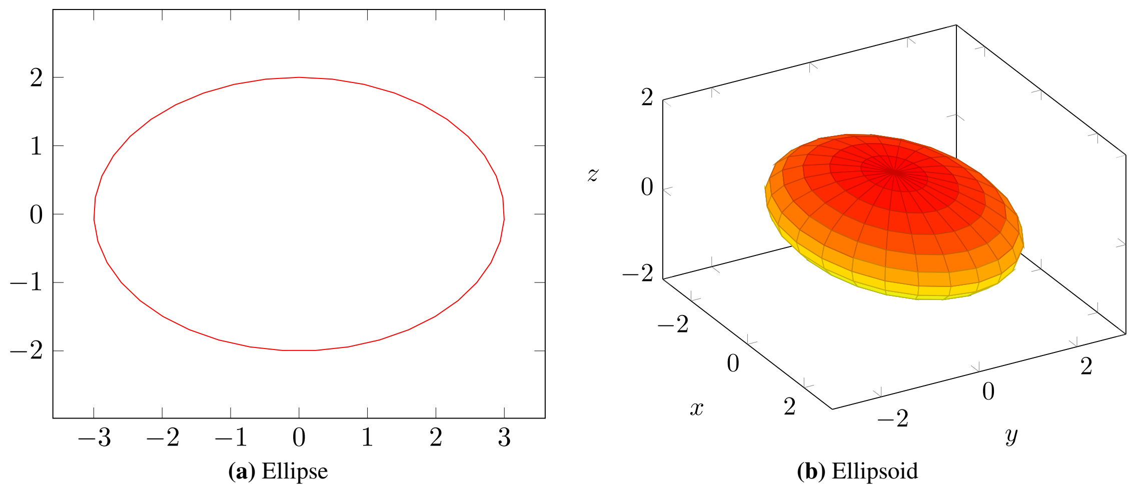


Figure 5.1: Quadric Surfaces in 2D and 3D (a) an Ellipse (b) an Ellipsoid

Solving this equation for $\mathbf{x}$ produces a family of points that collectively define a *quadric surface*, such as the ellipes in Figure 5.1a and ellipsoid in Figure 5.1b. Quadrics may be classified into 17 surfaces such as ellipsoids, elliptic parabaloids and hyperboloids. In particular if the quadratic form $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0 \forall x$ then A is positive definite, the quadric is an ellipsoid and can be expressed as:

$$(\mathbf{x} - \mathbf{v})^T \mathbf{A} (\mathbf{x} - \mathbf{v}) = 1$$

where $\mathbf{v}$ is the centre of the ellipsoid. Ellipsoids are relevent to this work as they describe cost function contours.

5.1.3 Convex Functions

Convex functions [20] are a group of functions classified by the relation between any two points on the surface of the function's graph. If the line between these two points itself never crosses the graph surface, the function is said to be convex. Figure 5.2a illustrates the convexity of the quadratic function while 5.2b illustrates a non-convex cubic function. Expressed algebraically, a function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if the following holds:

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$$
$$\forall x, y \in \mathbb{R}^n \text{ and } \forall \alpha, \beta \in \mathbb{R} \text{ where}$$
$$\alpha + \beta = 1, \alpha \geq 0, \beta \geq 0$$

It is useful to identify whether functions are convex, since convexity implies that a minimum will always exist, and if the function is *strictly convex*, it will be unique. In particular, quadratic functions are convex [20] and have analyic solutions. Other, more general convex functions may need to be

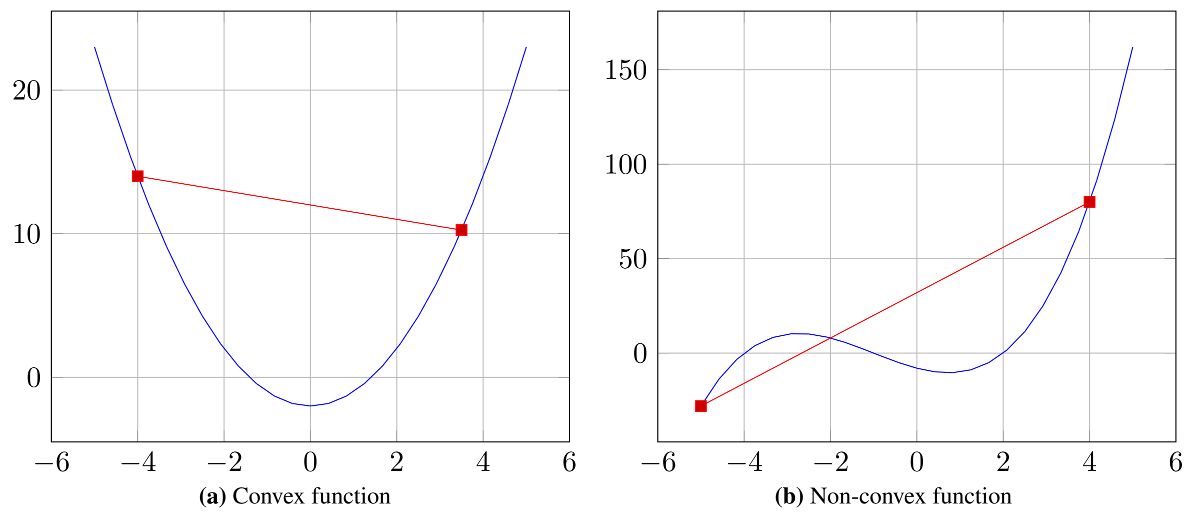


Figure 5.2: (a) x^2-2 is convex, since connecting two points do not cross the graph surface. (b) x^3+3x^2-6x-8 is non-convex since connecting two points does cross the graph.

solved using numeric techniques such as *Nonlinear optimisation*,

5.1.4 Nonlinear Programming and Optimisation

The mathematical field of Nonlinear Programming (NLP) [12] encompasses techniques for finding the optimum of an objective function, subject to a system of equalities and inequalities, termed *constraints*. The objective function and constraints may be nonlinear. More formally the requirement is to:

$$\begin{aligned}
&\text{Minimise} f(\mathbf{x}) \text{ where}\\
&f : \mathbb{R}^n \to \mathbb{R}\\
&\mathbf{x} \in \mathbb{R}^n\\
&\qquad \text{subject to}\\
&g_i(\mathbf{x}) \leq 0, i \in 1, \ldots, l\\
&h_j(\mathbf{x}) = 0, j \in 1, \ldots, m
\end{aligned}$$

The set of possible solutions that lie within the supplied constraints is called the *feasible region*. In Figure 5.3 for example, the feasible region lies below $2x + 2$ and $-3x + 1$ and above $x^2 - 2$.

5.1.5 Quadratic Programming

Quadratic Programming [102] is a nonlinear optimisation method, which aims to minimise or maximise a multivariate quadratic function, subject to linear constraints on the variables.

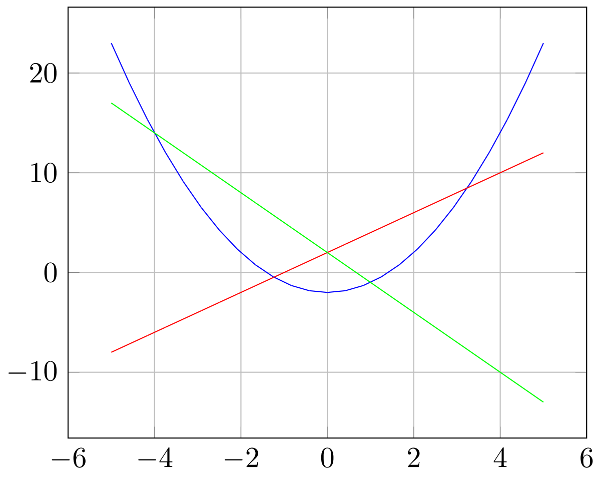


Figure 5.3: A nonlinear function $y = x^2 - 2,$ constrained by $2x + 2 - y \leq 0$ and $-3x + 1 - y \leq 0$

The quadratic function $f(\mathbf{x})$, is composed of a $\mathbf{Q}$, a symmetric, $n \times n$ square matrix and row vectors $\mathbf{c}$ and $\mathbf{x}$ of dimension n:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} \tag{5.3}$$

If $\mathbf{x}^T\mathbf{Q}\mathbf{x} \geq 0\,\forall\mathbf{x}$, then $\mathbf{Q}$ is said to be positive semi-definite and $f(\mathbf{x})$ is a *convex* function which, if bounded below by a feasible region, has a global minima. If this property is strengthened such that $\mathbf{x}^T\mathbf{Q}\mathbf{x} > 0\,\forall\mathbf{x}$ the $\mathbf{Q}$ is said to be positive definite and the global minimum will be unique.

In practice, the quadratic program is subject to multiple, linear constraints, such that the problem becomes one of finding a local minimum within these imposed bounds. In the most general sense, these constraints are expressed as linear inequalities, although linear equalities allow for an easier solution.

$$g_i(\mathbf{x}) \leq 0 \quad \forall i \in 1, \ldots, l$$
$$h_j(\mathbf{x}) = 0 \quad \forall i \in 1, \ldots, m$$

The method of *Lagrange Multipliers* [12] finds an optimum to a problem subject to linear equalities. For example, we may wish to minimise Function $f(\mathbf{x})$ subject to constraint $g(\mathbf{x}) = c$. This constraint is multiplied by a Lagrange Multiplier, λ and added to $f(\mathbf{x})$ to form a *Lagrange Function*:

$$\mathbb{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda\,(g(\mathbf{x}) - c)$$

The method of Lagrange Multipliers thus involves solving the following:

$$\begin{gathered}\nabla_{\mathbf{x},\lambda}\mathbb{L}\left(\mathbf{x},\lambda\right)=0 \text{ or}\\ \nabla_{\mathbf{x}}f=-\lambda\nabla_{\mathbf{x}}g \text{ where}\\ \nabla_{\mathbf{x}}f=\frac{\partial f}{\partial\mathbf{x}} \text{ and } \nabla_{\mathbf{x}}g=\frac{\partial g}{\partial\mathbf{x}}\end{gathered}$$

Lagrange Multipliers provides an analytic solution to an equality constrained quadratic program since it reduces the program to an easily solvable linear system. For example, Function 5.3 and constraint $\mathbf{E}.\mathbf{x}=d$ result in the following system:

$$\begin{bmatrix}\mathbf{Q} & \mathbf{E}^T\\ \mathbf{E} & 0\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ \lambda\end{bmatrix}=\begin{bmatrix}-c\\ d\end{bmatrix}$$

While the *Lagrange Multiplier* method can find an optimum subject to linear equalities, the use of *Karush-Khan-Tucker conditions* (KKT) [69, 76] generalises this method so that optima can be found subject to linear inequalities. KKT are first order *necessary conditions* for finding an optimal solution to nonlinear programming problems

The aim, once again, is to minimise a function, $f(\mathbf{x})$, this time subject to multiple inequality constraints $g_i(\mathbf{x}), i=1,\ldots,l$ and equality constraints $h_j(\mathbf{x})=0, j=1,\ldots,m$. If $\mathbf{x}*$ is some local minimum of $f(\mathbf{x})$, then there exist two sets of constants, $\mu_i \quad i=1,\ldots,l$ and $\lambda_j \quad j=1,\ldots,m$ called KKT multipliers, corresponding to the l inequality, and m equality constraints. Then, in order for $\mathbf{x}*$ to be optimal, a number of KKT conditions must also hold. The *Primary Feasibility* condition serves to re-iterate the original constraints.

$$\begin{gathered}h(\mathbf{x}*)=0\\ g(\mathbf{x}*)\leq 0\end{gathered}$$

The *Dual Feasibility* condition states that every element in u_i must be greater than or equal to zero.

$$\mu_i\geq 0\,\forall i\in 1,\ldots,l$$

Stationarity is a statistical concept which implies that the mean and variance of equally sized subsequences within a series are always constant. It is particularly useful in analysing time-related series, since data trends can be identified that are independent of time. The *Stationarity* of the system must equal zero.

$$\nabla f(\mathbf{x}*)+\sum_{j=1}^{l}\lambda_i\nabla h_i(\mathbf{x}*)+\sum_{i=1}^{m}\mu_j\nabla g_j(\mathbf{x}*)=0$$

Finally, the *Complementary slackness* condition requires that the product of μ_i and the inequality constraint corresponding to it should be zero.

$$\mu_i g_i(\mathbf{x}*) = 0$$

If the above KKT conditions hold for $\mathbf{x}*$, then it is the optimum for the problem.

5.2 Related Work

*3D Field D** [23] extends Field D* to operate on a uniform 3D grid by extending Field D*'s *direct* cost function to cubes. 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. This is accomplished, firstly, by approximating the minima along the four cube edges, and secondly, by connecting the minima of opposing edges so that two intersecting lines are formed. The point at which they intersect is considered to be the minimum. It is not clear whether the minima estimation is accurate or how much error exists in the approximation of the minimum – a discussion of this topic was presented in Chapter 3. This can be considered as interpolation error present in the cost function.

More problematically, 3D Field D* also operates on a uniform 3D grid, which suffers from same geometric error inherent in representing weighted regions with 2D grids and quadtrees: A finite number of cubes cannot, in general, subdivide a polyhedron exactly. Therefore, any algorithm attempting to solve the WRP on a uniform 3D grid, will necessarily require high levels of subdivision to ameliorate geometric error in the representation of a polyhedron. An extension of this technique to multi-resolution grids may ameliorate this error, but can never remove it completely.

Also, as dimension increases, the level of subdivision required to represent a polytope increases, since the polytope boundary occupies an increasing number of dimensions. For example, to accurately represent a triangle boundary using a 2D grid, it must be finely subdivided along the three triangle edges using grid cells. Similarly, a tetrahedron boundary must be finely subdivided on the four boundary triangles using grid cubes.

Therefore, an extension of Field D* to simplices is important, because, as discussed earlier, simplices subdivide polytopes exactly and consequently allow Field D* to operate on representations free from geometric error.

5.3 Notation

As much of the discussion in this work involves simplices, we now introduce some notation. We use upper-case letters to refer to points, bold lower-case letters for vectors and bold upper-case letters for matrices.

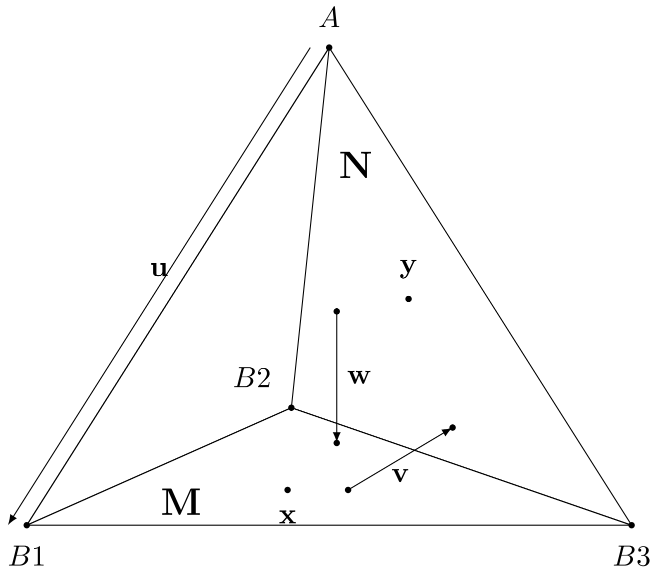


Figure 5.4: A tetrahedron with apex node A and base facet $B1B2B3$ consisting of node $B1$, $B2$ and $B3$. $\mathbf{M}$ is a matrix representing the basis of the vector subspace of the base facet, while $\mathbf{N}$ is a basis matrix representing the side facet $AB2B3$. $\mathbf{x}$ and $\mathbf{y}$ are coordinates that parameterise $\mathbf{M}$ and $\mathbf{N}$ respectively. $\mathbf{u}$ is a vector between A and $B1$, while $\mathbf{v}$ and $\mathbf{w}$ are vectors between points on the base facet and side facet respectively.

A *simplex* generalises the concept of a triangle in two dimensions and a tetrahedron in three dimensions to arbitrary dimensions. An n-simplex is a n-dimensional *polytope* constructed from $n + 1$ vertices, and is defined as the *convex hull* of those vertices.

The convex hull of any nonempty subset of the $n + 1$ vertices defining the simplex is a *face* of the simplex and is itself a simplex. An $m + 1$ subset of the original $n + 1$ vertices is an m-simplex, also termed an *m-face* of the n-simplex. Under this formulation, 0-faces are equivalent to *vertices*, 1-faces to *edges* and (n-1)-faces to *facets*.

The number of m-faces in an n-simplex, with $m < n$ is equal to the binomial coefficient $\begin{pmatrix} n+1 \\ m+1 \end{pmatrix}$. Using this formula, it can be seen that there are $n + 1$ facets in an n-simplex, for example.

Simplices may be connected together in a *Simplicial Complex*, sharing vertices and facets. In a 3D Simplicial Complex, two adjacent tetrahedra share a facet (triangle) and three vertices.

When referring to simplices, A will denote the *apex* vertex, or the node for which we are calculating the path cost $g(A)$, while the vertices $B1 \ldots Bn$ form a facet of the simplex opposite A, which we call the *base facet*. The other facets, involving A and each Bi save one, we denote the *side facet*. The path costs $g(Bi)\,\forall i \in (1, \ldots, n)$, form a linear weighting system on the base facet. We denote the interior weight of the simplex with λ, while we use β_i to denote the weights of simplices adjacent to the simplex under consideration.

In the mathematical derivations that follow, we refer to m-dimensional vector subspaces formed from the coordinate system of the base and side facets, as well as vectors expressing the distance between points and the origin. We use the following notation to refer to the bases of these subspaces and the associated vectors:

- $\mathbf{M}$: A matrix, the basis of the vector subspace formed by points on the base facet.
- $\mathbf{N}$: A matrix, the basis of the vector subspace formed by points on a side facet.
- $\mathbf{x}$: A coordinate with respect to basis $\mathbf{M}$.
- $\mathbf{y}$: A coordinate with respect to basis $\mathbf{N}$.
- $\boldsymbol{\mu}$: A vector, the gradient of a linear function defined over the vector subspace.
- $\mathbf{u}$: A vector, between the apex point, $A \in \mathbb{R}^n$ and the point $B1$.
- $\mathbf{v}$: A vector between two points in the vector subspaces described by $\mathbf{M}$ and $\mathbf{N}$.
- $\mathbf{w}$: A vector between two points in the vector subspaces described by $\mathbf{M}$ and $\mathbf{N}$.

5.4 Proof of separation of the Direct and Indirect Cases

In their development of the Field D* algorithm [49], the authors prove that a path through a triangle originating from an interpolated edge must either be a *direct* or *indirect* case - a combination of the two is not optimal. Similarly, here, we show that it is a path cannot be optimal if it includes both a point on the interior of the base facet and multiple points in the side facets. Thus, if a point originates on the interior of the base facet, it must travel straight to the apex node A, which we have defined as the direct case. However, if a point originates on the boundary of the base facet, it can either be an edge case of the direct case, or an indirect case. This proof was first published as a technical report [91].

Suppose that all global minima are for a path through some $P1$ in the interior of the base facet of the simplex and some $P2, P3, \ldots, Pm$ in the interior of the side facets. An example configuration is shown in Figure 5.5. The path travels from $P1 \ldots PmA$. We will show that this leads to a contradiction.

The path cost of a point $P1$ is linear in $P1$ by definition. We express the linear weighting for the sake of simplicity as $\mathbf{w} \cdot P1 + d$, where $\mathbf{w}$ is a n-dimensional vector representing a linear scaling and d is the offset of this linear scaling system.

Let T be the point where $AP2$ meets the face $B2B3 \ldots Bn$. We parameterise the line segments

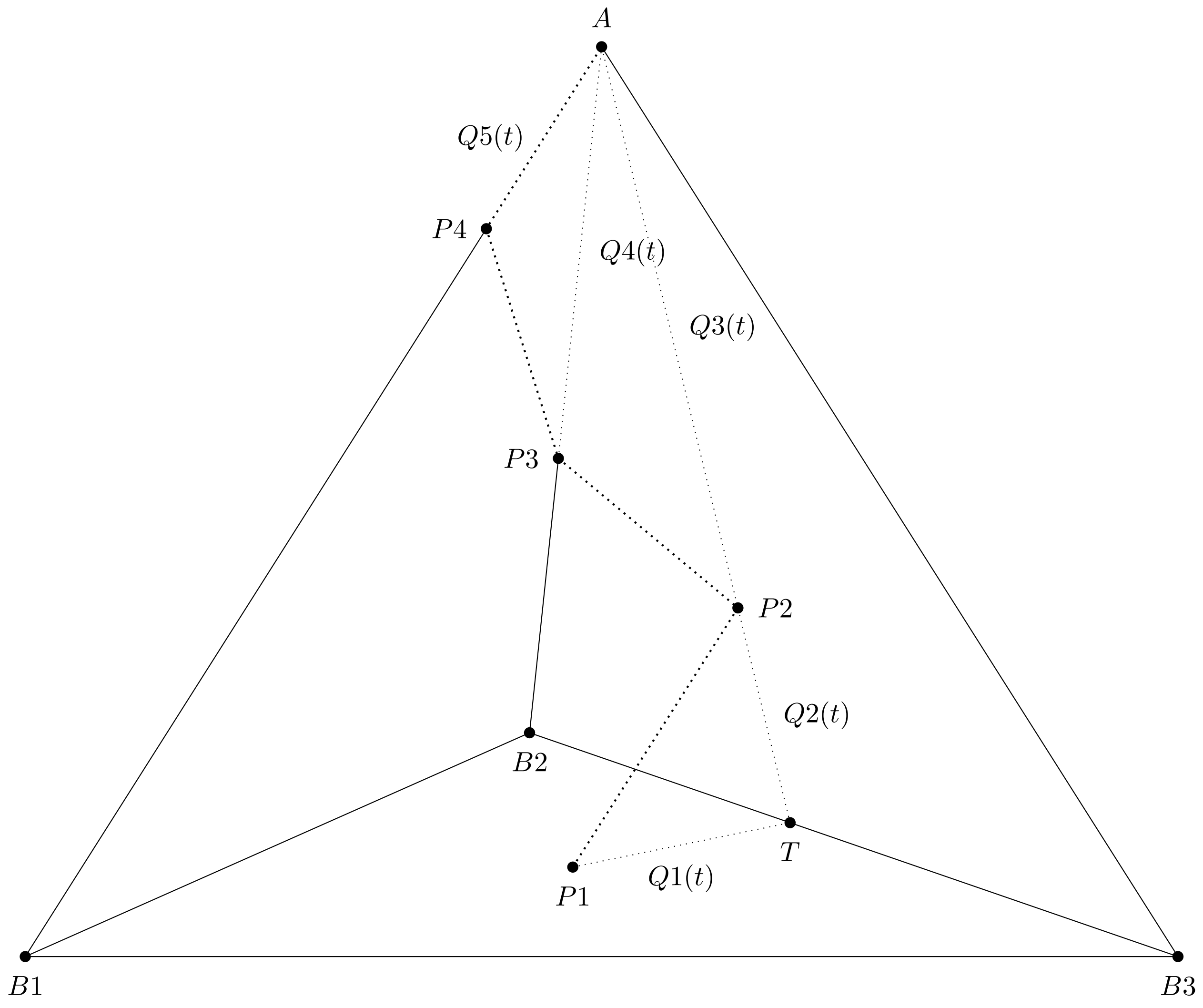


Figure 5.5: Depiction of the proof by contradiction, which shows that optimal paths originating on the *interior* of the base plane cannot travel on side facets, $P1T$ and $P2T$ are parameterised by t, while $P2A$, $P3A$ and $P4A$ are parameterised by u. $P1, P2, P3$ and $P4$ are points on a path that is assumed to be a global minimum. Since u is linear in t, and the cost function describing a path through these points, $G(t)$ is itself linear in t, a local minimum for this function can only occur when the slope is zero. However, starting from 1, t can be adjusted downwards until $P1$ and $P2$ reach T on the facet boundary or Pi reaches the base facet for $i > 2$ without changing the cost, contradicting the assumption that the global minimum occurs within the base plane.

constituting the path throught the simplex with t:

$$\begin{aligned} Q1(t) =& T + t(P1 - T) \\ Q2(t) =& T + t(P2 - T) \\ Qi(t) =& A + u(t)(Pi - 1 - A) \, \forall i > 2 \end{aligned}$$

where $u(t)$ is a linear function of t that we derive from the following relation:

$$\begin{aligned} u(t)(A - P2) + t(P2 - T) &= (A - T) \qquad (5.4) \\ \Rightarrow u(t)\|A - P2\| + t\|P2 - T\| &= \|A - T\| \\ \Rightarrow u(t) &= \frac{\|A - T\|}{\|A - P2\|} - t\frac{\|P2 - T\|}{\|A - P2\|} \end{aligned}$$

In particular, the term $\|Q3(t) - Q2(t)\|$ is linear in $u(t)$, and thus t, by using the relation expressed in Equation 5.4:

$$\begin{aligned} \|Q3(t) - Q2(t)\| &= \|u(t)(P3 - A) + A - t(P2 - T) - T\| \\ &= \|u(t)(P3 - A) - t(P2 - T) + (A - T)\| \\ &= \|u(t)(P3 - A) - t(P2 - T) + u(t)(A - P2) + t(P2 - T)\| \\ &= \|u(t)(P3 - A) + u(t)(A - P2)\| \\ &= u(t)\|P3 - P2\| \end{aligned}$$

The other distance components of consecutive sections of the path are also linear in t:

$$\begin{aligned} \|Q2(t) - Q1(t)\| &= \|t(P2 - T) + T - t(P1 - T) - T\| \\ &= \|t(P2 - P1)\| \\ &= t\|P2 - P1\| \\ \|Qi + 1(t) - Qi(t)\| &= \|u(t)(Pi + 1 - A) + A - u(t)(Pi - A) - A\| \\ &= \|u(t)(Pi + 1 - Pi)\| \\ &= u(t)\|Pi + 1 - Pi\| \\ \|A - Qm(t)\| &= \|A - u(t)(Pm - A) - A\| \\ &= u(t)\|Pm - A\| \end{aligned}$$

Note that $Pi = Qi(1)$. Therefore we can say that there is an open interval I including the value 1, containing a range of values for t such that $Qi(t)$ will always lie within the interior of their respective facets. Thus, $t \in I$ will always produce a legal path.

Let $G(t)$ be the path cost through $Qi(t)$. Since the points Pi are supposed to give the globally optimal path, $G(1)$ must be a local minimum on I. Now $G(t)$ can be expressed as:

$$\begin{aligned}G(t) =& \mathbf{w} \cdot Q1(t) + d + \lambda\|Q2(t) - Q1(t)\| + \beta_1\|Q3(u(t)) - Q2(u(t))\|+\\ &\ldots + \beta_o\|Qm(u(t)) - A\|\\ =& \mathbf{w} \cdot Q1(t) + d + \lambda t\|P2 - P1\| + \beta_1 u(t)\|P3 - P2\|+\\ &\ldots + \beta_n u(t)\|Pm - A\|\end{aligned}$$

Thus, G is a linear function of t. A linear function can only have a local minimum on an open interval if its slope is zero. But in that case, we can start with $t = 1$ and then adjust it upwards or downwards until any $Qi(t)$ reaches the boundary of its corresponding facet without changing the cost because of the zero slope. But this contradicts the assumption that there are no global minima except where Pi are in the interior.

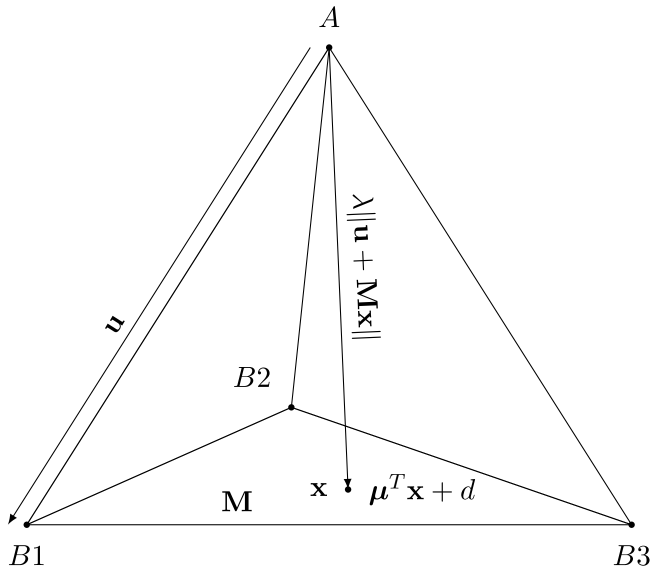
This proves that it is not possible for the path with the lowest cost to include points on both the interior of the base facet and the side facets. Therefore, in the direct case, the path must travel from a point on the base facet directly to the apex node A. However, it is possible for the shortest path to originate from points on the *boundary* [1] of the base facet and then travel to points on the side facets. These form the indirect cases in higher dimensions. In particular, we have not proved that the indirect cases may not have more than two path segments and we leave this for future work. The rest of this chapter only considers indirect cases involving two path segments.

5.5 N-Dimensional General Cost Functions

Extending Field D*'s cost functions to N-Dimensions requires a concise mathematical treatment. For example, in *Field D* pathfinding on weighted triangulated and tetrahedral meshes* [106], minimising a cost function on a 3D tetrahedron requires reduction to a two-dimensional case, at which point two-dimensional cost functions could be applied. While a similar process could be applied in higher dimensions, it quickly becomes cumbersome. Thus, to simplify higher dimensional cases, and to make the three-dimensional case easier to implement, we express the general cost function developed in Chapter 4 with *Linear Algebra*.

We first extend the general cost function by changing its arguments to use vectors and matrices, and develop an analytic solution for this function. This function does not describe the indirect cases as effectively as possible because it does not express the distance between two bases that the case must consider. To this end, we also present an extended version of the cost function which can represent these indirect cases. We do not have an analytic solution for the extended version, but for certain

[1] In 3D, boundary of the triangle that forms the base facet would consist of the triangle edges. In 4D, the boundary of the tetrahedron that forms the base facet, would itself consist of triangles.



A
$\mathbf{u}$
$\lambda\|\mathbf{u} + \mathbf{M}\mathbf{x}\|$
$B2$
$\mathbf{x}$
$\boldsymbol{\mu}^T\mathbf{x} + d$
$\mathbf{M}$
$B1$
$B3$

Figure 5.6: The components of the general cost function. $\mathbf{M}$ is a basis matrix of a vector subspace formed on the points of the base facet $AB1B2$ and $\mathbf{u}$ is a vector that expresses the distance between A and $B1$. Then, $\lambda\|\mathbf{u} + \mathbf{M}\mathbf{x}\|$ express the cost of travelling with weight λ from A to coordinate $\mathbf{x}$ in this vector subspace, weighted by linear function $\boldsymbol{\mu}^T\mathbf{x} + d$.

indirect cases it reduces to the simpler general cost function. This is especially the case in 3D, where all indirect cases can be reduced to the general cost function.

5.5.1 General Cost Function

The general cost function 4.1 described in Chapter 4 is expressed in terms of vectors and scalars:

$$G(x, \lambda, \mathbf{v}_1, \mathbf{v}_2, \boldsymbol{\mu}, d) = \lambda\|\mathbf{v}_1 + x\mathbf{v}_2\| + \mu x + d \tag{5.5}$$

Recall that $\mathbf{v}_1$ and $\mathbf{v}_2$ are linearly independent. Then, $\mathbf{v}_2$ is a 1D basis, representing a 1D vector subspace of $\mathbb{R}^2$ and x is a coordinate relative to this basis. v_1 is a position vector representing a point relative to the origin. Then $\lambda\|\mathbf{v}_1 + x\mathbf{v}_2\|$ expresses a distance, scaled by λ, between this point and a range of points within the vector subspace. $\mu x + d$ is a linear function defined on the vector subspace, with d the value at the origin. This formulation is convenient for expressing functions involving two edges of a triangle in 2D, but does not scale to higher dimensions.

We now extend these concepts to the Euclidean vector space $\mathbb{R}^n$. Let $\mathbf{u}$ be a n-dimensional position vector representing a point in $\mathbb{R}^n$ relative to the origin and let λ be the cost of travelling through space. Let $\mathbf{M}$ be an $n \times m$ basis matrix representing a m-dimensional vector subspace of $\mathbb{R}^n$, composed of m linearly independent, n-dimensional vectors, and with $m < n$. Then $\mathbf{x}$ is an m-dimensional vector expressing a coordinate relative to this basis. Also let $\boldsymbol{\mu}$ be an m-dimensional vector, defining a linear function $\boldsymbol{\mu}^T x + d$ over the vector subspace, where d is the scalar value of this function at the origin.

Then we define the general cost function as:

$$G\left(\mathbf{x}, \lambda, \mathbf{u}, \mathbf{M}, \boldsymbol{\mu}, d\right) = \lambda \|\mathbf{u} + \mathbf{M}\mathbf{x}\| + \boldsymbol{\mu}^T \mathbf{x} + d \tag{5.6}$$

In practice, we set $\mathbf{u} = B1 - A$ and compose $\mathbf{M}$ from m linearly independent vectors between base facet vertices, $Bi - B1 \; i \in 2, \ldots, m$ for example. Then $\|\mathbf{u} + \mathbf{M}\mathbf{x}\|$ represents the distance between A and a range of points on the hyperplane containing the base facet. Similarly, we compose $\boldsymbol{\mu}^T$ from the m scalar path cost differences, $g(Bi) - g(B1) \; i \in 2, \ldots, m$ and set $d = g(B1)$, so that $\boldsymbol{\mu}^T \mathbf{x} + d$ interpolates the path costs of the base facet vertices over the facet hyperplane. This configuration can be seen in Figure 5.6.

Also, in order for $\mathbf{x}$ to represent a coordinate within the facet, $\mathbf{x} > 0$ and $\|\mathbf{x}\| \leq 1$ must hold, otherwise $\mathbf{x}$ lies outside the simplex.

The analytic solution of Cost Function 5.6, derived in Section B.1 of the Appendix is:

$$\mathbf{x}^T = \left(\phi \boldsymbol{\mu}^T - \mathbf{u}^T \mathbf{M}\right) \left(\mathbf{M}^T \mathbf{M}\right)^{-1} \mathbf{M}^T \tag{5.7}$$

$$\text{where } \phi = -\sqrt{\frac{\mathbf{u}^T \left(\mathbf{I} - \mathbf{M}\left(\mathbf{M}^T \mathbf{M}\right)^{-1}\right) \mathbf{u}}{\lambda^2 - \boldsymbol{\mu}^T \left(\mathbf{M}^T \mathbf{M}\right)^{-1} \boldsymbol{\mu}}} \tag{5.8}$$

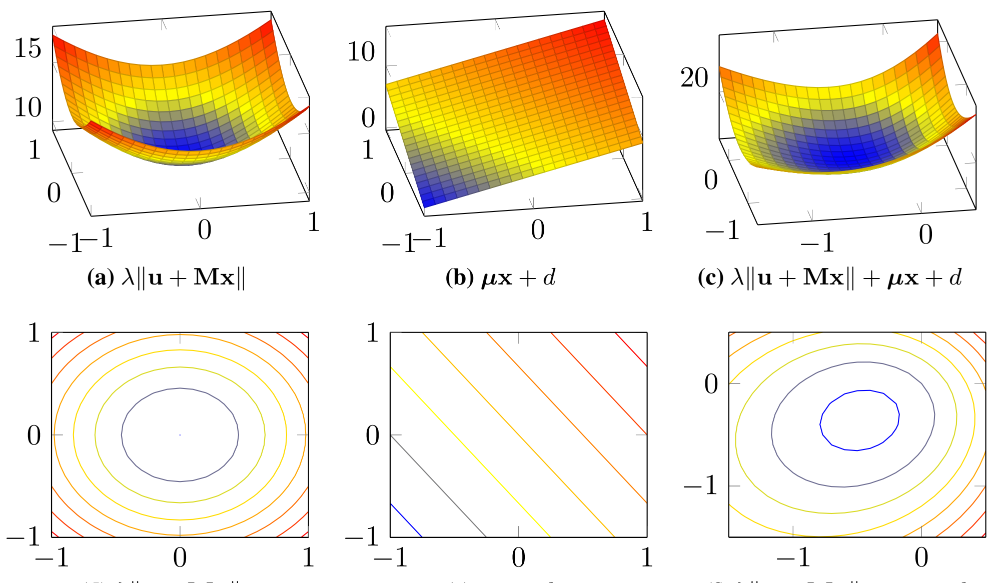
We note that $\mathbf{u}^T(\mathbf{I} - \mathbf{M}(\mathbf{M}^T\mathbf{M})^{-1})\mathbf{u}$ is the squared distance between position vector $\mathbf{u}$ and its orthogonal projection onto $\mathbf{M}$. Also, $\lambda^2 - \boldsymbol{\mu}^T \left(\mathbf{M}^T \mathbf{M}\right)^{-1} \boldsymbol{\mu}$ is the difference between the squared travel cost from $\mathbf{u}$ and a quadratic of μ.

Therefore, $\mathbf{x}$, depends on scalar ϕ, a ratio between a point's distance from the basis and the difference between the travel cost λ from this point and linear gradient $\boldsymbol{\mu}$. In particular $\lambda^2 > \boldsymbol{\mu}^T \left(\mathbf{M}^T \mathbf{M}\right)^{-1} \boldsymbol{\mu}$ is required for ϕ to be real. If this does not hold, then the intuition is that the linear component $\boldsymbol{\mu}^T \mathbf{x} + d$ dominates the distance component entirely and no global minima exists. However, a local minima will always exist on the exterior of the base facet. To find this local minima, ϕ can be set to a large value and the technique described in Section 5.5.1 used to find it.

The distance component, $\lambda \|\mathbf{u} + \mathbf{M}\mathbf{x}\|$ is in fact a *Least-norm* convex function [20] whose gradient can be visualised as a collection of hyperspheres (Figures 5.7a and 5.7d). If the linear component does not dominate (Figures 5.7b and 5.7e), it changes the distance component by "tilting" the solution in the direction of the plane formed by the linear component. The combination of the two results in a convex function whose gradient is a collection of ellipsoids (Figures 5.7c and 5.7f). The ellipsoid nature of the gradient is shown in Section B.1.2.

Edge Conditions

An analytic minimisation of G produces a minimum, $\mathbf{x}$, a coordinate relative to the basis $\mathbf{M}$, representing a coordinate system on the facet of a simplex. Minima lying outside the facet do not correspond



(a) $\lambda\|\mathbf{u}+\mathbf{Mx}\|$ **(b)** $\boldsymbol{\mu}\mathbf{x}+d$ **(c)** $\lambda\|\mathbf{u}+\mathbf{Mx}\|+\boldsymbol{\mu}\mathbf{x}+d$

(d) $\lambda\|\mathbf{u}+\mathbf{Mx}\|$ **(e)** $\boldsymbol{\mu}\mathbf{x}+d$ **(f)** $\lambda\|\mathbf{u}+\mathbf{Mx}\|+\boldsymbol{\mu}\mathbf{x}+d$

Figure 5.7: (a) and (d): $\lambda\|\mathbf{u}+\mathbf{Mx}\|$ is a convex function that produces hypersphere (circular) contour lines. (c) + (e):$\boldsymbol{\mu}\mathbf{x}+d$ produces lines on a plane. (d) + (f): Combining them results in another convex function (as long as $\boldsymbol{\mu}\mathbf{x}+d$ does not dominate) producing paraboloid contour lines.

to physically correct locations. Thus, the analytic minimisation must be constrained to lie within the bounds of this facet: specifically we require $\mathbf{x} \geq 0$ and $\|\mathbf{x}\| \leq 1$. If $\mathbf{x}$ does not satisfy these relations, a local minimum must be found on the exterior of the facet.

One possible method for finding the local minimum would solve G on the exterior of the facet: The $(m-1)$-faces on the exterior of the m-face. For example, the base facet of a tetrahedron is a triangle (2-face) with three edges (1-faces) on the exterior and base facet of a 4-simplex is a tetrahedron (3-face) with four triangles (2-faces) on the exterior. $\mathbf{M}$ and $\boldsymbol{\mu}$ can simply be reconfigured for each of the $(m-1)$-faces and G solved to find a local minimum on each of them.

However, the local minimum may still not lie on the $(m-1)$-face – the exterior triangle of a tetrahedron, for example – and thus the process must continue until local minima on edges (1-faces) are considered. Consequently, this is a computationally expensive solution which becomes more expensive as the dimension of the problem increases: The local minima for three edges of a triangle must be considered. For a tetrahedron, four triangles and six edges must be considered and for a 4-simplex, five tetrahedra, 10 triangles and 10 edges.

A more efficient method of determining which m-face contains the local minimum utilises the following information:

- $\mathbf{x}$ is a coordinate relative to the $\mathbf{M}$ basis.
- Within this coordinate system, the $m-1$-faces form hyperplanes with simple coordinates.

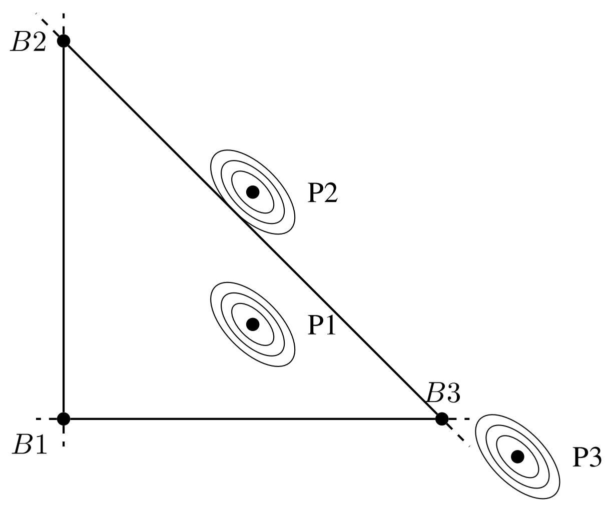


Figure 5.8: The contour of G is a family of ellipsoids expanding around the minimum. Their convex nature implies that we need only classify $\mathbf{x}$ with respect to the hyperplanes formed from the facet edges in the coordinate system of $\mathbf{M}$. $P2$ is within the positive half-space of the hyperplane of the hypotenuse, but not those of the top and bottom edge. Thus, the hypotenuse edge can be selected as the boundary condition to consider. $P3$ is within the positive half-space of the hyperplanes of the bottom and hypotenuse edges. Therefore we must consider both these boundary conditions.

- The gradient of G forms a family of ellipsoids in this coordinate system.

In Section B.1.2 we show that the contour lines of G are a family of ellipsoids, radiating outwards from $\mathbf{x}$. From the fact that these contour lines are convex, we can classify the position of $\mathbf{x}$ relative to the half-space of the hyperplanes formed by the facet edges. If $\mathbf{x}$ is within the positive hyperplane related to an edge, then $\mathbf{x}$ lies outside the facet and we should check for a local minimum along the related edge.

For example, in the tetrahedral case shown in Figure 5.8, $B1$ is at $(0, 0)$, $B2$ at $(1, 0)$ and $B3$ at $(0, 1)$ in the coordinate system of $\mathbf{M}$. Then $P1$ is within the negative half-space of all three hyper-planes and is therefore the local minima. $P2$, however, is within the positive half-space of the hypotenuse hyperplane and the negative half-space of the top and bottom hyperplanes. Therefore, we consider only the hypotenuse edge when finding a local minimum. $P3$ is within the positive half-space of both the hypotenuse and bottom hyperplanes and consequently we must check for local minima along these edges.

Thus, using this hyperplane classification system, we can reduce the number of edges that we should check for a local minimum.

5.5.2 Extended General Cost Function

This formulation of the cost function is more general in that it expresses the cumulative cost of travelling a distance between points on bases $\mathbf{M}$ and $\mathbf{N}$, weighted by λ, followed by the cost of travelling

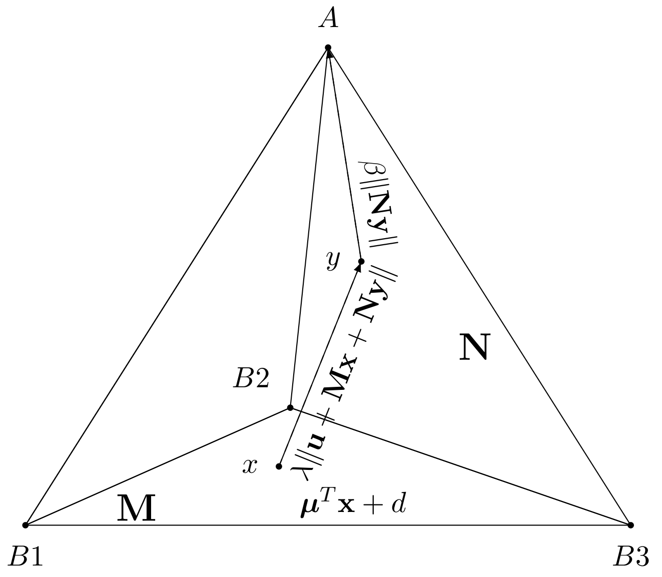

A
y
$\beta\|\mathbf{Ny}\|$
$\lambda\|\mathbf{u}+\mathbf{Mx}+\mathbf{Ny}\|$
$\mathbf{N}$
$B2$
x
$\mathbf{M}$
$\boldsymbol{\mu}^T\mathbf{x}+d$
$B1$
$B3$

Figure 5.9: Layout of the Field D* extended general cost function H. It expresses the cost of travelling from a coordinate $\mathbf{x}$, weighted by linear function $\boldsymbol{\mu}^T\mathbf{x}+d$, added to the weighted cost of travelling between $\mathbf{x}$ on basis $\mathbf{M}$ and y on basis $\mathbf{N}$, $\lambda\|\mathbf{u}+\mathbf{Mx}+\mathbf{Ny}\|$, added to the cost of travelling along basis $\mathbf{N}$, $\beta\|\mathbf{Ny}\|$, to A.

to point A, weighted by β. It provides a more natural expression of the various indirect cases that are encountered in higher dimensions:

$$H(\mathbf{x},\mathbf{y},\lambda,\beta,\mathbf{u},\mathbf{M},\mathbf{N},\boldsymbol{\mu},d) = \lambda\|\mathbf{u}+\mathbf{Mx}+\mathbf{Ny}\| + \beta\|\mathbf{Ny}\| + \boldsymbol{\mu}^T\mathbf{x}+d \tag{5.9}$$

An example of the physical configuration is shown in Figure 5.9. Two points, $P1$ and $P2$ are expressed by parameterising $\mathbf{M}$ and $\mathbf{N}$ with variables $\mathbf{x}$ and $\mathbf{y}$, respectively, such that $P1 = B1 + \mathbf{Mx}$ and $P2 = A + \mathbf{Ny}$. $\mathbf{u} = A - B1$ is a vector expressing the difference between the origins of the two bases. In particular, choosing A as the origin for basis $\mathbf{N}$ allows us to express the distance $\|P2 - A\| = \|A + \mathbf{Ny} - A\| = \|\mathbf{Ny}\|$. This distance is weighted by β. $\|\mathbf{u}+\mathbf{Mx}+\mathbf{My}\|$ expresses the distance between the two bases [2] and is weighted by lambda. Finally, $\boldsymbol{\mu}^T\mathbf{x}+d$ expresses a linear function defined over the vector subspace represented by $\mathbf{M}$.

[2]To express this distance more naturally, $\mathbf{Mx}$ could be re-expressed as $-\mathbf{Mx}$, but the sign can be incorporated into the matrix.

The solutions for $\mathbf{x}$ and $\mathbf{y}$ derived in Section B.2 of the Appendix, are:

$$\begin{aligned}
\mathbf{x}^T &= \left(\phi\boldsymbol{\mu}^T - \mathbf{u}^T\mathbf{M}\right)\left(\mathbf{M}^T\mathbf{M}\right)^{-1} \\
\mathbf{y}^T &= \theta\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1} \\
\phi &= -\sqrt{\frac{\mathbf{u}^T\left(\mathbf{I} - \mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\right)\mathbf{u}}{\lambda^2 - \boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu}}} \\
\theta &= -1 \pm \sqrt{\frac{\beta^2\left(\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{w} - \mathbf{w}^T\mathbf{w}\right)}{\left(\lambda^2 - \beta^2\right)\left(\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{w}\right)}} \\
\mathbf{v} &= \mathbf{u} + \mathbf{N}\mathbf{y} \\
\mathbf{w} &= \mathbf{u} + \mathbf{M}\mathbf{x}
\end{aligned}$$

Just as $\mathbf{x}$ is parameterised by scalar ϕ, $\mathbf{y}$ is also parameterised by scalar θ. θ expresses a ratio between the two weights, β and λ, as well as the ratio between the distance of the projection of $\mathbf{w}$ onto $\mathbf{N}$ and the distance between this projected point and the apex A. In order for θ to be real, $\beta < \lambda$ and $\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{w} \neq 0$ must hold.

Since $\mathbf{v}$ and $\mathbf{w}$ still contain $\mathbf{y}$ and $\mathbf{x}$, respectively, the solutions for $\mathbf{x}$ and $\mathbf{y}$ are not independent of each other. However, it is possible to eliminate $\mathbf{y}$ and θ from H, as shown in Section B.2.1 of the Appendix, to produce the following form:

$$\begin{aligned}
H\left(\mathbf{x}, \lambda, \beta, \mathbf{u}, \mathbf{M}, \mathbf{N}, \boldsymbol{\mu}, d\right) = \frac{\lambda^2 \pm \beta^2}{\sqrt{\lambda^2 - \beta^2}} \| \left(\mathbf{I} - \mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\right)\left(\mathbf{u} + \mathbf{M}\mathbf{x}\right) \| + \\
\beta \| \mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\left(\mathbf{u} + \mathbf{M}\mathbf{x}\right) \| + \boldsymbol{\mu}^T\mathbf{x} + d \qquad (5.10)
\end{aligned}$$

where $\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T$ is a matrix that projects orthogonally onto basis $\mathbf{N}$. There are two distance components: The first expresses the distance between vector $\mathbf{u}+\mathbf{M}\mathbf{x}$ and its projection onto $\mathbf{N}$. while the second expresses the magnitude of $\mathbf{u} + \mathbf{M}\mathbf{x}$ projected onto $\mathbf{N}$. These two distance components are orthogonal to each other, and thus the distance components from the original Equation have been transformed so that they lie on the *catheti* of a right-angled triangle.

We have not managed to obtain a general analytic solution for Equation 5.10 since two distance terms and one linear term contain $\mathbf{x}$. Attempts at solving 5.10 by minimisation suggest that it is necessary to solve for an eighth degree polynomial in $\mathbf{x}$. However, in certain cases described in Section 5.5.1, a distance term is linear and 5.10 reduces to 5.6, for which an analytic solution is available.

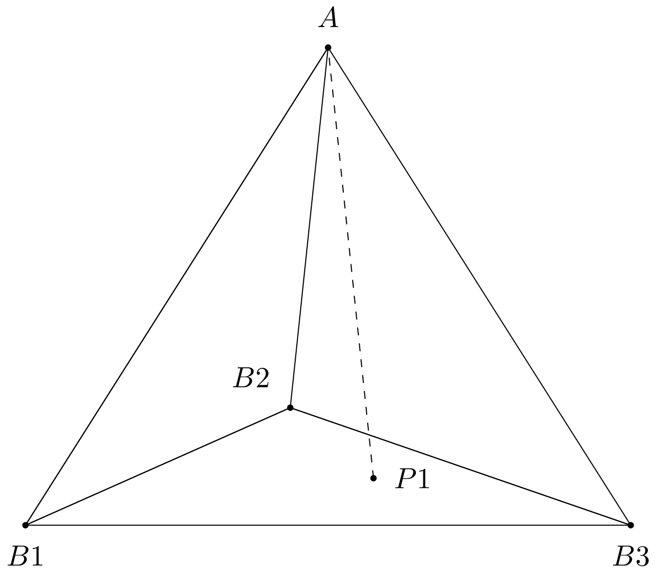


Figure 5.10: A direct case originating from point $P1$ on the base facet $B1B2B3$.

5.6 N-dimensional Direct and Indirect Cost Functions

In this section, we describe how the general cost functions from Section 5.5 can be applied to solve the direct and indirect cases in arbitrary dimensions.

5.6.1 Direct Case

In the direct case, the path travels directly from a linearly weighted base simplex to the apex node A. For example, in Figure 5.10, the path travels from point $P1$ on $\triangle B1B2B3$ to A.

In this example, we simply set

$$\begin{aligned}
\mathbf{v} &= B1 - A \\
\mathbf{M} &= \left[\begin{array}{c} B2 - B1, B3 - B1 \end{array} \right]^T \\
\boldsymbol{\mu} &= \left[\begin{array}{c} g(B2) - g(B1), g(B3) - g(B1) \end{array} \right]^T \\
d &= g(B1)
\end{aligned}$$

and solve the Cost Function 5.6 to obtain first ϕ and then $\mathbf{x}$. Thus, to solve the direct case, we simply set $\mathbf{M}$ to be the basis of the base facet and $\boldsymbol{\mu}$ to the differences between the linear weightings of the facet vertices. d is assigned the weight at the origin of the linear weighting system, $g(B1)$, while $\mathbf{v}$ is assigned the difference between the apex node and $B1$ the origin of basis $\mathbf{M}$.

5.6.2 Indirect Cases

The *indirect case* is the most difficult case to extend to higher dimensions. Firstly it must consider neighbouring simplices and the number of neighbouring simplices increases with each dimension. Secondly, the number of indirect cases rapidly increases with each dimension, because the number of ways in which simplices neighbour each other also increases. In the three-dimensional case for example, four tetrahedra each share a face with the primary tetrahedron and so we must consider the indirect cases involving these faces (Figure 5.11). Additionally, the three edges with the primary tetrahedron are also shared with an unspecified number of tetrahedra and consequently there are indirect cases involving these edges (Figure 5.12).

Each of Field D*'s cases originate from some point on the base facet of the simplex under consideration. The *direct* case originates from some internal point within the facet, while *indirect* cases originate from points on the facet exterior. In three-dimensions for example, the base facet is a triangle, *direct* cases originate from the interior of this triangle, while *indirect* cases originate from the edges of the triangle. Thus, there is also an interpolation component to the *indirect* case. [106] do not consider this interpolation component, and only present indirect cases originating from nodes. The interpolation component is not immediately apparent in the original two-dimensional case, since in two-dimensions the base facet is an edge, and the two *indirect* cases originate from the two nodes on either side of the edge. It is, however, present in the three-dimensional, and higher-dimensional cases.

Indirect Hyperplane ($n-1$) Case

Section 5.5.2 re-expressed H as:

$$H\left(\mathbf{x}, \lambda, \beta, \mathbf{u}, \mathbf{M}, \mathbf{N}, \boldsymbol{\mu}, d\right) = \frac{\lambda^2 \pm \beta^2}{\sqrt{\lambda^2 - \beta^2}} \| \left(\mathbf{I} - \mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\right)\left(\mathbf{u} + \mathbf{M}\mathbf{x}\right) \| + \\ \beta \|\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\left(\mathbf{u} + \mathbf{M}\mathbf{x}\right)\| + \boldsymbol{\mu}^T\mathbf{x} + d$$

In this form, $\mathbf{y}$ has been eliminated from H and can be used to solve certain indirect cases. The key to understanding how this is possible involves examining the *kernel* of the projection matrix $\mathbf{P}_N = \mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T$. The basis $\mathbf{N}$, composed of linearly independent vectors formed from facet edges is a *vector subspace* of the linear space inhabited by the primary simplex. If, in the general case, the linear space is $\mathbb{R}^n$, then the kernel of an projection matrix $\mathbf{P}_N$ onto basis $\mathbf{N}$ has one element if $\mathbf{N}$ consists of $n-1$ linearly independent vectors, two elements if $\mathbf{N}$ consists of $n-2$ vectors, and so on.

If $\mathbf{N}$ is composed of $n-1$ vectors, the basis $\mathbf{N}$ forms a hyperplane with normal $\mathbf{o}$, the single element in the kernel of $\mathbf{P}_N$. In this case, $\mathbf{o}\cdot\left(\mathbf{u} + \mathbf{M}\mathbf{x}\right) + e = \|\left(\mathbf{I} - \mathbf{P}_N\right)\left(\mathbf{u} + \mathbf{M}\mathbf{x}\right)\|$ for some e. The distance term then becomes linear in terms of $\mathbf{x}$ and can be combined with the $\boldsymbol{\mu}^T\mathbf{x}$ and d terms so that we have the following:

$$H\left(\mathbf{x},\lambda,\beta,\mathbf{u},\mathbf{M},\mathbf{N},\boldsymbol{\mu},d\right)=\beta\|\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\left(\mathbf{u}+\mathbf{M}\mathbf{x}\right)\|+\left(\frac{\lambda^2\pm\beta^2}{\sqrt{\lambda^2-\beta^2}}\mathbf{o}\cdot\mathbf{M}+\boldsymbol{\mu}^T\right)\cdot\mathbf{x}+ \left(\frac{\lambda^2\pm\beta^2}{\sqrt{\lambda^2-\beta^2}}\mathbf{o}\cdot\mathbf{u}+e\right)+d$$

which obeys the form of Cost Function 5.6 and thus has an analytic solution. However, this reduction only works for a basis of $n-1$ vectors, because the dimension, or *nullity* of $\mathbf{P}_N$ is one: it only contains one vector. In the following section we discuss why this is not possible in cases where the kernel of $\mathbf{P}_N$ has nullity greater than two.

Indirect ("inbetween") Cases between 2 and $n-2$ inclusive

If $\mathbf{N}$ is composed of $n-2$ linearly independent vectors, then the kernel of $\mathbf{P}_N$ contains two vectors and the nullity of this kernel is two. If $\mathbf{N}$ is composed of $n-3$ linearly independent vectors, the nullity will be three and so on. Then, the linear span of the kernel vectors defines a range of vectors orthogonal to the basis $\mathbf{N}$.

In these cases, the intuition is that there are many directions that are orthogonal to $\mathbf{N}$. In 3D for example, an infinite cylinder of vectors is orthogonal to a line. By contrast, when the kernel of $\mathbf{P}_N$ only contains one vector, it represents only one direction. While it is possible to scale this vector, the direction of the scaled vector does not change. In 2D for example, only one vector is orthogonal to a line and in 3D, only one vector is orthogonal to a plane.

Therefore, when the nullity $\mathbf{P}_N$ is greater than 2, a single direction, or normal vector cannot be chosen and $\|\left(\mathbf{I}-\mathbf{P}_N\right)\left(\mathbf{u}+\mathbf{M}\mathbf{x}\right)\|$ cannot conveniently be converted into a $\mathbf{o}\cdot\left(\mathbf{u}+\mathbf{M}\mathbf{x}\right)+e$ term, because $\mathbf{o}$ would need to be expressed as a linear function of the kernel vectors.

Alternatively, one could construct a normal for a point $\mathbf{u}+\mathbf{M}\mathbf{x}$ as:

$$\mathbf{o}=\frac{\left(\mathbf{I}-\mathbf{P}_N\right)\left(\mathbf{u}+\mathbf{M}\mathbf{x}\right)}{\|\left(\mathbf{I}-\mathbf{P}_N\right)\left(\mathbf{u}+\mathbf{M}\mathbf{x}\right)\|}$$

However, since the normal is dependent on $\mathbf{x}$, we cannot convert a distance component into a component linear in $\mathbf{x}$ and consequently to the form of Cost Function 5.6. Thus, there are a range of indirect cases without an analytic solution, for $\mathbb{R}^n$ where $n>3$. For ease of reference, we refer to these as the "inbetween" cases.

Indirect Edge Case

Below this range of analytically unsolvable cases lies one last indirect case which *does* have an analytic solution. This case originates from some interpolated side simplex of the base facet, travels through the primary simplex to an *edge* and then towards node A. This indirect case is solvable using Cost Function 5.6 because it is possible to include the vector describing the edge into the $\mathbf{M}$ basis, and the cost of travelling along this edge, into $\boldsymbol{\mu}$ vector.

For example in the 3D tetrahedral case shown in Figure 5.12, the path originates on side simplex $B1B3$ of the base facet $B1B2B3$, travels through the simplex to edge $AB2$ and then to node A. The cost of this path can be expressed as:

$$w(P1) + \lambda\|P2' - P1\| + \beta\|P2' - A\|$$

Now for some real s and t, the variables above can be expressed as:

$$\begin{aligned}
P1 &= (B3 - B1)s + B1 \\
P2' &= (B2 - A)t + A \\
w(P1) &= (g(B3) - g(B1))\, s + g(B1) \\
&\text{also} \\
P2' - A &= (B2 - A)t + A - A \\
\Rightarrow \|P2' - A\| &= t\|B2 - A\|
\end{aligned} \tag{5.11}$$

We can set:

$$\begin{aligned}
\mathbf{v} &= A - B1 \\
\mathbf{M} &= \begin{bmatrix} -(B3 - B1), B2 - A \end{bmatrix}^T \\
\boldsymbol{\mu} &= \begin{bmatrix} g(B3) - g(B1), \beta\|B2 - A\| \end{bmatrix}^T \\
d &= g(B1)
\end{aligned}$$

Thus $\lambda\|P2' - P1\| = \lambda\|\mathbf{v} + \mathbf{Mx}\|$, while $\beta\|P2' - A\|$ and $w(P1)$ combine to form $\boldsymbol{\mu}^T\mathbf{x} + d$.

We substitute these values into Cost Function 5.6 and solve for variable $\mathbf{x}$ to obtain an analytic solution.

An analytic solution can be obtained for this example because the basis for edge $B2A$ contains one vector. Thus, it can be parameterised by one scalar variable, which can be moved out of the distance term, as in 5.11. In terms of H, this distance term is related to $\|\mathbf{Ny}\|$.

The example above describes a 3D tetrahedral case, but this solution also works for higher dimensions: The basis formed from the base facet's side simplex will always form the first part of $\mathbf{M}$, while the

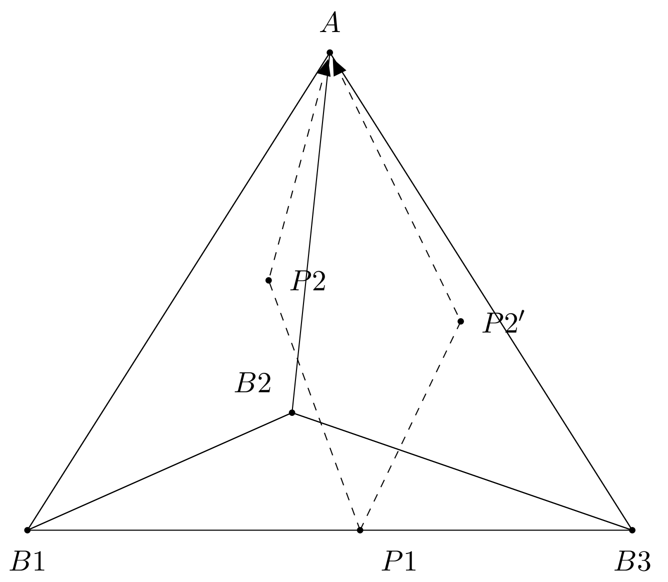


Figure 5.11: Two indirect hyperplane cases originating from a point $P1$ on edge $B1B3$ to a point $P2$ on facet $AB1B2$ and $P2'$ on facet $AB2B3$.

vector describing the edge will be the last entry in $\mathbf{M}$. Similarly, the differences between the linear weightings forms the first part of $\boldsymbol{\mu}$, while the cost of travelling along the edge will be the last entry. In terms of H, this represents a combination of matrices $\mathbf{M}$ and $\mathbf{N}$ and variables $\mathbf{x}$ and $\mathbf{y}$.

Number of Indirect Cases

Here we provide details on the number of indirect cases in each dimension. This is somewhat difficult to visualise as certain indirect cases reduce to a lower dimension.

Indirect cases originate from points on the exterior of the base facet. In 2D, this is a point on the end of a line, in 3D, an edge on the side of a triangle, and in 4D a triangle on the side of a tetrahedron. Firstly, the geometric entity from which the indirect case originates is always missing one of the vertices of the base facet. In 3D for example, an edge involves two vertices and leaves out the remaining vertex defining the triangle of the base facet.

Secondly, the geometric entity onto which the indirect case moves from its originating point contains this missing vertex, as well as the apex, A. In the 3D example in Figure 5.11, an indirect hyperplane case originates from edge $B1B3$ and moves to a point $P2$ on triangle $AB1B2$. Edge $B1B3$ leaves out vertex $B2$, but moves onto $AB2B3$, which does involve $B2$. Similarly in Figure 5.12, an indirect edge case originates from $P1$ on edge $B1B3$ and moves to a point $P2$ on edge $AB2$.

The reason for this is that is not possible for an indirect case to move onto a geometric entity that does not contain the missing vertex since this will reduce the dimension of the problem. Consider a point $P1$ originating from $B1B3$ and moving to point $P2'$ in edge $AB1$ of Figure 5.12: The dimension of the case is reduced to a 2D triangle $AB1B3$, and by the proof provided earlier, an indirect case

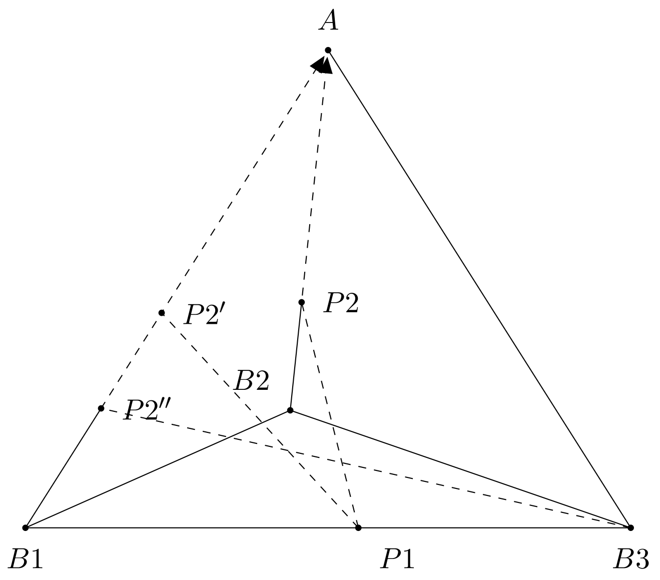


Figure 5.12: Indirect edge case. Path $P1P2A$ is a valid 3D indirect edge case. $P1P2'A$ reduces the dimensionality of the case to that of a 2D triangle. However, we know by proof that an indirect case cannot originate from a point on the base edge of triangle $AB1B3$ and so $P1P2'A$ is not a valid indirect edge case and can be safely ignored. Path $B3P2''A$ is also an example of reduction of dimensionality of the 3D indirect case to a valid 2D indirect case. It does not need to be solved as a separate case because it is a boundary condition of the indirect 3D edge case originating from edge $B2B3$ and moving to edge $AB1$.

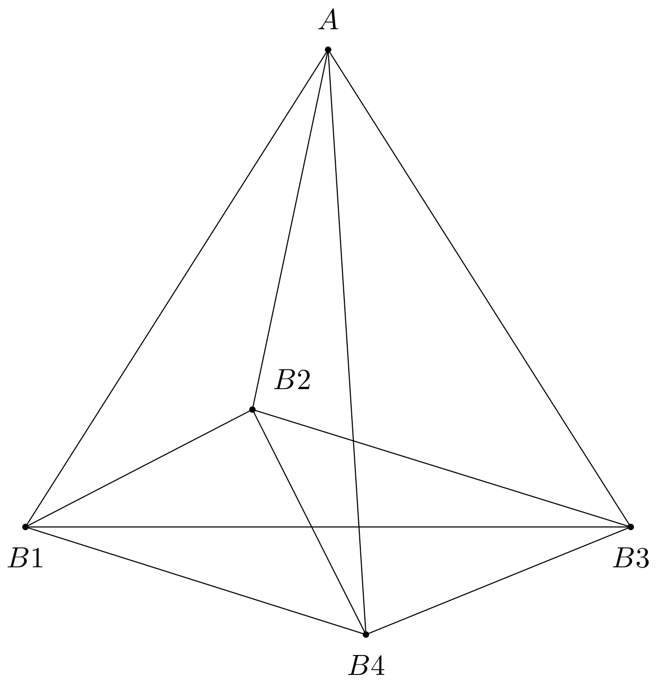


Figure 5.13: The projection of a 4-simplex into 3D.

cannot originate from the interior of a base facet. It is however possible for the path to originate from point $B3$ and move to edge $AB1$, but this is a valid 2D indirect case originating from a point and is in fact the boundary of another 3D indirect edge case originating from $B2B3$ and moving to edge $AB1$. This also applies in higher dimensions. In the 4D for example (Figure 5.13), paths originating from triangle $B1B3B4$ may not move onto triangle $AB3B4$ since the dimension of the case reduces to 3D.

Thus, indirect cases originate from n, $n-2$ simplices which form the *exterior* of the base facet, each composed of $n-1$ vertices. Each $n-2$ simplex produces a number of cases, each involving the missing vertex and the apex vertex. For example in 4D, the triangle $B1B2B3$ produces the following cases:

A	-	-	-	$B4$
A	-	-	$B3$	$B4$
A	-	$B2$	-	$B4$
A	-	$B2$	$B3$	$B4$
A	$B1$	-	-	$B4$
A	$B1$	-	$B3$	$B4$
A	$B1$	$B2$	-	$B4$

Note that the case involving the entire 4D 4-simplex ($AB1B2B3B4$) is excluded since ($B1B2B3$) is contained within it and would not produce a valid indirect case. Thus, the $n-1$ vertices in a $n-2$ simplex, in *combination* with the missing vertex produce a combinatorial total of $2^{n-1}-1$ cases. As there are n, $n-2$ simplices in the base facet, a total of $n(2^{n-1}-1)$ indirect cases exist in each dimension. The number of cases (hyperplane, edge) associated with each $n-2$ simplex can be determined by taking the binomial coefficient $\begin{pmatrix} n-1 \\ n-i \end{pmatrix}$, where i is the dimension of the subspace. For example, in 4D there are:

$$\begin{pmatrix} 3 \\ 1 \end{pmatrix} = 3 \text{ triangle to hyperplane (3 dim subspace) cases}$$

$$\begin{pmatrix} 3 \\ 2 \end{pmatrix} = 3 \text{ triangle to triangle (2 dim subspace) cases}$$

$$\begin{pmatrix} 3 \\ 3 \end{pmatrix} = 1 \text{ triangle to edge (1 dim subspace) cases}$$

It follows that there are two indirect cases in 2D, nine in 3D, 28 in 4D and 75 in 5D. Since the number of indirect cases is governed by an exponential term, their number increases rapidly in higher dimensions. In Section 5.8 we show how, at least in 3D, these indirect cases do not contribute significantly to the final path cost.

5.7 Results

Here, we describe results from our implementation of Field D* on Weighted 3D Tetrahedral Meshes. We show how Field D* can be used to find paths through medical data and simulated water data, and provide information about how the number of tetrahredra representing the environment produce different numbers of node expansions, path costs and running times. The results were first presented in [106], but this work did not include the extended range of indirect cases presented in this Chapter, instead implementing a cached indirect case originating from a node rather than from an interpolated point on an edge. The cost functions in this work were implemented *analytically*.

Subsequent to the work in [106] it was discovered that an extended range of indirect cases occur. We show that this extended range of indirect cases do not contribute significantly to the final path cost, and therefore the path costs produced in [106] are representative of a complete 3D Field D* implementation. Note that in these experiments we implemented the cost functions *numerically* due to time constraints and the running times are therefore not as fast as that of an analyic implementation.

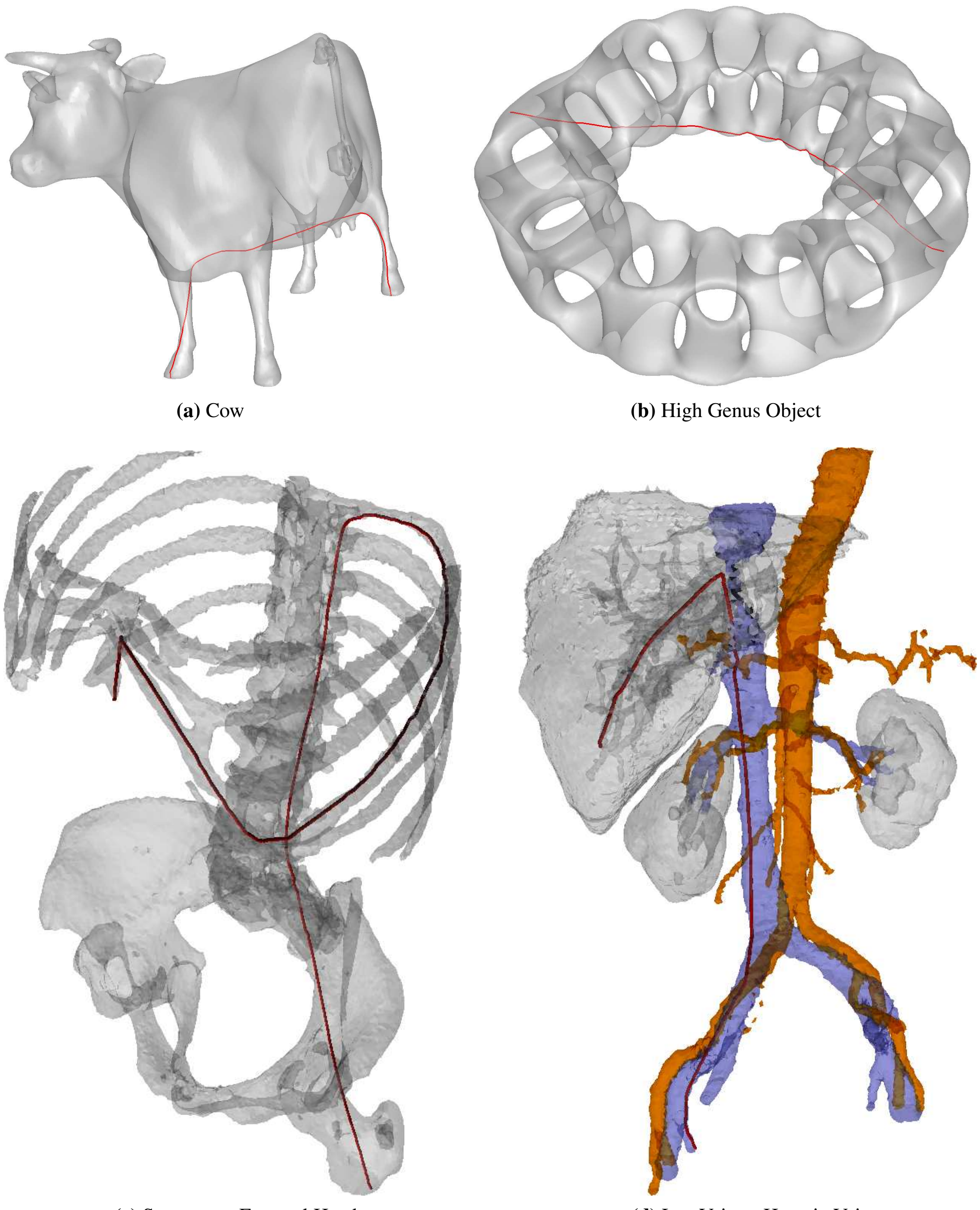
5.7.1 Pathing through 3D Models

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 5.1: 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.

We obtained a number of 3D surface models and tetrahedralized their interiors. The tetrahedra used to generate the path in Figure 5.14b and 5.14c 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) [4] 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

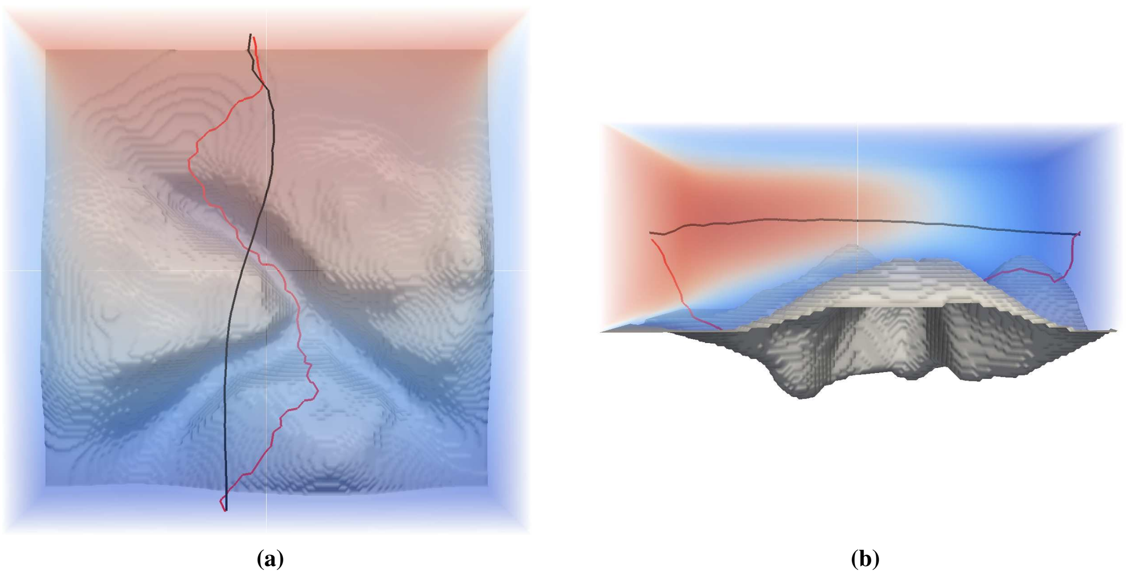

(a) Cow **(b)** High Genus Object

(c) Sternum to Femoral Head **(d)** Leg Vein to Hepatic Vein

Figure 5.14: 3D pathfinding. (a), (b) and (c) show paths through objects composed of uniformly weighted tetrahedra. In (d), tetrahedra representing the veins were weighted inexpensively, and other anatomical structures weighted expensively, resulting in the path following the veins.

winding, tubular structure of the intestinal tract without piercing the wall.

Figure 5.14a and 5.14b illustrates 3D pathing through cow and high genus objects, respectively. Figure



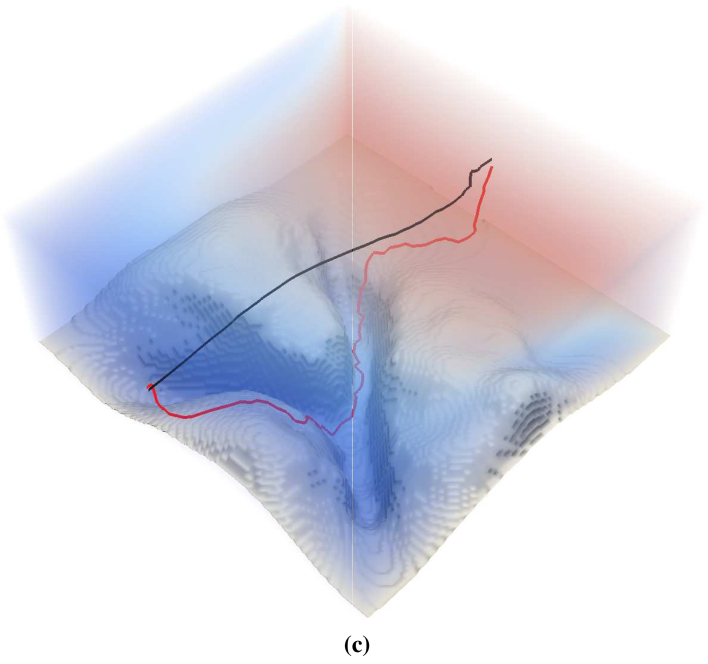


Figure 5.15: 3D pathfinding through a fluid simulation. (a) Top-down, (b) side and (c) three-quarter views. The red shading indicates areas of high fluid velocity, while blue indicates low velocity. The black path results from a tetrahedral weighting favouring the high velocity, while the red path favours low velocity, diving into the terrain crevices.

5.14c 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 5.14d 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 encourages 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 Figure 5.15. 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.

Table 5.1 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 tetrahedra representing veins and ribs. In both cases, increasing the number of tetrahedra decreases the path cost and path length at the cost of more node expansions and greater running time. The time taken for the algorithm to complete increases linearly as the number of faces and required node expansions increases.

5.8 Investigating the relevance of the indirect cases

As explained earlier, the number of indirect cases that must be considered increases with the dimension of the problem. In 3D for example, there are a total of 6 indirect *hyperplane* cases and 9 indirect *edge* cases. These cases have analytic solutions in three dimensions. In 4D, there are 12 indirect *hyperplane* cases, 12 *"inbetween"* cases and 4 indirect *edge* cases, and we have no analytic solutions for the inbetween cases. Numeric solutions would be required to solve these.

It is therefore apparent that, even in 3D, a significant amount of computation must be performed to obtain the costs for all the indirect cases. In higher dimensions, it would quickly become impractical to compute these costs due to the sheer number of cases that must be considered. Consequently, it is useful to investigate the degree to which indirect cases contribute to the final path and path cost.

Here, we first discuss how the indirect case is governed by two weights, λ and β, illustrating how sensitive the occurrence of an indirect case can be in certain circumstances. We then perform experiments showing how the indirect case in 3D, does not contribute significantly to the final path cost. These experiments were performed with numerical implementations of Field D*'s cost functions due to time constraints.

5.8.1 The relation between λ and β

To begin, we consider path costs without the involvement of g-values. This is accomplished by setting $\boldsymbol{\mu}$ and d to zero in cost function 5.9.

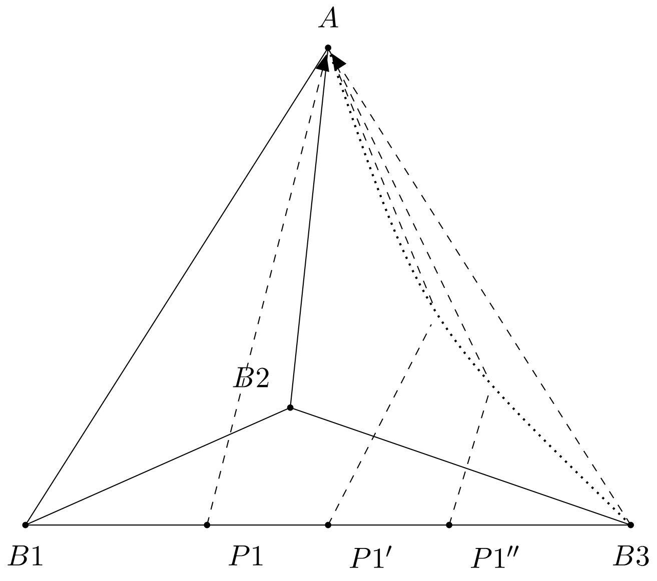


Figure 5.16: The range of an indirect case in 3D. At one end of the range, the indirect case is actually a boundary condition (a 2D direct case) of the direct case of the main tetrahedron: When $\lambda \geq \beta$ it travels from $P1$ straight to A. At the other end of the spectrum, if β is very small then $P1$ can be pushed onto $B3$ which is a boundary condition (1D direct case) of the direct case of the adjacent tetrahedron. Within the spectrum itself, there is a curve of points which the indirect case travels to on the face $AB2B3$. This curve is produced by varying β from λ to some small number ϵ. Indirect cases originating from $P1'$ and $P1''$ produce points along this curve.

The indirect case is governed by the relation between λ and β. If $\beta \geq \lambda$, then the indirect case devolves to a boundary condition of direct case. By the triangle inequality, it is cheaper to travel the shortest distance, rather than along two segments. Thus, for an indirect case to occur we must have $\beta < \lambda$. If β is very close to λ, then the shortest path involves only a short distance involving the adjacent simplex since the cost of travelling along it is only slightly cheaper than travelling through the main simplex.

If β is small and λ large, then the indirect case will tend towards travelling close to the projection of the point originating from the $\mathbf{M}$ basis onto the $\mathbf{N}$ basis, as this minimises the distance involving the large λ and maximises the distance involving β.

However, the smaller β is, the cheaper it is to travel through the adjacent simplex. Thus, we have a paradoxical situation where the better the indirect case becomes (because of a cheaper β), the better the direct case of the adjacent simplex also becomes. Additionally, the indirect case has the disadvantage of travelling an expensive distance involving λ, while the direct case of the adjacent simplex has the advantage of an inexpensive β. In fact, in cases higher than 2D, an expensive λ and inexpensive β has the effect of forcing the originating point on the M basis onto a boundary condition of the adjacent simplex since this avoids a distance involving λ.

Thus, in Figure 5.16, a high λ and low β can force $P1$ onto $B3$, since travelling along the adjacent tetrahedron weighted by β is much cheaper than travelling through the main tetrahedron weighted by

λ. By contrast a β value just below that of λ produces a direct case originating from $P1$.

This relation between β and λ produces extremely sensitive behaviour. An example of this is a tetrahedron with $A = \{3, 1, 3\}, B1 = \{1, 0, 0\}, B2 = \{5, 0, 0\}, B3 = \{3, 3, 0\}$ and $\lambda = 5, g(B1) = g(B2) = g(B3) = 0$ and the indirect case from edge $B1B3$ to the facet $AB2B3$. An actual indirect case where the indirect point lies on the interior of the face is only produced when $4.441 \leq \beta \leq 4.448$, which is only 0.14% of the range of β. The indirect cases originating from $P1'$ and $P1''$ in Figure 5.16 are examples of this. If $\beta < 4.441$ then $P1$ is forced onto $B3$, otherwise if $\beta > 4.448$, $P1$ becomes a boundary condition of the tetrahedron's direct case - an example of this is the path originating from $P1$ in Figure 5.16).

It is possible to set up pathological cases whereby environments are constructed from very thin simplices, reducing the distance involving λ. In practice our environments have been constructed using meshing algorithms that produce regularly shaped simplices with Delaunay properties [41]. This is not an unreasonable expectation since the original Field D* operates on a grid, for example. In such situations the indirect case is rare, as we show in the following section.

5.8.2 Prevalence of the indirect case

Nr of Verts	Nr of Tets	Cost		Length		Indirect Edge Instances	Indirect Plane Instances
		Direct Case Only	Indirect Cases Incl.	Direct Case Only	Indirect Cases Incl.		
1056	5797	108.713	108.713	36.2494	36.2721	4	0
1056	5748	109.257	109.257	36.0728	36.0452	6	1
1056	5797	111.774	111.774	39.4182	39.3827	1	1
1056	5797	111.941	111.941	36.6455	36.6426	0	2
1056	5797	108.705	108.705	36.8831	36.8831	4	0
1056	5726	107.648	107.648	35.7997	35.7997	2	1
1056	5797	108.999	108.999	35.7904	35.7904	1	0
1056	5796	113.135	112.708	35.787	35.782	4	2
1056	5797	112.513	112.513	36.7904	36.7904	1	0
1056	5797	110.389	110.378	37.5835	37.6166	8	5

Table 5.2: This table displays the results of ten experiments recording the path cost of travelling from corner to opposite corner of a tetrahedral mesh when firstly, only direct cases are considered and secondly, where indirect cases are also included in the search The individual tetrahedra were randomly weighted with a normal distribution, with values range from 0.1 to 256. Including indirect cases in the computation multiplies the time taken to find the shortest path by 5.35.

Each simplex requires only one direct case to be evaluated. By contrast, the number of indirect cases increases with each dimension and a number of them must be considered when calculating paths across a simplex. As the number of indirect cases contribute significantly to the total computational cost of a path across a simplex, it is important to identify the degree to which the indirect cases contribute to the final path cost.

β	Nr of Verts	Nr of Tets	Cost		Length		Indirect Edge Instances	Indirect Plane Instances
			Direct Case Only	Indirect Cases Incl.	Direct Case Only	Indirect Cases Incl.		
1.0	1056	5797	36.2344	36.2344	36.3397	36.3397	0	0
2.0	1058	5710	71.9232	71.9232	37.2419	37.2419	2	1
3.0	1056	5797	109.513	109.512	35.7271	35.7599	2	4
4.0	1056	5797	144.742	144.742	35.2571	35.2571	0	4
5.0	1056	5797	179.158	179.158	34.8183	34.8183	0	0

Table 5.3: This table displays the results of five experiments recording the path cost of travelling from corner to opposite corner of a tetrahedral mesh when firstly, only direct cases are considered and secondly, where indirect cases are also included in the search The individual tetrahedra were randomly weighted, with a 50% chance of been assigned the value in the β column and a 50% chance of being assigned 5.0. Including indirect cases in the computation multiplies the time taken to find the shortest path by 5.35.

To this end, we performed two experiments. In each case, we used CGAL's 3D Meshing Package [4] to mesh the interior of a 20x20x20 cube, centred on the origin and then found paths from corner to opposite corner of the cube.

In the first experiment, the tetrahedra within the cube were weighted randomly according to a *normal* distribution. This normal distribution has a mean of 0.5 and a σ of $1/6$, thereby ensuring that 99% of the values would be generated in the range between 0.0 and 1.0. The generated values were then multiplied by 16, cast to an integer and multiplied again by 16 so that the range of values lies between 0.1 and 256, separated by increments of 16. The aim of this experiment is to determine whether indirect cases occur in an environment where the weights have a normal distribution and where significant differences between the weights of adjacent tetrahedra can occur. Ten iterations of the first experiment were performed and are tabulated in Table 5.2.

In the second experiment we performed five iterations. 50% of the tetrahedra were weighted with value 5.0, while the remaining 50% were weighted with 1.0, 2.0, 3.0, 4.0 and 5.0 in each of the five experiments, respectively. The aim of this experiment is determine whether indirect cases tend to occur when there are "easier" paths to choose i.e. cells weighted with 1.0. The data from these experiments is tabulated in Table 5.3.

In the first experiment, only two out of the ten iterations showed a difference in path cost. In both cases, using the indirect case produced a slightly shorter path cost, both with 99.5% of the path cost using only the direct case. These two cases are highlighted in grey. In the second experiment, there was no difference in the path cost of all five iterations. The tabulated data also shows that both indirect hyperplane and indirect edge cases do occur. The occurrence of an indirect case means that a node or vertex derived its cost from an indirect case. In the second table row of the first experiment for instance, out of a total of 1056 nodes, only 7 derived their cost from an indirect case.

In terms of running time, only executing Field D* with the direct case took approximately 3.7 seconds, while including the indirect cases in the computation increasing the time taken to approximately 19.8

seconds with minor deviations of up to 0.02 seconds for all test runs - an average extra computation cost of 5.35 times. These results were produced from a numeric implementation of Field D*'s cost function due to time constraints.

From these experiments and data, we can conclude that indirect cases do not occur frequently in 3D, and also do not contribute significantly to the final path Cost. This is an important observation, since implementers of Field D* who are not overly concerned with exact path cost can simply implement the direct case, ignoring the six indirect hyperplane and three indirect edge cases, thereby saving significant computation. Indeed, Ferguson et. al. [49] ignore the indirect case when creating a lookup table in their implementation of Field D* for the Mars Rover.

These experiments show the likelihood of the indirect case occuring in 3D. We note that in higher dimensions, the connectivity of simplices is much higher compared to lower dimensions since they abut one another via their many facets in each dimension. For, example in 3D, tetrahedra may share edges and triangles, while in 4D, a 4-simplex may share tetrahedra, triangles and edges. This accounts for the increasing number of indirect cases. Whether these indirect cases significantly contribute to the path cost in higher dimensions remains to be seen.

5.9 Conclusion

This chapter describes an extension of Field D*'s cost functions to simplices in arbitrary dimensions. The analytic solutions for finding the minimum of these functions are fully provided for the 3D tetrahedral case, expressed in linear algebra. The direct case extends easily to higher dimensions and certain indirect cases do so too. Indirect cases involve projection onto a basis within a Euclidean space $\mathbb{R}^n$, and when the subspace representing this basis represents a hyperplane ($n-1$ basis vectors) or an edge (1 basis vector), analytic solutions exist. We have also documented a range of indirect cases in higher dimensions for which analytic solutions do not exist, specifically for subspaces with between $n-2$ and 2 basis vectors inclusive where $n > 4$.

Extending the cost functions to simplices in higher dimensions allows Field D*'s to solve the Weighted Region Problem on a representation free from geometric error: Simplices decompose polytopes exactly, compared to hypercubes which can, in general, only admit an approximate decomposition.

By providing the complete set of cost functions and their minimizations to tetrahedra in 3D, a full analytic extension of Field D* to 3D has been achieved, improving upon 3D Field D* where only an approximate minimization of the direct case was provided for a cube. These functions allow Field D* to operate on tetrahedral meshes, which subdivide a space partitioned by 3D polyhedra exactly. By contrast, 3D Field D* operates on weighted 3D grids composed of cubes, which, in general, can only approximate 3D polyhedrons.

Experimental evidence in 3D suggests that indirect cases do not contribute significantly to the final path cost. This is important since the indirect cases are more numerous than the single direct case, and increase the algorithm's running time by a factor of 5.35. Consequently, implementers of Field D* interested in maximising computational efficiency may choose to ignore these indirect cases. This has important implications for higher dimensions since the number of indirect cases is $n(2^{n-1} - 1)$ for dimension n, and this rapidly increases with each dimension.

Chapter 6

Creating Distance Fields with Field D*

The distance field, or distance transform, is a fundamental shape operator that describes the shape of an object and how it changes. Consequently, it has many applications such as extracting object skeletons, and producing Voronoi diagrams and Delaunay triangulations. They have also been applied to navigation problems. Distance fields are commonly represented with images or grids. In this chapter we present a technique for calculating an approximate distance field on the nodes of simplicial complexes using Field D*. As discussed in previous chapters, the simplicial complex offers space and time improvements over grid-based representations.

We first discuss literature related to distance fields, focusing on a *penalized volumetric skeleton algorithm* [15, 14] which computes a distance field and medial axis on a voxel grid, using Dijkstra's shortest path algorithm. Next we show how similar strategies can be applied to Field D* to compute distance fields on 2D and 3D simplicial complexes.

Field D* is adapted to perform a Dijkstra shortest path expansion on nodes existing on the boundary of the object for which we want to compute the distance field. This produces a distance field with some artifacts. We show how adding two conditions to Field D*'s *UpdateNode* function greatly reduces these artifacts.

Obstacle avoidance behaviour [111] is an important part of environment navigation as it allows agents or robots to avoid colliding with features within the environment. We show how the distance field that we compute can be used to weight a triangulation in such a way as to induce contour following behaviour from the Field D* algorithm.

6.1 Related Work

Distance maps or the *distance transform* were originally represented as an image [112, 113, 36], where each pixel contains the distance to an object of interest. The distance transform is closely linked to the

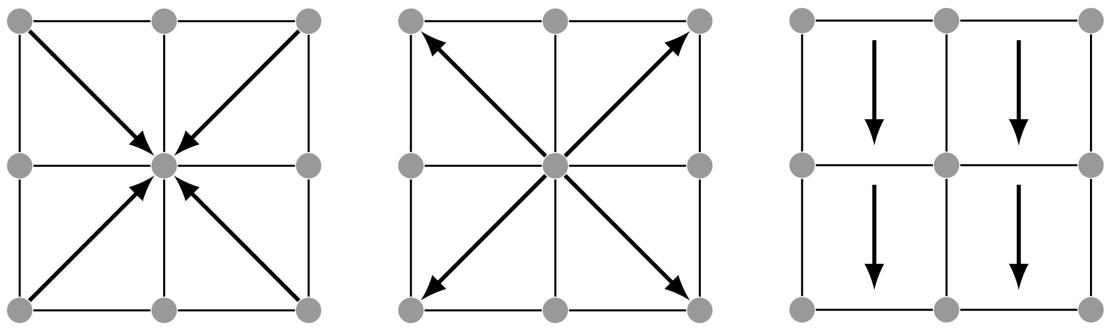


Figure 6.1: Classifying a cubic region according to the gradient vector field. (a) All vectors point towards a local maximum. (b) All vectors point away from a local minimum. (c) A region of uniform gradient.

skeleton and medial axis of an object, since these features occur at the furthest point from an object boundary.

A recent survey of the 2D Euclidean distance transform techniques can be found here [44]. Distance transforms have many applications. They can be used to separate overlapping objects in images. To recover the boundaries between these objects, a distance transform can be performed, followed by a watershed segmentation on the transformed image [145, 33].

They are also used in the computation of geometrical representations and measures. The distance transform can be used to produce an object's *skeleton* [36, 53, 120, 32, 30] as well as *Voronoi diagrams* and *Delaunay triangulations* [145].

Distance maps have also been applied to *robot navigation* [34, 26, 125], but are limited to finding shortest paths within images or voxel grids.

The distance transform is a fundamental operator in shape classification and is used for constructing *Shape measures related to distance* [112, 113, 34]. The maximum of an object's distance transform is its greatest width, for example, and the distribution of the distances within an object is useful for reasoning about it.

Distance transforms can be calculated in 3D; Jones [66] provides a survey of 3D distance field techniques. The shapes for which the distance transform is computed can be represented in a variety of formats such as triangle meshes and constructive solid geometry. Many techniques exist for making the calculation from these representations efficient. However, the distance field itself is still represented with a voxel grid or *adaptive distance fields* (ADF), which are essentially octrees, for efficiency of space representation.

Of particular relevance to our work in this chapter is the *penalized volumetric skeleton algorithm* [15, 14] since it uses pathfinding to find shortest paths from a point in a distance field to the closest boundary. This work utilises Dijkstra's shortest path algorithm [37] to find the shortest path from all voxels to a set of *boundary voxels*. To produce a graph suitable for Dijkstra's shortest algorithm, the authors construct an eight-connected dual graph from the voxel grid, linking adjacent voxels with edges. Computing the shortest path between a voxel and the closest boundary voxel produces a *distance from boundary field* (DBF), which stores path distances within the voxel and is post-processed to improve accuracy. Next, a gradient vector field (GVF) is constructed, representing the gradient of the distance field at each voxel. This is calculated from the DBFs of six neighbouring voxels, connected by faces in 3D. The GVF values of eight-voxel cubic regions are averaged and compared against the average gradient vector in order to classify local maxima, minima and regions of uniform gradient within the GVF, as shown in Figure 6.1. Relevent regions are connected together to form a skeleton or medial axis.

However, the underlying representation of the distance field is a grid, which suffers from the geometric error and resolution issues that were highlighted in Chapter 4. Polygonal boundaries require many grid cells to accurately represent increasing the space requirements of the algorithm, and consequently the running time of the algorithm increases due to the $O(n\, log\, n)$ worst-case time complexity.

6.2 Adapting Field D* to approximate distance fields

We have adapted Field D* to operate on weighted simplicial complexes and specifically constructed simplicial complexes whose members exhibit Delaunay properties. Thus, the simplicial complex representing the environment necessarily sub-samples the environment domain with triangles or tetrahedra. In this section we describe how to construct a distance field on the vertices or nodes of the simplicial complex using Field D*.

This is a useful representation because environments that are not axis-aligned can be regularly meshed with triangles of given sizes. Meshing strategies produce triangles/tetrahedra that follow polygonal/polyhedral boundaries. By contrast, grid representations must subdivide to accurately represent polygonal boundaries, as explained in Chapter 4. This results in high space requirements for representation, which increases the running time of algorithms using it. As a simplicial complex can represent an environment compactly, the representation space requirements and the running times of algorithms using it, are concomitantly decreased.

It is also possible to adaptively sample areas of the domain by adding extra triangles or tetrahedra. This allows algorithms operating on the simplicial complex to achieve greater accuracy in areas of the domain that contain polygons or are sensitive to noise.

6.2.1 Creating the distance field

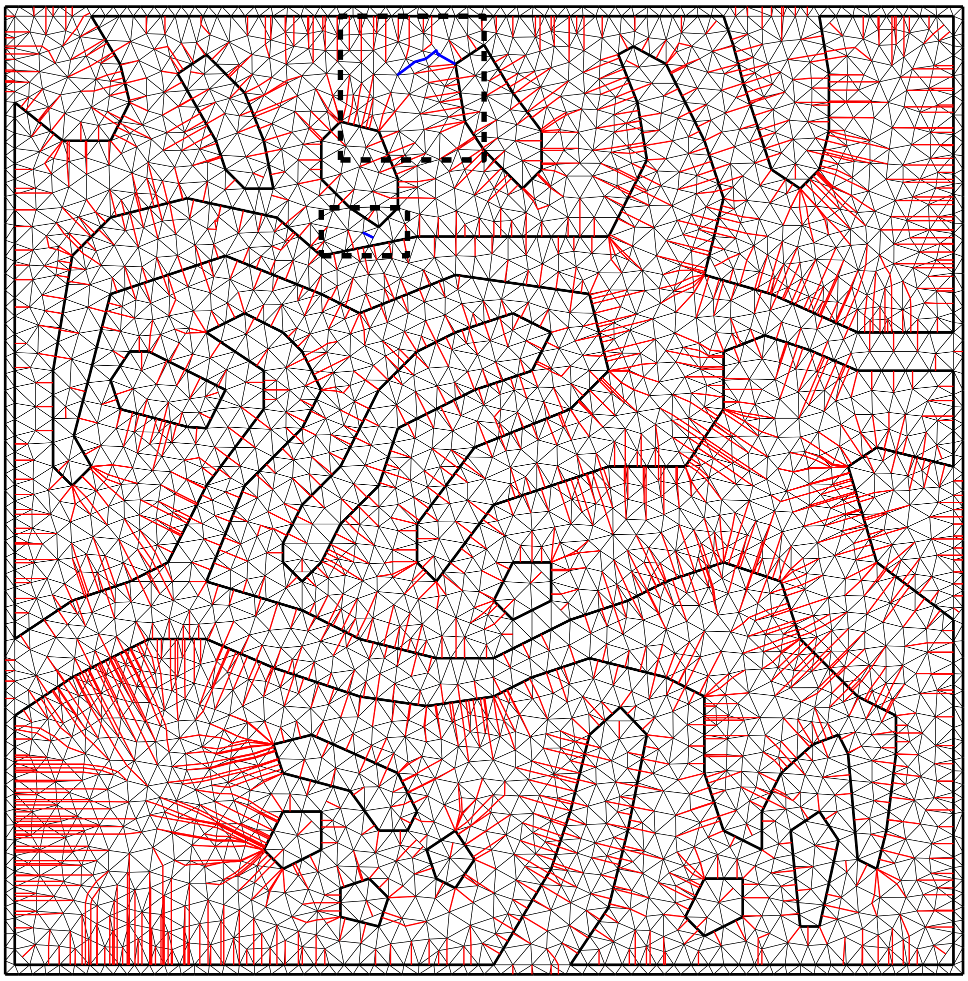
A distance field computes the distance from a boundary. The first step in creating the distance field is to identify all nodes on the boundary. Such nodes are initialised with zero path cost (g-value) and placed on a priority queue.

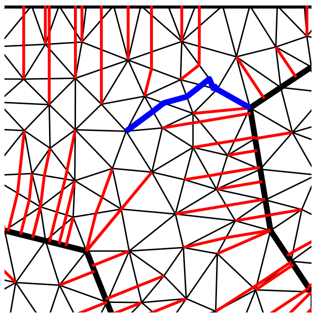
Algorithm 12 Dijkstra-like implementation of the Field D* algorithm. nbrs(s) denotes the neighbouring nodes of u, while connbrs(s) denotes the set of neighbouring node *pairs* surrounding u, $\{(s_1, s_2), (s_2, s_3), \cdots, (s_8, s_1)\}$.

```
function KEY($s$)
    return $[\min(g(s), rhs(s)), \min(g(s), rhs(s))]$
end function
function UPDATENODE($u$)
    if $s$ was not visited before then $g(s) = \infty$
    end if
    if $u \neq s_{goal}$ then
        rhs$(u) = \min_{s \in \text{trinbrs}(u)}$ComputeCost$(u, s)$
    end if
    if $u \in U$ then U.Remove($u$)
    end if
    if $g(u) \neq rhs(u)$ then U.Insert($u$, Key($u$))
    end if
end function
function DODIJKSTRA
    while U.Size() $> 0$ do
        u = U.Pop()
        if $g(u) > rhs(u)$ then
            $g(u) = rhs(u)$
            for all $s \in nbrs(u)$    UpdateNode($s$)
        else
            $g(u) = \infty$
            for all $s \in nbrs(u) \cup \{u\}$    UpdateNode($s$)
        end if
    end while
end function
function MAIN
    for all nodes $s$ on boundary edges do
        $g(s) = \infty$; $rhs(s) = 0$;
        U.insert($s$, Key($s$))
    end for
    DoDijkstra()
end function
```

All traversable triangles or tetrahedra in the environment are assigned a uniform path cost of 1. The Field D* cost functions will therefore consider the actual Euclidean distance within a triangle, rather than the weighted Euclidean distance. The resulting path cost, or, the $g(s)$ value at node s will then be set to the distance to the nearest boundary point upon completion of the algorithm. Triangles on the interior of a boundary's obstacle are assigned a large value, say 255, to distinguish them from traversible triangles.

Dijkstra's shortest path algorithm is then executed on the priority queue, using Field D*'s cost func-


(a) Basic distance field computed with Field D*


(b) First Artifact

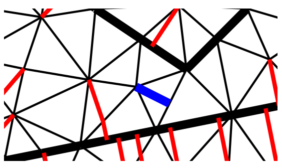
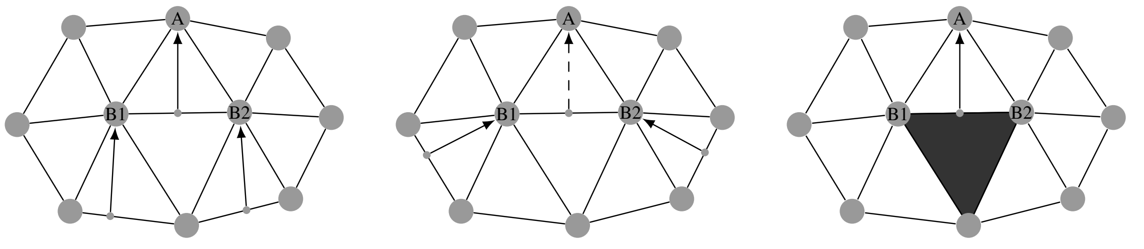

(c) Second Artifact

Figure 6.2: (a) An example of the basic computation of a distance field using Field D*. The red lines show Field D*'s estimate of the shortest path to the nearest boundary. Two problematic cases are highlighted by the dashed boxes. (b) illustrates how Field D*'s interpolation assumption for nodes equidistant from boundaries can produce artifacts in path extraction. The blue path is produced because Field D* interpolates between path costs produced by two different boundaries. (c) An edge whose nodes belong to two boundaries also results in Field D* assuming that the best point to traverse from lies between them, since they both have zero path cost.



(a) Valid **(b)** Invalid **(c)** Valid

Figure 6.3: Checking the child vector directions to decide whether an interpolation estimate is valid. (a) The directions that B1 and B2 derive their estimates from are similar and thus the angle between them is less than $90°$. In this case it is valid to derive costs from this triangle. (b) If however, the directions are dissimilar and the angle between them is greater than $90°$, we do not derive costs from the triangle. (c) If B1 and B2 have zero path cost, then $AB1B2$ must have a weight of 1, and its opposing triangle a weight of 255 for derivation of costs from this triangle to be valid.

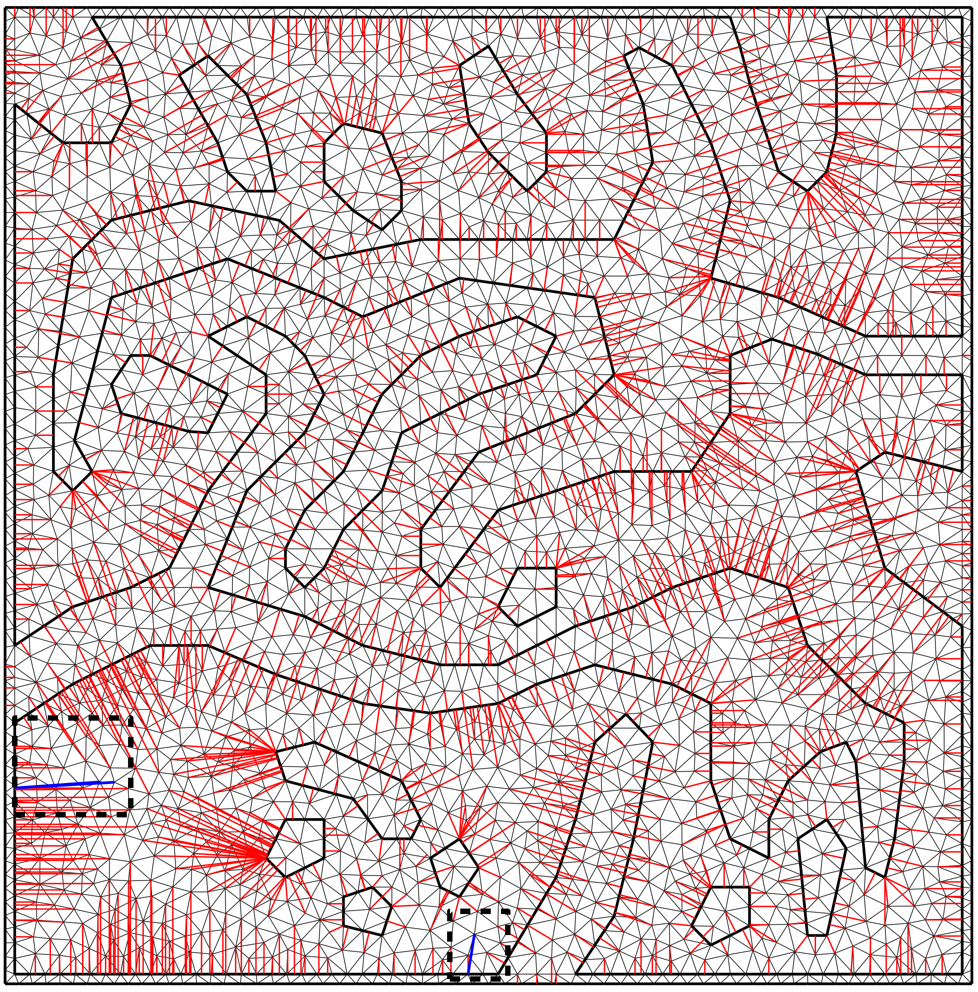
tions to evaluate the least cost path to each node until the priority queue is empty, and shortest distances to each node have been computed. This is similar to the way the penalized volumetric skeleton [15] uses Dijkstra's shortest path to calculate an approximate distance field on the dual graph of a voxel grid. However, Field D* allows this computation to be performed on a simplicial complex.

The Field D* algorithm, described in Chapter 3, and Chapter 4 can be used to perform Dijkstra's algorithm with minor modifications. This modified version is shown in Algorithm 12. Similarly to the way a single goal node is placed on the priority queue in anticipation of finding a shortest path to the start node, boundary nodes placed on the priority queue should have their rhs-values initialised to zero, and their g-values initialised to ∞, in the *Main* function. The heuristic value for each node can be effectively ignored by setting it to zero, since we are not directing the search towards a specific node, but rather expanding all nodes in the environment until their g-values and rhs-values are *consistent*[1]. The *Key* function therefore becomes simpler. Once all the boundary nodes have been placed on the priority queue, the *DoDijkstra* can be executed to expand all the nodes in the complex until the priority queue is empty.

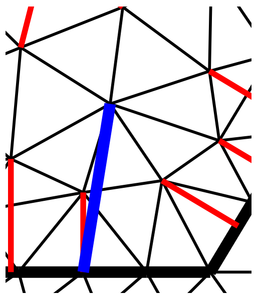
Once the Dijkstra expansion has been executed, path extraction must be performed for each node in the simplicial complex to find the shortest path to the boundary. Algorithm 11 described in Chapter 4 can be applied to each node, with a slight adaptation: Instead of extracting points on the path until we reach the start node, we extract nodes until we obtain a point with zero path-cost (g-value). Thus, the extracted paths trace back from the node under consideration to a point on the boundary. It is important to highlight that this final point is not necessarily a node: It can also be an point on the boundary due to Field D*'s use of interpolation.

An example of the distance field created by this process can be seen in Figure 6.2a, which illustrates Field D*'s estimate of the shortest path from a node to the nearest boundary point. The paths from

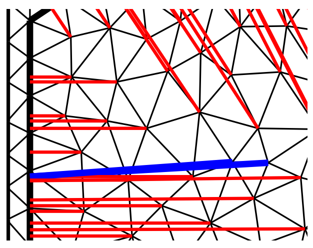
[1]See Chapter 3 Section 2.2.3



(a) Improved Distance Field



(b) First Artifact



(c) Second Artifact

Figure 6.4: (a) A distance field computed using the two extra conditions imposed upon the Field D* *UpdateNode* function. Some of the field lines in (b) and (c) are still not orthogonal to their respective boundary.

each node are shown in order to two problematic cases. Firstly, Figure 6.2b shows a particularly erroneous path extraction caused by Field D*'s interpolation assumption. In this case, Field D*'s cost function has assumed that it should interpolate the path costs – the distance from a boundary – produced by two completely different boundaries. This results in a path that initially follows the medial axis before turning to a boundary. In practice, one can simply connect the initial node to the final point on the boundary and in the provided example, this would produce a good distance field that is orthogonal to the boundary. However, in other cases, most obviously visible in the lower left hand corner, this is not the case.

Secondly, in Figure 6.2c, a single edge connects two boundaries. As they are both on separate boundaries, they have zero path cost. Therefore, Field D*'s cost function assumes that the interpolated path cost along the entire edge is zero. Consequently, the cost function ends up only minimising the distance component, which, by definition produces a path from the node, down the perpendicular to the triangle edge to a point between each boundary.

These two cases occur during the Dijkstra expansion when distance values are propagated outwards to the nodes of the complex. To prevent their occurrence we check two conditions to decide whether a triangle is a valid neighbour.

- Firstly, when considering whether to call *ComputeCost* on node A at the apex of a triangle $AB1B2$, we check the nodes over which interpolation occurs, $B1$ and $B2$, to see whether they derive their costs from similar *directions*. To accomplish this, we consider the points from which $B1$ and $B2$ derive their cost, $P1$ and $P2$, for instance, and take the dot product, $(B1-P1)\cdot(B2-P2)$. If $(B1-P1)\cdot(B2-P2) \leq 0$ then the angle between the two vectors is greater than 90 degrees and we ignore $AB1B2$ when computing the path cost at A. The idea is similar to the penalized volumetric skeleton's [15] classification of cubic regions of uniform gradient. This condition ensures that interpolation only occurs between nodes deriving their costs from similar gradients as in Figure 6.3a.
- Secondly, if both $B1$ and $B2$ have zero path cost, and the two triangles shared by edge $B1B2$ both have the initial uniform cost of 1, then we ignore $AB1B2$. This ensures that interpolation does not occur between two zero path cost boundary nodes from different boundaries. It does, however, permit interpolation between zero path cost boundary nodes on the same boundary, since the triangles shared by such nodes will have values of 1 and 255, respectively, as in Figure 6.3c.

If these two conditions do not hold, then the triangle is not considered to be a member of the set trinbrs(u) in Algorithm 12 and *ComputeCost* is not called. Implementing these two extra conditions eliminates the most egregious problems with our distance field technique. Connecting each node to its corresponding boundary point, as opposed to displaying the full path extraction for each node produces Figure 6.4.

The distance field was computed in 0.01 seconds on a mesh of 3086 vertices and 5916 triangles. To estimate the accuracy of the distance field, a brute-force approach was used to find the actual distance to the closest boundary: the shortest distance between a node and all boundary edges were calculated.

This took 0.63 seconds to compute. Then, the Normalised Root Mean Square Deviation (NRMSD) between the path distances produced by Field D* and the brute force approach were calculated. Thus, given a set of b_i $i \in 0 \ldots, n$ brute forced distances at n vertices, we calculated the NRMSD against a set of d_i $i \in 0, \ldots, n$ Field D* distances using the following:

$$\text{NRMSD} = \sqrt{\frac{\sum_0^n (d_i - b_i)^2}{n}} \times \frac{1}{d_{\text{max}} - d_{\text{min}}} \tag{6.1}$$

where d_{max} and d_{min} are the maximum and minimum distances estimated by Field D* respectively. The NRMSD of the mesh was 0.00656. Expressed as a percentage, the residual variance is 0.65%, indicating a low level of error in Field D*'s distance field approximation.

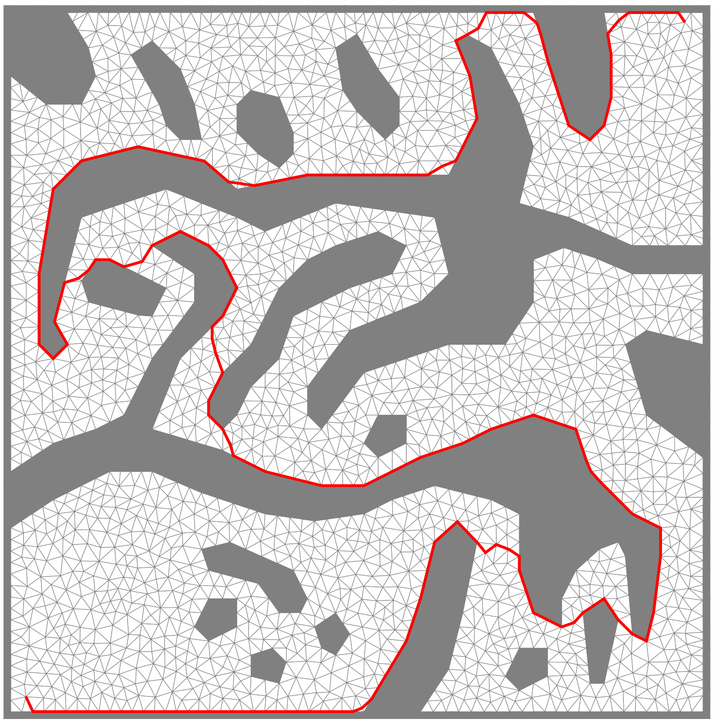
Some of the distance field lines are not completely orthogonal to their corresponding boundaries: This is because implementing the first condition prevents Field D* from interpolating through certain triangle edges, and thus the resulting paths traverse around them. The penalized volumetric skeleton [15] shares similar artifacts during path extraction and post-processing is used to fix this [117]. We leave this for future work as our creation of a distance field using Field D* is exploratory in nature.

6.3 Weighting the triangulation with distance

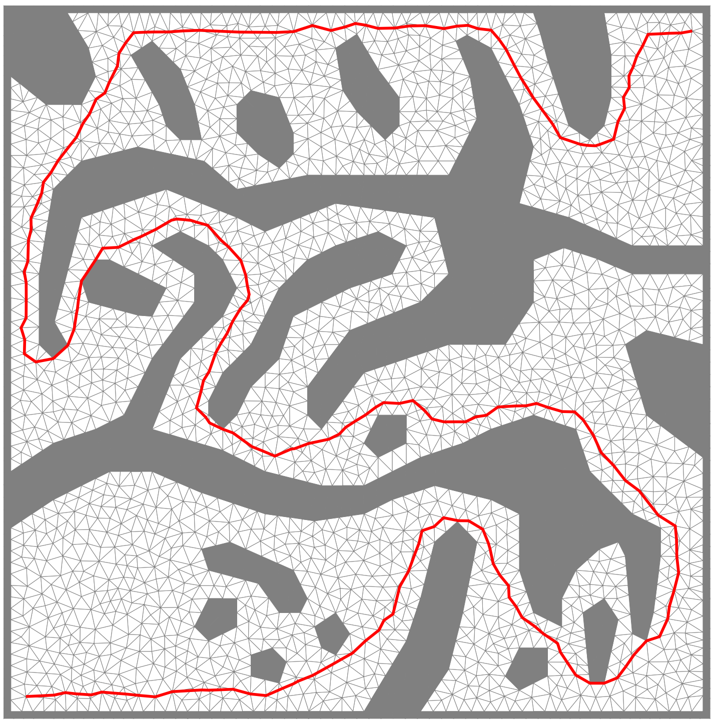
The distance field on the simplicial complex nodes, produced above, is useful for weighting the triangulation to produce different behaviours in the Field D* algorithm. We show how contour following behaviour can be induced from this weighting.

Firstly, we select a particular contour distance, c, that we wish Field D* to favour. Then for each triangle, we average the distances at the triangle nodes to produce a weight, a. We assign the final triangle weight with the following expression – $\lambda = \text{MAX}(\|a - c\|, 0.1)$ – to assign the triangle with a non-zero distance from the contour. We then run Field D* on the resulting triangulation. When $c = 0$, we obtain wall hugging behaviour, as demonstrated in Figure 6.5a. Setting $c = 1$ produces contour following behaviour, shown in Figure 6.5b

In narrow sections of the world that are covered by only one or two triangles across the breadth of the corridor, there is insufficient resolution in the mesh for the algorithm to represent the desired contour, and this results in the path adhering to walls. This could be solved by introducing greater levels of subdivision in these areas, or by extending the Field D* cost functions to cater for a *barycentric weighting* across the interior of the triangle, rather than the uniform λ weighting. The latter approach is discussed further in Chapter 7.



(a) Wall hugging



(b) Contour following

Figure 6.5: (a) Wall following behaviour, produced by weighting triangles close to contour 0 with a cheap weight. (b) Contour following behaviour, produced by weighting triangles close to contour 1 with a cheap weight.

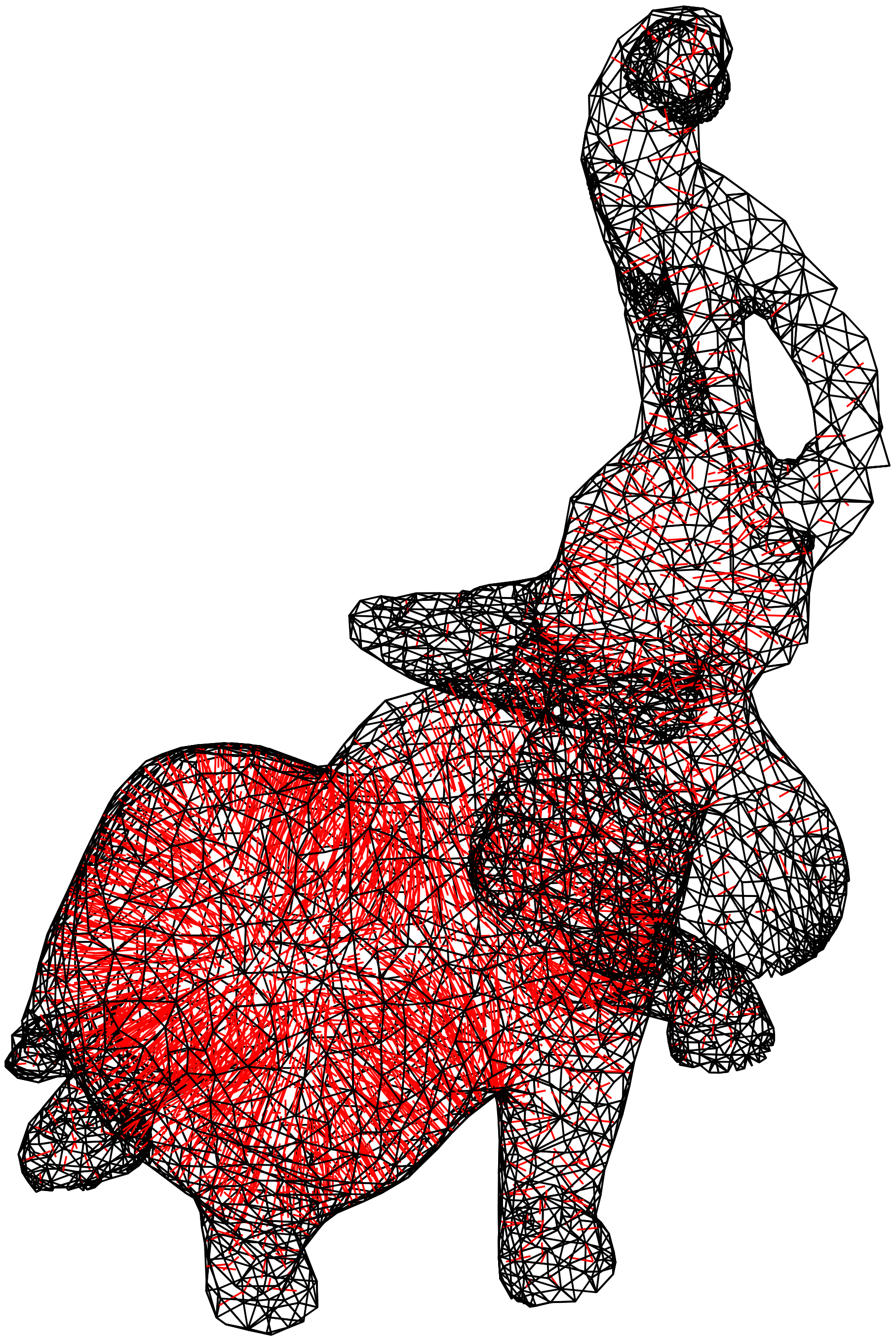


Figure 6.6: The distance field of an elephant mesh

6.4 3D distance fields

The distance field technique described in Section 6.2.1 can be adapted to 3D. The additional two conditions are modified as follows:

- The first condition now checks whether it is possible to update a node A on tetrahedron $AB1B2B3$.

In the 2D case only two vectors needed to be compared against each other. In the 3D case, three vector dot products must be formed for the three, $(B1-P1)\cdot(B2-P2)$,$(B2-P2)\cdot(B3-P3)$ and $(B3-P3)\cdot(B1-P1)$. If any of these dot products is less than or equal to zero, the tetrahedron should be ignored.

- The second condition changes to: If at least two of $B1$,$B2$ and $B3$ are zero, then the weights of the tetrahedrons sharing facet $B1B2B3$ must be checked. If they are both 1, then the facet is not a boundary facet as it lies between at least two other boundaries. Thus, it too should not be considered in this case.

We show an example of a 3D distance field computed on a elephant mesh in Figure 6.6. The mesh is composed of 6557 vertices and 41351 tetrahedra, while the distance field was computed in 0.42 seconds. Similarly to the 2D example, we calculated the shortest distance between a node and all mesh boundary triangles using a brute force method and compared this actual distance to that computed by Field D*. The brute force method took 9.7 seconds to complete and the NRMSD was 0.66%, indicating low amounts of error in 3D Field D*'s approximation.

6.5 Discussion and Future Work

The distance field that we have computed on a simplicial complex offers savings in terms of space, compared to a distance field on an image. This is analagous to the space benefits offered by our extension of Field D* from weighted grids to weighted simplicial complexes.

The triangulation that we use to present our work in the previous sections is composed of 3086 vertices and 5916 faces. If we store a floating point distance at each node, and six integers to store the connectivity information for each face, we require a total of $3086 * 4 + 5916 * 6 * 4 = 154,328$ bytes to store the distance field. By comparison, if a 512x512 image of floating point distances is used for the representation, $512^2 * 4 = 1,048,576$ bytes is required. We can do a similar calculation for 3D: The elephant mesh was composed of 6592 vertices and 41409 tetrahedra. Using 8 integers to store connectivity information, we require a total of $6592 * 4 + 41409 * 8 * 4 = 1,341,456$ bytes, while a 512x512x512 voxel grid would require $512^3 * 4 = 12,582,912$ bytes.

In terms of running time, the *penalized volumetric skeleton algorithm* would have to expand $512^2 = 262,144$ nodes to compute a distance transform, compared to our customised Field D*'s expansion of 3086 nodes. Both algorithms perform a Dijkstra type expansion and run in $O(n\,\text{log}n)$ time, but the n factor is significantly lower for the customised Field D*.

Additionally, the distance within a triangle can be interpolated over the distances stored at the node corners using a linear weighting scheme, giving the ability to approximate the distance field within triangles. We note that this interpolation would only be valid if the distance fields related to the nodes shared a uniform gradient, similar to Figure 6.1c. Such interpolation would not be valid for triangles containing local maxima and minima (Figures 6.1a and 6.1b), but in these cases, subdividing the triangle by introducing a node where the distance fields intersect, might solve this problem. We leave this for future work.

Future work can expand the exploratory nature of the work in this chapter. The distance field that we have calculated is approximate in the sense that it does not guarantee that the distance to the boundary is exact. In particular, we have documented cases where subtle artifacts still exist and the line between the simplex node and the boundary point is not completely orthogonal. It should be possible to fix these cases by locally perturbing the point until it's position is completely orthogonal to the simplex node. However, the overall error in the distance field, represented by Normalised Root Mean Square Deviation is less than a percent for both the 2D and 3D examples presented.

We also note that once the distance field has been computed, it is possible to extract contour lines in 2D and iso-surfaces in 3D, by interpolating along edges to find the appropriate contour values.

6.6 Conclusion

In this chapter we have described how a strategy, similar to that employed with voxel grids in the *penalized volumetric skeleton algorithm* [15], can be applied to Field D* in order to calculate an approximate distance field on a simplicial complex in two and three dimensions. The Field D* algorithm is modified to perform a Dijkstra's shortest path algorithm on the vertices of the simplicial complex. Paths across certain triangles are ignored when performing this expansion in order to avoid serious artifacts in the distance field.

Once Dijkstra's algorithm has been applied, Field D*'s path extraction is performed for each node in the complex. This produces a path from each node to a corresponding point on the boundary. Such boundary points are not restricted to vertices of the complex and may be interpolated across a triangle edge, or tetrahedron face. In both a 2D and 3D example, the Normalised Root Mean Square Deviation in Field D*'s approximation was less than a percentage point, indicating a low level of error.

A distance field on a simplicial complex saves space over a grid-based distance field, since simplicial complexes do not require a grid's high level of subdivision to accurately represent polyhedral boundaries. This saving in space positively impacts the time complexity of our algorithm, since far fewer nodes need to be considered in a complex, compared to the nodes in a grid.

We have also shown how this distance field can produce wall hugging and contour following behaviour. This functionality is useful as obstacle avoidance is an important part of path-planning.

Chapter 7

Conclusion

The development of shortest path algorithms to navigate through environments is a continuing area of research. When paths through environments can be logically decomposed into a collection of routes and intersections, such environments can be represented by a *graph*. Edges of the graph can be assigned a cost, representing the expense of traversing a route between two graph nodes. Algorithms such as Dijkstra's shortest path and A* operate on these graphs to find paths with a minimal summed edge cost. Thus, different routes can be favoured or avoided by weighting graph edges with some metric associated of the environment. The distance travelled along a route is the most frequently used, but other measures such as traffic congestion can also be used.

Such formulations are useful and commonly applied when routes through an environment can logically be mapped to a graph, such as a road network. In other cases, this mapping is not trivial. When environments are represented by a set of *weighted regions*, a continuous range of paths can exist between them. The Weighted Region Problem (WRP) formulates the challenge of finding the least cost path between two points in a weighted planar polygonal subdivision. Steiner point techniques introduce extra graph nodes on region boundaries, as well as edges connecting these new nodes, but this requires pre-processing of the graph and space to store extra edges and nodes.

By contrast, the Field D* algorithm takes the approach of minimising a cost function representing a range of paths across a weighted square cell to a node on the cell corner. Field D* then finds shortest paths through a grid of weighted cells and the resulting path is able to travel through cells. Grids are a simple and easy representation to work with, but suffer from resolution issues when representing polygonal structures, since subdivision of the grid is required to represent polygons to within an error tolerance. Multi-resolution grids can ameliorate this to some extent by aggregating smaller, similar cells into larger cells. However, subdivision is still necessary on polygon boundaries, since it is not possible to aggregate dissimilar cells. Consequently the Field D* solution to the WRP is subject to geometric error, since grid subdivision cannot, in general, represent a polygon exactly.

Triangulations, by contrast, represent polygons both accurately and compactly. For example, Triangulated Irregular Networks (TINs) are favoured over image-based height maps of Digital Elevation Models (DEM) in the field of Geographic Information Systems since they can accurately model environment boundaries, leading to savings in the space required for representation. A triangulation is a 2D *simplicial complex*, a mathematical structure which generalise to a mesh of tetrahedrons in 3D, and general simplices in higher dimensions.

The core contribution of this thesis is an extension of the Field D* [49] algorithm to *weighted simplicial complexes*, in order to take advantage of the space and time savings associated with this representation. This also allows Field D* to solve the WRP on a representation which is free from geometric error. Polygons can be exactly decomposed into a finite number of weighted triangles. Similarly, polytopes can be exactly decomposed into a finite number of simplices in higher dimensions. The same does not apply to squares in 2D, or hypercubes in general.

We extended Field D* to 2D triangulations and 3D tetrahedral meshes by adapting the algorithm's cost functions to triangles with an efficient and compact linear algebra formulation. This differs from Generalized Field D* [119] which uses a formulation based on trigonometric relationships and length ratios of a triangle, and which performed limited experiments to justify the extension to triangulations. We showed our formulation offers up to a 56% performance improvement over this technique. We also performed extensive testing showing how such a triangle extension offers significant benefits in terms of algorithmic time and space requirements, in comparison to *Multi-resolution Field D**.

The reasons these benefits acrue are clearly illustrated by our experiments. Grids and multi-resolution grids require a large degree of subdivision to represent a polygon with a geometric error bound. By contrast, we found that an order of magnitude fewer triangles than quadtree cells produced similar path costs, since triangles can represent polygons exactly. This reduction in elements, or space, leads to a reduction in running time, since between 10 and 20 times fewer node expansions are required to calculated a shortest path. Indeed, reducing the number of faces and vertices – F and V – directly effects Field D*'s worst case performance of $O(F+V\log V)$. We also identified cases in which Field D*'s cost functions can be precomputed and cached, offering modest performance improvements of up to 16% of running time.

Building upon this 2D mathematical and theoretical basis, we proceeded to extend Field D* to 3D tetrahedra and simplices in higher dimensions. This improves upon 3D Field D* [23] which only approximates a *direct* cost function on a 3D grid. Our formulation adapted the vector mathematics used in our 2D extension to express the Field D* cost functions with linear algebra. We provided a complete set of analytic solutions in 3D. Next, we documented a range of cases that exist in higher dimensions that are not analytically solvable, and showed how, in 3D, these cases do not contribute significantly to the final path cost. We also presented results for 3D pathfinding queries through medical data, as well as a simulated ocean environment.

Finally, we demonstrated how Field D* can be used to create an approximate distance field on 2D and 3D simplicial complexes. A simplicial complex is a more compact representation than the typical voxel grids or octree distance field representations. We adapted Field D* to perform a Dijkstra-like algorithm, propagating distances from relevent environment features to the nodes of the simplicial complex using Field D*'s cost functions. In both a 2D and 3D example, the Normalised Root Mean Squared Deviation between Field D*'s approximation and the actual distance field was less than 1%. We also showed how this distance field is useful for producing contour-following and obstacle-avoidance behaviour in the Field D* algorithm by weighting triangles appropriately.

Our results demonstrate the importance of selecting a good *representation* for the Field D* algorithm to operate on. While grid representations are simple to work with, they require subdivision to represent environment features with low geometric error. Due to this, the space requirements for the algorithm, represented by the number of nodes in the environment, is high and consequently increases the running time of the Field D* algorithm. By contrast, a *simplicial complex* can represent environments more compactly, lowering the space requirements and, by implication, the running time of the algorithm.

Furthermore, triangulations and tetrahedral meshes are frequently used to represent environments. TINs represents terrain data, and triangulations are the dominant environment representation in computer graphics and visual effects. Triangulations and tetrahedral meshes are frequently used in Finite Element Methods (FEM) [55] to represent the domain of computation, and in the field of Medical Imaging to represent the various tissues, bones and organs within a body [46, 63]. Many packages exist that efficiently mesh such domains. Examples include *Tetgen* [126] and the *Computational Geometry and Algorithms* (CGAL) library's 3D meshing package [4].

Simplices are also easily subdivided into other simplices. For example, by placing a new node on a triangle, a subdivision of the triangle into other triangles can be achieved by connecting it to the other triangle nodes. As Field D* must consider interpolated points on triangle boundaries during path extraction, triangle cost functions can be applied during the extraction process. It is also possible to use this subdivision strategy to place temporary start and goal nodes at arbitrary locations *inside* triangles. However, this is not necessarily the case for grid squares or cubes, which, to be subdivided into squares and cubes, must have a node inserted precisely at their centre. Therefore, the cost functions for these elements cannot be used during path extraction.

To summarise: We have extended the Field D* algorithm to a space-efficient representation – the simplicial complex – commonly used to represent complex domains in a wide range of computationally challenging problems. This extension does not merely benefit from savings in terms of space: By reducing the space requirements, the running time of the algorithm is also reduced. Additionally, the simplicial complex allows Field D* to solve the Weighted Region Problem on a representation free from geometric error.

7.1 Future Work

Field D*'s cost functions dominate computation and a number of possible options to reduce this are discussed. Simplicial complexes are also amenable to *adapative meshing*, whereby areas of the mesh relevent to the solution can be finely subdivided to reduce computation. The Field D* cost functions that we have presented assume constant weighting of a simplex; a linearly interpolated barycentric weighting may be appropriate for certain applications. Finally, our Spatial Awareness Framework is a good candidate for further development.

7.1.1 Accelerating Field D*'s cost functions

Field D* introduces a set of cost functions that must be minimised to calculate the shortest path across a triangle or a tetrahedron. By contrast Steiner point techniques discretise the triangle by introducing extra edges between additional Steiner points added on the triangle borders. Field D* emphasises computation over memory access, and one of the subtle advantages of Field D* is that the CPU speed of modern computers is fast eclipsing the speed of memory access. Nevertheless, evaluating cost functions is a dominant part of the algorithm, consuming up to 80% of the computation in our benchmarks, and strategies to reduce cost function evaluation time would be useful.

One simple approach would be to implement the cost functions using the standard *Streaming Single Instruction Multiple Data* (SIMD) *Extensions* (SSE) on modern CPUs. SIMD instructions process vector operations in parallel, and are thus good candidates for evaluating Field D*'s cost functions, which we have framed using linear algebra.

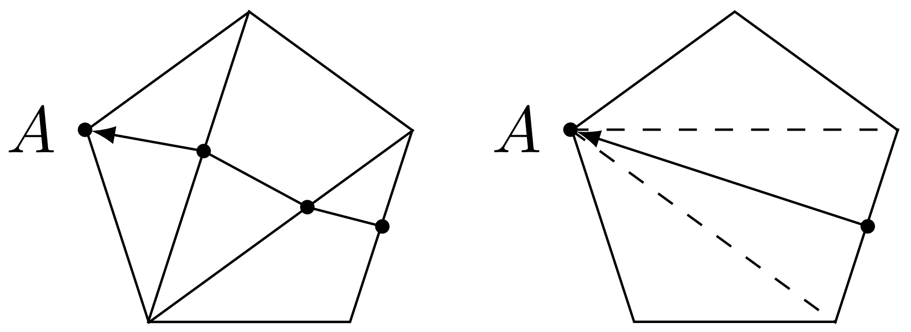
Another possible approach would be to implement all or part of the algorithm on a *Graphics Processing Unit* (GPU). GPUs also use a SIMD approach by applying a single *kernel*, or program, to multiple data points. The difference between GPU and CPU SSE instruction sets is the degree of parallelism exhibited by a GPU, since hundreds of threads may execute the same kernel, whereas SSE instruction only execute four operations concurrently.

Thus, GPU's have the potential to compute many of Field D*'s cost functions in parallel. In our experience, implementing many parallel cost functions on a GPU is not a difficult task, but integrating this cost function calculation with the priority queue driving the algorithm is challenging. A priority queue is not an easily parallelised data structure and, if naïvely implemented on a GPU, is a bottleneck in the algorithm's performance.

An alternative approach is to implement the priority queue on the CPU and leave the calculation of the cost functions to the GPU. This approach is relatively simple, but the latency of the PCI-Express bus between the CPU and the GPU then impacts the algorithm's performance. Modern GPU's specifically offer methods for interleaving memory transfers and kernel executions however, and it may be possible to ameliorate this latency, by interleaving the cost functions of multiple searches.

We note that a GPU implementation of the A* algorithm [16] exists, and while this implementation does obtain speedups of up to 50X, they are obtained by performing many searches in parallel on relatively small graphs of up to 340 nodes and 2150 edges.

7.1.2 Polytope Subdivision



(a) Static sub-division **(b)** Temporary sub-division

Figure 7.1: Travelling to node A through a weighted complex may end up being more complicated with static sub-division since there are more triangles to travel through. By contrast, temporary sub-division may make this less complex and possibly shorter.

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

The advantage of this approach over a sub-division into static triangles (Figure 7.1a) is that temporary subdivision may result in longer path segments (Figure 7.1b) and slightly shorter paths since the path to node A has to travel through fewer triangles. The disadvantage is that the number of neighbouring nodes that must be updated by the *UpdateNode* function will increase, potentially increasing the running time of the node expansion phase of the algorithm.

7.1.3 Adapative Mesh Refinement

Adaptive mesh refinement is frequently used in *numerical analysis* techniques, such as *finite element methods* (FEM) to save both space and time during computation. For example, [71] models the collapse and fragmentation of a stars in a molecular cloud. The vast distances involved preclude the use of a uniform grid, and so the authors implement a multi-resolution grid which is only refined in regions containing stellar gas. Similar concepts apply to triangulations: *Delaunay triangulation and 3D adaptive mesh generation* [55] describes a general adaptive meshing technique for 3D tetrahedral meshes.

High levels of subdivision reduce the interpolation error inherent in Field D*, but the path cost decreases too slowly to justify the increase in space and time requirements. Utilising the adaptive meshing paradigm, one can calculate an approximate shortest path on a coarse triangulation and adaptively refine portions of the triangulation involved with the path. Similar ideas have been applied to grids: *Partial Pathfinding Using Map Abstraction and Refinement* [131], for example, builds an abstraction hierarchy from a grid which is refined during path-planning operations. As a further enhancement, it may be possible to propagate heuristic estimates to nodes of the refined mesh from path costs of the coarse mesh.

7.1.4 Field D* on Barycentrically Weighted Simplices

The current formulation of the Field D* cost functions only allows a uniform weighting of the simplex with some value λ. *Finite Element Methods* allow for a barycentric weighting of the interior of a simplex, derived from values stored at the vertices of the simplex. For example, in our distance field technique, the distances stored at the vertices of a triangle can be barycentrically interpolated over the triangle's interior, instead of averaging these distances and weighting the triangle with a uniform average distance.

Converting Field D*'s cost function to a formulation involving barycentric coordinates could provide this functionality, since this coordinate system can elegantly represent the linear interpolation within the simplex. It may be possible to formulate an *isoparameteric* simplex, which has easy convertability between barycentric and cartesian coordinate systems.

The benefit of a barycentrically weighted simplex is that less subdivision of the simplicial complex is required to represent a gradated weighting of sections of the environment. For example, when using our distance field to weight a simplicial complex, we averaged the node distances at the corners of each triangle to create a uniform distance weight for the triangle. However, it would be more accurate to plot paths across a distance gradient. This can be applied to many other measures such as temperature and lighting for instance.

7.1.5 Field D* Pathfinding in Higher Dimensions

This dissertation has focused on pathfinding in 2D and 3D, as these are the standard Cartesian spaces in which pathfinding takes place. The cost functions that we have developed have been specified with linear algebra, and can thus be applied in higher dimensions.

Applications for pathfinding in 4D exist. [84] represents blood vessels in a 4D space: The first three dimensions are the standard x, y, z coordinates while the fourth represents the *radius* at that coordinate. This coordinate system specifies the 3D surface of a blood vessel as a 4D curve, through which a minimal path is found. [87] plans shortest paths for vehicles in a 2D plane while incorporating the vehicle's turning and velocity constraints into the final path. Cartesian coordinates x, y, as well as angle and velocity θ, v form a 4D coordinate space. Points in this 4D space are connected in a graph, on which the shortest path is performed.

Another example, *Cooperative A** [128] provides cooperative pathfinding between multiple agents within an environment by giving agents with the positions of other agents *in time*. The time dimension is incorporated with a 2D cartesian coordinate grid to form a 3D grid. Shortest paths across this grid are calculated from an x, y coordinate in time step 0 to a coordinate in time step n, where n is the dimension of the time coordinate. The cells that the path traverses in the grid are marked as impassable, reserving the range of *coordinates in time* occupied by the path so that subsequent path queries on the grid will avoid it. *Windowed Hierarchical Cooperative A** (WHCA*) [128] incorporates an improved heurisitic for handling such searches and a windowing strategy to limit the search domain. *Improving Collaborative Pathfinding Using Map Abstraction* [132] combines WHCA* with *Partial Refinement A** (PRA*) [131] to form the Cooperative PRA* (CPRA*) algorithm, which uses hierarchical environment abstraction to improve search efficiency.

A similar cooperative pathfinding strategy can be constructed with Field D* to allow multiple agents to find paths within a 3D space. A 3D tetrahedral mesh can be created by subdividing the space according to some meshing strategy. Then, the dimension of the mesh can be *raised* by adding a fourth dimension, time, to produce a 4D mesh of 4-simplices on which Field D* searches can be performed. 4-simplices occupied by such paths can be weighted with some value representing the degree to which an agent occupies the 4-simplex, thereby making the 4-simplex more expensive for other agents to travel through.

7.1.6 Spatial Awareness Framework

The 2D Spatial Awareness Framework documented in Appendix A was developed as a prototype to explore how qualities of space can aid autonomous agent behaviour within that space. This framework is useful for analysing and representing these qualities, giving agents an understanding of the space in which they operate. It can also be used to weight regions within that space for use by Field D*. For example, regions within areas of high curvature can be weighted expensively.

Due to time constraints we have not achieved an integration of this framework and Field D*, or extended the concepts of this framework into 3D. A denouement to these avenues of work still remains.

Appendix A

Prototype Spatial Awareness Framework

This appendix introduces an exploratory Spatial Awareness Framework used to investigate region-based analysis of environments, for the purposes of providing *Artifical Intelligence* (AI) agents with a sense of the qualities of the space that they operate in. We implement simple agents that use this framework in a bottom-up manner to tailor their behaviour to the local area of space.

This follows from the field of *Nouvelle AI* which, by the *Physical Grounding Hypothesis* [21], proposes that systems must be grounded within a physical environment to produce intelligence. Thus, to behave intelligently according to this paradigm, AI agents must be able to react to the conditions within their environment. For example, straight and gently curving corridors of space are appropriate for running, while spaces with sharp corners are not. Wide-open spaces provide good vantage points for observing other objects, but are difficult to hide in. By contrast, enclosed, dimly lit spaces are good hiding places.

In order to create this form of intelligence, agents must be provided with an understanding of the space in which they operate. It would be appropriate for an agent representing a bird to fly in wide-open spaces, but hop or walk in narrow, enclosed spaces. An agent behaving in a “sneaky” manner may favour dimly lit areas with poor visibility to more exposed areas. To provide this information to agents, it is necessary to perform some form of spatial analysis on the environment.

Creating an abstract graph representing an environment is a form of spatial analysis specifically designed to assist the navigation of agents within a environment. Graphs or navigation meshes can be constructed representing the space within which an agent navigates. Graph edges are frequently weighted with characteristics pertaining to the routes they represent, so that shortest path algorithms such as Dijkstra’s shortest path [37] and A* [61, 101] can find paths that minimise certain criteria.

Such types of spatial information are extracted during the analysis of an environment, and while such information may not be specifically extracted with agents in mind, it may still be useful to them. For example the visibility between different points in a environment allows one to avoid rendering invisible environment structures [1, 136, 89, 83]. This information may also be useful to an agent wishing to strategise about the suitability of a vantage point.

Similarly, the lighting in various areas of an environment is calculated to improve the realism of a rendered image [29, 57], but this information may also be useful if agents are designed to react to lighting information. Some agents may see better in the dark and should favour dark areas to gain an advantage over opponents with poor vision.

These areas have been well researched, but less work has been done on providing information about the *intrinsic* qualities of a space. For example, the way a space curves is useful to an agent when planning when to accelerate and decelerate, especially if its turning behaviour is physically modelled. The width or openess of a space is also a useful measure for evaluating proximity to environment structures and strategising about wide or narrow spaces.

We present a novel data structure that provides useful data on higher-order connectivity, curvature and width. We also describe the process for automatically generating this structure from a environment defined by 2D polyons. The data from this structure can be combined with data extracted from other sources, such as visibility and lighting. We implement simple agents and demonstrate how their effectiveness can be improved by utilising this data in two different scenarios. In the first, racing car agents use data about track curvature to improve their racing line. In the second, "battle" robots use data about path intersections to orientate themselves towards areas of high traffic. The agents use almost no planning capability since our intention is to produce bottom-up behaviour based on the environment they occupy.

This appendix is structured as follows: We describe previous work and background material relevent to our work in Section A.1. Section A.2 provides a description of the framework as a pruned medial axis, with linear mappings between it and the environment boundary. Section A.3 describes the creation of this data structure, while Section A.4 describes the agents that used to test the framework. Finally, Section A.5 describes our testing and results and we conclude in Section A.6.

A.1 Related Work

Binary Space Partition Trees Binary Space Partition (BSP) Trees [52] are commonly used in spatial analysis. They are binary trees that recursively divide a space using half-planes. They are typically created from a set of polygons, using the polygon planes as splitting half-planes. BSP trees are commonly used in environments as they are useful in calculating visibility and performing collision detection.

Navigation Data Structures *Navigations Graphs* are commonly used to enable agents to navigate within an environment by providing perception of paths within the environment. They provide a simplified representation of the spatial areas that an agent may occupy. This reduces the time needed to search through positions in order to plan routes and make decisions. At first designers created navigation graphs by manually placing waypoints in the environment [85]. As environments increase in size and complexity, this task became more time-consuming and a candidate for automation.

Binary Space Partition (BSP) Trees [52] recursively partition an environment into spaces on either side of a hyperplane: each leaf node corresponds to a convex polytope. The traversability of these leaf nodes can be examined and linked together to automatically create a navigation graph [143]. The *Navigation Mesh* [104] simplifies navigation graph creation by representing walkable environment areas as polygons which are connected together in a mesh. As noted by [142], such navigation algorithms are useful for calculating the shortest path between two locations, but they do not assist agents in understanding the surrounding terrain.

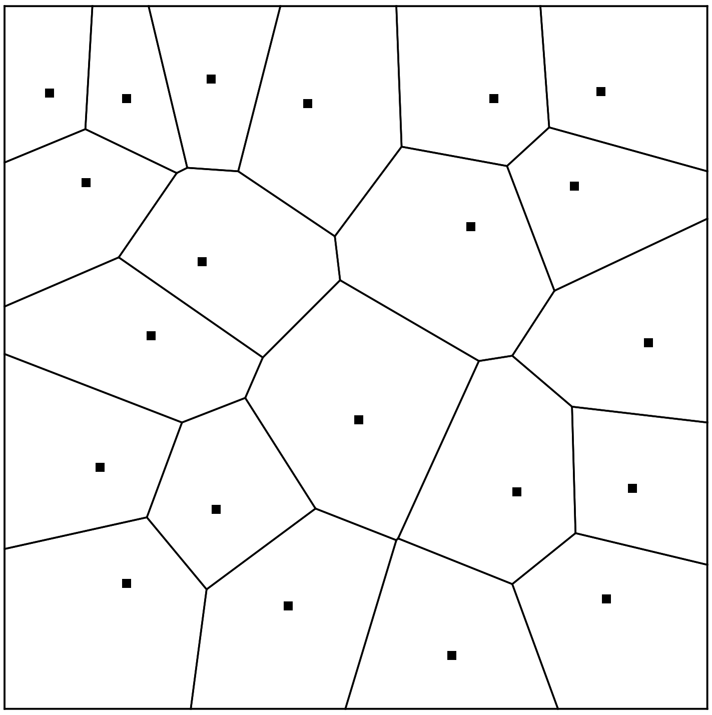


Figure A.1: Simple Voronoi Diagram

Terrain Reasoning Pottinger [110] discusses *influence maps* which are created by applying terrain influences to a 2D array to determine the best position to site objects. He also discusses grouping logical areas together for AI use via area decomposition and the importance of establishing connectivity between such areas for pathfinding purposes. However, this information is derived from features external to the terrain and not from the terrain itself. Morgan [97] evaluates a number of algorithms for determining suitable locations for a soldier to take a cover within an environment including using *Shadow BSP Trees* [29] to detect regions of concealment. For reasons of efficiency, a sensor grid evaluating visibility around an agent at different heights is used to decide if a location may be used for cover.

Van der Sterren [142] discusses terrain reasoning by examining the relationship between waypoints in a navigation graph. The connectivity and visibility between waypoints is used to make estimates about the effectiveness of a waypoint as a firing position. This effectiveness is then modified by the actual performance of agents at the waypoint. Van Der Sterren recognises the need for annotating waypoints with higher-order terrain information such as visibility and lighting. However, the focus on waypoints inherently discards intrinsic geometric data that may be useful to agents and introduces resolution issues. For example, the curvature of a section of space would be difficult to reconstruct from waypoints and resolution issues would complicate the matter further. It also difficult to reason about higher-order connectivity because waypoints are distributed throughout the environment in order to provide agents with sensory information. The proliferation of waypoints maybe obscure the fact that there is a single logical path that all the waypoints in a region may belong to. In contrast our method focuses on retaining such information since it may be useful to agent designers.

Voronoi diagrams *Voronoi diagrams* [146] are another useful tool for spatial analysis. Given a set of points S in a plane, a Voronoi diagram partitions the plane into convex polygons, each containing one point $p \in S$ and having the property that every point in the polygon is closer to p than any other point in S. Each point is called a *Voronoi Site* and lies within a *Voronoi Cell*. Figure A.1 illustrates a simple example.

Voronoi diagrams have various applications but for our work they are pertinent for the generation of a *Medial Axis*. Sample points are introduced on the boundary of an object, and the Voronoi Diagram of these points is computed. A sufficient density of samples is required to represent a surface accurately – refer to [82] and [40] for an in-depth discussion.

Medial Axes and Skeletons In 2D, the medial axis [17] of a shape can be defined as a set of curves. Each curve is defined as the locus of points lying between the boundaries of a shape [31]. It is a *shape descriptor* representing the shape of an object, since it constitutes a connected set of curves passing through the central parts of an object.

Originally introduced as the *topogical skeleton* [17], the terms *medial axis* and *skeleton* have been used interchangeably to refer to the same concept, while in other cases they are considered to be different but related concepts. In this work we use the term *skeleton* when referring to the *medial axis*.

When the shape for which the skeleton is to be computed is represented with an image, it may be calculated using a *distance transform* [112, 113] or *morphological thinning* [65]. Images are not compact representations of a shape, and polygons are frequently used to represent environments and objects. Voronoi diagrams are frequently used to approximate the skeleton [103] of polygonal shapes. In Mathematical Morphology, they are also referred to as *Skeletons by Influence Zones* (SKIZ) [144]. This process is used in path planning [13] and robotic motion planning [58, 62] to generate a skeleton, from which a navigation graph is derived.

Computing the skeleton of an object with complex boundaries may produce many vestigial "spurs" near the boundary, which do not contribute significantly the basic skeleton shape. Ogniewicz [103] prunes very fine spurs on the object boundary, and groups the remaining spurs as primary, secondary and tertiary according to a hierarchical scheme. These spurs may still connect to the object boundary at particularly jagged points of convexity.

Amenta et. al [5] use Voronoi Diagrams to reconstruct surfaces from unorganised sample points. They use a process they call “Voronoi filtering” to choose faces of the Delauney simplices to remove. Of interest here is their use of the medial axis to construct a metric that relates sampling density to surface curvature, and from which they can prove the accuracy and topological validity of their surface reconstructions.

Mobile Robot Mapping Mobile robots explore and interact with physical environments. In order to successfully navigate between regions of explore space, they build maps of the environment. Mapping strategies tend towards two approaches:

- *Metric Maps*, most commonly represented by *Occupancy Grids* [43].
- *Topological Maps*, commonly represented by mathematical graphs.

Metric Maps are fine-grained, highly-detailed representation of an environment. Simple formulations of Occupancy Grids [43, 42] accomplish this by marking grid cells as occupied or empty, but more complex versions store statistical information about cell occupancy [96, 18]. Due to the resolution required to represent large environments, metric maps can be demanding in terms of space and time complexity.

By contrast, the Topological Map, is a simplified representation of an environment, represented with a graph. Important topological locations are assigned vertices and edges are assigned to paths linking these locations. Early formulations [77, 28, 78] developed from the concept of a *Cognitive Map* [127] of an environment. Theoretically, Topological Maps scale well to large environments, but may struggle to adequately represent salient features of an environment [78].

Efforts have been made to integrate the two approaches, alternatively by building Topological Maps from Metric Maps [139, 138], and Metric Maps from Topological Maps [140].

Frequently, the robot must perform mapping without any absolute positioning data, requiring it to simultaneously build a map and localise itself within the map. This process referred to as Simultaneous Localisation and Mapping [59] (SLAM). SLAM identifies important geometric features in an environment and stores their location in a sparse map. Correlating the robot pose and the estimated features has a time complexity of $O(n^2)$ which has performance implications for large environments.

The DenseSLAM problem aims to integrate much denser sampling of environment features, while maintaining efficiency in the correlation process[99, 100]. *Hybrid Metric Maps* (HYMM) [99] accomplish this by sampling features, and partitioning the feature plane into local triangle regions (LTR) triangles with some, but not all, features as triangle corners. The triangles are themselves subdivided into grids in which dense feature information is stored. This multi-resolution representation provides correlation efficiency and dense environment mapping. Guo et. al. [60] use Voronoi diagrams to partition of the feature plane by using features as Voronoi sites. This partitions the plane uniquely and the contours of the Voronoi cells can describe the environment contour.

Metric maps do not explicitly store any width, curvature or logical connectivity. Topological maps are useful structures for representing the logical connectivity of an environment, and possibly, the curvature of the environment if it is adequately sampled and graph edges correctly placed. However, as the environment is represented as a graph, information about the width of a environment is not stored.

A.2 Spatial Awareness Framework

We aim to develop a Spatial Awareness Framework representing the intrinsic qualities of an environment, such as width, curvature and connectivity. Here we describe the structure and mathematical description of this framework. The process for creating this structure is described in Section A.3.

The core of this framework is a *medial axis* or *skeleton*, generated from a Voronoi diagram, and represented with a mathematical graph. The Voronoi diagram is produced by sampling points on the environment boundaries, which are used as input points to the process creating the diagram. Note that we manually estimate the number of samples required for an environment, but techniques do exist [82, 40] for computing a Delaunay triangulation that accurately approximates a surface, and its dual, the Voronoi diagram.

An illustration of a skeleton produced from a Voronoi diagram can be seen in Figure A.2. This skeleton can also be thought of as a topological map of the environment. Our framework builds on this graph by establishing a correspondence between the skeleton and boundaries of the environment.

A *skeleton* is an orthodox structure for describing the shape of an environment, since it approximates these structures by their centre lines – lines which are equidistant from the borders of the environment. As such, it is useful for representing the logical connectivity of an environment since different sections will be connected along their centre lines. For our purposes however, not all curves in the skeleton are appropriate for describing the curvature and width of sections of an environment, *as they are not sufficiently parallel to the object boundaries*. In particular, many of the skeleton leaf vertices extend towards concavities on the environment boundary.

Consider Figure A.4a. The *major concavity* introduces significant changes to the width, curvature and connectivity of the space and the skeleton segment that extends into it is parallel to the environment boundary. By contrast the *minor concavity* in Figure A.4b does not introduce major changes to the space and the skeleton segment is not parallel to the boundary. We therefore impose an orthogonality condition on skeleton leaf vertices that require the graph edge containing them to be perpendicular to

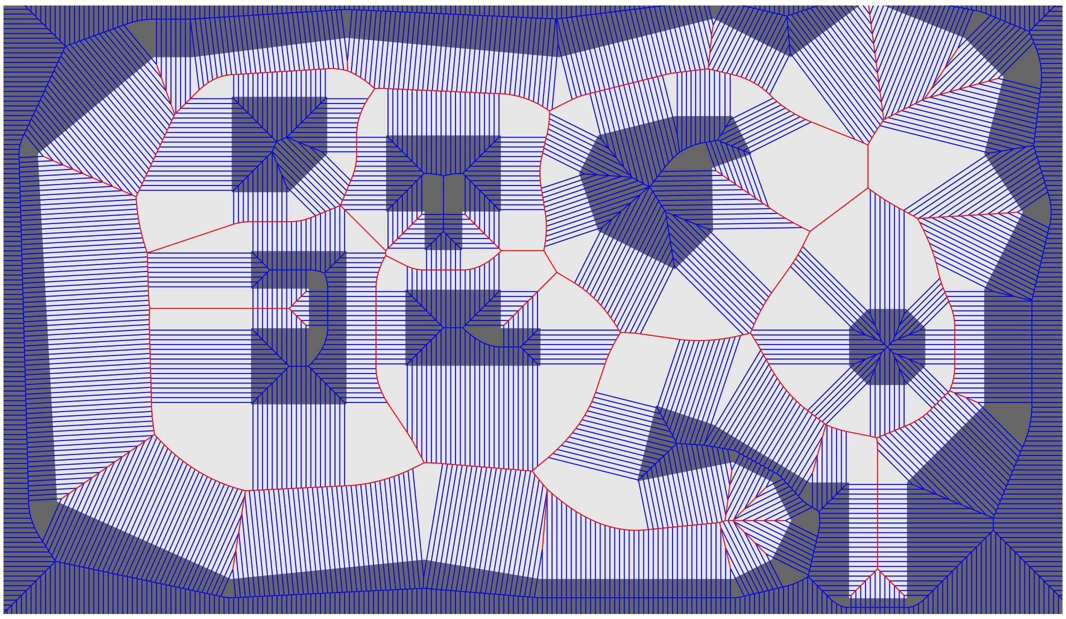


Figure A.2: The Voronoi Diagram generating the skeleton. Edges marked in red are considered to be on the skeleton.

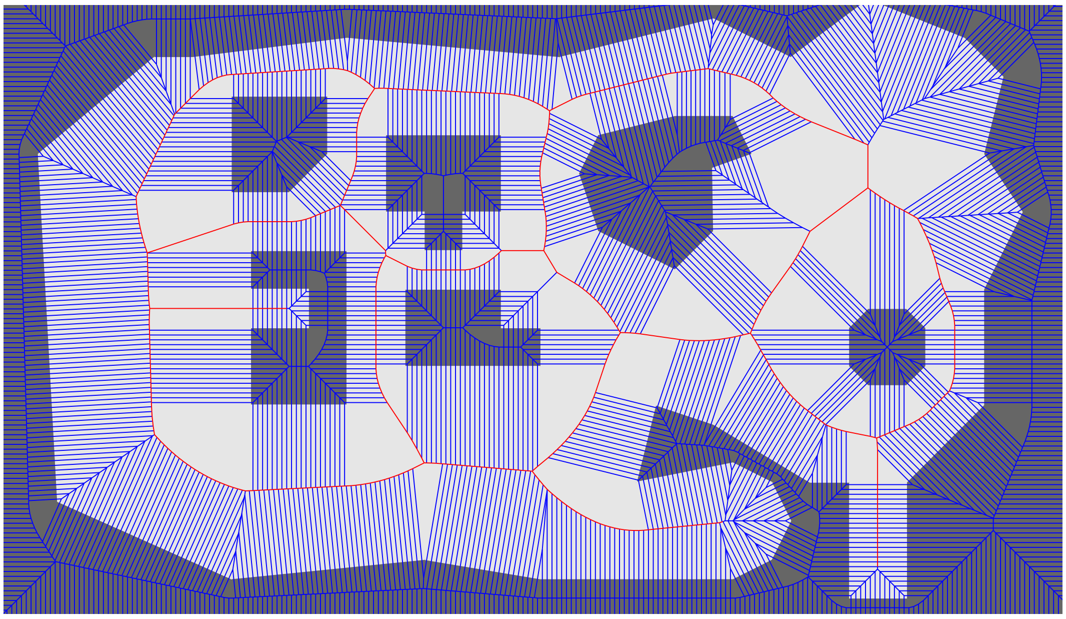
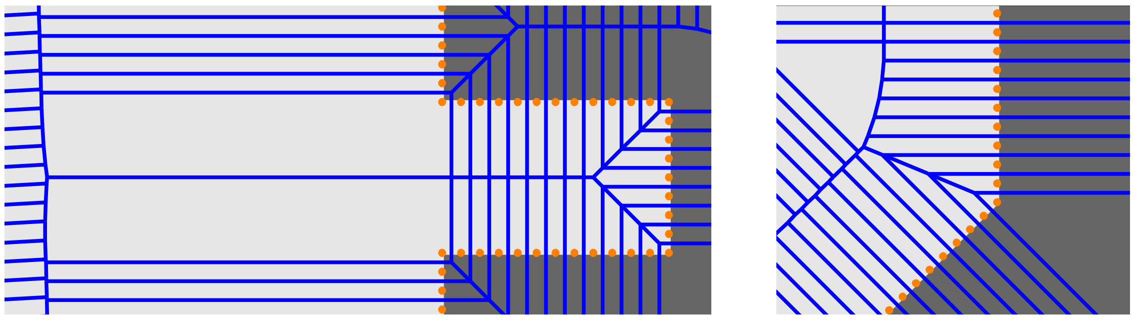


Figure A.3: The skeleton after pruning.

at least one environment boundary edge. Leaf vertices that do not satisfy this requirement are pruned from the skeleton. This pruned skeleton resulting from the pruning process applied to Figure A.2 is shown in Figure A.3.

This pruned skeleton forms the core of our Spatial Awareness Framework, representing the logical connectivity of the environment, as well as the general curvature of the space. Then, to estimate



(a) Major Concavity **(b)** Minor Concavity

Figure A.4: Major concavities introduce significant changes into the width and curvature of a space and therefore warrant a skeleton extension into the space. The changes introduced by minor concavities do not.

the width of the environment around the skeleton, we establish a bi-directional mapping between the skeleton and environment boundaries, derived from collections of Voronoi sites, which estimate the closest boundary points on either side of the skeleton.

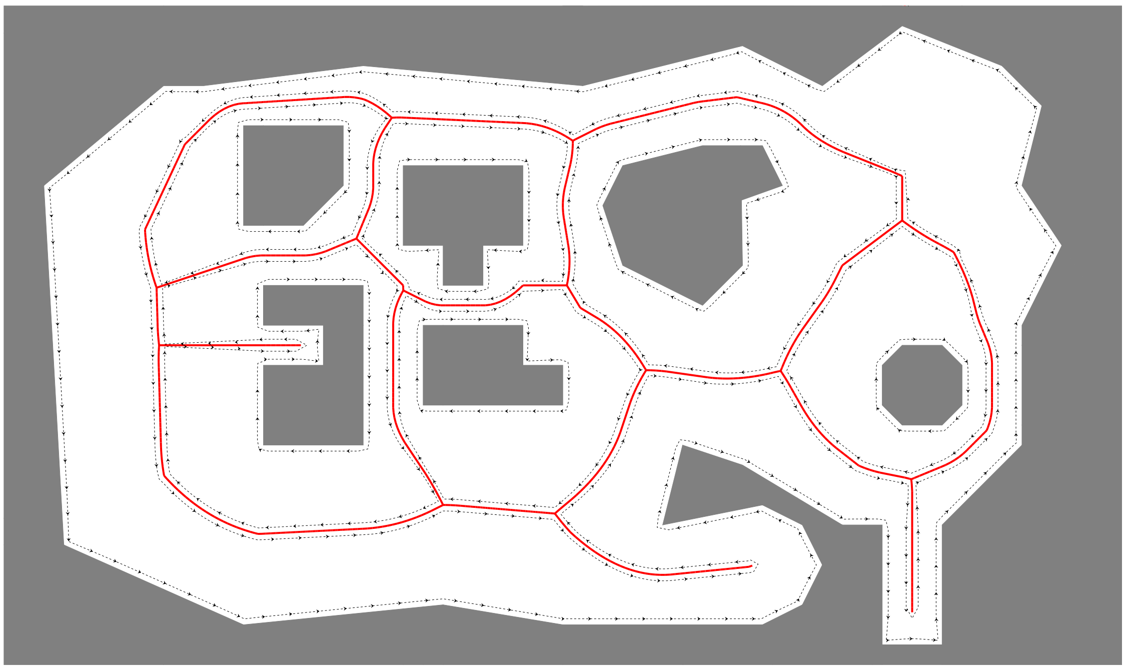


Figure A.5: Environment boundaries and closed walks on the skeleton graph are converted into closed parameterised lines, represented by a piecewise linear function. These functions are paired togethered and mappings established between their domains.

We require the boundaries of our environment to be specified by parameterised lines which form closed loops. Each boundary line is paired with a *closed walk* on the graph skeleton – a cycle in a graph possibly containing repeated vertices – also represented as a closed parameterised line. These pairings may be seen in figure A.5. These parameterised lines are defined as *piecewise linear functions*. We also create mappings between intervals on the domains of these functions, also through the use of

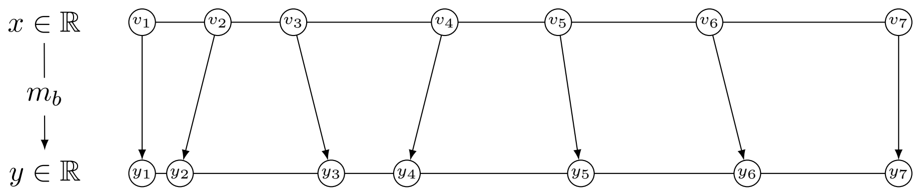


Figure A.6: A piecewise linear map is established between the domain of the function describing the skeleton closed walk, and the domain of the function describing the environment boundary.

piecewise linear functions.

For a particular line pair, let $\{p_1, \ldots, p_j\}$ with $p_1 = p_j$ be the j points forming the closed parameterised boundary, and let $\{q_1, \ldots, q_k\}$ with $q_1 = q_k$ be the k points forming the parameterised closed walk. Also let the values that parameterise these points be $\{t_1, \ldots, t_j\}$ and $\{u_1, \ldots, u_k\}$ respectively. Then we can define piecewise linear functions, b and s representing these parameterised lines:

$$b(x) = \begin{cases} (p_2 - p_1)\,x + p_1 & t_2 < x \leq t_1 \\ \cdots & \\ (p_j - p_{j-1})\,x + p_{j-1} & t_j < x \leq t_{j-1} \end{cases}$$

$$s(y) = \begin{cases} (q_2 - q_1)\,y + q_1 & u_2 < y \leq u_1 \\ \cdots & \\ (q_k - q_{k-1})\,y + q_{k-1} & u_k < k \leq u_{k-1} \end{cases}$$

and $b : \mathbb{R} \to \mathbb{R}^2$

and $s : \mathbb{R} \to \mathbb{R}^2$

Since s and b parameterise closed lines, they are *periodic*, and we choose this periodic interval to be $[0, 1]$. Thus $s(0) = s(1)$. We also define piecewise linear functions m_b and m_s that map intervals of the domain of b onto intervals of the domain of s:

$$m_s(x) = \begin{cases} (y_2 - y_1)\,x + y_1 & v_2 < x \leq v_1 \\ \cdots & \\ (y_l - y_{l-1})\,x + p_{l-1} & v_l < x \leq v_{l-1} \end{cases}$$

$$m_b(y) = \begin{cases} (x_2 - x_1)\,y + x_1 & w_2 < y \leq w_1 \\ \cdots & \\ (x_m - x_{m-1})\,y + x_{m-1} & w_k < y \leq w_{m-1} \end{cases}$$

and $m_b : \mathbb{R} \to \mathbb{R}$

and $m_s : \mathbb{R} \to \mathbb{R}$

See Figure A.6 for an illustration of how the domain of b is mapped onto the domain of s. m_b and m_s are also periodic on the $[0, 1]$ interval. To determine how these domain intervals are mapped, Voronoi cells created by sampling points along a parameterised boundary are grouped into polygons.

The points at which these polygons intersect the environment boundary and skeleton determine the interval mappings.

We describe the process for creating the skeleton, pruning it, creating parameterised lines representing boundary and skeleton sections and establishing piecewise linear mappings between them in Section A.3, which follows.

A.3 Approach

We aim to develop a data structure representing the intrinsic qualities of a space, such as width, curvature and connectivity. A *medial axis* or *skeleton* is useful for representing these qualities since in 2D it is a set of curves that run through centre of a space. From the skeleton, logical paths through the space can be identified, providing connectivity information and allowing the path curvature to be used to identify the curvature of the surrounding space.

We first perform a BSP decomposition of the environment. Since this structure subdivides the environment using half-planes, convex regions representing the solid areas of a environment can conveniently be determined and grouped together. The boundaries of these grouped regions can then be identified by traversing the outer boundary of the group. This approach means that a level designer can conveniently construct environment structures out of separate polygons. It also allows for environment structures with holes.

As we are dealing with geometric representations of environments, we adapt the popular geometric technique of extracting a medial axis from a Voronoi diagram [103]. The boundaries extracted from the BSP process are sampled and used as input to the Voronoi tesselation process.

A.3.1 Skeleton Extraction

We create a modified medial axis from a Voronoi tesselation [146] of environment objects. This medial axis is then pruned to fit our definiton of the skeleton.

Sampling polygons: We perform a Binary Space Partition (BSP) [52] of the polygons describing the environment and extract the convex regions defined by the BSP tree half-plane intersections. Solid regions representing a distinct environment structure are grouped together and the counter-clockwise boundaries of this structure are extracted. This boundary is stored as a closed, parameterised line.

Voronoi Tessellation and Initial Skeleton: Points are sampled on the parameterised boundaries and are used as input points to the Voronoi tesselation process. We use *qhull* [7] to perform the tesselation. For each input point, a Voronoi facet is generated, consisting of a number of Voronoi vertices. The vertices and the edges between them are linked together in a graph as shown in Figure A.7. Initially, a Voronoi vertex is labelled as "off" the skeleton if it lies within a environment structure, otherwise it is considered to be on the skeleton. A graph edge is considered to be on the skeleton if both its starting and ending point are on the skeleton.

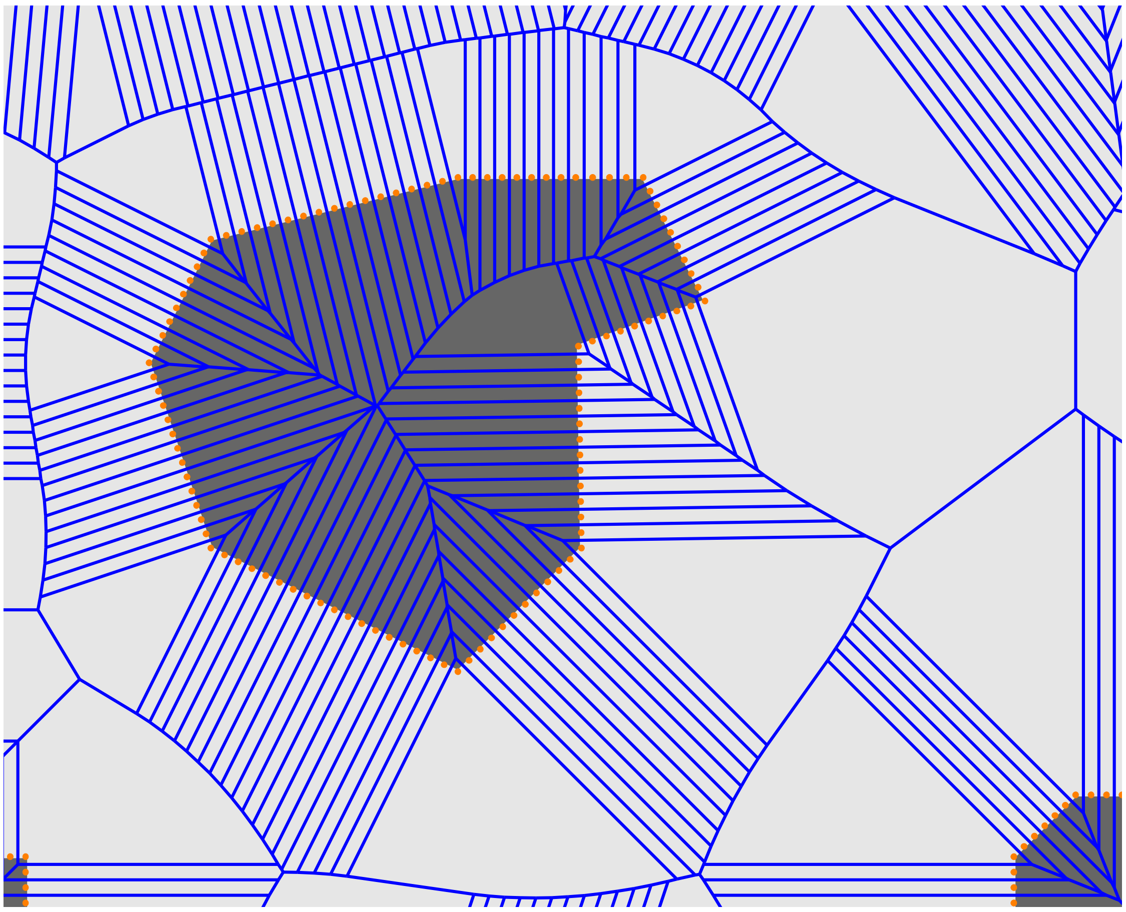


Figure A.7: Voronoi tesselation created from orange input points, sampled from the boundary of the gray environment structures. The blue edges and vertices of this tesselation are linked together in a graph.

Pruning the skeleton: According to our definition of the skeleton, skeletal sections extending towards minor regions of concavity represent too much detail and should be pruned. Skeletal sections that extend into major regions of concavity are orthogonal to sections of the environment. We therefore test the endpoints to see if the orthogonality criterion is met. This is accomplished by examining the skeletal endpoints, e, that only have one neighbour n on the skeleton. An endpoint e is considered *strong* if there exists at least one adjacent non-skeletal neighbour a such that the angle between the vectors $\overrightarrow{en}$ and $\overrightarrow{ea}$ lies in the interval $[\frac{\pi}{2} - \epsilon, \frac{\pi}{2} + \epsilon]$ for some tolerance ϵ. Otherwise the endpoint is considered to be *weak*. *Weak* endpoints are pruned from the skeleton. The process continues until no endpoints remain or only *strongly* supported endpoints remain. Figure A.8 shows two examples of this process.

Skeleton Parameterisation: Prior to creating a mapping between environment boundaries and the skeleton, parameterisation must be performed to facilitate the mapping. To accomplish this, we need

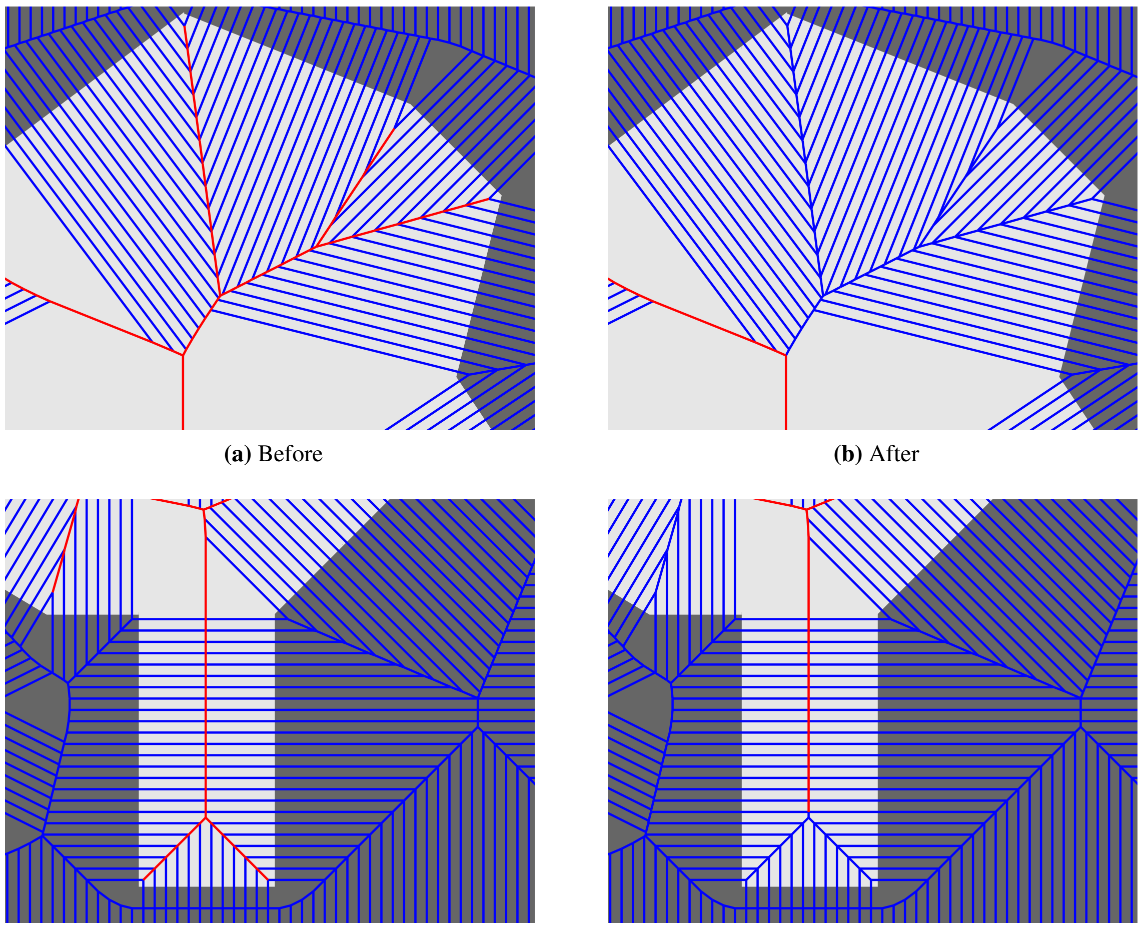

(a) Before **(b)** After **(c)** Before **(d)** After

Figure A.8: Two examples of skeleton pruning: A.8a shows a number of skeletal endpoints with *weak* supporting neighbours. This results in recursive removal until no endpoints remain as shown in A.8b. In A.8c the two endpoints are recursively pruned to the configuration in A.8d, where only one endpoint remains supported by two *strong* neighbours.

to parameterise both a environment boundary and a section of skeleton with the intent of creating a correspondence between the two. An intuitive way of visualising this is to realise that each environment boundary is enclosed by a part of the skeleton. Parameterising a boundary is simple since we can extract a closed counter-clockwise sequence of points from the BSP tree to describe it. However, our skeleton at this point exists as a graph – a set of vertices connected by edges – with no implied direction and no way to choose which path to take at intersecting points.

To parameterise the section of skeleton surrounding a environment boundary, we utilise the Voronoi facets derived from the tesselation process. By sampling the parameterised environment boundaries, a sequence of points is extracted that is input to the Voronoi tesselation process. The tesselation produces a facet for each input point and therefore produces a corresponding sequence of facets that intersect the parameterised boundary. Facets that do not have skeleton vertices can be safely ignored. Once the facet ordering has been established, the ordering of the skeleton points lying on the facets

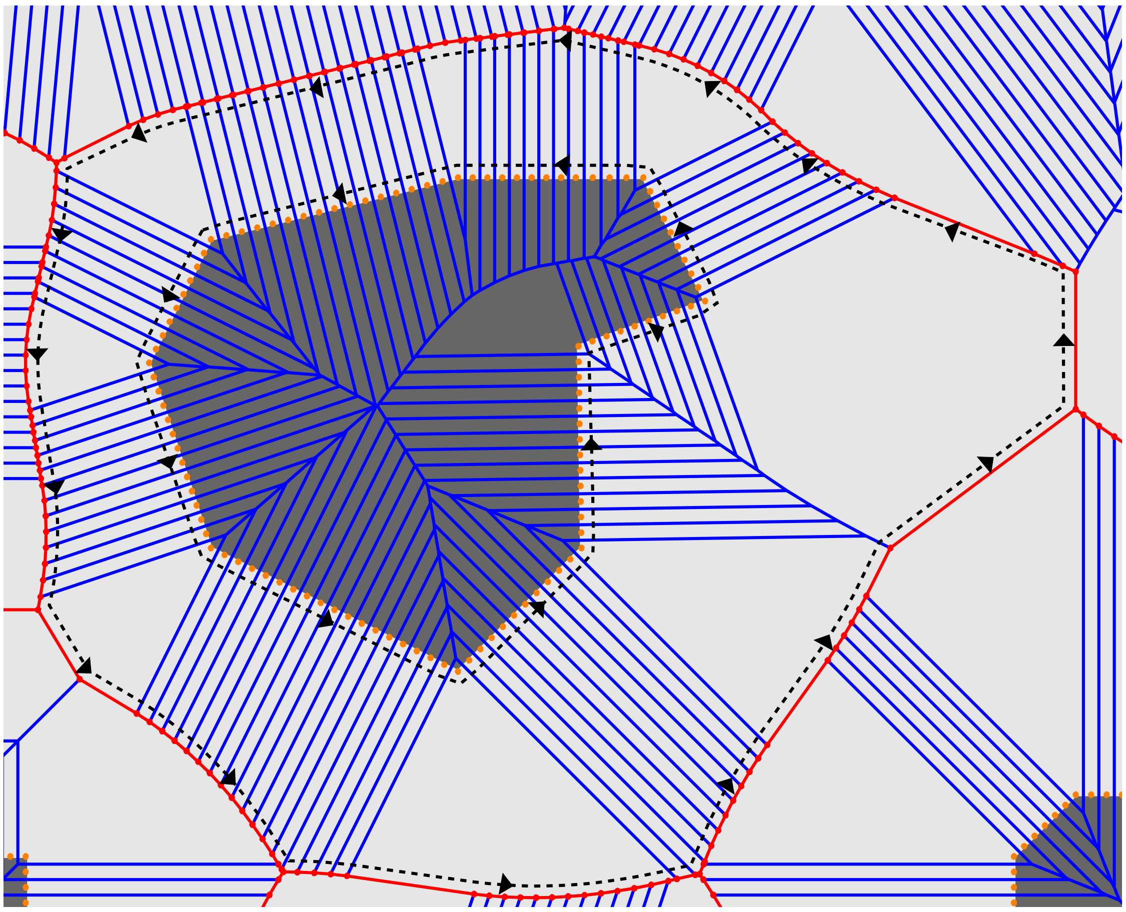


Figure A.9: Skeleton parameterisation: The orange input points to the Voronoi tesselation process produce the blue Voronoi facets, which have edges and vertices that lie on the skeleton and are marked in red. The ordering of the input points produces a corresponding Voronoi facet ordering, which allows us to establish an ordering of the skeleton points surrounding the environment structure.

can also be established. This sequence of points is then parameterised. The relation between the Voronoi input points and the skeleton points is shown in Figure A.9.

At the end of the parameterisation process a parameterised environment boundary b, and a section of parameterised skeleton s, that correspond to each other are produced. They are periodic functions on the interval $[0, 1]$.

A.3.2 Mapping Generation

Once a skeleton has been derived from the environment structure, and parameterisations for sections of skeleton and environment boundaries have been established, we create a mapping between the skeleton and environment boundaries. The aim of this mapping is to establish the best possible correspondence between points on the skeleton and points on the environment boundary in order to accurately represent the width of the space. Once again, this is accomplished by examining the

Voronoi facets that intersect the environment boundaries and have an edge on the skeleton. We use these facets to categorise sections of environment boundary and skeleton into three classes as shown in Figure A.10.

Perpendicular Sections: Consist of one or more consecutively ordered Voronoi facets which have edges that all intersect the environment boundary at the same angle. The intersecting edges are perpendicular to each other.
Folded Out Sections: Consist of one Voronoi facet that "fans outward" from the environment boundary. The intersecting edges diverge from each other as they move from the environment boundary towards the skeleton.
Folded In Sections: Consist of one or more consecutively ordered Voronoi facets which have had weak skeleton endpoints pruned away. These facets "fan in" towards skeleton endpoints that have been removed.

These sections are used to establish local mappings between environment and skeleton boundaries and form the building block of the final mapping. Each section is assigned parameterised values for the starting and ending skeleton points (v_s and v_e) and starting and ending boundary points (w_s and w_e). Two section lists are maintained, a *skeleton ordered section list* ordered by the v_s of each section and a *boundary ordered section list*, ordered by w_s.

When performing a mapping from the parameterised skeleton onto the parameterised boundary for some parameterised skeleton value v, the *skeleton ordered section list* is used to look up a section such that $v_s \leq v_v \leq v_e$. Linear interpolation is then performed to derive a corresponding parameterised value w for the boundary.

$$w = \frac{(v - v_s)}{v_s - v_e}.(w_s - w_e)$$

Folded in sections converge on a single skeleton point such that $v_s = v_e$ and linear interpolation fails in this case. To deal with this we simply map v_s and v_e to $\frac{w_s - w_e}{2}$. Mapping from the boundary on to the skeleton can be performed by reversing the process, with no special cases needing to be dealt with, since sections never converge onto a single boundary point.

A.4 Agent Implementation

In order to test the spatial awareness framework, we implemented an agent-based crowd simulation system. This system is briefly described here, but for greater implementation insight, the reader is referred to Chapter 6 of [64].

The simulation consists of environment geometry represented as a set of polygons and a crowd of autonomous, embodied agents. Using forward Euler integration, the agents update their state at each time step based on their perception of the environment as well as constraints imposed on them.

The autonomous agents interact with the environment (as well as other simulated agents) through a set of senses as shown in Figure A.11. These senses take information from the environment (such as environment geometry or enemy agent positions), as well as information about the local agent (such as turn rate or movement speed), and produce a two dimensional output according to the sense interface. This output is used by the agent to steer.

The senses are tailored for a particular type of agent. Examples include **distance_to_friends**, the distance to each friendly agent in the environment, and **geom_vision_x**, the amount of area obscured by environment geometry measured along the agent's x-axis. The senses may be customised to allow the agents a greater or lesser knowledge of the environment or to allow a certain type of behaviour. For example, the **geom_vision_x** sense may be used for navigation purposes and the **angle_to_friend** sense for tactical decision purposes.

The input from the senses are grouped into a perception module and used as input to the agent's brain. This brain then alters control values for the agent which are in turn used to update the agent's internal state.

An agent's brain is a collection of fuzzy rules, well documented in the field of control systems [148]. These rules may be visualised as a network of fuzzy logic nodes. Each rule, of a standard **if** a **then** b form, operates on fuzzy variables which, in contrast to the standard boolean variety, take on a value in the range [0, 1]. For a full explanation of fuzzy variable and fuzzy inference, the reader is referred to standard texts [72].

A fuzzification step takes the two dimensional input from the agent's senses and determines the values of the fuzzy variables used by the brain. This is done in a number of different ways, depending on the properties required, usually involving a scaling step followed by a summation or maximum operation. Once the fuzzy inference has been conducted, a defuzzification step is required in order to determine the real values for the agent's controls as well as combine rules that act upon the same control. We use the height method for this defuzzification [94] due its computational efficiency.

A.5 Results

To demonstrate the usefulness of the spatial awareness framework, we created two simple games using the crowd simulation agents designed to play the games at a basic level. We then created a new group of agents based on the original agents, but with additional rules making use of extra sensory information provided by the spatial awareness framework. By observing the performance of the modified agents, we evaluated whether or not the framework has enhanced their behaviour.

A.5.1 Racing Car Scenario

The racing car game involves the agents moving around a simple track as efficiently as possible while avoiding the walls. We observed that the basic agent, with inter-agent and wall avoidance behaviour, did not take an optimal line around corners. This is due to the agent being purely reactive with no knowledge or recollection of the way in which the track turns.

We created a sense which provided information on the curvature of the upcoming section of track by considering the angle changes along upcoming sections of skeleton. To accomplish this, polygons created from the boundary local mapping sections in the *skeleton ordered section list* were placed in a quadtree. Then, during the game, the agent's position was used to query the quadtree and return the local mapping section containing the position. The skeleton t value, defining the line containing the position within the section was obtained using a binary search and changes in skeleton curvature after this t value were provided to the agent. Using this information, the agent was able to keep the inside wall of the track in view, hugging the walls and taking a better line around corners.

We created five racing tracks (See Figure A.12a to A.12e) to test the performance of the agents. Eight racing agents took part in each race, four of which were normal agents and the other four being enhanced with curvature awareness. The agents' starting positions were arranged in the traditional staggered, two column configuration, with the curvature aware agents placed at the back.

In all of the five tracks, the spatially aware agents overtook all the normal agents by the second lap. Taking the inside line on the track shortened the distance they travelled and gave them a better line making it more difficult for normal agents to pass.

The four spatially aware agents queried the framework in realtime. The complexity of this query is $O(\log N)$ since it involves a quadtree lookup followed by a binary search.

A.5.2 Robot War Scenario

To test the use of the spatial awareness framework in a setting somewhat similar to a first-person shooting game, we created a robot war simulation. Each agent was able to shoot in the direction that they are facing with some degree of randomness in their accuracy. The basic behaviour for a robot agent is the standard agent-and wall-avoidance, as well as a "targetting" behaviour in which an agent turns to face any enemy agent that it sees.

In this scenario, the improved agents were given a sense of how many skeleton intersection points – locations where three or more skeleton sections connected – were visible to them. The rationale for knowledge of these areas being advantageous is that places where paths intersect are likely to have a lot of traffic. This sense is therefore a combination of the intrinsic quality of connectivity provided by the framework and the secondary quality of visibility.

Contest	Smart Agent Kills	Normal Agent Kills
Map 1 Team	5	3
Map 2 Team	5	0
Map 3 Team	5	1
Map 1 Team Reversed	5	0
Map 2 Team Reversed	5	2
Map 3 Team Reversed	5	0
Map 1 1v1	1	0
Map 2 1v1	1	0
Map 3 1v1	1	0
Map 1 1v1 Reversed	1	0
Map 2 1v1 Reversed	1	0
Map 3 1v1 Reversed	1	0

Table A.1: The outcomes of the agent contests. The number of kills for each type of agent are listed for each contest.

To this end, we generated a *strategy map* from the spatial awareness framework. The strategy map is generated by traversing the skeleton and sampling points on it and to either side of it. At each point, we compute the number of visible skeleton intersection points and two sense values, **best_vis_angle** and **position_goodness**. **best_vis_angle** is set to the angle at which the most skeleton intersection points can be seen in a 30° arc. **position_goodness** is set to the number of intersection points in the 30° divided by the number of visible skeleton intersection points. Thus, if a point has many intersection skeleton points visible in a single 30° arc, it will have a high **position_goodness**, representing the advantage of being able to see many areas of high traffic. The sample points were placed in a kd-tree [11] to facilitate fast lookup of point-based data.

The first sense, **best_vis_angle**, was used by adding two rules to the brain which turn the agent toward the best angle if there are no enemies currently visible. The second sense, **position_goodness** was used to direct the agent to stop and wait for enemies in a location (also known as "camping") if it has a high **position_goodness** and is not too close to other friendly agents (to stop agents grouping together in one spot). The combination of these two behaviours results in the agents occasionally camping in a strategically valuable area while facing in the direction from which an enemy is most likely to come.

The contests between the agents took place in three environments (See Figure A.13a to A.13c). Since we wished to evaluate the effect of one environmental variable or sense at a time, we constructed environments which did not give an undue advantage to the normal agents. For example, we broke up outer circuits on the edge of the environment since normal agents tended to congregate on them and surprise the more spatially aware agents looking inward. While it would be easy enough to use additional environmental variables to eliminate this advantage, our intention was to assess the utility of the extra spatial information to agent behaviour, not to design an optimal agent.

For each environment two different contests took place: A team contest were a group of five "smart" agents competed against a group of five "normal" agents and a one-on-one contest where one "smart" agent competed against one normal agent. Each contest was then repeated with the starting positions reversed in order to ensure that the environments did not unduly favour one side. The results of these contents are listed in Table A.1.

In each case, the agents with awareness of path intersection points won the contest. Additionally, the more spatially aware agents "camped" near positions with a high **position_goodness** sense, managing to surprise normal agents who wandered across areas of high traffic. Since the agents react to sensory information, they have no higher-level planning behaviour besides "camping."

A.5.3 Complexity of Data Structure Queries

In each scenario, the more spatially aware agents queried the framework in realtime. To lookup curvature information for a racing agent, a quadtree lookup was performed followed by a binary search, yielding an $O(\log N)$ complexity. To lookup up a point sample for the warring agents, a nearest neighbour search was performed on a kd-tree, which again produces $O(\log N)$ complexity.

A.6 Conclusion

The spatial awareness framework presented in this appendix introduces a new system which allows agents to query the *intrinsic* geometric qualities of the space that they are operating in, namely width, curvature and connectivity. To our knowledge, this is the first system which automatically extracts such qualitative geometric information from an environment. An agent crowd simulation system was implemented to test whether awareness of these qualities could improve the performance of agents within an environment.

Even though the agents were primarily designed to react to sensory information and only implemented the most basic of planning capabilities, their effectiveness was increased with access to geometric information. We also showed how the connectivity information derived from our spatial awareness framework could be combined with the commonly used visiblity information.

The framework conveniently stores information about the space that we have termed *intrinsic*. In our testing scenario, secondary visibility information is embedded within the graph of the skeleton, but this information is point-sampled at each vertex. This strategy may not be ideal for representing regions of secondary information such as lighting or temperature. This information could be incorporated into the polygonal data on either side of the skeleton, but subdivision of polygons may be necessary to accurately represent regions of various characteristics.

Additionally, while the autonomous agents used to test the Spatial Awareness Framework were useful for testing purposes, it would be impractical to actually use these types of agents in a real-time scenarios, since simulating vision is costly. The agents can also only respond to what they can "see" around them, but introducing responses to features further away is limited by this sense, since it does not incorporate an abstract representation of the environment, such as a graph. The most useful aspect of this approach to agent programming is that it is bottom-up: the agents behaviour is defined by the way it "senses" its environment.

Once the environmental feature that the agent should respond to is further away from the agent, the notion of path-planning naturally comes into consideration. Integrating path-planning across regions, or a subdivision of these regions, so that the agents react to the properties within them, prompted our investigation of various path-planning algorithms capable of finding shortest paths through a set of weighted regions. The challenging nature of this set of problems, as well as the interesting issues stemming from the *coupling* between the environment representation and the algorithm, resulted in our focus on these algorithms in the rest of this work.

Our criteria for a candidate pathfinding algorithm are:

- it finds shortest paths through weighted *regions*;
- it should be able to efficiently represent and find paths through irregularly shaped environments. By this, we mean that the environments are not necessarily axis-aligned;
- it should be able to efficiently replan paths when the weightings within the environment change. Lighting, for example, can be a dynamic when light sources move;

This criteria led us to adapt and extend Field D*, a shortest path algorithm that operates on weighted grids.

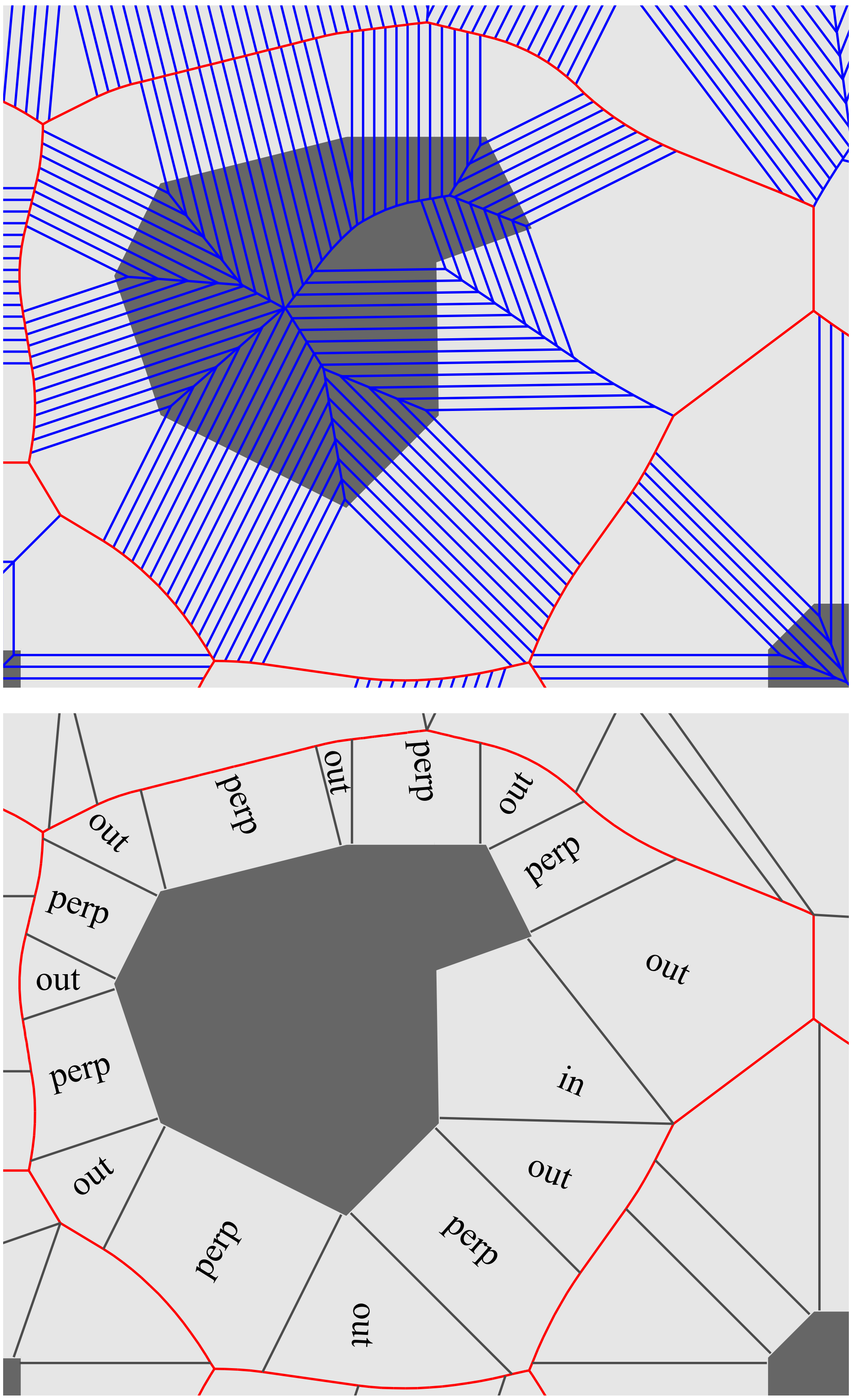


Figure A.10: Classifying sections: Sections of space between the environment boundary and skeleton are classified into *folded out*, *folded in* and *perpendicular* sections, based on the Voronoi facets found within the space.

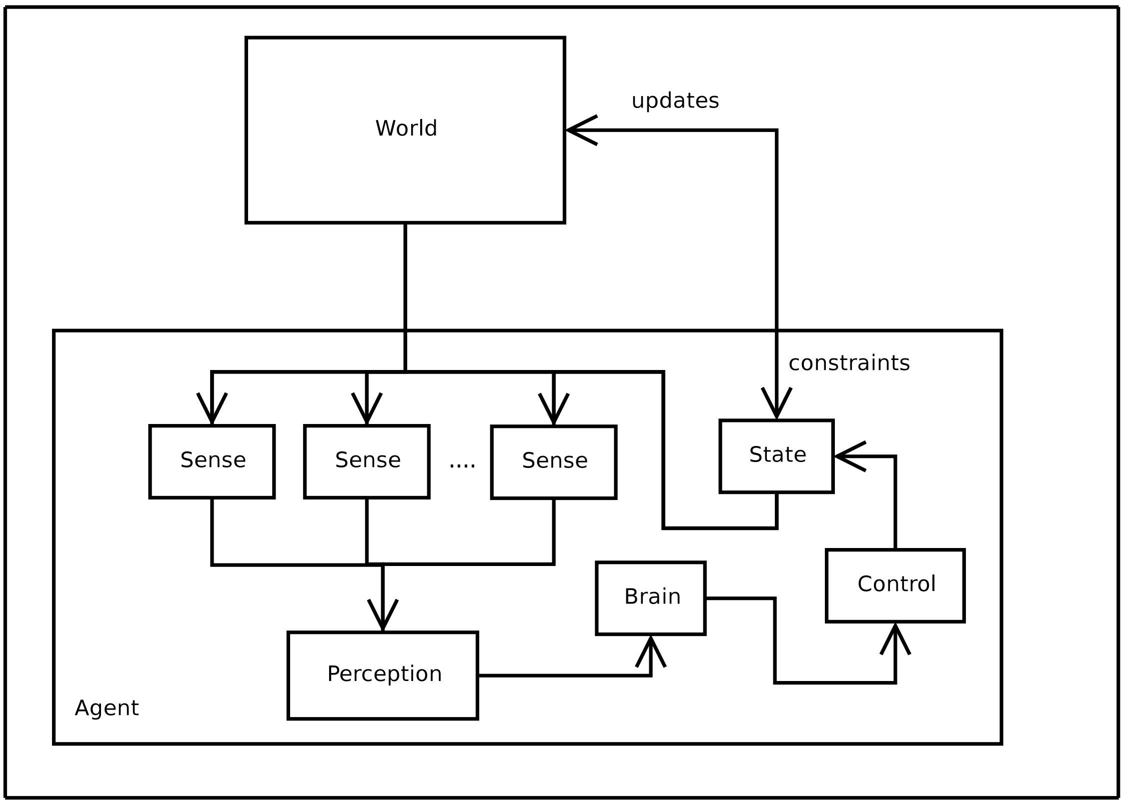
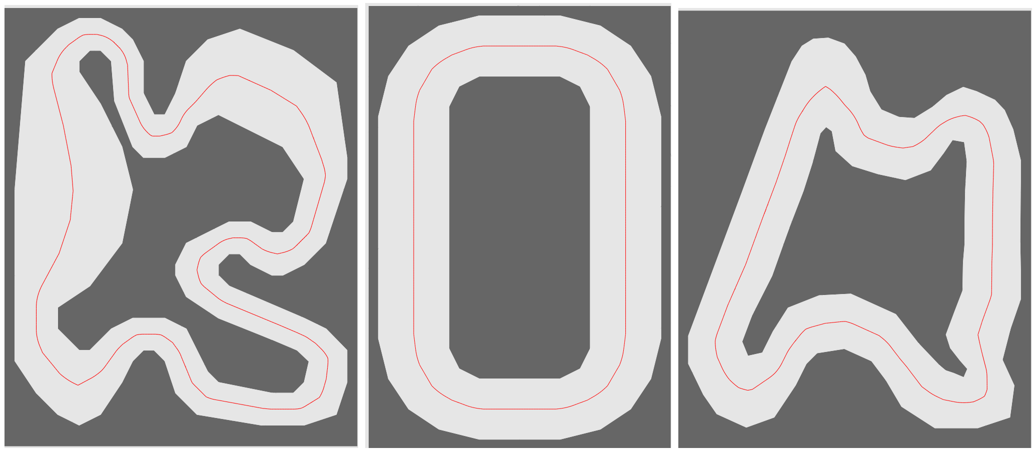
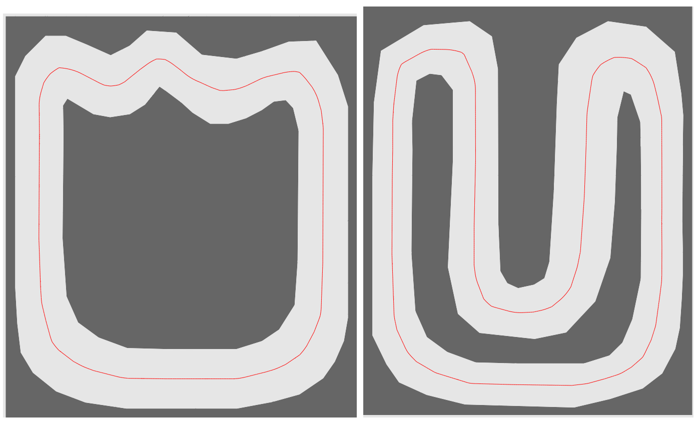


Figure A.11: A visual overview how an agent's brain interacts with the environment.

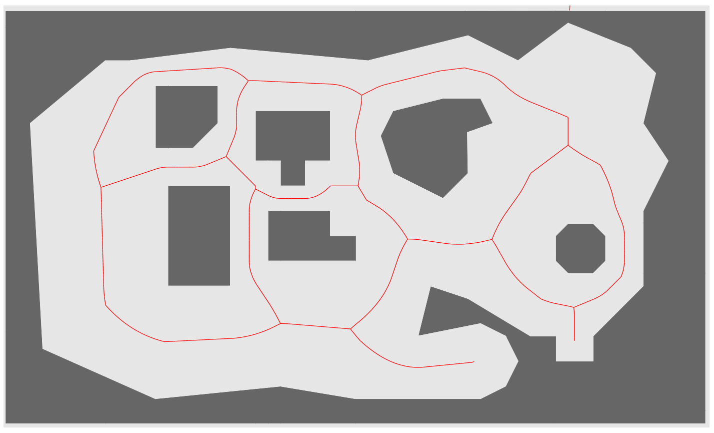


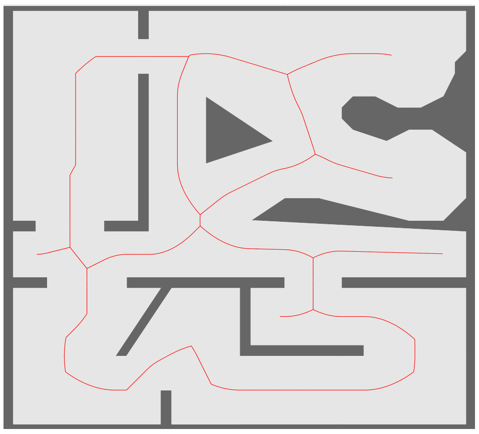
(a) Racetrack 1 **(b)** Racetrack 2 **(c)** Racetrack 3



(d) Racetrack 4 **(e)** Racetrack 5

Figure A.12: The racetrack environments and their skeletons, used in testing.


(a) Robot War 1


(b) Robot War 2

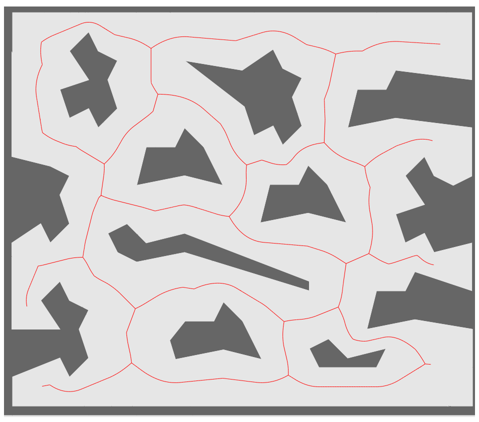

(c) Robot War 3

Figure A.13: The robot war environments and their skeletons, used in testing.

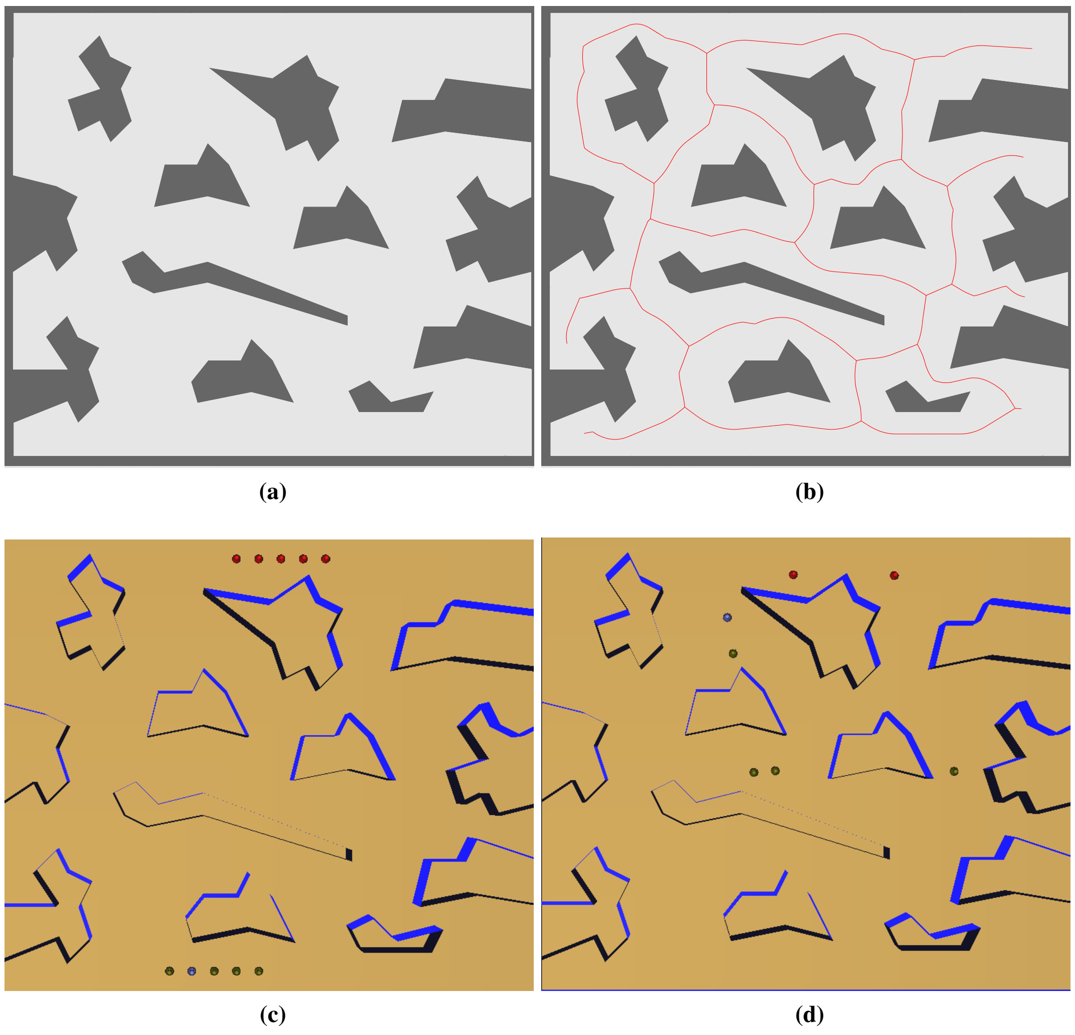


Figure A.14: Starting with the geometry defining an environment in A.14a, information on connectivity, width and curvature is extracted in A.14b. This information is used by agents to enhance their behaviour within the environment. In A.14b and A.14c the beige and blue agents use this enhanced behaviour to defeat their red opponents.

Appendix B

N-Dimensional Cost Function Solutions and Reductions

In this appendix, we derive the analytic solutions for the n-dimensional cost functions. In each case we will be considering the space $\mathbb{R}^n$ with dimension n. Within this space, we will be constructing bases $\mathbf{M}$ and $\mathbf{N}$ from linearly independent vectors within $\mathbb{R}^n$. In particular these matrices will have an $n \times m$ shape, with $m < n$ since $\mathbf{M}$ and $\mathbf{N}$ represent subspaces of $\mathbb{R}^n$. Since these matrices are *full rank* and $m < n$, a *left inverse*, $\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\mathbf{M}^T$ for example, exists for both $\mathbf{M}$ and $\mathbf{N}$.

B.1 General Cost Function

The simple general cost function is expressed as:

$$G\left(\mathbf{x}, \lambda, \mathbf{u}, \mathbf{M}, \boldsymbol{\mu}, d\right) = \lambda \|\mathbf{u} + \mathbf{M}\mathbf{x}\| + \boldsymbol{\mu}^T\mathbf{x} + d \tag{B.1}$$

Expressing the distance component as the vector dot product to the power of a half yields:

$$G\left(\mathbf{x}, \lambda, \mathbf{u}, \mathbf{M}, \boldsymbol{\mu}, d\right) = \lambda \left(\mathbf{u}^T\mathbf{u} + 2\mathbf{u}^T\mathbf{M}\mathbf{x} + \mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x}\right)^{\frac{1}{2}} + \boldsymbol{\mu}^T\mathbf{x} + d$$

Taking the derivative with respect to $\mathbf{x}$ and setting, $dG/dx = 0$

$$-\lambda \left(\mathbf{u}^T\mathbf{M} + \mathbf{x}^T\mathbf{M}^T\mathbf{M}\right) . \left(\mathbf{u}^T\mathbf{u} + 2\mathbf{u}^T\mathbf{M}\mathbf{x} + \mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x}\right)^{-\frac{1}{2}} + \boldsymbol{\mu}^T = 0 \tag{B.2}$$

Setting $\phi = \left(\mathbf{u}^T\mathbf{u} + 2\mathbf{u}^T\mathbf{M}\mathbf{x} + \mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x}\right)^{\frac{1}{2}}/\lambda$ and rearranging, we obtain the following expression for $\mathbf{x}^T$ and $\mathbf{x}$:

$$\begin{aligned}
\mathbf{u}^T\mathbf{M} + \mathbf{x}^T\mathbf{M}^T\mathbf{M} &= \phi.\boldsymbol{\mu}^T \\
\Rightarrow \mathbf{x}^T\mathbf{M}^T\mathbf{M} &= \phi.\boldsymbol{\mu}^T - \mathbf{u}^T\mathbf{M} \\
\Rightarrow \mathbf{x}^T &= \left(\phi\boldsymbol{\mu}^T - \mathbf{u}^T\mathbf{M}\right)\left(\mathbf{M}^T\mathbf{M}\right)^{-1} \\
\Rightarrow \mathbf{x} &= \left(\mathbf{M}^T\mathbf{M}\right)^{-1}\left(\phi\boldsymbol{\mu} - \mathbf{M}^T\mathbf{u}\right)
\end{aligned}$$

Now, we rearrange Equation B.2 and square it. By squaring we introduce an extra solution. Later on, we show that it is possible to always pick a particular solution.

$$\begin{aligned}
&\lambda\left(\mathbf{u}^T\mathbf{M} + \mathbf{x}^T\mathbf{M}^T\mathbf{M}\right) = \boldsymbol{\mu}^T\left(\mathbf{u}^T\mathbf{u} + 2\mathbf{u}^T\mathbf{M}\mathbf{x} + \mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x}\right)^{\frac{1}{2}} \\
&\Rightarrow \lambda^2\left(\mathbf{u}^T\mathbf{M} + \mathbf{x}^T\mathbf{M}^T\mathbf{M}\right)\left(\mathbf{M}^T\mathbf{u} + \mathbf{M}^T\mathbf{M}\mathbf{x}\right) = \boldsymbol{\mu}^T\boldsymbol{\mu}\left(\mathbf{u}^T\mathbf{u} + 2\mathbf{u}^T\mathbf{M}\mathbf{x} + \mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x}\right) \\
&\Rightarrow \lambda^2\mathbf{u}^T\mathbf{M}\mathbf{M}^T\mathbf{u} + \lambda^2\mathbf{u}^T\mathbf{M}\mathbf{M}^T\mathbf{M}\mathbf{x} + \lambda^2\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{M}^T\mathbf{u} + \lambda^2\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{M}^T\mathbf{M}\mathbf{x} = \\
&\boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{u} + 2\boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{M}\mathbf{x} + \boldsymbol{\mu}^T\boldsymbol{\mu}^T\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x}
\end{aligned}$$

Grouping by $\mathbf{x}^T$ and $\mathbf{x}$:

$$\begin{gathered}
\left(\lambda^2\mathbf{u}^T\mathbf{M}\mathbf{M}^T\mathbf{u} - \boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{u}\right) + \left(\lambda^2\mathbf{u}^T\mathbf{M}\mathbf{M}^T\mathbf{M} - 2\boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{M}\right)\mathbf{x} + \\
\mathbf{x}^T\left(\lambda^2\mathbf{M}^T\mathbf{M}\mathbf{M}^T\mathbf{u}\right) + \mathbf{x}^T\left(\lambda^2\mathbf{M}^T\mathbf{M}\mathbf{M}^T\mathbf{M} - \boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{M}^T\mathbf{M}\right)\mathbf{x} = 0
\end{gathered}$$

We now substitute the expressions for $\mathbf{x}^T$ and $\mathbf{x}$ into the above:

$$\begin{aligned}
&\left(\lambda^2\mathbf{u}^T\mathbf{M}\mathbf{M}^T\mathbf{u} - \boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{u}\right) + \\
&\left(\lambda^2\mathbf{u}^T\mathbf{M}\mathbf{M}^T\mathbf{M} - 2\boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{M}\right)\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\left(\phi\boldsymbol{\mu} - \mathbf{M}^T\mathbf{u}\right) + \\
&\left(\phi\boldsymbol{\mu}^T - \mathbf{u}^T\mathbf{M}\right)\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\left(\lambda^2\mathbf{M}^T\mathbf{M}\mathbf{M}^T\mathbf{u}\right) + \\
&\left(\phi\boldsymbol{\mu}^T - \mathbf{u}^T\mathbf{M}\right)\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\left(\lambda^2\mathbf{M}^T\mathbf{M}\mathbf{M}^T\mathbf{M} - \boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{M}^T\mathbf{M}\right)\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\left(\phi\boldsymbol{\mu} - \mathbf{M}^T\mathbf{u}\right) = 0
\end{aligned}$$

Note that $\left(\mathbf{M}^T\mathbf{M}\right)^{-1}$ multiplied by $\left(\mathbf{M}^T\mathbf{M}\right)$ produces the identity matrix. Expanding out produces:

$$\begin{aligned}
&\left(\lambda^2\mathbf{u}^T\mathbf{M}\mathbf{M}^T\mathbf{u} - \boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{u}\right) + \\
&\lambda^2\phi\boldsymbol{\mu}^T\mathbf{u}^T\mathbf{M}\boldsymbol{\mu} - \lambda^2\mathbf{u}^T\mathbf{M}\mathbf{M}^T\mathbf{u} - 2\phi\boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu} + 2\boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\mathbf{M}^T\mathbf{u} + \\
&\lambda^2\phi\boldsymbol{\mu}^T\mathbf{M}^T\mathbf{u} - \lambda^2\mathbf{u}^T\mathbf{M}\mathbf{M}^T\mathbf{u} + \\
&\lambda^2\phi^2\boldsymbol{\mu}^T\boldsymbol{\mu} - \lambda^2\phi\boldsymbol{\mu}^T\mathbf{M}^T\mathbf{u} - \lambda^2\phi\mathbf{u}^T\mathbf{M}\boldsymbol{\mu} + \lambda^2\mathbf{u}^T\mathbf{M}\mathbf{M}^T\mathbf{u} - \\
&\phi^2\boldsymbol{\mu}^T\boldsymbol{\mu}\boldsymbol{\mu}^T(\mathbf{M}^T\mathbf{M})^{-1}\boldsymbol{\mu} + \phi\boldsymbol{\mu}^T\boldsymbol{\mu}\boldsymbol{\mu}^T(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T\mathbf{u} + \\
&\phi\boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{M}(\mathbf{M}^T\mathbf{M})^{-1}\boldsymbol{\mu} - \boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\mathbf{M}^T\mathbf{u} = 0
\end{aligned}$$

Taking into account that the transpose of a scalar value is equal to itself, and that many of the above terms cancel each other, we obtain:

$$\lambda^2\phi^2\boldsymbol{\mu}^T\boldsymbol{\mu} - \phi^2\boldsymbol{\mu}^T\boldsymbol{\mu}\boldsymbol{\mu}^T(\mathbf{M}^T\mathbf{M})^{-1}\boldsymbol{\mu} + \boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{M}(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T\mathbf{u} - \boldsymbol{\mu}^T\boldsymbol{\mu}\mathbf{u}^T\mathbf{u} = 0$$

This expression can hold if $\boldsymbol{\mu}^T\boldsymbol{\mu} = 0$. Assuming it does not, we can cancel the $\boldsymbol{\mu}^T\boldsymbol{\mu}$ terms out and rearrange to obtain:

$$\phi^2 \left(\lambda^2 - \boldsymbol{\mu}^T \left(\mathbf{M}^T\mathbf{M}\right)\right)^{-1} \boldsymbol{\mu}\Big) = \mathbf{u}^T\mathbf{u} - \mathbf{u}^T\mathbf{M}(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T\mathbf{u}$$

Thus we have the following value for ϕ:

$$\phi = \pm\sqrt{\frac{\mathbf{u}^T\left(\mathbf{I} - \mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\mathbf{M}^T\right)\mathbf{u}}{\lambda^2 - \boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu}}}$$

Note that $\lambda^2 - \boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu} > 0$ is required for a real solution of ϕ to exist. Also, there are two possible solutions for ϕ, and consequently, $\mathbf{x}$. In fact, one always chooses the negative value for ϕ as we explain in the next section.

B.1.1 Choosing the root

It is possible to substitute both the positive and negative values of ϕ into cost function B.1 and select the value of ϕ that ultimately minimises the function. However, it would involve less computation if we could detect this root initially. Firstly we consider its effect on the distance term $\|\mathbf{v} + \mathbf{M}\mathbf{x}\| = \left(\mathbf{v}^T\mathbf{M} + 2\mathbf{v}^T\mathbf{M}\mathbf{x} + \mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x}\right)^{\frac{1}{2}}$. Substituting $\mathbf{x}^T$ and $\mathbf{x}$ into the expression within the distance term yields:

$$\begin{aligned}&\mathbf{v}^T\mathbf{v} + 2\mathbf{v}^T\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\left(\phi\boldsymbol{\mu} - \mathbf{M}^T\mathbf{v}\right) + \\ &\qquad\left(\phi\boldsymbol{\mu}^T - \mathbf{v}^T\mathbf{M}\right)\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\mathbf{M}^T\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\left(\phi\boldsymbol{\mu} - \mathbf{M}^T\mathbf{v}\right)\end{aligned}$$

Expanding out produces:

$$\begin{aligned}&\mathbf{v}^T\mathbf{v} + 2\phi\mathbf{v}^T\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu} - 2\mathbf{v}^T\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\mathbf{M}^T\mathbf{v} + \\ &\qquad\phi^2\boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu} - 2\phi\mathbf{v}^T\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu} + \mathbf{v}^T\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\mathbf{M}^T\mathbf{v} \\ \Rightarrow\ &\mathbf{v}^T\mathbf{v} - \mathbf{v}^T\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\mathbf{M}^T\mathbf{v} + \phi^2\boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu}\end{aligned}$$

Since only ϕ^2 contributes to this expression, the sign of ϕ does not matter here. Next we substitute $\mathbf{x}$ into the $\boldsymbol{\mu}^T\mathbf{x}$ term:

$$\begin{aligned}&\boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\left(\phi\boldsymbol{\mu} - \mathbf{M}^T\mathbf{v}\right) \\ \Rightarrow\ &\phi\boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu} - \boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\mathbf{M}^T\mathbf{v}\end{aligned}$$

Now, $\boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu} = \boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\left(\mathbf{M}^T\mathbf{M}\right)\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu} = \|\mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu}\|^2 > 0$. Thus, to minimise the $\boldsymbol{\mu}^T\mathbf{x}$ term and G in general, we always use the negative root of ϕ.

B.1.2 Proof that the General Cost Function's contours are sometimes ellipsoid

Here we show that the contours of the cost function B.1 are ellipsoid under certain conditions. Let c be the cost of G at a particular contour. Then we have:

$$\lambda||\mathbf{u} + \mathbf{M}\mathbf{x}|| + \boldsymbol{\mu}^T\mathbf{x} + d = c$$

Substituting $e = d - c$ and re-arranging:

$$\boldsymbol{\mu}^T\mathbf{x} + e = -\lambda||\mathbf{u} + \mathbf{M}\mathbf{x}||$$

We then square both sides of the equation and re-arrange:

$$\begin{aligned}
&\left(\boldsymbol{\mu}^T\mathbf{x}\right)^2 + 2e\boldsymbol{\mu}^T\mathbf{x} + e^2 = \lambda^2\left(\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x} + 2\mathbf{u}^T\mathbf{M}\mathbf{x} + \mathbf{u}^T\mathbf{u}\right)\\
\Rightarrow&\mathbf{x}^T\boldsymbol{\mu}\boldsymbol{\mu}^T\mathbf{x} + 2e\boldsymbol{\mu}^T\mathbf{x} + e^2 = \lambda^2\left(\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x} + 2\mathbf{u}^T\mathbf{M}\mathbf{x} + \mathbf{u}^T\mathbf{u}\right)\\
\Rightarrow&2e\boldsymbol{\mu}^T\mathbf{x} - 2\lambda^2\mathbf{u}^T\mathbf{M}\mathbf{x} + e^2 - \lambda^2\mathbf{u}^T\mathbf{u} = \lambda^2\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x} - \mathbf{x}^T\boldsymbol{\mu}\boldsymbol{\mu}^T\mathbf{x}\\
\Rightarrow&2\left(e\boldsymbol{\mu}^T - \lambda^2\mathbf{u}^T\mathbf{M}\right)\mathbf{x} + e^2 - \lambda^2\mathbf{u}^T\mathbf{u} = \mathbf{x}^T\left(\lambda^2\mathbf{M}^T\mathbf{M} - \boldsymbol{\mu}\boldsymbol{\mu}^T\right)\mathbf{x}\\
\Rightarrow&\mathbf{x}^T\left(\lambda^2\mathbf{M}^T\mathbf{M} - \boldsymbol{\mu}\boldsymbol{\mu}^T\right)\mathbf{x} - 2\left(e\boldsymbol{\mu}^T - \lambda^2\mathbf{u}^T\mathbf{M}\right)\mathbf{x} - e^2 + \lambda^2\mathbf{u}^T\mathbf{u} = 0
\end{aligned}$$

This is a quadratic polynomial equation, defining a quadric whose nature is determined by the $m \times m$, square, symmetric and real matrix $\left(\lambda^2\mathbf{M}^T\mathbf{M} - \boldsymbol{\mu}\boldsymbol{\mu}^T\right)$. If it is *positive definite*, then the quadratic polynomial equation represents an *ellipsoid quadric surface*. Otherwise, it represents a *hyperboloid*. To be positive definite, the following must hold for all $x \neq 0$:

$$\begin{aligned}
&\mathbf{x}^T\left(\lambda^2\mathbf{M}^T\mathbf{M} - \boldsymbol{\mu}\boldsymbol{\mu}^T\right)\mathbf{x} > 0\\
\Rightarrow&\lambda^2\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x} - \mathbf{x}^T\boldsymbol{\mu}\boldsymbol{\mu}^T\mathbf{x} > 0\\
\Rightarrow&\lambda^2\mathbf{x}^T\mathbf{M}^T\mathbf{M}\mathbf{x} > \mathbf{x}^T\boldsymbol{\mu}\boldsymbol{\mu}^T\mathbf{x}\\
\Rightarrow&\lambda^2||\mathbf{M}\mathbf{x}||^2 > \left(\boldsymbol{\mu}^T\mathbf{x}\right)^2
\end{aligned}$$

Now, if the above did not hold, i.e. $\lambda^2||\mathbf{M}\mathbf{x}||^2 \leq \left(\boldsymbol{\mu}^T\mathbf{x}\right)^2 \forall x$, it would imply that the weighted distance $\lambda||\mathbf{M}\mathbf{x}||$ to the origin is cheaper than the linear component's value at $\mathbf{x}$.

Geometrically, this means that after travelling weighted distance $\lambda||\mathbf{u} + \mathbf{M}\mathbf{x}||$, it is cheaper to take the weighted distance $\lambda||\mathbf{M}\mathbf{x}||$ to the origin instead of using the linear component's value at $\mathbf{x}$, i.e. $\lambda||\mathbf{u} + \mathbf{M}\mathbf{x}|| + \lambda||\mathbf{M}\mathbf{x}|| \leq \lambda||\mathbf{u} + \mathbf{M}\mathbf{x}|| + \mathbf{u}^T\mathbf{x}\ \forall x$.

But by the triangle inequality $\lambda||\mathbf{u}|| \leq \lambda||\mathbf{u} + \mathbf{M}\mathbf{x}|| + \lambda||\mathbf{M}\mathbf{x}||\ \forall x$ – It is always cheaper to travel directly to the origin. In practice, these situations occur when the gradient of the linear component, $\boldsymbol{\mu}^T\mathbf{x} + d$, is very steep compared to the weighting of the distance component. The linear component dominates the distance component to the extent that a global minimum no longer exists.

Thus, for a global minimum to be present, $\left(\lambda^2\mathbf{M}^T\mathbf{M} - \boldsymbol{\mu}\boldsymbol{\mu}^T\right)$ must be positive definite, in which case the contour lines of the cost function will be ellipsoids. Note that the positive definite requirement on this matrix is similar to the requirement that $\lambda^2 - \boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu}$ be positive for a real value of ϕ to exist.

B.2 Extended General Cost Function

The extended general cost function is expressed as:

$$G\left(\mathbf{x}, \mathbf{y}, \lambda, \beta, \mathbf{u}, \mathbf{M}, \mathbf{N}, \boldsymbol{\mu}, d\right) = \lambda\|\mathbf{u} + \mathbf{M}\mathbf{x} + \mathbf{N}\mathbf{y}\| + \beta\|\mathbf{N}\mathbf{y}\| + \boldsymbol{\mu}^T\mathbf{x} + d \tag{B.3}$$

If we set $\mathbf{v} = \mathbf{u} + \mathbf{N}\mathbf{y}$ take the partial derivative with respect to $\mathbf{x}$, $\partial G/\partial\mathbf{x}$, set it to zero and solve we obtain the same solution for $\mathbf{x}$ and ϕ as in the Section B.1.

Now we wish to solve for $\mathbf{y}$. We set $\mathbf{w} = \mathbf{u} + \mathbf{M}\mathbf{x}$ so that:

$$G\left(\mathbf{x}, \mathbf{y}, \lambda, \beta, \mathbf{u}, \mathbf{M}, \mathbf{N}, \boldsymbol{\mu}, d\right) = \lambda\|\mathbf{w} + \mathbf{N}\mathbf{y}\| + \beta\|\mathbf{N}\mathbf{y}\| + \boldsymbol{\mu}^T\mathbf{x} + d$$

and take the partial derivative with respect to $\mathbf{y}$, $\partial G/\partial\mathbf{y}$:

$$\frac{\partial G}{\partial \mathbf{y}} = -\lambda\frac{\left(\mathbf{w}^T\mathbf{N} + \mathbf{y}^T\mathbf{N}^T\mathbf{N}\right)}{\|\mathbf{w} + \mathbf{N}\mathbf{y}\|} - \beta\frac{\left(\mathbf{y}^T\mathbf{N}^T\mathbf{N}\right)}{\|N\mathbf{y}\|} \tag{B.4}$$

Setting $\partial G/\partial\mathbf{y} = 0$, $\xi = -\lambda/\|\mathbf{w} + \mathbf{N}\mathbf{y}\|$ and $\delta = -\beta//\|\mathbf{N}\mathbf{y}\|$, we have:

$$\begin{aligned}
&\xi\left(\mathbf{w}^T\mathbf{N} + \mathbf{y}^T\mathbf{N}^T\mathbf{N}\right) + \delta\mathbf{y}^T\mathbf{N}^T\mathbf{N} = 0\\
\Rightarrow&\mathbf{y}^T\mathbf{N}^T\mathbf{N}\left(\xi + \delta\right) = -\xi\mathbf{w}^T\mathbf{N}\\
\Rightarrow&\mathbf{y}^T = -\frac{\xi}{\xi + \delta}\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}
\end{aligned}$$

For convenience we combine the ξ and δ terms into θ and express $\mathbf{y}^T$ and $\mathbf{y}$ as follows:

$$\mathbf{y}^T = \theta\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1} \tag{B.5}$$

$$\mathbf{y} = \theta\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{w} \tag{B.6}$$

Working from Equation B.4 we obtain the following expression:

$$-\lambda\|\mathbf{N}\mathbf{y}\|\left(\mathbf{w}^T\mathbf{N} + \mathbf{y}^T\mathbf{N}^T\mathbf{N}\right) = \beta\|\mathbf{w} + \mathbf{N}\mathbf{y}\|\mathbf{y}^T\mathbf{N}^T\mathbf{N}$$

Squaring both sides yields:

$$\lambda^2\left(\mathbf{y}^T\mathbf{N}^T\mathbf{N}\mathbf{y}\right)\left(\mathbf{w}^T\mathbf{N}+\mathbf{y}^T\mathbf{N}^T\mathbf{N}\right)\cdot\left(\mathbf{N}^T\mathbf{w}+\mathbf{N}^T\mathbf{N}\mathbf{y}\right)=$$
$$\beta^2\left(\mathbf{w}^T+\mathbf{y}^T\mathbf{N}^T\right)\cdot\left(\mathbf{w}+\mathbf{N}\mathbf{y}\right)\left(\mathbf{y}^T\mathbf{N}^T\mathbf{N}\mathbf{N}^T\mathbf{N}\mathbf{y}\right)$$

$$\Rightarrow\quad\lambda^2\left(\mathbf{y}^T\mathbf{N}^T\mathbf{N}\mathbf{y}\right)\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}+2\mathbf{y}^T\mathbf{N}^T\mathbf{N}\mathbf{N}^T\mathbf{w}+\mathbf{y}^T\mathbf{N}^T\mathbf{N}\mathbf{N}^T\mathbf{N}\mathbf{y}\right)=$$
$$\beta^2\left(\mathbf{w}^T\mathbf{w}+2\mathbf{y}^T\mathbf{N}^T\mathbf{w}+\mathbf{y}^T\mathbf{N}^T\mathbf{N}\mathbf{y}\right)\left(\mathbf{y}^T\mathbf{N}^T\mathbf{N}\mathbf{N}^T\mathbf{N}\mathbf{y}\right)$$

Substituting the values of $\mathbf{y}^T$ and $\mathbf{y}$ from Equation B.5 and B.6 into the above produces:

$$\Big(\lambda^2\theta^2\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\left(\mathbf{N}^T\mathbf{N}\right)\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{w}\Big)$$
$$\Big(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}+2\theta\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{N}\mathbf{N}^T\mathbf{w}+\theta^2\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{N}\mathbf{N}^T\mathbf{N}\left(\mathbf{N}^{\mathbf{N}}\right)^{-1}\mathbf{N}^T\mathbf{w}\Big)$$
$$=\beta^2\theta^2\Big(\mathbf{w}^T\mathbf{w}+2\theta\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{w}+\theta^2\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{w}\Big)$$
$$\Big(\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{N}\mathbf{N}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{w}\Big)$$

Note that $\left(\mathbf{N}^T\mathbf{N}\right)^{-1}$ multiplied by $\left(\mathbf{N}^T\mathbf{N}\right)$ produces the identity matrix. We substitute $\mathbf{P}_N=\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T$ for convenience noting that $\mathbf{P}_N$ is a matrix defining the orthogonal projection onto $\mathbf{N}$. Expanding out produces:

$$\lambda^2\theta^2\mathbf{w}^T\mathbf{P}_N\mathbf{w}\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}+2\theta\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}+\theta^2\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\right)$$
$$=\beta^2\theta^2\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\left(\mathbf{w}^T\mathbf{w}+2\theta\mathbf{w}^T\mathbf{P}_N\mathbf{w}+\theta^2\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right)$$

Expanding further produces:

$$\begin{aligned}
&\lambda^2\theta^2\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right)\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\right)+\\
&2\lambda^2\theta^3\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right)\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\right)+\\
&\lambda^2\theta^4\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right)\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\right)+\\
&\beta^2\theta^2\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\right)\left(\mathbf{w}^T\mathbf{w}\right)+\\
&2\beta^2\theta^3\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right)\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\right)+\\
&\beta^2\theta^4\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\right)\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right)=0
\end{aligned}$$

Grouping by θ:

$$\begin{aligned}
&\theta^4\left(\lambda^2-\beta^2\right)\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right)\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\right)+\\
&2\theta^3\left(\lambda^2-\beta^2\right)\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right)\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\right)+\\
&\theta^2\left(\lambda^2\mathbf{w}^T\mathbf{P}_N\mathbf{w}-\beta^2\mathbf{w}^T\mathbf{w}\right)\left(\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}\right)=0
\end{aligned}$$

The above can hold if $\mathbf{w}^T\mathbf{N}\mathbf{N}^T\mathbf{w}=0$ or $\theta=0$. If this is not the case however, we need to solve the following quadratic:

$$\begin{aligned}
&C\theta^2 + 2C\theta + D = 0 \\
\text{where}& \\
&C = \left(\lambda^2 - \beta^2\right)\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right) \\
&D = \lambda^2\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right) - \beta^2\left(\mathbf{w}^T\mathbf{w}\right)
\end{aligned}$$

According to the quadratic formula:

$$\begin{aligned}
\theta &= -\frac{-2C \pm \sqrt{4C^2 - 4CD}}{2C} \\
\theta &= -1 \pm \frac{\sqrt{C^2 - CD}}{C} \\
\Rightarrow \theta &= -1 \pm \sqrt{\frac{C - D}{C}}
\end{aligned}$$

Utilising the fact that $C - D$ reduces to the following:

$$C - D = \beta^2\left(\mathbf{w}^T\mathbf{w} - \mathbf{w}^T\mathbf{P}_N\mathbf{w}\right)$$

we obtain the following formula for θ:

$$\theta = -1 \pm \sqrt{\frac{\beta^2\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w} - \mathbf{w}^T\mathbf{w}\right)}{\left(\lambda^2 - \beta^2\right)\left(\mathbf{w}^T\mathbf{P}_N\mathbf{w}\right)}}$$

B.2.1 Eliminating $\mathbf{y}$

For the formula:

$$G\left(\mathbf{x}, \mathbf{y}, \lambda, \beta, \mathbf{u}, \mathbf{M}, \mathbf{N}, \boldsymbol{\mu}, d\right) = \lambda\|\mathbf{u} + \mathbf{M}\mathbf{x} + \mathbf{N}\mathbf{y}\| + \beta\|\mathbf{N}\mathbf{y}\| + \boldsymbol{\mu}^T\mathbf{x} + d \tag{B.7}$$

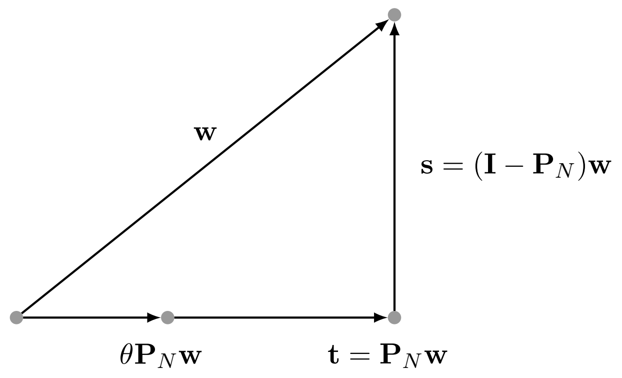

$\mathbf{w}$
$\mathbf{s} = (\mathbf{I} - \mathbf{P}_N)\mathbf{w}$
$\theta \mathbf{P}_N \mathbf{w}$
$\mathbf{t} = \mathbf{P}_N \mathbf{w}$

Figure B.1: The vectors $\mathbf{s}$ and $\mathbf{t}$ are in the nullspace and range of projection $\mathbf{P}_N$ respectively. Therefore, they are orthogonal to one another.

we have the following solution for $\mathbf{x}$ and $\mathbf{y}$:

$$\begin{aligned}
\mathbf{x}^T &= \left(\phi\boldsymbol{\mu}^T - \mathbf{v}^T\mathbf{M}\right)\left(\mathbf{M}^T\mathbf{M}\right)^{-1} \\
\mathbf{y}^T &= \theta\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1} \\
\phi &= \pm\sqrt{\frac{\mathbf{v}^T\left(\mathbf{I} - \mathbf{M}\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\right)\mathbf{v}}{\lambda^2 - \boldsymbol{\mu}^T\left(\mathbf{M}^T\mathbf{M}\right)^{-1}\boldsymbol{\mu}}} \\
\theta &= -1 \pm \sqrt{\frac{\beta^2\left(\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{w} - \mathbf{w}^T\mathbf{w}\right)}{\left(\lambda^2 - \beta^2\right)\left(\mathbf{w}^T\mathbf{N}\left(\mathbf{N}^T\mathbf{N}\right)^{-1}\mathbf{N}^T\mathbf{w}\right)}} \\
\mathbf{v} &= \mathbf{u} + \mathbf{N}\mathbf{y} \\
\mathbf{w} &= \mathbf{u} + \mathbf{M}\mathbf{x}
\end{aligned}$$

Since $\mathbf{v}$ and $\mathbf{w}$ still contain $\mathbf{y}$ and $\mathbf{x}$ respectively, the solutions for $\mathbf{x}$ and $\mathbf{y}$ are not independent of each other. We will now eliminate $\mathbf{y}$ and θ from G.

First we substitute $\mathbf{w}$ and $\mathbf{y}$ into Equation B.7, substituting $\mathbf{P}_N$, the orthogonal projection onto basis $\mathbf{N}$ from the previous section:

$$G\left(\mathbf{x}, \mathbf{y}, \lambda, \beta, \mathbf{u}, \mathbf{M}, \mathbf{N}, \boldsymbol{\mu}, d\right) = \lambda\|\mathbf{w} + \theta\mathbf{P}_N\mathbf{w}\| + \beta\|\theta\mathbf{P}_N\mathbf{w}\| + \boldsymbol{\mu}^T\mathbf{x} + d \tag{B.8}$$

Now, setting $\mathbf{s} = (\mathbf{I} - \mathbf{P}_N)\mathbf{w}$, a vector in the *nullspace* of $\mathbf{P}_N$ and $\mathbf{t} = \mathbf{P}_N\mathbf{w}$, a vector in the *range* of $\mathbf{P}_N$, we proceed to transform the distance components of Equation B.8, taking advantage of the fact that $\mathbf{s} \cdot \mathbf{t} = 0$, since elements of the nullspace and range of a projection are orthogonal to one another, as shown in Figure B.1.

The first distance component can be represented in terms of an $\mathbf{s}$ and $\mathbf{t}$ as follows:

$$\begin{aligned}
\lambda\|\mathbf{w} + \theta\mathbf{P}_N\mathbf{w}\| &= \lambda\|\mathbf{t} + \mathbf{s} + \theta\mathbf{t}\| \\
&= \lambda\|\mathbf{s} + (1+\theta)\,\mathbf{t}\| \\
&= \lambda\sqrt{\|\mathbf{s}\|^2 + 2\,(1+\theta)\,\mathbf{s}\cdot\mathbf{t} + (1+\theta)^2\,\|\mathbf{t}\|^2} \\
&= \lambda\sqrt{\|\mathbf{s}\|^2 + (1+\theta)^2\,\|\mathbf{t}\|^2}
\end{aligned}$$

now working with the definition of θ:

$$\begin{aligned}
(1+\theta)^2 &= \left(\frac{\beta^2}{\lambda^2-\beta^2}\right)\left(\frac{\mathbf{w}^T\mathbf{w} - \mathbf{w}^T\mathbf{P}_N\mathbf{w}}{\mathbf{w}^T\mathbf{P}_N\mathbf{w}}\right) \\
&= \left(\frac{\beta^2}{\lambda^2-\beta^2}\right)\left(\frac{\|\mathbf{t}+\mathbf{s}\|^2 - \|\mathbf{t}\|^2}{\|\mathbf{t}\|^2}\right) \\
&= \left(\frac{\beta^2}{\lambda^2-\beta^2}\right)\left(\frac{\|\mathbf{s}\|^2}{\|\mathbf{t}\|^2}\right)
\end{aligned}$$

Thus:

$$\begin{aligned}
\lambda\sqrt{\|\mathbf{s}\|^2 + (1+\theta)^2\,\|\mathbf{t}\|^2} &= \lambda\sqrt{\|\mathbf{s}\|^2 + \frac{\beta^2}{\lambda^2-\beta^2}\|\mathbf{s}\|^2} \\
&= \lambda\sqrt{\|\mathbf{s}\|^2\left(1 + \frac{\beta^2}{\lambda^2-\beta^2}\right)} \\
&= \lambda\|\mathbf{s}\|\sqrt{\frac{\lambda^2}{\lambda^2-\beta^2}}
\end{aligned}$$

Working with the second distance component:

$$\begin{aligned}
\beta\|\theta\mathbf{P}_N\mathbf{w}\| &= \beta\|\theta\mathbf{t}\| \\
&= \beta\sqrt{\theta^2\|\mathbf{t}\|^2}
\end{aligned}$$

and utilising the fact that:

$$
\begin{aligned}
\theta &= -1 \pm \frac{\|\mathbf{s}\|}{\|\mathbf{t}\|}\sqrt{\frac{\beta^2}{\lambda^2-\beta^2}} \\
\Rightarrow \theta^2 &= 1 \pm 2\frac{\|\mathbf{s}\|}{\|\mathbf{t}\|}\sqrt{\frac{\beta^2}{\lambda^2-\beta^2}} + \|\mathbf{s}\|^2\frac{\beta^2}{\lambda^2-\beta^2} \\
\Rightarrow \theta^2\|\mathbf{t}\|^2 &= \|\mathbf{t}\|^2 \pm 2\|\mathbf{s}\|\|\mathbf{t}\|\sqrt{\frac{\beta^2}{\lambda^2-\beta^2}} + \|\mathbf{s}\|^2\frac{\beta^2}{\lambda^2-\beta^2} \\
\Rightarrow \theta^2\|\mathbf{t}\|^2 &= \left(\|\mathbf{t}\| \pm \|\mathbf{s}\|\sqrt{\frac{\beta^2}{\lambda^2-\beta^2}}\right)^2 \\
\Rightarrow \sqrt{\theta^2\|\mathbf{t}\|^2} &= \|\mathbf{t}\| \pm \|\mathbf{s}\|\sqrt{\frac{\beta^2}{\lambda^2-\beta^2}}
\end{aligned}
$$

Thus,

$$
\begin{aligned}
\lambda\|\mathbf{w}+\theta\mathbf{P}_N\mathbf{w}\| + \beta\|\theta\mathbf{P}_N\mathbf{w}\| &= \lambda\|\mathbf{s}\|\sqrt{\frac{\lambda^2}{\lambda^2-\beta^2}} + \beta\|\mathbf{t}\| \pm \beta\|\mathbf{s}\|\sqrt{\frac{\beta^2}{\lambda^2-\beta^2}} \\
&= \|\mathbf{s}\|\frac{\lambda^2}{\sqrt{\lambda^2-\beta^2}} + \beta\|\mathbf{t}\| \pm \|\mathbf{s}\|\frac{\beta^2}{\sqrt{\lambda^2-\beta^2}} \\
&= \|\mathbf{s}\|\frac{\lambda^2 \pm \beta^2}{\sqrt{\lambda^2-\beta^2}} + \beta\|\mathbf{t}\| \\
&= \|\left(\mathbf{I}-\mathbf{P}_N\right)\mathbf{w}\|\frac{\lambda^2 \pm \beta^2}{\sqrt{\lambda^2-\beta^2}} + \beta\|\mathbf{P}_N\mathbf{w}\|
\end{aligned}
$$

we have eliminated $\mathbf{y}$ from Equation B.7. which can now be expressed as:

$$
\begin{aligned}
G\left(\mathbf{x},\lambda,\beta,\mathbf{u},\mathbf{M},\mathbf{N},\boldsymbol{\mu},d\right) = \frac{\lambda^2 \pm \beta^2}{\sqrt{\lambda^2-\beta^2}}\|\left(\mathbf{I}-\mathbf{P}_N\right)\left(\mathbf{u}+\mathbf{M}\mathbf{x}\right)\|+ \\
\beta\|\mathbf{P}_N\left(\mathbf{u}+\mathbf{M}\mathbf{x}\right)\| + \boldsymbol{\mu}^T\mathbf{x} + d
\end{aligned}
\tag{B.9}
$$

There are two distance components: The first expresses the distance of a vector $\mathbf{u} + \mathbf{M}\mathbf{x}$ and its projection onto $\mathbf{N}$. while the second expresses the magnitude of $\mathbf{u} + \mathbf{M}\mathbf{x}$ projected onto $\mathbf{N}$. These two distance components are orthogonal two each other, and thus the distance components from the original Equation have been transformed so that they lie on the *catheti* of a right-angled triangle.

We have not managed to obtain a general analytic solution for Equation B.9 since two distance terms and one linear term contain $\mathbf{x}$. Attempts at solving B.9 by minimisation suggest that it is necessary to solve for an eighth degree polynomial in $\mathbf{x}$. However, in certain cases, a distance term is linear and 5.10 reduces to 5.6, for which an analytic solution is available.

Bibliography

[1] J. M. Airey, J. H. Rohlf, and F. P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Proceeeding of ACM Symposium on Interactive 3D Graphics*, pages 41–50, 1990.

[2] Lyudmil Aleksandrov, Hristo Djidjev, Hua Guo, Anil Maheshwari, Doron Nussbaum, and Jrg-Rdiger Sack. Approximate Shortest Path Queries on Weighted Polyhedral Surfaces. In Rastislav Krlovic and Pawel Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006*, volume 4162 of *Lecture Notes in Computer Science*, pages 98–109. Springer Berlin / Heidelberg, 2006.

[3] Lyudmil Aleksandrov, Anil Maheshwari, and Jörg-Rüdiger Sack. Approximation algorithms for geometric shortest path problems. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 286–295, New York, NY, USA, 2000. ACM.

[4] Pierre Alliez, Laurent Rineau, Stphane Tayeb, Jane Tournois, and Mariette Yvinec. 3D mesh generation. In *CGAL User and Reference Manual*. CGAL Editorial Board, 3.9 edition, 2011. `http://www.cgal.org/Manual/3.9/doc_html/cgal_manual/packages.html#Pkg:Mesh_3`.

[5] Nina Amenta, Marshall Bern, and Manolis Kamvysselis. A new Voronoi-based surface reconstruction algorithm. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 415–421, New York, NY, USA, 1998. ACM Press.

[6] R. Balakrishnan and K. Ranganathan. *A Textbook Of Graph Theory*. Springer (India) Pvt. Ltd., 2007.

[7] C. B. Barber, D. P. Dobkin, and H. T. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[8] M. Bardi and M. Falcone. An Approximation Scheme for the Minimum Time Function. *SIAM Journal on Control and Optimization*, 28(4):950–965, 1990.

[9] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[10] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[11] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[12] Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2nd edition, 1999.

[13] Priyadarshi Bhattacharya and Marina L. Gavrilova. Voronoi diagram in optimal path planning. In *ISVD '07: Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering*, pages 38–47, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Ingmar Bitter, Arie E. Kaufman, and Mie Sato. Penalized-Distance Volumetric Skeleton Algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):195–206, 2001.

[15] Ingmar Bitter, Mie Sato, Michael Bender, Kevin T. McDonnell, Arie Kaufman, and Ming Wan. CEASAR: a smooth, accurate and robust centerline extraction algorithm. In *Proceedings of the conference on Visualization '00*, VIS '00, pages 45–52, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[16] Avi Bleiweiss. GPU Accelerated Pathfinding. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 65–74, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[17] Harry Blum. A Transformation for Extracting New Descriptors of Shape. In Weiant Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, Cambridge, 1967.

[18] J. Borenstein and Y. Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *Robotics and Automation, IEEE Transactions on*, 7(3):278–288, 1991.

[19] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near Optimal Hierarchical Path-finding. *Journal of Game Development*, 1:7–28, 2004.

[20] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[21] Rodney A. Brooks. Elephants don't play chess. *Robot. Auton. Syst.*, 6(1-2):3–15, 1990.

[22] Keegan Carruthers-Smith. A Faster Field D* for Path Planning in Weighted Regions. 2012.

[23] J. Carsten, D. Ferguson, and A. Stentz. 3D Field D*: Improved Path Planning and Replanning in Three Dimensions. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 3381–3386, 2006.

[24] Bernard Chazelle. A Theorem on Polygon Cutting with Applications. In *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 339–349, Washington, DC, USA, 1982. IEEE Computer Society.

[25] Siu-Wing Cheng, Hyeon-Suk Na, Antoine Vigneron, and Yajun Wang. Querying Approximate Shortest Paths in Anisotropic Regions. *SIAM Journal on Computing*, 39:1888–1918, 2010.

[26] Y. T. Chin, H. Wang, L. P. Tay, Wang H., and W. Y. C. Soh. Vision Guided AGV Using Distance Transform. In *Proceedings of the 32nd International Symposium on Robotics*, pages 19–21, 2001.

[27] Sunglok Choi and Wonpil Yu. Any-angle path planning on non-uniform costmaps. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 5615–5621, 2011.

[28] Eric Chown, Stephen Kaplan, and David Kortenkamp. Prototypes, Location, and Associative Networks (PLAN): Towards a Unified Theory of Cognitive Mapping. *Cognitive Science*, 19(1):1–51, 1995.

[29] Yiorgos Chrysanthou and Mel Slater. Shadow volume BSP trees for computation of shadows in dynamic scenes. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 45–50, New York, NY, USA, 1995. ACM Press.

[30] David Coeurjolly and Annick Montanvert. Optimal Separable Algorithms to Compute the Reverse Euclidean Distance Transformation and Discrete Medial Axis in Arbitrary Dimension. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(3):437–448, 2007.

[31] Nicu D. Cornea and Patrick Min. Curve-Skeleton Properties, Applications, and Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):530–548, 2007.

[32] Michel Couprie, Andr Vital Sade, and Gilles Bertrand. Euclidean homotopic skeleton based on critical kernels. In *SIBGRAPI*, pages 307–314. IEEE Computer Society, 2006.

[33] O. Cuisenaire and B. Macq. Fast and exact signed Euclidean distance transformation with linear complexity. In *Proceedings of the Acoustics, Speech, and Signal Processing, 1999. on 1999 IEEE International Conference - Volume 06*, ICASSP '99, pages 3293–3296, Washington, DC, USA, 1999. IEEE Computer Society.

[34] O. Cuisenaire and B. Macq. Fast Euclidean distance transformation by propagation using multiple neighborhoods. *Comput. Vis. Image Underst.*, 76(2):163–172, 1999.

[35] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-Angle Path Planning on Grids. *Journal of Artificial Intelligence (JAIR)*, pages 533–579, 2010.

[36] P.-E. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

[37] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, pages 269–271, 1959.

[38] I. Capuzzo Dolcetta. On a discrete approximation of the Hamilton-Jacobi equation of dynamic programming. *Applied Mathematics and Optimization*, 10:367–377, 1983.

[39] I. Capuzzo Dolcetta and M. Falcone. Discrete dynamic programming and viscosity solutions of the Bellman equation. *Annales de l'institut Henri Poincar (C) Analyse non linaire*, S6:161–183, 1989.

[40] Ramsay Dyer, Hao Zhang, and Torsten Möller. Surface sampling and the intrinsic Voronoi diagram. In *Proceedings of the Symposium on Geometry Processing*, SGP '08, pages 1393–1402, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[41] Herbert Edelsbrunner, M. J. Ablowitz, S. H. Davis, E. J. Hinch, A. Iserles, J. Ockendon, and P. J. Olver. *Geometry and Topology for Mesh Generation (Cambridge Monographs on Applied and Computational Mathematics)*. Cambridge University Press, New York, NY, USA, 2006.

[42] A. Elfes. Using Occupancy Grids for Mobile Robot Perception and Navigation. *Computer*, 22(6):46–57, 1989.

[43] Alberto Elfes. *Occupancy grids: a probabilistic framework for robot perception and navigation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1989.

[44] R. Fabbri, L. da F. Costa, J. C. Torelli, and O. M. Bruno. 2D Euclidean Distance Transform Algorithms: A Comparative Survey. *ACM Computing Surveys*, 40(1):1–44, 2008.

[45] M. Falcone. A numerical approach to the infinite horizon problem of deterministic control theory. *Applied Mathematics and Optimization*, 15:1–13, 1987.

[46] A. Fedorov, N. Chrisochoides, R. Kikinis, and S. Warfield. Tetrahedral mesh generation for medical imaging. Technical report, 2005.

[47] David Ferguson and Anthony Stentz. Multi-resolution Field D*. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)*, 2006.

[48] David Ferguson and Anthony (Tony) Stentz. The Field D* Algorithm for Improved Path Planning and Replanning in Uniform and Non-Uniform Cost Environments. Technical Report CMU-RI-TR-05-19, Robotics Institute, Pittsburgh, PA, 2005.

[49] David Ferguson and Anthony (Tony) Stentz. Using Interpolation to Improve Path Planning: The Field D* Algorithm. *Journal of Field Robotics*, 23(2):79–101, 2006.

[50] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, 1962.

[51] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[52] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980. ACM Press.

[53] Yaorong Ge and J. Michael Fitzpatrick. On the Generation of Skeletons from Discrete Euclidean Distance Maps. *IEEE Trans. Pattern Anal. Mach. Intell.*, 18(11):1055–1066, 1996.

[54] H. Goldstein, C. Poole, and I. Safko. *Classical Mechanics (3rd Edition)*. Addison Wesley, 3 edition, 2001.

[55] N. A. Golias and R. W. Dutton. Delaunay Triangulation and 3D Adaptive Mesh Generation. *Finite Elements in Analysis and Design*, 25(3-4):331–341, 1997.

[56] R. Gonzalez and E. Rofman. On Deterministic Control Problems: An Approximation Procedure for the Optimal Cost I. The Stationary Problem. *SIAM Journal on Control and Optimization*, 23(2):242–266, 1985.

[57] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 213–222, New York, NY, USA, 1984. ACM.

[58] L. Guibas, C. Holleman, and L. Kavraki. A probabilistic roadmap planner for flexible objects with a workspace medial axis. In *Proceedings of the IEEE International Conference on Intelligent Robots*, 1999.

[59] J. Guivant, E. Nebot, and H. Durrant-Whyte. Simultaneous localization and map building using natural features in outdoor environments. *Intelligent Autonomous Systems*, 6(1):581–586, 2000.

[60] Shuai Guo, Shugen Ma, Bin Li, Minghui Wang, and Yuechao Wang. A new hybrid metric map representation by using voronoi diagram and its application to SLAM. In *Information and Automation (ICIA), 2012 International Conference on*, pages 400–405, 2012.

[61] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[62] C. Holleman and L. E. Kavraki. A framework for using the workspace medial axis in PRM planners. In *Proceedings of the International Conference on Robotics and Automation*, pages 1408–1413, 2000.

[63] Ping Hu, Hui Chen, Wen Wu, and Pheng-Ann Heng. Multi-Tissue Tetrahedral Mesh Generation from Medical Images. In *Bioinformatics and Biomedical Engineering (iCBBE), 2010 4th International Conference on*, pages 1–4, 2010.

[64] David Jacka. MSc Dissertation: High-Level Control of Agent-based Crowds by means of General Constraints. Technical report, University of Cape Town, 2009.

[65] B. K. Jang and R. T. Chin. Analysis of Thinning Algorithms Using Mathematical Morphology. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(6):541–551, 1990.

[66] Mark W. Jones, J. Andreas Baerentzen, and Milos Sramek. 3D Distance Fields: A Survey of Techniques and Applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, 2006.

[67] G. Kalai and G.M. Ziegler. *Polytopes : Combinatorics and Computation*. Basel: Birkhuser Verlag, 2000.

[68] M. Kallman. Path Planning in Triangulations. In *Proceedings of the IJACI Workshop on Reasoning, Representation and Learning in Computer Games*, 2005.

[69] W. Karush. Minima of Functions of Several Variables with Inequalities as Side Constraints. Master's thesis, Department of Mathematics, University of Chicago, 1939.

[70] Donald E. Kirk. *Optimal Control Theory: An Introduction*. Dover Publications, 2004.

[71] Richard I. Klein. Star formation with 3-D adaptive mesh refinement: the collapse and fragmentation of molecular clouds. *Journal of Computational and Applied Mathematics*, 109(12):123–152, 1999.

[72] George J. Klir and Bo Yuan, editors. *Fuzzy sets, fuzzy logic, and fuzzy systems: selected papers by Lotfi A. Zadeh*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[73] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning A*. *Artificial Intelligence*, 155:93–146, 2004.

[74] K. Konolige. A gradient method for realtime robot control. In *Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, volume 1, pages 639–646 vol.1, 2000.

[75] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, September 1985.

[76] H. W. Kuhn and A. W. Tucker. Nonlinear Programming. In Jerzy Neyman, editor, *Proceedings of the 2nd Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492. University of California Press, Berkeley, CA, USA, 1950.

[77] Benjamin Kuipers. Modeling Spatial Knowledge. *Cognitive Science*, 2:129–153, 1978.

[78] Benjamin Kuipers and Yung-Tai Byun. A Robot Exploration and Mapping Strategy Based on a Semantic Hierarchy of Spatial Representations. *JOURNAL OF ROBOTICS AND AUTONOMOUS SYSTEMS*, 8:47–63, 1991.

[79] H. Kushner. Numerical Methods for Stochastic Control Problems in Continuous Time. *SIAM Journal on Control and Optimization*, 28(5):999–1048, 1990.

[80] Mark Lanthier, Anil Maheshwari, and Jörg-Rüdiger Sack. Approximating weighted shortest paths on polyhedral surfaces. In *Proceedings of the thirteenth annual symposium on Computational geometry*, SCG '97, pages 485–486, New York, NY, USA, 1997. ACM.

[81] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[82] Greg Leibon and David Letscher. Delaunay triangulations and Voronoi diagrams for Riemannian manifolds. In *Proceedings of the sixteenth annual symposium on Computational geometry*, SCG '00, pages 341–349, New York, NY, USA, 2000. ACM.

[83] A. Lerner, Y. Chrysanthou, and D. Cohen-Or. Breaking the walls: scene partitioning and portal creation. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, pages 303–312, 2003.

[84] Hua Li and Anthony Yezzi. Vessels as 4D Curves: Global Minimal 4D Paths to Extract 3D Tubular Surfaces. In *Proceedings of the 2006 Conference on Computer Vision and Pattern Recognition Workshop*, CVPRW '06, pages 82–, Washington, DC, USA, 2006. IEEE Computer Society.

[85] L. Lidén. The Integration of Autonomous and Scripted Behaviour through Task Management. In *Artificial Intelligence and Interactive Entertainment, AAAI Spring Symposium*, 2000.

[86] Koenig, S. and M. Likhachev. D* Lite. In *Eighteenth national conference on Artificial intelligence*, pages 476–483, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[87] Maxim Likhachev and Dave Ferguson. Planning Long Dynamically Feasible Maneuvers for Autonomous Vehicles. *Int. J. Rob. Res.*, 28(8):933–945, 2009.

[88] Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22:560–570, 1979.

[89] David Luebke and Chris Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 105–ff., New York, NY, USA, 1995. ACM Press.

[90] Christian S. Mata and Joseph S. B. Mitchell. 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, pages 264–273, New York, NY, USA, 1997. ACM.

[91] B. Merry, S. Perkins, P. Marais, and J. Gain. Proof of Field D*'s Case Separation for Arbitrary Simplices. Technical Report CS12-07-00, Department of Computer Science, University of Cape Town, 2012.

[92] Joseph S. B. Mitchell, David M. Mount, and Christos H. Papadimitriou. The discrete geodesic problem. *SIAM Journal of Computing*, 16:647–668, 1987.

[93] Joseph S. B. Mitchell and Christos H. Papadimitriou. The Weighted Region Problem: Finding Shortest Paths through a Weighted Planar Subdivision. *J. ACM*, 38(1):18–73, 1991.

[94] Masaharu Mizumoto. Defuzzification. In *Handbook of Fuzzy Computation*. IOP Publishing Ltd., 1998.

[95] Edwin E. Moise. Affine Structures in 3-Manifolds: V. The Triangulation Theorem and Hauptvermutung. *Annals of Mathematics*, 56(1):pp. 96–114, 1952.

[96] Hans Moravec. Sensor fusion in certainty grids for mobile robots. *AI Mag.*, 9(2):61–74, 1988.

[97] David Morgan. Algorithmic Approaches to Finding Cover in Three-Dimensional, Virtual Environments. Masters thesis, MOVES Institute, 2003.

[98] A. Nash, K. Daniel, Koenig S., and Felner A. Theta *: Any-Angle Path Planning on Grids. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1177–1183, 2007.

[99] J. I. Nieto, J. E. Guivant, and E. M. Nebot. The HYbrid metric maps (HYMMs): a novel map representation for denseSLAM. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 1, pages 391–396 Vol.1, 2004.

[100] Juan Nieto, Jose Guivant, and Eduardo Nebot. DenseSLAM: Simultaneous localization and dense mapping. *The International Journal of Robotics Research*, 25(8):711–744, 2006.

[101] Nils J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, 1993.

[102] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2000.

[103] R. Ogniewicz. A Multiscale MAT from Voronoi Diagrams: The Skeleton-Space and its Application to Shape Description and Decomposition. *Aspects of Visual Form Processing*, pages 430–439, 1994.

[104] J. C. O'Neill. Efficient Navigation Mesh Implementation. *Journal of Game Development*, 2002.

[105] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, 1988.

[106] S. Perkins, P. Marais, J. Gain, and M. Berman. Field D* path-finding on Weighted Triangulated and Tetrahedral Meshes. *Autonomous Agents and Multi-Agent Systems*, pages 1–35, 2012.

[107] T K Peucker, R J Fowler, J J Little, and D M Mark. The triangulated irregular network. *Amer Soc Photogrammetry Proc Digital Terrain Models Symposium*, 516:96–103, 1978.

[108] Roland Philippsen. A Light Formulation of the E* Interpolated Path Replanner. Technical report, Autonomous Systems Lab, Ecole Polytechnique Federale de Lausanne, 2006.

[109] Roland Philippsen and Roland Siegwart. An Interpolated Dynamic Navigation Function. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2005.

[110] D. C. Pottinger. Terrain Analysis in Realtime Strategy Games. In *Proceedings of Computer Game Developer Conference*, 2000.

[111] C. W. Reynolds. Steering Behaviors For Autonomous Characters. In *Proceedings of Game Developers Conference*, San Francisco, California, 1999. Miller Freeman Game Group.

[112] Azriel Rosenfeld and John L. Pfaltz. Sequential Operations in Digital Picture Processing. *J. ACM*, 13(4):471–494, 1966.

[113] Azriel Rosenfeld and John L. Pfaltz. Distance functions on digital pictures. *Pattern Recognition*, 1(1):33–61, 1968.

[114] B. Roy. Transitivité et connexité. *C. R. Acad. Sci. Paris*, 249:216–218, 1959.

[115] Stuart Russell. Efficient memory-bounded search methods. In *Proceedings of the 10th European conference on Artificial intelligence*, ECAI '92, pages 1–5, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[116] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2 edition, 2010.

[117] Toyofumi Saito and Jun-Ichiro Toriwaki. New algorithms for euclidean distance transformation of an n-dimensional digitized picture with applications. *Pattern Recognition*, 27(11):1551–1565, 1994.

[118] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[119] Lenny Sapronov and Alberto Lacaze. Path planning for robotic vehicles using Generalized Field D*. volume 6962, pages 69621C–69621C–12. SPIE, 2008.

[120] André Vital Saúde, Michel Couprie, and Roberto Lotufo. Exact euclidean medial axis in higher resolution. In *Proceedings of the 13th international conference on Discrete Geometry for Computer Imagery*, DGCI'06, pages 605–616, Berlin, Heidelberg, 2006. Springer-Verlag.

[121] J. A. Sethian. A Fast Marching Level Set Method for Monotonically Advancing Fronts. In *Proc. Nat. Acad. Sci*, pages 1591–1595, 1995.

[122] Jonathan R. Shewchuk. What is a Good Linear Element? Interpolation, Conditioning, and Quality Measures. In *Proceedings, 11th International Meshing Roundtable*, pages 115–126, 2002.

[123] Jonathan Richard Shewchuk. Mesh Generation for domains with small angles. In *SCG '00: Proceedings of the sixteenth annual symposium on Computational geometry*, pages 1–10, New York, NY, USA, 2000. ACM.

[124] Jonathan Richard Shewchuk. Delaunay Refinement Algorithms for Triangular Mesh Generation. *Computational Geometry: Theory and Applications*, 22:1–3, 2001.

[125] Frank Y. Shih and Yi-Ta Wu. Three-dimensional Euclidean distance transformation and its application to shortest path planning. *Pattern Recognition*, 37(1):79–92, 2004.

[126] Hang Si. TetGen: A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator. http://tetgen.berlios.de/.

[127] Alexander W. Siegel and Sheldon H. White. The Development of Spatial Representations of Large-Scale Environments. volume 10 of *Advances in Child Development and Behavior*, pages 9–55. JAI, 1975.

[128] David Silver. Cooperative Pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 117–122, 2005.

[129] Panagiotis E. Souganidis. Approximation schemes for viscosity solutions of Hamilton-Jacobi equations. *Journal of Differential Equations*, 59(1):1–43, 1985.

[130] Anthony Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*, pages 1652–1659, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[131] Nathan Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *Proceedings of the 20th national conference on Artificial intelligence - Volume 3*, AAAI'05, pages 1392–1397. AAAI Press, 2005.

[132] Nathan R. Sturtevant and Michael Buro. Improving Collaborative Pathfinding Using Map Abstraction. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 80–85, 2006.

[133] Nathan R. Sturtevant, Ariel Felner, Max Barrer, Jonathan Schaeffer, and Neil Burch. Memory-based heuristics for explicit state spaces. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 609–614, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[134] Zheng Sun and John Reif. BUSHWHACK: An approximation algorithm for minimal paths through pseudo-euclidean spaces. In Peter Eades and Tadao Takaoka, editors, *Algorithms and Computation*, volume 2223 of *Lecture Notes in Computer Science*, pages 160–171. Springer Berlin / Heidelberg, 2001.

[135] Zheng Sun and John Reif. Adaptive and Compact Discretization for Weighted Regions Optimal Path Finding. In *Proc. 14th Sympos. on Fundamentals of Computation Theory, LNCS*, pages 258–270. Springer-Verlag, 2003.

[136] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics*, 25(4):61–68, 1991.

[137] The CGAL Project. *CGAL User and Reference Manual.* CGAL Editorial Board, 3.9 edition, 2011. `http://www.cgal.org/Manual/3.9/doc_html/cgal_manual/packages.html`.

[138] S. Thrun. Learning Metric-Topological Maps for Indoor Mobile Robot Navigation. *Artificial Intelligence*, 99(1):21–71, 1998.

[139] S. Thrun and A. Bücken. Integrating Grid-Based and Topological Maps for Mobile Robot Navigation. In *Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, 1996.

[140] Sebastian Thrun, Jens steffen Gutmann, Dieter Fox, Wolfram Burgard, and Benjamin J. Kuipers. Integrating topological and metric maps for mobile robot navigation: A statistical approach. In *Proceedings of the AAAI Fifteenth National Conference on Artificial Intelligence*, 1998.

[141] John N. Tsitsiklis. Efficient Algorithms for Globally Optimal Trajectories. *IEEE Transaction on Automatic Control*, 40(9):1528–1538, 1995.

[142] W. van der Sterren. Terrain Reasoning for 3D Action Games. *Game Programming Gems 2*, pages 307–316, 2001.

[143] Jean-Paul van Waveren. The Quake III Arena Bot. Masters thesis, Delft University of Technology, 2001.

[144] Luc Vincent. Graphs and mathematical morphology. *Signal Processing*, 16(4):365–388, 1989.

[145] Luc Vincent and Pierre Soille. Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(6):583–598, 1991.

[146] Georgy Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. *Journal für die Reine und Angewandte Mathematik*, 133:97–178, 1907.

[147] Stephen Warshall. A Theorem on Boolean Matrices. *J. ACM*, 9(1):11–12, 1962.

[148] A. L. Zadeh. A rationale for fuzzy control. In *Fuzzy sets, fuzzy logic, and fuzzy systems: selected papers by Lotfi A. Zadeh*, pages 123–126. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[149] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The finite element method: its basis and fundamentals*. Butterworth-Heinemann, 2005.