University of Edinburgh
Division of Informatics

Route Planning for the Dervish Mine Detonating Vehicle

4th Year Project Report
Computer Science

Will Bryson

September 12, 2000

Abstract: The Dervish is a mine-detonating vehicle that is remotely controlled. De-miners need to be certain that the Dervish has covered every part of a minefield before it can be declared safe. This report describes implementations of a prototype route planner for the Dervish, such that the generated routes take the Dervish across every patch of ground within the minefield region specified.

One algorithm, 'Contour', is based on the concept of contouring – generating offset curves or buffers within the shape of the minefield, and using the contours as parts of the path. 'Chomper', a simpler, iterative algorithm is also described. The design of the route generators is explained, and a quantitative basis for comparison of routes is shown and used.

Acknowledgements

I would particularly like to thank David McLeod [15][16], whose Avenue training and advice were as helpful as his interest and enthusiasm for my work. I am very grateful to the patience and generosity of Mark Jerrum [13], who spared time to help me brainstorm on this project, and suggested many useful ideas.

Thanks also to Martin Held, whose book [8] was key to my work, and who kindly allowed me to reprint diagrams from his web site (see Figure 4 and Figure 6) [9].

Finally, and perhaps most crucially, I am indebted to Don Sanella for giving me invaluable feedback and advice, and generally keeping me on track throughout this project.

Contents

1 Introduction 1
1.1 About the Dervish 1
1.2 About the minefields 2
1.3 Assumptions 2
1.4 Route criteria 3
1.5 ArcView and Avenue 5
1.6 Unattempted Implementations 6
1.6.1 Travelling Salesman Problem (TSP) 6
1.6.2 Plane Filling Curves. 6
1.6.3 Axis-Parallel Milling 7
1.6.4 Iterative Refinement 7
1.6.5 Voronoi Diagrams 8
2 Discussion 10
2.1 The route generating framework 10
2.1.1 Highlighting missed areas 10
2.2 Local algorithm variations – the Chompers 11
2.2.1 Chomper1 – Arbitrary chomper 12
2.2.2 Chomper2 – Nearest-to-start chomper 12
2.2.3 Chomper3 – Beyond nearest chomper 13
2.2.4 Chomper4 – ReverseChomper 13
2.3 Global variations 14
2.4 A ‘global’ approach - naïve contour-following 14
2.5 ‘Contour’ – the multi-phase solution 16
2.5.1 Contour Generation 17
2.5.2 Corner-fixing 17
2.5.3 Shape Simplification (Weeding) 21
2.5.4 Appending Missed Areas 22
2.5.5 Joining Resultant Contours 22
2.5.6 Small-Part Merging (Clustering) 23
2.5.7 Combined Solution 23
2.6 Obstacles 25
2.6.1 Effect of obstacles on creating contours 25
2.6.2 Effect of obstacles on connecting contours 26
2.6.3 PathFind algorithm 26
3 Comparison 30
3.1 What I compare 30

3.2 Data used for comparison 31
3.3 Discussion of Results 31
4 Testing 34
4.1 What I aim to learn from testing 34
4.2 Testing the route planning systems 34
4.2.1 Errors to look for 34
4.3 Reliability of ArcView 35
4.3.1 ReturnBuffered 36
4.3.2 Distance 37
4.3.3 Intersection, Difference and Union 37
4.4 Conclusions 37
5 Conclusions 39
5.1 Review of Achievements 39
5.2 Directions for Further Work 39
5.2.1 Grid following approach 39
5.2.2 Unknown obstacles 39
5.2.3 Return to base 39
5.2.4 Multiple Dervishes 40
5.2.5 Improving upon a predefined path 40
5.2.6 Concentrating more on certain areas 40
5.2.7 Implementation language 40
6 Bibliography 41
Appendix A – Implementation issues 44

1 Introduction

The main aim of this project was to create a prototype that could be used as a basis for a route planner for the Dervish, a remotely controlled mine-detonating vehicle. Given a map of a minefield, the system should be able to generate a *route* which, when followed by the Dervish, will mean that every region that can be reached by the Dervish will have been checked. If there are any unreachable regions, then their locations must also be returned. More detail about criteria for generated routes is given in §1.4. I have implemented example algorithms using two very different approaches, and I describe and compare the different approaches and design choices throughout this document. Both implementations use ArcView and its programming language, Avenue, described in §1.5.

1.1 About the Dervish

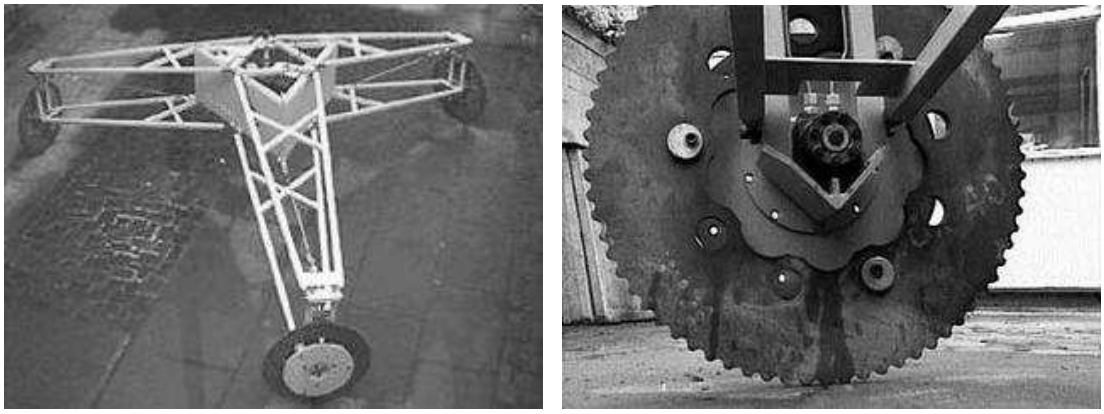

Figure 1 - The Dervish (left), and a close-up of one of its wheels (right). Note how little of the machine's surface area would be exposed to any blast, due to its frame-like structure. The left picture also shows how the orientation of the Dervish's wheels enables its spinning motion. Images from [6].

The Dervish, described in detail by Salter in [20], is a robot designed to detonate anti-personnel mines on agricultural land [6]. The project is run as a charity by Dervish Mine Clearance Ltd. Mines are set off by the weight of the machine, which at each leg is comparable to that of an adult human. The Dervish avoids damage to itself thanks to its open-frame design, which allows the blast to pass straight through.

The positioning of the Dervish's wheels gives it a zero-radius turning circle. This means that although some paths may be more efficient than others, the Dervish is capable of following any path. With all wheels turning at the same rate, the Dervish just whirls on the spot. By varying the rate of wheel turn, the Dervish can be made to slowly edge in one direction whilst it spins [19]. When in use checking ground, each wheel traces out loops with just 3cm between each track, and a path 5m wide is swept [6].

With most robots, variations in motor efficiency and surface conditions can mean that it becomes difficult to establish what state the machine is in. The Dervish can cope with such problems thanks to its control technique, where the on-board electronics turn the wheels appropriately until the Dervish is lined up with a vertex on a grid over the minefield. The grid is created by transmitting radio signals from three transmitters placed around the minefield. Where peaks in signal match up, we get points whose

positions can be controlled precisely by changing parameters of the transmitters. By keeping the instructions for the Dervish independent of the Dervish's current state, the Dervish is immune to inefficiencies, most slopes, and variations in surface roughness.

Another important aspect of the Dervish's capabilities is the fact that it will almost certainly have a fast-moving non-sweeping mode[1] [19]. This would involve locking one wheel and turning about that point, then locking another wheel and turning back, and so on, in a similar way to that used by people 'walking' large items of furniture across a room. Although such a mode would not be suitable for detecting mines, it would be very useful for moving across already-checked ground quickly, or for quickly returning to a start point for refuelling.

Because of the way theDervish continuously rotates in order to achieve complete coverage along a path, it can reasonably be treated as a disc that is translated around the minefield. This makes things much simpler for the purposes of route planning, because we reduce the state space of Dervish configurations to two dimensions (x and y-axis coordinates within the minefield) from three (x, y and heading)[2].

1.2 About the minefields

Unfortunately, in the real world, minefields can be almost any size or shape. Salter [19] claims that almost any minefield shape is possible, giving an example of a minefield one metre wide and a mile long. Also, minefields may be already mapped, but this is extremely unlikely. If maps are available at all, they may be low precision, out of date or just inaccurate. In practice, maps may be created in the field by sight, probably from some distance. Again, it seems maps will be unreliable.

Another issue is that minefields might contain *obstacles.* Obstacles are regions of the minefield that the Dervish must not enter. They could be regions that could disable the Dervish (such as becoming tangled in barbed wire fencing, or falling into a river or off a cliff), regions into which it would be futile for the Dervish to even attempt entering (such as a house, a car, or a tree), or areas we want the Dervish to avoid for other regions (maybe a human mine clearance expert is working there).

Note that the issue here is not that the Dervish need not enter these regions, but that the Dervish *must* not enter them.

1.3 Assumptions

Discussions with Stephen Salter [19] made me realise I need to narrowly define my aims. I could not aim for a system that does everything – it would be impossible to design from scratch, and would take far too long to create given the intended scale of this project. So I have considered an abstraction of the problem, choosing to put aside some of the real issues involved. This means modifications and additions will need to be made to this project before it could be used by the Dervish team, but it also means

---

[1] The motion control software design had not been finalised at time of writing. It is likely that the Dervish will be capable of this.

[2] There are some situations where knowing the orientation of the Dervish would enable us to manoeuvre it more closely into awkward corners and reduce the amount of missed area. For sharper corners, even this manoeuvrability would not be enough to guarantee complete coverage.

the process has been for me much cleaner and more focussed, and means more concrete conclusions can be drawn from the implementations. Hopefully this work will prompt further improvements so it can be used in the field (see section 5.2 below).

These assumptions have been made to give me a clear set of requirements and to make the project practicable within the time available.

- Assume perfect data provided (that is, the maps of minefields). If an error occurs, we can always re-plan with an updated world view.
- Initially, assume no obstacles. This means that there are no regions that the Dervish may not enter. Here I also include the area outside the minefield. Note that I drop this assumption in §2.6. The later addition of obstacles allows us to prevent the Dervish leaving the minefield if we so desire, by placing obstacles around the minefield appropriately, as in Figure 26.
- Assume no refuelling. This means we assume there is no limit on how far the Dervish can travel before returning to base. The Dervish's quick mode of movement should allow any route to be interrupted easily in time for it to make its way back to base for refuelling. Alternatively, as long as there is a safe route to the Dervish, people could bring the fuel to it where it runs out.
- Assume completely accurate execution of planned route. This means we need not make any allowance for the control software misjudging headings or positions during route execution.

1.4 Route criteria

It is important to make explicit the criteria on which I will judge the route generator.

Keeping Area/Route length high would mean the Dervish covers more new ground per metre travelled. If the Dervish isn't covering new ground, it's either leaving the minefield, or it's covering old ground – that is, there's *overlap*. Travelling in a straight line over unvisited ground clearly gives us minimum overlap by having maximum Area/Route length, and travelling over visited ground reduces Area/Route length and increases overlap. However, overlap can also occur when the Dervish turns a corner. This can be shown with a simple example.

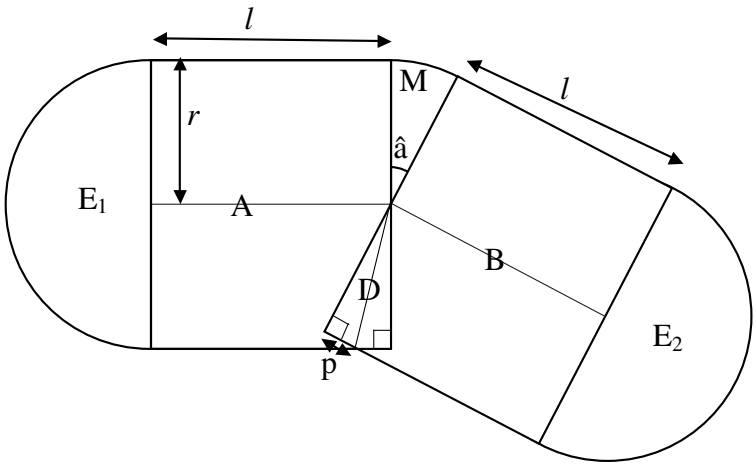

Figure 2 - Calculating overlap for a simple route section

In Figure 2, the Dervish travels a distance l, turns by angle â, then travels l again. When â = 0, the Dervish is just travelling a distance $2l$ in a straight line, so the area covered is πr^2+4rl. For the general case, where â is greater than 0, we have to subtract the twice-counted area D, and add the missing segment M. p is $r.\tan(â \div 2)$ by simple trigonometry, so D has area pr, which is $r^2\tan(â \div 2)$.

This gives us –

$$\begin{aligned}\text{Area} &= E1 + E2 + A + B + M - D \\ &= \pi r^2 + 4lr + (â \div 2)r^2 - r^2\tan(â \div 2)\end{aligned}$$

When $â = \pi \div 2$, i.e. we turn 90°, the area covered is

$$\begin{aligned}&= \pi r^2 + 4lr + \pi.r^2 \div 4 - r^2 \\ &= \pi r^2 + 4lr + r^2.(\pi \div 4 - 1)\end{aligned}$$

Which means that the area covered for a path of the same length covers an area smaller by about 20% of the area[3] of the Dervish.

This tells us that we should avoid having too many corners in a route, but it doesn't say what 'too many' is. There are no systems for me to compare a route generator against. Inevitably, the routes the generator creates will involve some overlap, but what is an acceptable amount? I believe my only option here is to compare a few slightly different implementations. I will not be able to say, this is a 'good' implementation, only that implementation A is better than implementation B.

So I judge routes on a subset of these criteria:

- Must be an easy to follow/explain algorithm. People must have confidence in it working.
- Must cover all coverable area. There are some areas that the Dervish cannot physically reach due to its shape. All other areas, though, should be covered.
- Must return area that wasn't covered. Those areas that the Dervish cannot reach must be checked by humans, so we need to know exactly where they are.
- Must keep ratio of area covered to route length high, i.e. keep overlap low
- Should cover ground near starting point first. In the event of failure, the Dervish should be easily retrievable. If the Dervish has reached a point, we know there is at least one safe path by which to retrieve it, but the wider that safe path the better.
- Must be generated in a reasonable amount of time. Note however, that given the slow rate of movement of the Dervish, calculation time can get quite large before it becomes significant.
- Must cope well with concave minefields. That is, Area/Route length should not drop significantly for shapes with concave parts.

This is how I have divided up the criteria –

[3] From the fact that $\pi / 4 - 1 = -0.214$ (3 s.f.)

Requirement	Importance	Explanation
Most cover all coverable area	Absolutely essential	A given algorithm must cover every area that is coverable. The only uncovered space should be physically unreachable by the Dervish.
Must return uncovered area	Absolutely essential	Uncovered space is required information for the human operator.
Keep overlap low	Important	Difficult to evaluate. I can at least examine routes by hand to look for unnecessary overlap, and compare different routes by considering the ratio of area covered to length of route (see §3.1).
Covers ground near starting point first	Not essential, but desirable.	To enable easy and safe retrieval of the Dervish were it to break down during route traversal.
Executable in a reasonable time.	Important	Running time of the algorithm should be O(L) or O(A) at most, where L is route length and A is the minefield area. Showing running time may not be straightforward, as the ArcView documentation does not explain its implementations of shape operations, or their running times.
Avoids obstacles	Important, but should be added later.	Must avoid known obstacles, not entering their space at all.
Return to base	Fairly desirable, but not important	If the Dervish can be made to finish up near or at where it began, that might be helpful to its operators.

1.5 ArcView and Avenue

ArcView is a commercial Geographic Information System (GIS) created by ESRI, and designed to allow non-technical people to perform spatial analysis tasks. It features its own programming language, Avenue, allowing customisation of ArcView, or even new applications to be created using ArcView's facilities [18].

Avenue, whilst powerful, is not an intuitive programming language to use, and ArcView's programming environment is not conducive to good programming practice. However, I believe ArcView to be a good platform for a proof-of-concept implementation such as this; many complex geometric operations are provided and are easy to use – saving a great deal of time in testing out new ideas. For example, the ReturnIntersection request [7], calculates the intersection of two shapes, the ReturnLength request can be used to find the length of a PolyLine[4], and so on.

[4] A polyline is made up of line segments where, with the exception of the two end line segments, each one joins two other line segments. In this way, three or more points define a polyline. In my implementations, I use polylines to represent the route the Dervish should take.

1.6 Unattempted Implementations

As well as the algorithms I describe in Section 2, there are some proposed solutions that I chose not to implement, yet which are worthy of discussion. I will describe these solutions now, before beginning the discussion of the algorithms I did implement.

1.6.1 Travelling Salesman Problem (TSP)

Tiling the plane with hexagons, and creating a graph of adjacent centre points, we can then treat the problem as an instance of TSP, as suggested by Arkin, Fekete and Mitchell (in [1], where they also give a good overview of this problem area). It is possible to generate an optimal tour of those points, and [1] demonstrates how to generate an approximation no more than 6/5 times the length of an optimal tour in time O(n), where n is number of points visited.

It is important to note, however, that this approach guarantees a certain amount of overlap between different parts of the path.

Looking at the problem as TSP ignores one important point, though – that routes with many corners lead to greater overlap. Jerrum [13] suggests using a square grid with spacing equal to the diameter of the Dervish. This means that traversing horizontally or vertically would leave no overlap at all. If we want the Dervish to end where it began, we can use known algorithms to find a Hamiltonian cycle of the grid points. If the Dervish changes direction within a unit square on this grid, and leaves by a different side to the one at which it enters, an area is missed. Therefore turns would have to be supplemented by small detours to cover the missed area.

We could use cost functions to take into account the extra distance incurred by corners. This modification to the grid follower technique would certainly reduce the amount of overlap, and the basic approach would be the same. In section 5.2, I propose this technique as one that should be investigated further. Due to time constraints, I have not attempted it as part of this project.

1.6.2 Plane Filling Curves.

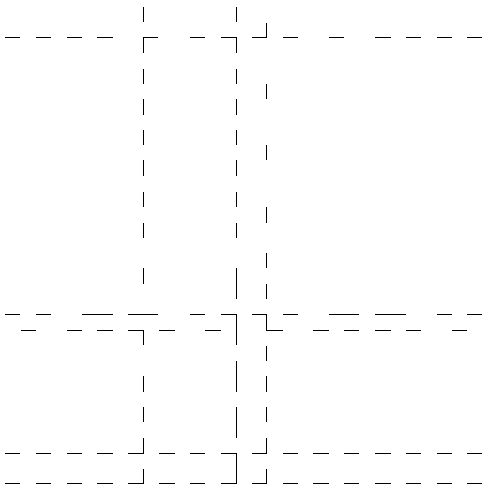

Figure 3 - Hilbert's plane-filling curve

Space-filling curves are mathematically defined objects that pass through every point in a (usually two dimensional) space. Balayoghan [2] particularly mentions the Moore curve, which fills a square, and finishes adjacent to where it begins. The fact that these curves visit every point in a region could indicate they have some application in this problem area. However, it is very difficult to generalise their mathematical descriptions to fit areas of arbitrary shape. Because of this difficulty, I have chosen not to follow this up any further.

1.6.3 Axis-Parallel Milling

Axis-parallel milling is described by Held in [8] and [9] in the context of pocket machining, and briefly by Arkin, Fekete and Mitchell [1], and involves zig-zagging back and forth across the shape to be covered. Each 'zig-zag segment' is a straight line parallel to some initial reference line. These segments are joined up to form a path by following the shape boundary until the next segment is reached, or there are no more segments. The gap between segments is selected to ensure that the whole region will be covered if the path is followed, and the orientation of the initial reference line can be selected to give the smallest number of retractions[5].

The more complex the shape, the less likely there is to be a route that doesn't require retractions. When obstacles are present, such routes are very unlikely to exist. For these reasons, I've deemed them not applicable to the problem of route finding for the Dervish.

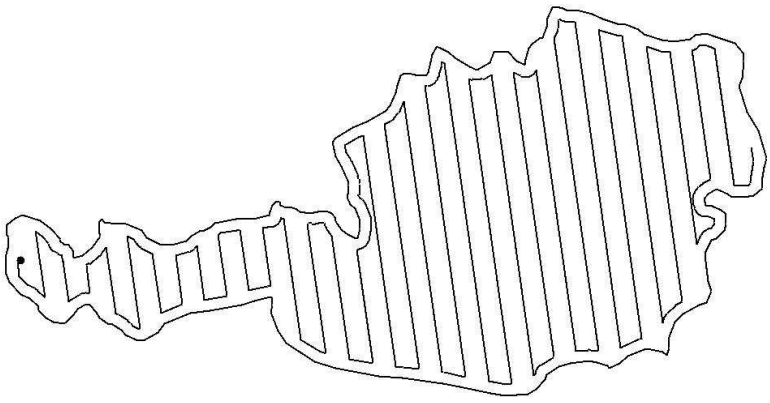

Figure 4 - Zig-zag pocket machining. (Image courtesy of Martin Held [9]) In this picture the initial angle has been chosen so as to avoid having to retract the cutter at all. For some shapes there won't be such a configuration.

1.6.4 Iterative Refinement

The idea presented itself as a way of avoiding some of the limitations of the Chomper and contour-following algorithms whose implementations I describe below. The aim here would be to begin with some initial path, possibly of very poor quality, and then iteratively make small improvements to it. The algorithm would identify problems with the suggested route, and make improvements appropriately. For example, if a

[5] Retractions are discontinuities in the machine path; at these points the cutter is lifted from the material and reapplied where the path continues. Obviously this is not possible when planning a path for the Dervish, which cannot choose to fly above areas it cannot visit.

section of the minefield is not visited, the route could be altered so the route point nearest that unvisited area is extended into it.

This technique lends itself to interactivity; although the initial path could be created by one of the other algorithms mentioned here, a human using a pointing device could specify a rough version of the desired route just as well. This sort of iterative algorithm would also have the advantage that it could be stopped at any point, depending on how good quality a route the user needs.

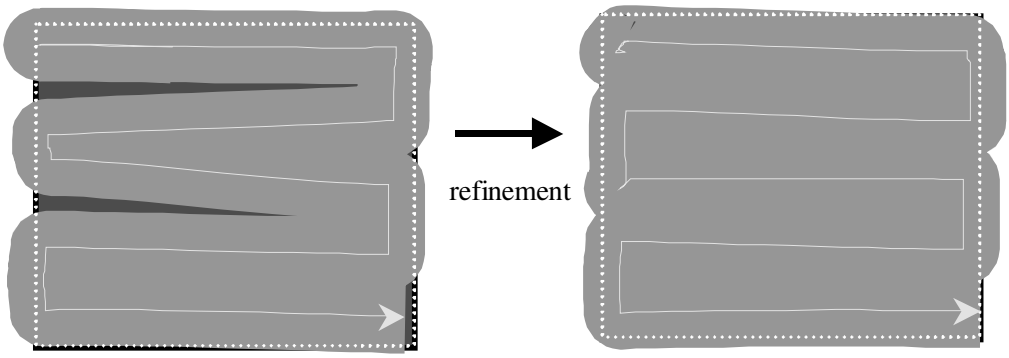

Figure 5 - How a few steps of iterative refinement might work. Vertices in the route are adjusted to ensure more minefield area is covered. In some places new vertices are added.

For this algorithm to be useful, though, a method for creating the initial route is required. Therefore this algorithm may be seen as a next step to be considered now path-generators have been implemented.

1.6.5 Voronoi Diagrams

One possible improvement would be to use Voronoi Diagrams for the generation of contours, as illustrated in Figure 6. A Voronoi Diagram for a set of points P divides the Euclidean plane into separate regions, where each region contains those points which are nearer to one member of P than any other. The edges in the diagram can be used to calculate the positions of contours. Held [8] uses two different types of region: those that contain points nearest a vertex in the original shape, and those containing points nearest an edge in the original shape.

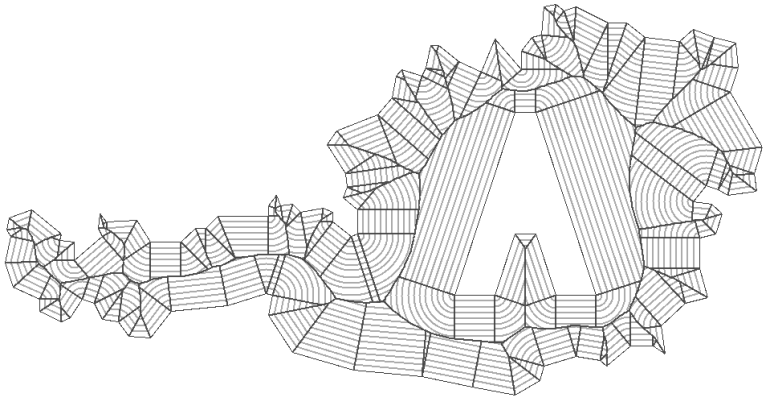

Figure 6 - Composite picture of Voronoi Diagrams and contours. Created by and used with permission of Martin Held [9].

Held [8] explains how to construct and use Voronoi Diagrams to efficiently generate contours within a shape. However, in [10], he says there are very few

implementations of Voronoi Diagram code, and even fewer reliable ones. Fortunately, contour-generation is handled automatically (though probably less efficiently) by ArcView using the ReturnBuffered request. However, were ArcView's implementation to be shown to be faulty, or were the route generator to be implemented in another language, it may be required to use Voronoi Diagrams. I therefore make note of it here as being of possible use in the future. I draw the reader's attention to a particular case I found which caused ReturnBufferd to exhibit incorrect behaviour – see §4.3.1.

2 Discussion

The route generator algorithms described below simply generate a route; a separate piece of code calculates the area covered, and is responsible for highlighting any regions missed. These functions I count as being part of the *route generating framework*. Note that I postpone comparison and evaluation of the route generators until §3.

2.1 The route generating framework

For testing and comparison purposes, it was important to use the same framework for each algorithm, so only one piece of code was responsible for storing and displaying results, and only one piece of code calculated and displayed the regions (if any) missed by the route. Listing 1 shows the route-algorithm independent framework for calculating routes and displaying results.

```
/* DervishGenerateRoute */
1.  Make a note of the time for statistics reporting purposes
2.  If a start point is selected,
3.     set startPoint = first selected startpoint,
4.  Else
5.     Set startPoint = first existing startpoint.
6.  Endif
7.  Get the minefield shape, fieldPoly, by calling GetFieldAreaPoly, which returns
    the selected minefield area (or a default one, if there is no selection), and
    subtracts obstacles from that area if we're set to avoid obstacles.
8.  If there is no route algorithm chosen, return nil, and finish
9.  Call the chosen route algorithm with parameters -
    Minefield region = fieldPoly
    Start point = startPoint
10. Record the route in the appropriate themes within ArcView (this also calculates
    and records any missed regions).
11. Display timing information to show execution time.
```

Listing 1 - Outline for route generating framework, DervishGenerateRoute

2.1.1 Highlighting missed areas

As noted in §2.5.4, there are some areas that are unreachable by the Dervish. In the event of failure of the route planner, there may also be *reachable* areas that have been missed. In both these situations, we need to know where these areas are. With this information, if the area is unreachable, the human operator may –

- Get human mine clearance experts to check the area
- Mark the area as dangerous (say, put a fence up around the region, with a warning sign)
- Inspect the area to see if it is unreachable – if the map is inaccurate, it may be possible to update it and run the route generator again before getting the Dervish to actually follow the route.

If the area is reachable, the information is equally useful, and shows a weakness in the implementation. This is not a dangerous situation as long as missed areas are highlighted. Action that could be taken here would be –

- Run the algorithm again over the missed areas. In the short term (for example, if the problem occurs in the field), this may be all that's feasible.
- Improve that implementation. This is a course of action only open to the software developers. Missed, reachable regions highlight problems with the implementation that need to be fixed.

The missed areas can be calculated quite simply. First, we take the whole of the route returned by whichever algorithm we are using, and buffer it by the radius of the Dervish. This gives us a shape representing the checked area; that is, area covered by the Dervish when following the route. By subtracting this region from the original minefield shape, we get the regions of the minefield that are not within the checked area. In my implementation, these are added to the display in bright red so they are very visible. Highlighting them further would probably be useful in the field as even very small missed regions are important.

2.2 Local algorithm variations – the Chompers

The algorithms I have implemented can be divided into two distinct classes. Those algorithms that do little or no analysis of the minefield shape I call *local*, and those algorithms that examine the minefield in detail before committing any part of the path I call *global* algorithms. The 'Chomper' algorithms are all in the former category.

The Chomper implementations repeatedly ask 'where now?' and use various heuristics to provide a good answer. For example, the three implementations listed below always position the Dervish's centre on the nearest so-far-unvisited point in the minefield. The differences between them are due to various different ways of breaking ties for nearest so-far-unvisited points (see the more detailed descriptions which follow). The algorithm ends when there are no more points to visit, so by definition it always visits every point[6]. The final few iterations of the algorithm usually involve the Dervish racing back and forth cleaning up odd tiny regions it missed before.

None of these algorithms prevent the Dervish from escaping the minefield region, and none of them have any provision for dealing with obstacles.

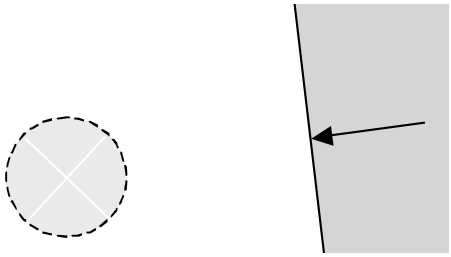

Figure 7 - Initial situation using a Chomper algorithm. The arrow indicates the unvisited point which is nearest to the Dervish.

[6] This assumes, of course, that the system's implementation of Boolean operations on shapes is completely reliable. There are a few known examples of data where this is not the case for ArcView. See Section 4.3.

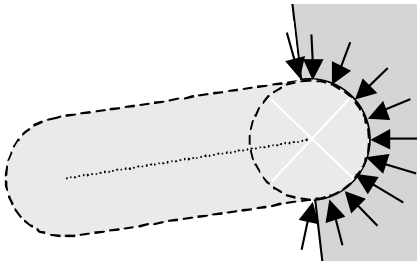

Figure 8 – First iteration. Again, the arrows indicate the unvisited minefield points nearest to the Dervish. A tie-breaker is needed to pick between them, and the choice of tie-breaker is what distinguishes the different Chomper algorithms.

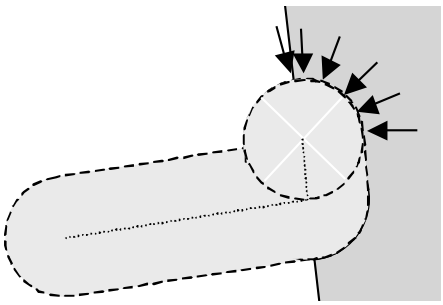

Figure 9 - Second iteration. There are still several nearest unvisited points. Note that only about a quarter of the Dervish is over new ground.

2.2.1 Chomper1 – Arbitrary chomper

This repeatedly visits the nearest so-far-unvisited point, and solves ties arbitrarily. My implementation tends to favour following along edges, due to ArcView's implementation of irregular polygons.

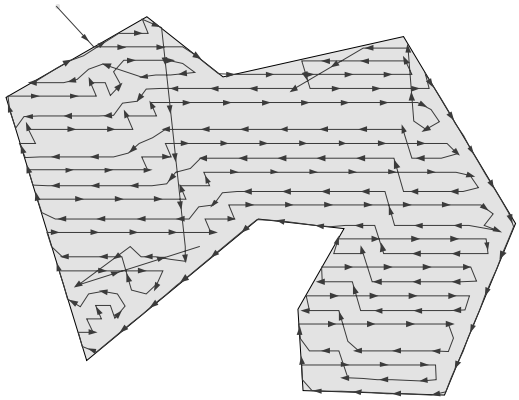

Figure 10 - Chomper1 covering an irregular minefield. The minefield has area 10,215m^2. This route took 8 seconds to calculate. The route length is 2591m, and the area:distance ratio is 3.94 (the calculated upper limit on the area:distance ratio was 8.04). The Dervish radius was set to 4m.

2.2.2 Chomper2 – Nearest-to-start chomper

This repeatedly visits the nearest so-far-unvisited point. It solves ties where there are several nearest points to the Dervish by picking the one of those that is nearest to the start. The thinking here is to keep the Dervish near the base for as long as possible to allow for easy retrieval in case of failure. This implementation causes the route to trace out ripple shapes from the start point. A downside was that the Dervish always

finished right on the other side of the minefield, when it would be preferable to have it finish at or near the start point. Note that the route created is even with few turns and a regular appearance, but that the gap between parts of the path is consistently around the same as the Dervish radius. Chomper3 is an attempt to improve on this.

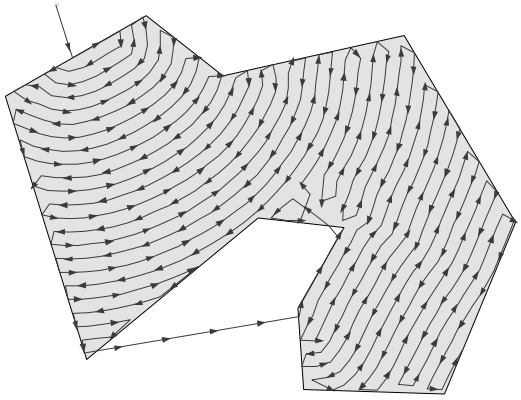

Figure 11 - Chomper2 covering an irregular minefield. This route took 32 seconds to calculate. The route length is 2781m, and the area:distance ratio is 3.67.

2.2.3 Chomper3 – Beyond nearest chomper

Chomper3 was an attempt to improve on the overly conservative spacing between route parts created by Chomper2. Each time the Dervish takes a step in the Chomper algorithms, its centre moves to the edge of an unvisited region. In Chomper3, we move the Dervish into that unvisited region, going as far as the edge and then some more. Unfortunately, the results were not as anticipated. The generated routes were substantially longer than with the other Chompers, and appeared particularly random.

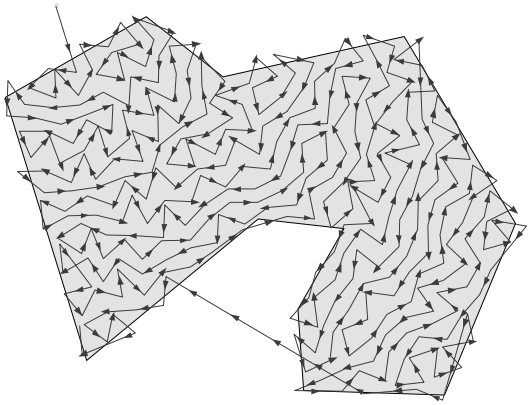

Figure 12 - Chomper3 covering an irregular minefield. This route took 9 seconds to calculate. The route length is 3129m, and the area:distance ratio is 3.26.

2.2.4 Chomper4 – ReverseChomper

This repeatedly visits the nearest so-far-unvisited point. It solves ties where there are several nearest points to the Dervish by picking the one of those that is furthest from the start. The whole route generated was then reversed. The thinking here was to allow the Dervish to return to base. Before we reverse the route, the first thing it does is to travel, in small steps, to the far end of the shape. When there, it begins working its way back, covering most of the minefield on the way.

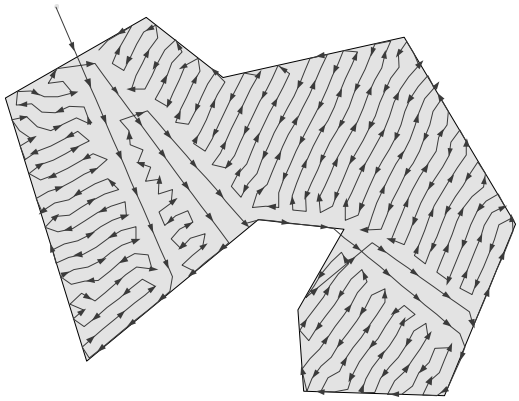

Figure 13 – Chomper4 covering an irregular minefield. The route took 9 seconds to calculate. The route length is 2661m, and the area:distance ratio is 3.84.

When reversed, such a route would stay near the starting point on the whole, but leave itself some minefield to cover on the way home.

2.3 Global variations

The 'global' approaches to this problem work by analysis of the geometry of the minefield area. They suffer from the fact that they usually require a large number of operations such as intersection and union on 2D shapes, which can be very computationally intensive. The advantage is that the resulting route may be easier to reason about because it was created using a high-level mathematical description. Below, I describe the simple contour-following approach I used as a start point for ideas on this problem, and then go on to describe the different phases of the solution I built as a result.

2.4 A 'global' approach - naïve contour-following

David McLeod implemented this algorithm. It was used as a starting-point for brainstorming on this project. The algorithm takes the minefield shape, and generates a *buffer*[7] or *contour* within it, and again within that, until we cannot create a smaller shape. The algorithm then steps towards some 'central' point from a starting point outside the minefield, and whenever it reaches a contour, it follows the full circuit of that contour and then continues inwards.

---

[7] It is worth highlighting this use of terminology. Literature regarding ArcView and Avenue consistently uses the term 'buffer' to describe these regions (examples are [15], [16], [7]). Others, including those approaching the problem from a machine-tooling angle, such as Held [8] and [9], and others, such as McLeod [24], notice the resemblance of a set of these shapes to a map with contours.

Both these terms are flawed. 'Contours' tends to imply, wrongly, that points on the boundary of each shape share some characteristic, and that different contours differ in that characteristic. 'Buffer' has a particular meaning in Computer Science, but here I use the term only in the same sense as ArcView.

Saying region S' is shape S buffered by a distance of d is to say that S' is the points within S which are at least a distance d from any point on the perimeter of S[8]. Note S' may be a disconnected region. Also, one can buffer externally to a shape in a similar fashion.

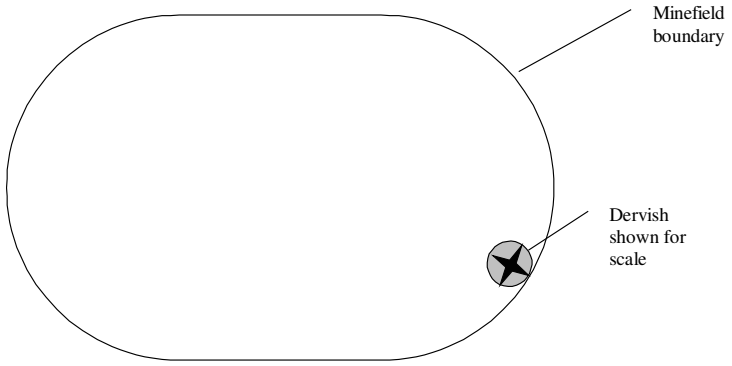

Figure 14 - The minefield shape before calculating the first contour.

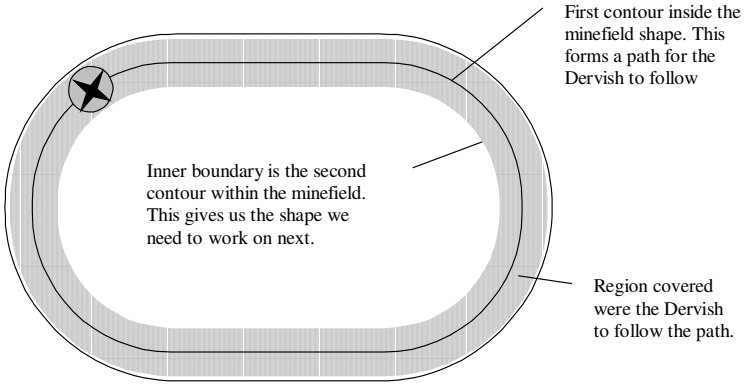

Figure 15 - The first contour. The thin black line at the centre of the grey band is a path contour. Note that the second contour is the inner edge of the region covered by the Dervish.

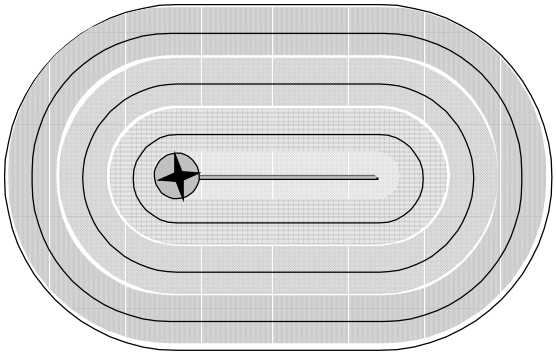

Figure 16 - The minefield with all the contours calculated. Note that the innermost contour is a very thin ring, and a large amount of overlap occurs here. The thin white lines visible between bands represent the contour lines generated which specified remaining area to contour at each iteration. We can calculate boundaries repeatedly until there is no shape left to cover, alternating between path and next-shape boundaries, each time moving into the

[8] Latombe [14] describes this process as growing the shape B isotropically by the radius of a circle A, to give a new shape CB. The boundary of CB is the path described by the centre of A when we roll A around B's boundary.

shape by the radius of the Dervish. Note that this illustration uses a particularly convenient example to keep the explanation simple.

Figure 14 to Figure 16 show three stages in calculating the contours. Note that they show a very conveniently shaped minefield for this algorithm. In general, a contour may have several disconnected parts, so stepping in through each contour in turn may not make any sense. Also, normal methods for picking a central point may give a result that is outside of the minefield itself. For example, calculating centre of gravity, or even taking the mean of all the vertices, can result in a point not within the shape itself. Avenue's ReturnCenter request returns the centre point of a polygon's bounding box, and the X coordinate adjusted as necessary to put the point inside the polygon.

Another problem, illustrated in Figure 17, is that generating the contours can leave an innermost region that is not visited, yet which is too small to generate a contour for. Finally, a straightforward contouring technique results in missed regions at each corner, as shown in Figure 19.

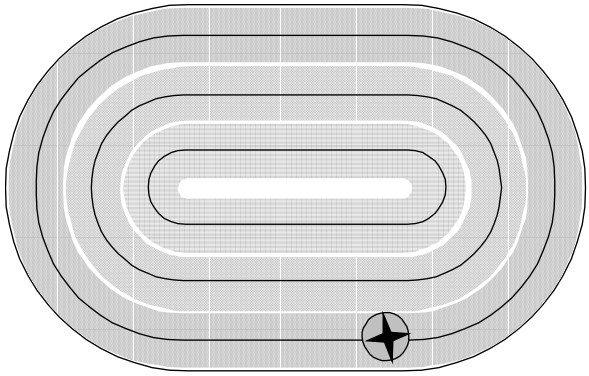

Figure 17 - Minefield with small unbufferable region

However, contours do have many inherent desirable properties. The contours never cross, and we can choose the inter-contour gap such as to have very little or no overlap. They are definable mathematically, so their behaviour can be understood more easily and with more certainty.

2.5 Contour – the multi-phase solution

The *Contour* algorithm I have designed and implemented divides route generation using contours into several distinct phases. This keeps each aspect of route generation independent, and means improvements can be made more easily. It also means the algorithm can be more easily understood.

The phases are

- Contour generation
    - Corner fixing
    - Shape simplification
- Appending Missed Areas
- Joining resultant contours
    - Small part merging

Note that shape simplification does not improve the quality of the output, but reduces the time taken to generate the output by reducing the data set. All these stages are described in more detail in the sections that follow.

Contour has several advantages over McLeod's algorithm, described in the previous section. It can cope with concave minefield shapes, or even disconnected minefields; it correctly compensates for corner regions missed during the contour generation process; it detects and visits regions which contouring simply cannot cover; it joins the resulting route parts in a sensible and efficient way, and it does all this whilst avoiding convex obstacles within the minefield.

2.5.1 Contour Generation

Contour generation could be performed by means of Voronoi Diagrams, but as described in §1.6.5, the contours can more easily be generated using Avenue's ReturnBuffered request. Therefore, I look to the 'naïve' algorithm and try to improve upon its shortcomings.

We note that repeatedly generating smaller and smaller contours in some concave shapes (such as an hourglass shape) will give us disconnected contours (I call them *islands*) at some stages.

We can deal with this by maintaining a list of shapes that need to be buffered. Initially this list contains just one shape – the whole minefield. Each time we buffer a minefield region, we get a set of one or more shapes defining path segments. By buffering each of those, we get a (possibly empty) set of remaining unvisited areas, which can be added to our list of shapes that still need processing. As we go, we can keep track of all the path segments we made, and when the list of shapes to process is empty, we have finished collecting route parts and can navigate them in some way.

2.5.2 Corner-fixing

A second improvement upon the naïve approach is that we tackle the small missed sections at corners. In the context of milling, Held [8] discusses the 'uncut triangular-shaped areas' found when tracing around internal corners. He identifies two known strategies for tackling the problem –

Putz's Suggestion	–	Move from the corner point directly into the uncut region before returning to the previous path.
Czeranowsky's Approach	–	Choose an inter-contour gap small enough such that there is never any uncut region.

Czeranowsky's approach suffers from the problem of covering much space more than once; there is a high degree of overlap even in the best case. However Putz's suggestion involves short cutter path segments and rapid changes of direction, which result in uneven, extremely slow milling. In particular, a tiny detour involving a 180° turn would be especially problematic. This makes Putz's suggestion unfeasible for machine tooling. Therefore, Held proceeds using Czeranowsky's approach as a solution.

For the purposes of the Dervish however, short path segments have no unpleasant side effects at all. The Dervish does not need to turn – whatever its orientation it can

move equally quickly in any direction. So Putz's suggestion is probably more appropriate for our needs.

I will now explain my implementation of corner fixing in more detail, as well as some of the choices made during its design. This technique involves altering the path, to move from each vertex to a point sufficiently within the unchecked region, then returning to the followed path, as illustrated by Figure 20. NP is the Nearest Point on the path to the missed region. This is the vertex where the problem occurs, and is also the point from which the correction is made. FP is the point in the missed region that is furthest from NP. TP is the target point for the correction. TP lies on the line NP → FP, at a distance from FP equal to the radius of the Dervish. The dotted arc passing through FP shows the extent of the extra area covered given the correction.

I have done some work to calculate the extent of the missed areas. This is important, as it means the correct point to travel to can be derived. If we are allowed to roam outside the region[9], the point we want to visit is at a distance *r* from the furthest point of the unvisited region, where *r* is the radius of the Dervish.

There are several issues which need considering for implementing Putz's suggestion. In the outline above, I assume that we know the location of FP, the point in the missed region furthest from the current vertex in the path. Recall that FP will be a point on the next-outer contour; all we need to do is calculate which vertex on that contour led to the creation of NP.

However, as ArcView is generating the contours for us, we do not know which vertex in the next-outer contour corresponds with our NP. There are two possible ways we can deal with this. One option would be to analyse the various contour parts to find which vertex corresponds to which.

An alternative, which I have implemented, is to derive the location of FP by analysis of the turn angle.

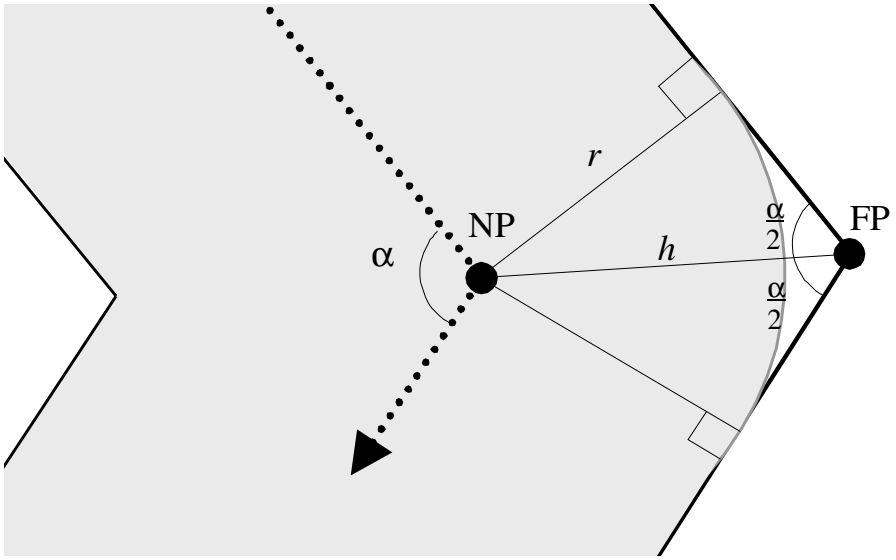

Figure 18 - Deriving TP from turn angle

In Figure 18, we use the angle α at the vertex NP to let us calculate h.

[9] We always are, unless this is the outermost boundary, and we drop the assumption that the Dervish is allowed to leave the minefield.

$$h = \frac{r}{\sin(\frac{\alpha}{2})}$$

As FP is the furthest point we need to reach then to calculate TP, the target point for the Dervish's centre to reach if the correction is to be successful, we need to find how far TP is along NP → FP.

$$|\,\mathrm{TP} \rightarrow \mathrm{FP}\,| = r$$

so $$|\,\mathrm{NP} \rightarrow \mathrm{TP}\,| = h - r.$$

$$= r.\left(\frac{1}{\sin(\frac{\alpha}{2})} - 1 \right)$$

Calculating the orientation of the line NP → TP is trivial, so now the algorithm can calculate the coordinates of TP. It then replaces the entry for NP in the list of coordinates for the path with the three points {NP, TP, NP}.

Note that if the Dervish is not allowed to escape the minefield, we cannot apply corner fixing to the outermost contour (i.e. the first one generated). In these cases, some areas are unavoidably missed. In general, though, the Dervish may leave the minefield, so this problem does not arise.

Figure 21 shows a case where calculating TP as described above can misjudge the situation. This need not be a problem, though, as such situations are detected and fixed later on (see §2.5.4).

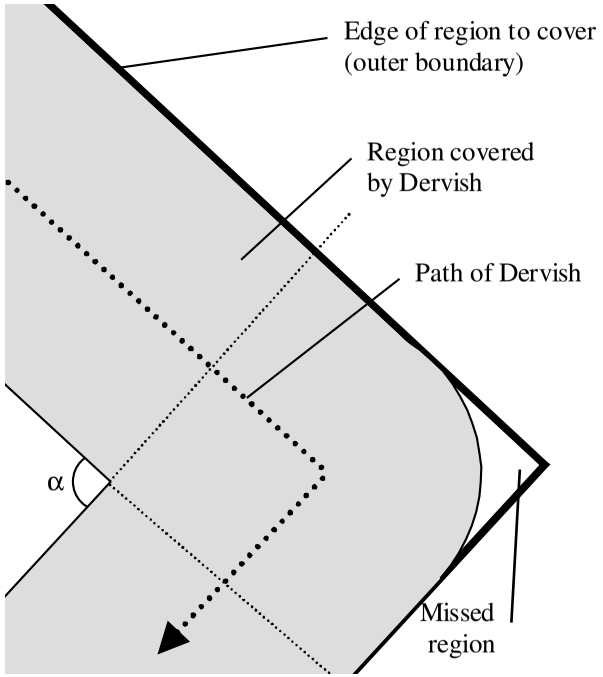

Figure 19 - Contour following and the missed region problem

α, the angle formed at the problem corner, allows us to calculate the extent of the missed region.

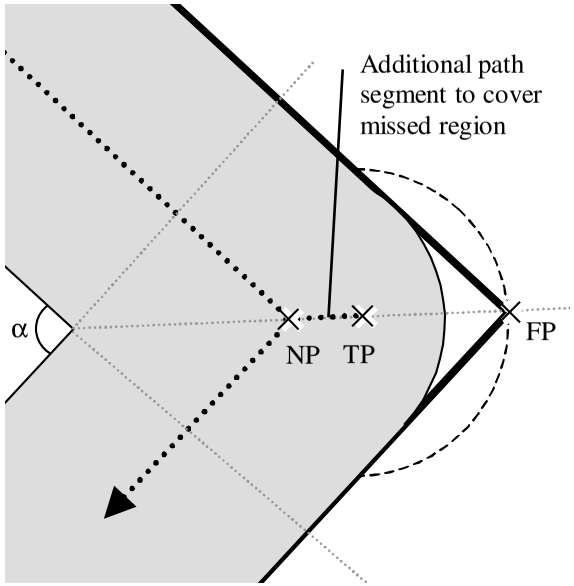

Figure 20 - Putz's suggestion for curing the missed region problem

NP is nearest point on the Dervish's path to the problem area.

FP is point in the problem area furthest from NP.

TP is the target point, to where the path has to be extended to correct the problem.

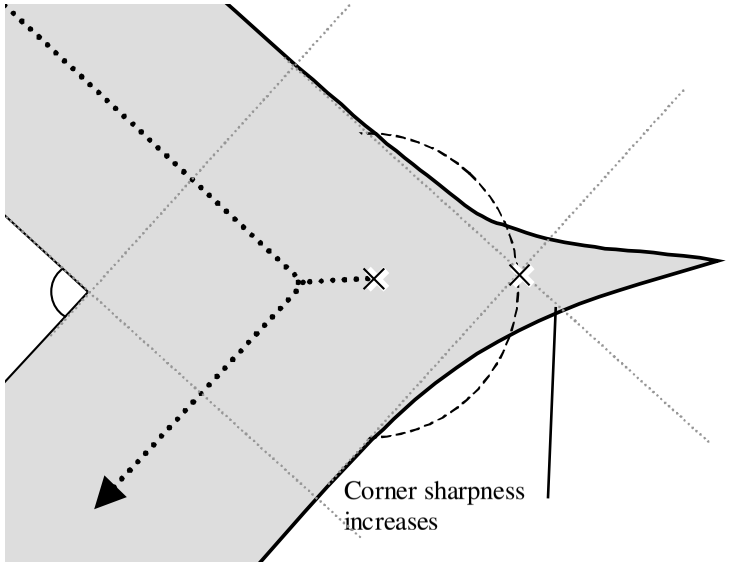

Figure 21 – Case when finding FP to choose TP is not so easy.

The corner-fixing I have implemented ignores the contour outside the path section it is fixing. Above is an example how when the turn angle at a path vertex is very different from the outer contour's corner angle, the algorithm may underestimate the size of the missed region.

2.5.3 Shape Simplification (Weeding)

In an effort to improve run-time of the contouring algorithm, I introduce a shape simplification phase. The motivation for this is that every right-turn, when going clockwise around the minefield, will result in a corner which needs fixing. This means that path complexity, and therefore run-time, can increase very quickly. In GIS circles, this process of removing vertices not required to adequately define a feature, is referred to as *weeding*. [17]

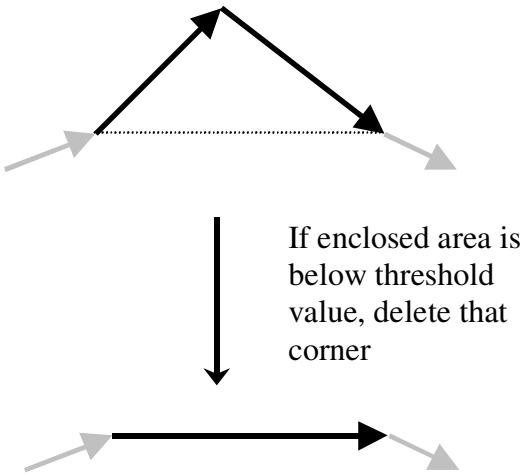

Figure 22 - Shape simplification removing insignificant vertices at a convex corner

During shape simplification, the system scans each vertex of a shape. If the turn at that vertex is into the shape (to the right if we're traversing the boundary in a clockwise direction), then this vertex is a convex corner, and deleting the vertex will reduce the area of the shape. If the turn is out of the shape (e.g. to the left when going clockwise), then this point describes a concave corner, and deleting the vertex will increase the shape's area. Depending on the type of shape we're simplifying, we can then enforce that we want the resultant shape to contain the original region, or to be contained by the original region.

We need to know if we're allowing the shape to grow during this phase because if the shape is that of an obstacle or of a visited area, we may be left in a situation where the area still to visit is underestimated. Similarly, if we're simplifying the boundary of the area visited, and the shape is allowed to grow, the system may ignore areas that haven't been checked, under the false belief that they have. Another option is that we don't enforce any restriction; the missed areas resulting from such a policy would have to be picked up at a later phase.

Finally, to prevent the shape simplification phase reducing every shape to nothing, we only remove a vertex if the affect on the shape's area is below a certain threshold. This means that important vertices will be unaffected by the simplification process.

The speed improvements gained from simplifying the shapes returned by ArcView are considerable, reducing execution time by up to 90%. It could be argued that whatever the saving, considering that the Dervish takes several hundred times longer to follow a path than it takes for the path to be calculated, whether the run time is ten seconds or a hundred seconds is unimportant. However, from the point of view of a developer experimenting with the system, such a difference has a considerable effect on how easily variations and modifications can be tested.

2.5.4 Appending Missed Areas

When a full set of contours have been generated, and corners have been fixed where required, as described in section 2.5.2, we hope to have all the path segments necessary to ensure that the whole minefield is covered. In other words, we should have a *minimum* set of path segments. However, there may still be some regions that would be missed. In roughly descending order of likelihood, these are:

(a) Regions missed because they are physically unreachable

(b) Small regions that cannot be contoured (see Figure 17 on page 16)

(c) Regions missed because of limitations of the corner-fixing algorithm (see Figure 21 on page 20)

(d) Regions missed because of shape simplification (see §2.5.3)

Case (a) is unavoidable and incurable; such regions are left, and will be detected and highlighted later[10]. Fortunately, all the remaining cases are easily detectable. We can create a contour around the path segments found so far (treating each existing contour as a series of line segments rather than as a shape), and subtract this region from the original minefield area to give us the unvisited minefield area as a (probably disconnected) region.

To deal with these missed regions, I take advantage of the fact that in each case the missed regions can be no more than the diameter of the Dervish across[11]. This means that if the Dervish were to trace the perimeter of these regions, they would be covered. So we simply add the outlines of these areas to the list of route parts. In many cases this seems overly cautious; often the whole of the missed region is smaller than the Dervish, so tracing the region's perimeter is overkill. Although this is an area ripe for improvement, the relatively small sizes of these missed regions means that any performance gains will be small.

2.5.5 Joining Resultant Contours

At last we have a list of shape outlines sufficient to cover as much of the minefield as is physically possible. The only question now is to find the most efficient order in which to visit these route parts.

It's useful to recall here that the Dervish will probably have a fast-walking mode that it can use when complete coverage is not required (see §1.1). As we already have enough route to cover the minefield, any extra route parts added to join the contours together can be traversed in this speedier, but less thorough way. This means that although we want to aim to add as little as possible to the route here, what we do add will have only a small effect on the execution time for traversal of the path.

Because any contour may have multiple contours within it, we cannot just trace serially into the shape from the outside, traversing contours as we meet them. Instead, the contour-joining algorithm keeps track of a notional 'current point' of the Dervish,

---

[10] Obviously, modifying the map's shape, or reducing the radius of the Dervish would minimise this problem, but neither are practical.

[11] If a region was missed because it was too small to contour, it has no points within it at a distance greater than the Dervish radius from the region's perimeter. Such a region can therefore be no thicker than the Dervish diameter at any point.

and adds line segments to a list containing the path so far. The algorithm searches for the nearest unvisited route part to the current point, and traces that, then does the same from that point, and so on, until no more route parts are left.

The disadvantage with this approach is that route parts made when appending missed areas are often very small, and we can end up making large detours for tiny route parts. We deal with this by joining small route parts to nearby neighbours before joining other parts. This process is described in more detail in §2.5.6.

This phase is seriously affected if we remove the assumption that there are no obstacles, as described in §2.6.2.

In general, this contour-joining algorithm visits contours from the outside in. This has the beneficial side effect that, assuming all areas outside the defined minefield are safe, the Dervish is always easily retrievable.

2.5.6 Small-Part Merging (Clustering)

This clustering phase, where route-parts with area below a threshold are merged with nearest neighbours, helps to reduce the likelihood of the Dervish having to make long detours to reach small missed regions. In my implementation a threshold value of $1/200^{th}$ of the area of the Dervish is enough to catch all the smallest missed parts. The merging process works by finding the vertices in each contour outline that are nearest the other contour. In the larger shape, that vertex becomes a point of detour, where the Dervish then follows the loop outline of the smaller shape and then returns. By modifying the larger contour in this way, we can count the whole lot as one route part, and discount the smaller contour, with just two new small route segments.

In practice, this is not always effective in improving the efficiency of the route generated; sometimes just moving from contour to nearest contour works well. For example, in the best case when not using clustering, joining contours adds rn to the route length, where n is the number of contours and r is the Dervish radius. When using clustering, we always add roughly $2rn$ to the route length, as each time we merge a contour to another we have a path to the other contour (length approximately r) and a path back again.

2.5.7 Combined Solution

I have completed a contour-following algorithm to implement all these features.

Listing 2 outlines a route algorithm which can plug into the framework described in §2.1, and which demonstrates all the features listed earlier in §2.3.

```
/* DervishRoute.Buffer2 */
1.   Get fieldPoly and startPoint from parameters
2.   Call GetRouteParts to return a list of shape outlines, routeParts, which need to
     be traversed.
3.   If we're merging small parts to larger ones, then
4.      routeParts = MergeSmallParts(routeParts)
5.   End If
6.   wholeRoutePoints = JoinRouteParts(startPoint, routeParts)
7.   Make a polyline from wholeRoutePoints, the list of points in the whole route,
     and return it.
```

Listing 2 - Outline for contouring route generator, DervishRoute.Buffer2

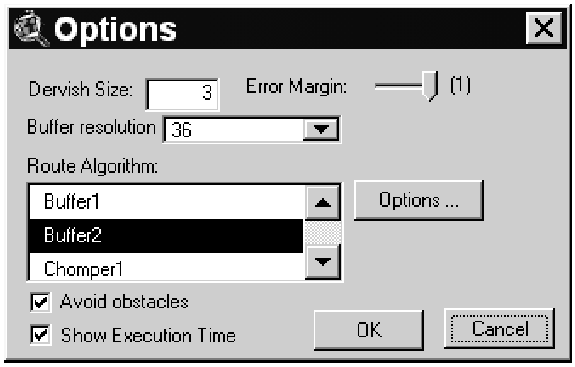

Figure 23 - General route generator options screen. The "Options…" button is only enabled when there are more options available for the selected route algorithm.

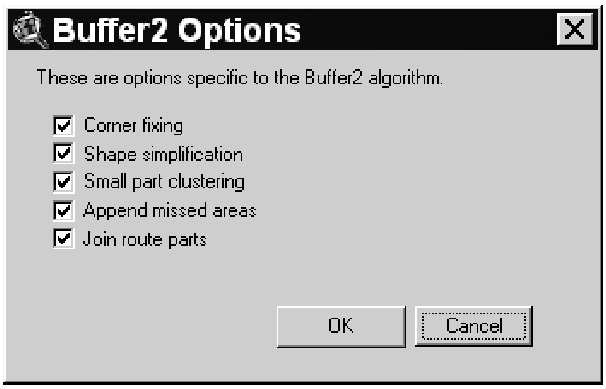

Figure 24 - Options specific to the 'Buffer2' contour-based algorithm

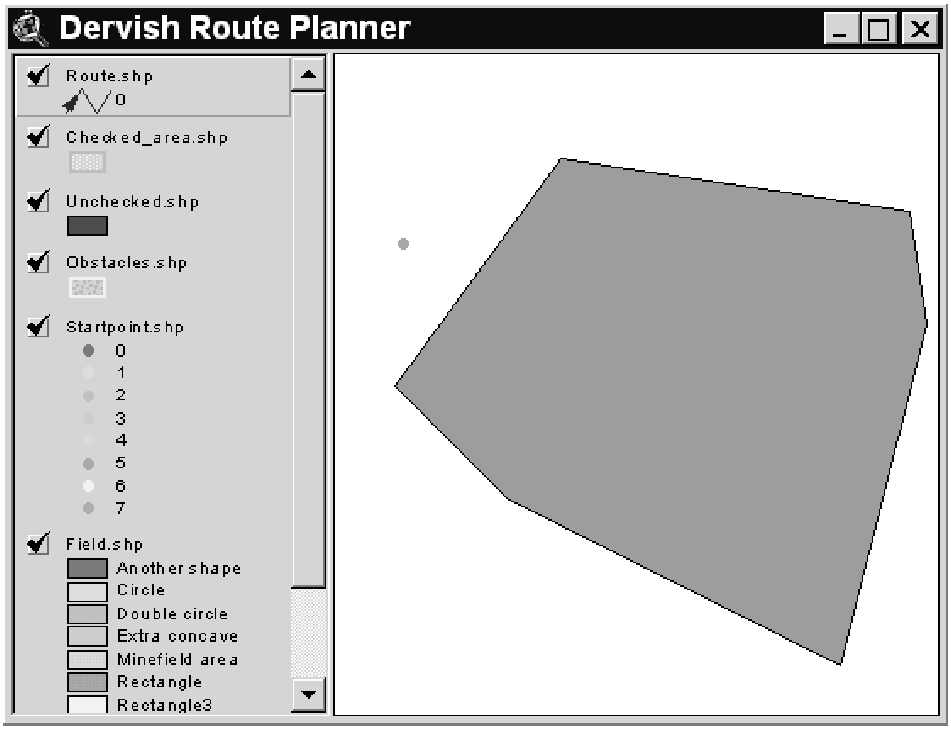

Figure 25 - A minefield and a start point. Neither are selected in this screen.

DervishGenerateRoute and DervishRoute.Buffer2 are both designed so that any phase of the algorithm can easily be disabled or replaced with an improved implementation. For example, the phases of merging small parts and of joining route parts together are completely self-contained.

A very simple interface was created to allow the user to experiment with different route algorithms, and to compare the same algorithm with different parameters (see Figures 23 and 24).

As mentioned in Listing 1, the program will use the first selected start point and minefield if there are any selected, and the first existing start point and minefield otherwise. Start points and minefield shapes can be selected easily, using ArcView's usual tools. ArcView highlights the selected item from each theme by drawing it in yellow. Modifications, including creating new minefields or start points can also be made using ArcView's built-in tools. If ArcView were to be used in the field, some simpler, yet more precise, interface may be devised for inputting minefield data.

2.6 Obstacles

Obstacles, introduced in §1.2, have a considerable effect on the contour-based algorithm. The two main implications are that contour generation must take obstacles into account, and that contour joining must avoid obstacles. Accepting that obstacles may be present also allows us to prevent the Dervish leaving the minefield, if that is what we desire, simply by regarding the boundary as an obstacle.

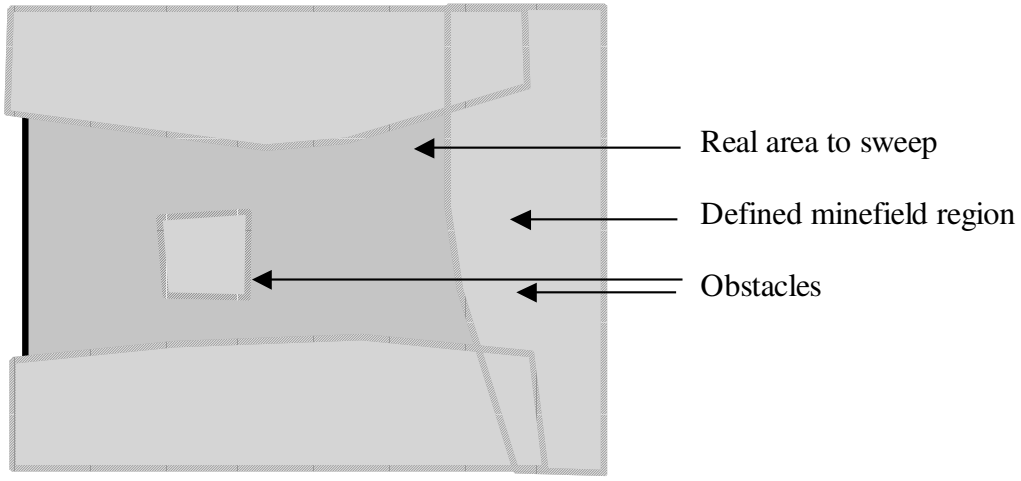

Figure 26 - Defining a minefield region using obstacles. In this example, only the left hand side of the minefield is unobstructed. This is where the Dervish would enter or exit the region.

2.6.1 Effect of obstacles on creating contours

When calculating contours, we previously created a buffer within the minefield shape to give us one or more route parts, then buffered within those to give us the remaining minefield to process on the next iteration. If obstacles are present, we need to subtract their shapes from that of the minefield before we calculate the first contour. ArcView can calculate a buffer of this shape with holes using the same ReturnBuffered command used for ordinary contouring. Each time we contour, the remaining region's perimeter shrinks, whilst holes in that region grow. When considering route parts, both the outer perimeter and the outlines of holes must be counted.

Another side effect of obstacles on contour calculation occurs when we fix corners, as described in §2.5.2. The small detours created by implementing Putz's suggestion may hit obstacles. As these detours are very small though, it makes more sense to curtail the detour if it hits an obstacle, rather than attempting any complex obstacle

avoidance. In some circumstances this may result in small missed regions, but these would be picked up at the appending missed areas stage (see §2.5.4)

Importantly, the process of appending missed areas has to be altered slightly to prevent the Dervish entering an obstacle whilst tracing the perimeter of a missed region. This occurs in particular if there are missed regions immediately adjacent to obstacles, a situation which is more than likely. Therefore those regions should have buffered versions of the obstacles subtracted from them before their perimeters are traversed.

2.6.2 Effect of obstacles on connecting contours

In the contour-joining phase, we add straight route segments which join different contours; usually each contour is joined its nearest unvisited neighbour. When we add obstacles, this situation changes in two ways. First, the definition of 'nearest' is different, as crow-flies distance ignores the extra distance required to avoid obstacles. Secondly, we actually need to find a *path*[12] past those obstacles. If the only obstacles are convex and the Dervish can fit between them, then a very simple algorithm can find a way past them. If we allow more complex obstacles, or accept that they may be too close for the Dervish to fit between, then a correspondingly more general path finding algorithm is required.

Allowing these generalisations means that we're allowing the obstacles to be arranged like a maze, in the worst case. One solution, described by Latombe in [9], is to use *visibility graphs*. The nodes of a visibility graph are the initial point and the target point for the route, plus all the vertices of the obstacles. Arcs join nodes if and only if either the line segment joining them is an obstacle edge or the line joining them does not intersect any obstacles [9]. Finding a path across this graph is a much simpler problem. Constructing the visibility graph can be done in $O(n^2)$ [22].

A variation, which improves graph search time at the expense of graph creation time, is to use (reduced) *generalised visibility graphs*. Note that the graph need only be created once for any minefield map, so extra cost here doesn't matter if we improve the time required to search the graph, an operation we may need to perform several times during route generation. Here it is noted that just because one vertex is visible from another one does not mean it will be used in any shortest paths. Latombe [9] defines these graphs when curved edges are allowed. All convex corners are included in the graph, and points on curved edges are connected if the line joining those points is unobstructed and would form tangents to the curves at each end.

2.6.3 PathFind algorithm

The algorithm I use for finding paths past obstacles is more simplistic than those described above, but can easily be replaced when a better algorithm is coded in Avenue. 'PathFind', as I call this algorithm, assumes that the space that the path must not go through (i.e. the obstacle region buffered by the radius of the Dervish) is

[12] I use the term 'path' here to mean sequence of steps to get from an initial point to a destination point, in contrast to the term 'route', used to refer to the planned motion of the Dervish in covering the whole minefield.

convex. By buffering the obstacles first, the problem is reduced to finding a path for a point, rather than trying to move an object with area.

PathFind maintains a stack of *waypoints* it needs to visit. Each waypoint is tagged to say either we will need to find a path to it ('S_FIND_ROUTE'), or we can go directly to that waypoint ('S_GO_DIRECT'). Initially, the waypoint stack only contains the destination point, tagged to say we need to find a path to it, and the list of points in our final path is empty. When we pop an item from the waypoint stack, if the tag is S_GO_DIRECT, we can add that point straight to the path.

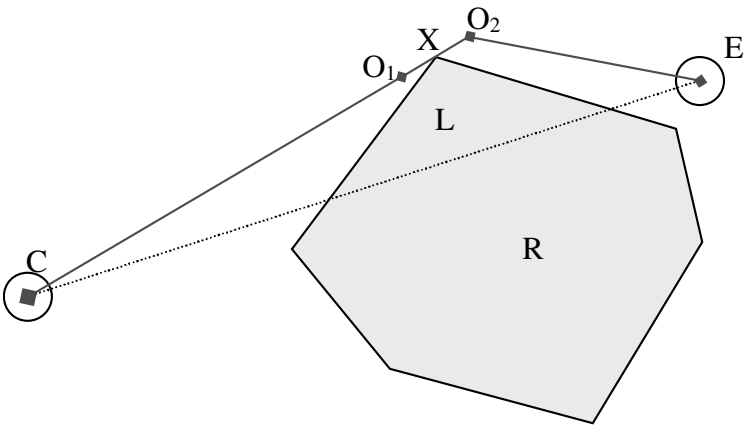

Figure 27 - PathFind avoiding a convex obstacle. The start point is C. E is the end point. CE is the direct path which is used to split the obstacle into two regions, L and R. Because L has the smaller area, PathFind looks for an extremity on the left, and finds X. From X it can calculate the offsets of O_1 and O_2. E, O_1 and O_2 are then put on the stack tagged so that a path is found to O_1, then we go direct to O_2, and then a path is found to E.

For popped items tagged S_FIND_ROUTE, we need to check for obstacles intersecting the line from the last point in the path so far to the waypoint just popped (the *direct path*). If there are none, we can again just add this point to the list of points so far. In cases where there are obstacles in the way, we pick an *extremity*, and avoid the obstacle by going around it past that point. An extremity is a left-most or right-most point in the obstacle looking along the direct path. Because this algorithm assumes obstacles are convex, we know that there will be exactly two extremities. PathFind decides which to pick by dividing the obstacle along the direct path and then looking for an extremity on the left or right depending on which side has the smaller area.

PathFind detects extremity points by tracing through each vertex in the obstacle, and recording which point is most left and most right when looking along the direct path. Having found the extremity we want, we set up the waypoint stack so as to find a path to the extremity, then find a path to the destination. However, finding a path to the extremity will always fail, because the line to the extremity by definition includes a point within an obstacle. So we find a path to a point a little bit nearer than the extremity, then go direct to a point on the other side of the extremity, and from there find a path to the destination. This is illustrated in Figure 27. Note that in Listing 3, these are pushed onto the stack in reverse order, so the order is correct when they are later popped.

The waypoint stack is an important part of this algorithm as it enables PathFind to cope with any number of obstacles getting in the way at any point. As long as there is room for the Dervish to get past, PathFind can. The routes are often not optimal, but

they aren't bad. At the beginning of the While loop in Listing 3, my implementation also includes a check to ensure that the stack doesn't grow beyond a certain size (suggesting no progress is being made), or that the route doesn't have above a threshold number of vertices (indicating some other failure[13]).

```
/* PathFind */
1.  Push (destination point, S_FIND_ROUTE) onto waypoint stack
2.  Set PointsSoFar = {startPoint}
3.  Set PointCurrentPosition = startPoint
4.  While stack size ≠ 0,
5.  Let (endPoint, tag) = Popped item from stack
6.  If (tag = S_GO_DIRECT) Then
7.      Append endPoint to PointsSoFar
8.      Let PointCurrentPosition = endPoint
9.  ElseIf (tag = S_FIND_ROUTE) Then
10.     If (PointCurrentPosition or endPoint intersect any obstacles) Then
11.         report error and return nil
12.     End If
13.     Check if line(PointCurrentPosition -> endPoint) intersects any obstacles
14.     If (no intersections found), then
15.         Append endPoint to PointsSoFar
16.         Let PointCurrentPosition = endPoint
17.     Else
18.         For the nearest obstacle, find an extremity point, extPoint.
19.         Let offset = small vector parallel to (PointCurrentPosition -> endPoint).
20.             Push (endPoint, S_FIND_ROUTE) onto waypoint stack
21.             Push (extPoint + offset, S_GO_DIRECT) onto waypoint stack
22.             Push (extPoint – offset, S_FIND_ROUTE) onto waypoint stack
23.     End If
24. End If
```

Listing 3 - Outline of PathFind

[13] This occurs, for example, if an obstacle has concave parts and the destination is within the concave region. Given the assumptions for PathFind, this problem should not arise.

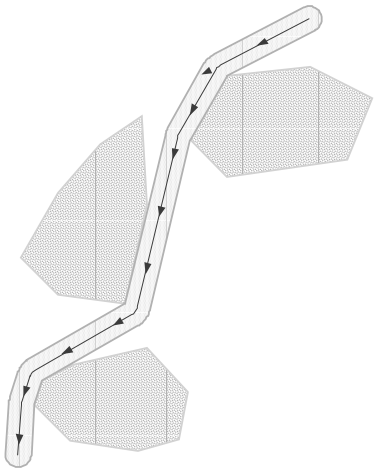

Figure 28 - Screen capture of PathFind coping with several convex obstacles

This algorithm has the advantage that it does not involve the creation of particularly complex data structures, such as visibility graphs, so can operate quite quickly. Nor does it require recursion (which isn't supported by Avenue).

3 Comparison

3.1 What I compare

Obvious measures for comparing route generators on a given route are the *calculation time* and the total *route length*. Routes that do not manage to cover the whole of the visitable area of the minefield, I count as having failed completely.

It is worth acknowledging that the measure of route length is flawed however, if we note the likelihood that the Dervish has a second mode of operation that is quicker but unsuitable for the purposes of mine detection (see §1.1). This means that time and fuel required to follow the route are no longer dependent entirely on the total route length, but also depend on what proportion of the route was spent in each mode. Without figures indicating the difference in speed between the modes, it is unfeasible to use that as a measure of comparison. Therefore, I do not use how much time could be spent in the quicker mode in the following analysis.

A third quantitative measure of the efficiency of a route is the ratio of minefield area to route length (the *area:route length ratio*). After all, route length is what dictates execution time, so we want a measure that reflects its importance. If there is a lot of overlap in a route, then the route will need to be longer to cover the same area, so such a route would have a lower ratio. For a mathematical justification of this measure, see §1.4.

Note that the optimal ratio value depends upon the shape of the minefield and the size of the Dervish, so it makes no sense to compare ratios calculated for different minefields. However, for a given minefield and Dervish size, we can calculate an upper bound on the area:route length ratio. We can imagine the simplest case when the minefield is exactly the same shape as a straight line buffered by the Dervish radius, and long enough keep the area unchanged. In reality, the minefield shape is no simpler than that, and therefore no generated route that covers the whole area can be shorter than the straight line we imagined.

We can easily calculate the length of this hypothetical straight line:

$$\mathrm{A} = \pi r^2 + 2rl$$

$$l = \frac{\mathrm{A} - \pi r^2}{2r}$$

So the optimal area:route length ratio can be derived:

$$\frac{\mathrm{A}}{l} = \frac{\mathrm{A}}{\left(\frac{\mathrm{A} - \pi r^2}{2r}\right)} = \frac{2\mathrm{A}r}{\mathrm{A} - \pi r^2}$$

ArcView can be programmed to automatically display this value for comparison purposes, whenever a route is generated. It is important to bear in mind that it is very unlikely that any real route can achieve this ‘optimal’ ratio. However it does give us something to aim for, a finite point by which we can measure improvement.

An additional measure of comparison I use is the extent to which ground near the starting point is covered earlier in a route.

3.2 Data used for comparison

The minefields used for the comparison of route generators are shown in Figure 29. For fair comparison, the size of the Dervish is constant throughout, at radius 6m. The minefields, numbered 1 through 4, are progressively more complex. Minefield 1 tests just how good a route can be generated for a simple, convex shape. Minefield 2 has the additional problem of concave features. Minefield 3 is an hourglass shape, which tests to see if the route generator creates excess overlap at the neck of the hourglass. Minefield 4 is a larger, more complex shape, with sharp convex and concave corners.

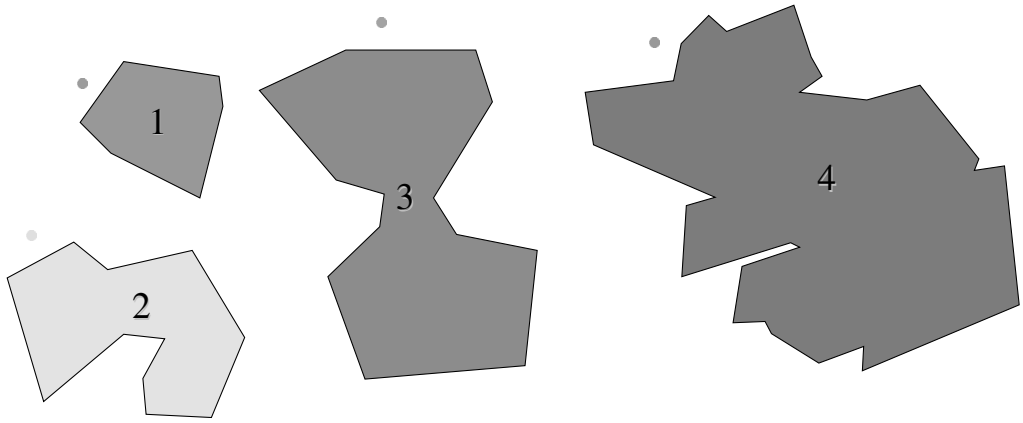

Figure 29 - Minefield shapes used for comparitive evaluation. The areas of minefields 1 through 4 are 4991.9m^2, 10215.9m^2, 20505.8m^2, and 35463.9m^2 respectively. For comparison, the Dervish is a circle of radius 6m, giving it an area of 113.1m^2.

3.3 Discussion of Results

The results are summarised in Table 1. Statistics are shown for each algorithm variation and for each of the four minefield shapes in Figure 29, which comes to 24 different routes generated. The upper-limit route length and ratio values for each minefield are also shown.

The results show that the Contour algorithm generates consistently more efficient routes than any of the Chomper algorithms. The area:route length ratios for Chomper are invariably less than half the upper limit, while Contour does much better. The random appearance of routes generated by Chomper3 is reflected in their area:route length ratios, which are the worst I recorded.

It is difficult to draw any firm conclusions on the value of the clustering phase. Minefield 4 using Contour shows a slight improvement with clustering, but Minefield 2 is slightly worse. As shown in §2.5.6, joining route parts by using clustering has a much better worst case than without clustering, and without more conclusive data, I am inclined to believe that in general clustering is a good choice.

It was a surprising result that the Chomper1 (‘arbitrary’) was very nearly as good as the best of the Chomper variants, the ‘ReverseChomper’. A final interesting point to note is that the Chompers’ routes took roughly the same amount of time to calculate as those generated by Contour.

An informal analysis of the extent to which ground near the starting point was covered first showed that Chomper2 gave the best results, unsurprisingly. However Contour gave good results in this respect, in that it visited outer parts of the minefield before diving into the middle, so access to a disabled Dervish would be safe.

Chomper4 slightly compromised some of Chomper2's obvious propensity to stay near the beginning, but gained the ability to finish near the starting point, a fairly useful characteristic not exhibited by any of the other algorithms.

	Minefield Shape											
	1			2			3			4		
	Execution time (s)	Route length (m)	Area:Route length (m^2/m)	Execution time (s)	Route length (m)	Area:Route length (m^2/m)	Execution time (s)	Route length (m)	Area:Route length (m^2/m)	Execution time (s)	Route length (m)	Area:Route length (m^2/m)
Calculated upper limit	-	407	12.28	-	842	12.13	-	1699	12.07	-	2946	12.04
Chomper1 (Arbitrary tie-break)	1	965	5.17	4	1773	5.76	7	3756	5.46	18	6050	5.86
Chomper2 (Nearest to start)	2	948	5.27	9	1901	5.37	10	3963	5.17	21	6373	5.56
Chomper3 (Beyond nearest)	1	1052	4.74	2	2116	4.83	10	4183	4.90	12	6965	5.09
Chomper4 (Reverse Chomper)	2	927	5.39	4	1820	5.61	14	3496	5.87	21	6149	5.77
Contour (no clustering)	1	547	9.12	2	1252	8.16	12	2050 †	10.0	34	3983	8.90
Contour (w/clustering)	1	547	9.12	2	1263	8.09	11	2050 †	10.0	34	3954	8.96

Table 1 - Comparison of algorithms on convex minefield. Evaluated for minefield 2 shown in Figure 29. The Dervish radius is 6m. The run times shown were of execution on a 500Mhz PC with 128Mb of memory.
† These runs gave routes which did not cover the whole route area. Investigation revealed that the incomplete route was caused by Avenue failing to buffer one of the shapes. This bug is described in more detail in §4.3.1.

4 Testing

4.1 What I aim to learn from testing

Here I am concerned with two things – the reliability of my implementation, and the reliability of ArcView. It is important to remember that what I have implemented is intended as a prototype, to prove that the concept of an automated route generation system is plausible.

4.2 Testing the route planning systems

The route planning systems need to be tested against a variety of minefield shapes to ensure they will be able to cope with both likely and unlikely situations. For this reason, I chose six different classes of minefield, representing different possible situations (some of which were particularly complex and unlikely).

The algorithms were run with three different examples from each of the following classes of minefield, i.e. 18 test runs per algorithm:

1. Narrower than the Dervish.
   These regions will definitely lead to regions that cannot be contoured by the Contour algorithm, yet which must be covered.
2. With acute corners.
   These shapes will test how well algorithms cope with regions that the Dervish cannot enter.
3. With noisy or very irregular edges.
   These regions will test whether the implementations can deal with poorly digitised minefield maps, with many small acute corners. Note that I ensured the minefield shapes were still properly defined polygons by not allowing self-intersecting polygon boundaries.
4. Disconnected minefield.
   This is to ensure that route generators are prepared to cover ground that doesn't need to be tested, if required.
5. 3-shape
   The 3-shape is likely to lead to a situation where the Dervish can take a shortcut across ground which isn't in the minefield, but doesn't have to.
6. With obstacles
   These minefields are necessary to ensure that the algorithms which claim to be able to avoid obstacles actually can.

The Dervish size was kept constant for each test, at 6m.

4.2.1 Errors to look for

The errors I looked for were:

(a) Did not complete (i.e. the computer froze)

(b) Reported an error and did not finish

(c) Missed regions of the minefield which were reachable

(d) Entered an obstacle region

(e) Other miscellaneous problems

Errors (a) and (b) are clearly the most serious. If the algorithm cannot generate a route, then no work has been done and the user is left with few courses of action. Error (d) is similar, but it may be assumed that the Dervish at least covers some area before running into the obstacle. Such an event could be a real problem if the obstacle region is a hole, a lake, or the known location of an anti-tank mine. Error (c) is least problematic because missed regions, reachable or not, can at least be highlighted to allow a human to check them later, or can be used to create a new map to allow the software to have another go. Clearly, though, we still want to avoid error (c) whenever possible.

I did not measure the efficiency of any of the routes generated during this testing phase; that was studied in §3. However, I looked for any ridiculous routes (such as the Dervish taking a 20m detour out of the minefield), with the intention of classing those as error (e), along with any other obvious problems not already classified.

		Algorithm					
		Chomper1	**Chomper2**	**Chomper3**	**Chomper4**	**Contour** no clustering	**Contour** clustering
Minefield Shape	**1**	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓
	2	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓
	3	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓
	4	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓
	5	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓
	6	d d d	d d d	d d d	d d d	✓✓✓	✓✓✓

Table 2 - Results of testing the route generating algorithms. ✓means no error observed. Letters indicate class of error observed, as listed in §4.2.1. Because I used three representatives of each class of minefield, there are three results in each table cell.

4.3 Reliability of ArcView

There is some doubt over the reliability of some of ArcView's shape manipulation operations, as expressed by Held [10]. Note that it is impossible to perform white-box testing without knowing something about how Avenue and ArcView themselves are implemented. ArcView is not published as 'Open Source' software, so this is not feasible. However, we can perform testing to try to establish the severity of known bugs in ArcView, as well as performing some testing of the Avenue routines most crucial to the route finding algorithm's operation.

DeLaune [5][4] maintains a list of bugs found in ArcView while developing the Xtools suite of ArcView add-ons. I include relevant bugs below.

Obviously, ArcView has some limit to its precision[14], so very small inaccuracies I will count as a limitation rather than a bug in ArcView.

4.3.1 ReturnBuffered

The ReturnBuffered request returns a buffered version of a shape[15]. Because ArcView cannot represent curves, any curved portions of the buffered version are approximated to line segments; this means ReturnBuffered always returns an approximation when the buffered version has curves.

It is important to know whether Avenue approximates consistently larger or smaller than the buffered region, or not consistently at all, because there are consequences for the algorithms I have implemented. In particular, I consider the Contour algorithm here.

Possible implications include -

- Internal buffering too small – Leave too much gap around done areas.
- Internal buffering too big – Overestimate work still to do, leading to unnecessary overlap.
- External buffering too small – Underestimate work done when buffering path-so-far. May cause unnecessary overlap.
- External buffering too big – Overestimate work done – possibly missing some sections. Leave too much gap around obstacles.

If it sometimes does either, I should try to establish the range of values by which it is approximating, so I can compensate for their effect.

By buffering a point to create a unit-radius circle, I used Avenue's ReturnArea request to compare the result with the desired area, namely π. The measured result was 3.12567, a 0.5% reduction in area. I also tested the distance from the origin of each vertex in this approximation to a circle, and this distance never exceeded 1. The perimeter's nearest point to the origin was at a distance of 0.996195, meaning that the distance from the centre is up to 0.38% too small. Buffering a square internally created a new square indistinguishable from the desired one.

The implications of the inaccuracies mentioned above are not serious. The worst effect would be for a route generator to cover one section of ground twice, which is hardly disastrous.

More concerning is the discovery when performing comparison testing, of a shape which ArcView fails to buffer correctly. This bug led to an incomplete route being generated. The missed region was highlighted by the route generation framework code, so the region was visible, and would not have posed a safety problem in the field. This is the only time I have found such a shape in the eight months I have spent developing and testing route generation software in Avenue, so I believe it to be a very rare situation. I have sent code that reproduces the bug to ESRI and Mike DeLaune, who maintains an Avenue bugs list.

---

[14] Avenue reports that 1/3 = 0.33333333333333331, so I can probably assume 15 significant figures of accuracy in Avenue's mathematics.

[15] Buffering is defined on page 14.

4.3.2 Distance

Errors calculating distance, using Avenue's Distance request, may result in nearest points that aren't nearest for the Chomper algorithms, and nearest contours that aren't nearest for the Contour algorithms. If distance were particularly unreliable, it would have serious consequences for the framework code that calculates the length of a route when recording it.

I test the Distance request by creating a set of randomly generated right-angled triangles. I calculate the length of the hypotenuse of each triangle using the Distance request, and compare the result with that achieved by using Avenue's mathematics operators to perform the simple geometry. I used 200 test triangles. The first hundred had widths and heights in the range 0-100. The second hundred were in the range 0-0.1. The calculated lengths matched up exactly for every result.

4.3.3 Intersection, Difference and Union

There are three main geometry commands provided by Avenue that I use extensively throughout my code. Any errors in the implementations of these Boolean operations on geometric shapes would have serious implications for any route generator implemented in ArcView.

ReturnIntersection is used when calculating the nearest point to a shape (see Appendix A), which affects all the route generators. ReturnDifference is used to calculate the area remaining to be visited at various stages in route generation by the Contour and Chomper algorithms, as well as when recording a generated route. ReturnUnion is used to union the collection of obstacle shapes before processing, as well as to record route segments in the Chomper algorithms. Other algorithm variants do not use the union feature.

I performed various tests using an assortment of shapes to ensure that these operations work as expected. However, confirming correct operation for all but the simplest shapes is a non-trivial problem. No issues arose.

DeLaune [5] reports that if most, but not all of polygon A overlaps polygon B, then A.ReturnDifference(B) should return a small remainder, but sometimes returns a null polygon. I was not able to reproduce this error.

4.4 Conclusions

Testing the route generation algorithms, involving a total of 108 generated routes, showed that the algorithms work on the routes for which they were intended. The Chomper algorithms fail to properly avoid obstacles in minefields which feature them, but the design of the Chomper algorithm makes no allowance for obstacles, so that is no surprise. It was particularly interesting and reassuring to see that the algorithms work well on minefields that are disconnected or are particularly complex (shape 4 and 3).

Testing Avenue didn't reveal any problems, which while reassuring, does not mean that Avenue is totally reliable. The known bugs, in addition to the ReturnBuffered case I discovered, mean that much more extensive testing of Avenue is required to ascertain exactly what situations it has trouble with. Preferably, the final platform on which the route generation software is implemented would have geometry code that

the Dervish development group have access to, so they can look for and fix bugs themselves. More thorough testing would require a larger set of minefield shapes, so a system to generate random ones, or even a library of real minefield data, would be very useful. However, for the purposes of prototyping, ArcView and Avenue are clearly good enough.

5 Conclusions

5.1 Review of Achievements

I have designed and implemented two quite different algorithms for generating routes for the Dervish, such that following the generated routes will result in complete coverage of a minefield. The framework for the implementations allows for testing and comparison of different algorithms, and I show how generated routes can be quantitively evaluated against each other. I have shown that of the route generators I implemented, the contour-following approach is substantially better; it generates more efficient routes and copes well with known obstacles in the minefield.

This system can act as proof of concept and as a working prototype for the route generation system for the Dervish. The algorithms have limitations (specifically, no obstacles when using Chomper, and no concave obstacles when using Contour), but otherwise they do generate routes.

5.2 Directions for Further Work

There are many things I would have liked to implement, but for which there was not time within the constraints of this project. These are things I decided against attempting, but which present themselves as reasonable extensions – possibly to be considered as projects in later years. These extensions bring the project closer to matching the general requirements for a route planning system for the Dervish.

5.2.1 Grid following approach

Arkin, Fekete and Mitchell suggest in [1] placing a hexagonal grid over the minefield, and treating it as a travelling salesman problem, where the grid vertices are the places that are visited. This was described in more detail in section 1.6.1, along with Jerrum's suggestion of a square grid and a Hamiltonian Cycle finder. An implementation of either variation would make for an interesting alternative.

5.2.2 Unknown obstacles

Minefields may contain such hazards as tanks, tree stumps, wrecked cars and so on, which would obstruct the Dervish trying to follow a path. These items are unlikely to be mapped, so we can't easily avoid just driving straight into them. In this event, we could make an immediate note that there is some obstruction, and then recalculate a route with this updated map. Alternatively, there could be some special procedure for dealing with unknown obstacles, involving tracing around the outside of the object to determine its extent, and then we can record the shape of the whole object on the map. In fact, the Dervish could have a second function, of mapping a region using various sensors mounted upon it [19].

5.2.3 Return to base

The Dervish carries a finite quantity of fuel. Possibly it could be refuelled wherever it stops, but a safer and more convenient approach would be for it to return to base before it stops, so it can be refuelled safely off the minefield. Stephen Salter [19][20]

suggests that the return-to-base action could use an alternative travelling mode, akin to walking a wardrobe. This action doesn't necessarily cover every point of ground along the path, but would allow much quicker travel, so in terms of time and fuel used, the return-to-base action might only have a small impact.

5.2.4 Multiple Dervishes

The remote-control mechanism means that two or more Dervishes can be controlled as easily as one – with the caveat that they would all make the same movements. Sending out three or four Dervishes to cover the ground as a team could mean that large regular shapes, such as rectangular fields, could be covered quite quickly. To be able to cope with more general cases, taking full advantage of the Dervishes would be a non-trivial problem, as sending one Dervish into a clear area may have the side effect of sending another Dervish into an obstacle.

5.2.5 Improving upon a predefined path

Another potential extension to a project could be a path-improving algorithm. In the event that we hit an obstacle halfway through a route, the chances are that most of the remaining planned route is fine – we only need to make a few tweaks. Alternatively, a human operator with a pen-based computer may want to sketch out a preferred route in advance, and the computer can then fine-tune it to make a route with minimal overlap. Calculating area:route length for possible changes would indicate which change would lead to the greatest benefit.

5.2.6 Concentrating more on certain areas

If a human operator or sniffer dog has a suspicion about a particular part of the minefield, it would be desirable to have the Dervish be more thorough in that area. Possibly certain regions on the map could be highlighted for a second pass by the Dervish.

5.2.7 Implementation language

During development of the route generators, I have learnt a great deal about ArcView and Avenue. Avenue is a powerful language, and very complex results can be achieved. The ease with which many abstract geometric operations can be performed makes this the ideal platform on which to build such a prototype.

I would not recommend Avenue as the implementation language for the final route generator, though. For large programs involving much code, Avenue is particularly unsuited. Avenue does not support recursion; debugging facilities are weak; function calls and parameter passing are clumsy, and error trapping is non-existent. In short, Avenue is not conducive to good programming practice. I would suggest using an industry standard implementation language such as C++ or Java, despite the possibility of extra work involved implementing geometric operations.

6 Bibliography

[1] E. M. Arkin, S. P. Fekete, and J. S. B. Mitchell. *Approximation Algorithms for Lawn Mowing and Milling*. Available as ZPR report 97-255, 1997.

[2] V. B. Balayoghan. *Space-Filling Curves*. On the web at **http://www.cs.utexas.edu/users/vbb/misc/sfc/Oindex.html**

[3] A. Bogomolny. *Plane Filling Curves*. On the web at **http://www.cut-the-knot.com/do_you_know/hilbert.html**, 1996.

[4] M. DeLaune, *Avenue Bugs Affecting Spatial Overlay Operations – ArcView 3.2*. On the web at **http://www.odf.state.or.us/SFGIS/Avenbugs/AvenBugs32.htm**

[5] M. DeLaune, *Avenue Bugs Affecting Xtools.* On the web at **http://www.odf.state.or.us/SFGIS/Avenbugs/AvenBugs.htm**

[6] Dervish Website, **http://www.dervish.org/**

[7] ESRI, Inc. ArcView Online Help 1998.

[8] M. Held. *On the Computational Geometry of Pocket Machining*. Lecture Notes in Computer Science, Vol. 500, Springer-Verlag, June 1991.

[9] M. Held. Personal website. **http://www.cosy.sbg.ac.at/~held/**

[10] M. Held. Private communication by e-mail. May 2000

[11] J. Hershberger, *Finding the Visibility Graph of a Simple Polygon in Time Proportional to its Size*. 1987, Stanford. In Proceedings of the Third Annual Symposium on Computational Geometry. ACM Press 1987.

[12] B. Huber, Private communications by e-mail. Dec 9, 1999 and Dec 11, 1999. Quantitative Decisions is at http://www.quantdec.com/

[13] M. Jerrum. Meeting with Mark Jerrum to discuss approaches to computational geometry problems such as this one. 1999.

[14] J. Latombe, *Robot Motion Planning*. 1991 Stanford. Kluwer Academic Publishers

[15] D. McLeod. *Getting Started With Avenue – Course Notebook*. Unpublished draft, copyright 1998.

[16] D. McLeod. *Working With Avenue – Course Notebook*. Unpublished draft, copyright 1998.

[17] D. McLeod. Private communications on ArcView, Avenue and GIS. 1999-2000.

[18] A. H. Razavi. *ArcView GIS/Avenue Developer's Guide, Second Edition*. OnWord Press, 1997.

[19] S. H. Salter. Meeting with Professor Stephen Salter to discuss background to Dervish project and long-term requirements for the route-planner. Nov 9, 1999.

[20] S. H. Salter, C. N. G. Gibson. *Map-Driven Platforms for Moving Sensors in Mine Fields*, Department of Mechanical Engineering, University of Edinburgh. Journal of Mine Action, Volume 3, Number 2, 1999.

[21] S. H. Salter. *Cartesian to Hyperbolic Conversion for Dervish Decca Control in Command Mode from CAD Based Minefield Maps* in International Workshop

on Robots for Humanitarian Demining. Laas-Curs, Toulouse pp. 89-108, Sept 1998.

[22] E. Welzl, *Constructing the Visibility Graph for n Line Segments in O(n^2) Time*, Information Processing Letters 20, p.167-171, 1985.

Appendix A – Implementation issues

Finding a nearest point

Here I explain how I implemented NearestPointInShape.

ArcView provides a request to return the minimum distance, *d*, of a shape S from a point P, but doesn't tell us which part of S is at that distance. We can buffer P by $d + \delta$, and get ArcView to return the intersection of that circle with S. This gives us a set of points in S at a distance $\leq d + \delta$. Note that due to precision errors, letting $\delta = 0$ gives us no intersection at all.

Sometimes, particularly in the Chomper algorithm, there are several nearest points. If ArcView could represent curves, we would find cases were there are infinitely many points. Normally, we're only interested in one such point, so a tie-breaker is needed to eliminate all but one potential nearest point. It is the choice of tie-breaker that distinguishes the different versions of Chomper, and which give three different NearestPoint variations.

Chomper1 picks whichever is the first point ArcView returns in the list of points within the intersection region. The ordering ArcView uses is arbitrary, so this means that the nearest unvisited point that Chomper1 picks is also arbitrary.

Chomper2 uses distance from the start point as a tie-breaker, always choosing the point the least distance from the initial position in the route. This means that the distance from each vertex of the intersection region to the start point needs to be calculated.

Chomper3 uses the same tie-breaker as Chomper2, but goes beyond the winning point.

Chomper4 uses distance from the start point again, but this time the point furthest from the start point wins.

It seems strange that Avenue does not have a built in ReturnNearestPoints(shape1, shape2) request. Having to work around this oversight will lead to some inefficiency when ArcView must be doing much of the work involved anyway to calculate the distance between two shapes.