An optimal pathfinder for vehicles in real-world digital terrain maps

by

F. Markus Jönsson

[Note: named Markus Dimdal since 2004]

[This paper was published in April 1997 by:]

The Royal Institute of Science, School of Engineering Physics, Stockholm, Sweden
This paper is presented by the Department of Numerical Analysis and Computing Science

Abstract

This paper describes an algorithm for approximately finding the fastest route for a vehicle to travel between two points in a digital terrain map, avoiding obstacles along the way. There can optionally be one or more ‘enemies’ located in the terrain which should, whenever possible, be avoided. Specifically, the terrain map model consists of a 2D height raster and a terrain class raster. There are also roads, in the form of vector data. The vehicle speed is a function of the terrain or road class as well as constrained by a maximum allowed slope. The enemies are avoided by staying out of their line of sight. However, the general results of this paper should be feasible for a much wider range of applications ranging from complex GIS systems to home computer games.

The approach taken in this work is to translate the problem into a ‘least cost path’ graph problem with an associated cost function on the graph edges. Standard graph algorithms can then be used to solve the graph problem exactly. In order to be feasible for use on standard personal computers, a simple progressive scheme is needed for very large graphs (containing many millions of nodes) giving approximate solutions in reasonable time and memory space.

[Swedish title and abstract:]

En optimal stigfinnare för fordon i verklighetsbaserade digitala terrängkartor

Referat

Detta arbete beskriver en algoritm för att finna den snabbaste stigen för ett fordon att färdas mellan två punkter i en digital terrängkarta, undvikande eventuella hinder på vägen. Det kan även finnas en, eller flera, ’fiender’ i terrängen som bör undvikas när så är möjligt. Mer specifikt består terränkartan av ett 2D höjdraster och ett terrängklassraster. Det finns även vägar i form av vektordata. Fordonets hastighet är en funktion av terräng- och vägtyp. Underlaget får ej heller luta för mycket. Fiender undviks genom att hålla sig utanför deras siktfält. De generalla resulatateten i arbetet bör dock vara tillämpbara på allt från stora GIS system till enklare spel för hemdatorer.

Ansatsen som görs i detta arbete är att reducera problemet till grafproblemet ’lägsta kostnadsstig’ med en associerad kostnadsfunktion på grafens kanter. Kända grafalgoritmer kan sedan användas för att lösa grafproblemet exakt. För att vara användbart på vanliga persondatorer behövs en enkel progressiv metod för riktigt stora grafer (grafer med många miljoner noder) som ger approximativa lösningar med rimlig tids och minnesförbrukning.

Preface

This work is the product of my master's thesis project, formally performed for the Department of Numerical Analysis and Computing Science (NADA) at the Royal Institute of Technology, Stockholm, Sweden. The practical work has been commissioned by, and performed at, S&T Datakonsulter AB, Stockholm, Sweden.

The ultimate results of the algorithm research, implementation and testing being done for this project is due to be used as a small part of a larger military troupe simulator in use with the Swedish defense force. In this simulator a number of more or less autonomous units, called 'actors', interact with each other and their surroundings according to rules dictated by a knowledge database. In this scenario, an armed vehicle travelling to a given destination, along a pre-planned route of roads, may sometimes be discovered by an enemy. Thus, the necessity arises for the vehicle to autonomously find a new path to its target, avoiding the enemy if possible, perhaps venturing into the terrain and avoiding obstacles along the way. That is the practical focus of this work.

I wish to express my gratitude to all the people at S&T, who has provided me with invaluable help, support and companionship. An honorary mention goes to my supervisors: Prof. Stefan Arnborg at NADA, and Richard Elg at S&T.

Contents

1 Introduction

In a number of applications, the problem of determining the, in some sense, optimum path occurs. This could be anything from finding the fastest path in a network to determining the safest path for at robotic craft wandering upon the surface of Mars. In this context, we shall limit our scope to the special case of finding paths in Euclidean three-dimensional space. Even more so we shall limit ourselves to movements along a surface that can be projected onto a two dimensional grid. To be specific, we shall look at the case of finding the optimum path for a vehicle moving along the surface of our earth (1.2) as represented in today's geographic information systems (1.1).

1.1 Digital maps

Representing traditional cartographic information (e.g. various kinds of maps as in figure 1) in a computer accessible format presents both a number of challenges as well as many new opportunities.

The vast amount of information that is often present requires efficient storage techniques, while performance issues place high demands on fast retrieval and indexing methods. This is mainly a database problem and shall not be dealt with further in this paper. It is assumed that we can efficiently retrieve a properly formatted set of data for the portion of a map of interest.

Figure 1 - Example of a digital map covering central Stockholm

The blending of different data types presents many new opportunities for looking at data in ways unconceivable with traditional techniques. Computer aided analysis (interactive or automated), processing, evaluation and visualization of geographic data represents currently several active fields of research. To name a few diverse fields: remote sensing, image processing and visualization, AI, environmental monitoring, land use, numerical analysis, database design, et c. et c. The present work is one humble example of what can be done with the aid of a computer and a blend of different types of digital map data.

An extensive introduction to digital maps and computer cartography can be found in [CLAR90].

1.1.1 Raster data

Raster data is a two dimensional grid of numerical values representing some measurable characteristic where the spatial data surface is projected onto the grid plane. The values at each discrete grid point, called pixels, are usually interpreted as averages over the area projected onto each grid point (see figure 2). Other interpretations are of course possible depending on the source of the data (sensor, assimilation process or whatever). This is all intuitively quite simple since, except for the discretization, it is quite analogous to how geographic data (maps) has always been traditionally represented. We shall not here delve into the further complexities involved (e.g. projection methods).

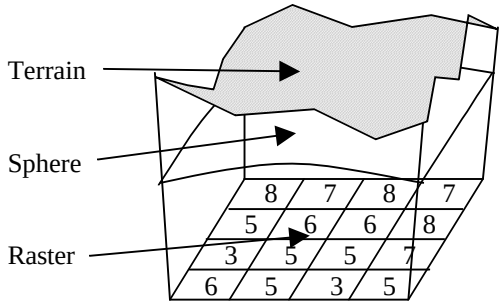


Figure 2 - Example of altitude represented as a height raster

We are here mainly interested in two types of raster data:

1.1.1.1 Terrain class raster

Based on for example satellite imagery coupled with ground validation, it is possible to construct a classification of the dominant terrain type. Quite like the traditional land use maps. The data used for the examples presented in this paper assigns each raster pixel to one of 16 terrain classes; e.g. exposed rock, water, dry marshes, et c. The data has a pixel resolution (grid spacing) of 25m.

1.1.1.2 Height raster

The earth surface height above sea level can be measured by air born radar or by other means. The data used for the examples presented in this paper have a 50m pixel resolution and 1m vertical resolution.

1.1.2 Geometric data

Geometric data, often referred to as ‘vector’ data, consists of lists of discrete points (surface coordinates) and associated values interpreted to define the shapes of lines, curves or other geometric objects that might be useful.

1.1.2.1 Road vectors

The data used for the examples presented in this paper has three classes of road vectors, small, medium or large roads, defined as polygon chains.

1.1.2.2 Obstacles

Unpassable ‘lines’, e.g. rivers or minefield borders, may also be represented as different types of vector data.

1.2 Problem definition

The incentive for this project was to find, in a military unit simulator, an efficient way for vehicles to autonomously find the fastest way to a given destination. It should not be to hard for the reader to imagine several other possible applications of such an ‘optimum-path finder’. The immediate, practical, problem that this paper sets out to solve is the following problem instance:

Given data as described in 1.1, find the fastest path for a vehicle to travel between two points, here called the source and the destination. Obstacles along the way should of course be avoided. There can optionally be one or more ‘enemies’ located in the terrain which also should, whenever possible, be avoided. The vehicle speed is a function of the terrain or road class as well as constrained by a maximum allowed slope. The enemies are avoided by staying out of their line of sight. The memory usage should be limited so that it is possible to run the algorithm on a standard personal computer, even with raster sizes on the order of a million pixels in the map. For sufficiently small maps and static enemy positions, it should have near real-time behavior.

1.3 Objective

This paper is presented as a part of the authors master’s thesis project. The objective of the project - to research, develop, implement and test a solution to the problem as defined in (1.2). This paper presents the general results found, and algorithms used during the work.

1.4 Outline

Section 2 provides a short overview of a few different approaches to solving related problems. In section 3 a method is described to approximate the problem by reducing it to essentially a ‘shortest path’ graph problem. Efficiency and performance issues of the graph representation are also discussed. Section 4 tackles the problem of searching for the optimum path in the graph. A procedure to obtaining approximate solutions while minimizing memory usage on very large graphs using a progressive scheme is discussed. Section 5 presents a number of behaviorally illustrative test runs of the method. The paper is rounded of with a brief section of conclusions and ideas for areas of future work.

2 Overview of path finding methods

Through the years, a few different approaches to the problem of finding paths though space have been proposed within various disciplines where the problem naturally occurs. Most of these leads to some sort of graph search problem. Here is given a very brief introduction to the most common of these approaches.

2.1 Line intersection methods

This is a common approach to the problem when the data consist of geometric objects and we have a binary type of terrain. I.e. all objects are absolute obstacles in the sense that they cannot be passed through and all the terrain not occupied by an object is considered unobstructed with no variation in vehicle speed or other parameters. In this case, the Euclidean-distance shortest path is also the fastest path. The basic idea is as follows: first construct the convex hull of all objects. The corners of all the hull polygons define a set of vertices. Connect all vertices with edges. Then do some more or less clever filtering of these edges and finally search for the shortest valid path between source and destination along a series of edges using standard graph algorithms. The main difference between methods in this class is how the filtering is done. The purpose of the filtering is to remove lines that cannot possibly belong to a shortest path and so dramatically reduce the number of paths that need to be examined. The first step is usually to remove the edges that (geometrically) intersect any other hull, i.e. edges that would pass through an obstacle. Then various ‘distance norm’ based rules can be applied to remove too far-reaching paths. Finally, many algorithms use some kind of heuristics to remove additional unlikely edges, often giving up the requirement of finding exact solutions in every case and gaining instead large reductions in the number of edges. Some attempts at line intersection methods can be found in [ELGR92], [MONT87] and [HOLM92].

These methods are unfortunately not very well suited to the problem at hand, both because of their binary nature and because of their inability to handle raster-based data very well. Of course, we could always transform every raster pixel to a polygon but the sheer amount of ‘lines’ that that would produce would simply be too staggering for our purposes. There are ways to deal with non-binary terrain but they also tend to suffer from the same problem of rapidly increasing graph sizes, especially for more complex terrain.

2.2 Weighted graph methods

Although they share the final graph search nature with the line intersection methods, these methods take a radically different, and in a sense opposite, approach to constructing the search graph. The basic idea is to divide space into discrete regions, here called cells, and restrict movement from a particular cell to its ‘neighbors’. Neighboring cells are those that can be directly reached from a particular cell. A directed graph is constructed by taking the cells as graph vertices and the possible movements to its neighboring cells as (directed) edges between the vertices. A weight function is defined by assigned a cost to every edge, corresponding to the ‘cost’ of moving along the edge in question (time, length, or

whatever function is appropriate for the problem). The space division, the definition of neighbors and the edge cost function can all differ between different methods in this class. One method based upon this approach is detailed in the following sections since this is the approach also chosen for this paper. Choosing the space division so that it coincides with the raster-based nature of our main data makes for a very flexible and yet efficient model. Some examples can be found in [STEF95], [WOOD97], [LONN96] and [PATE97].

2.3 Other methods

Various people have tried a few approaches of a more mathematical nature. For example, [KIMM95] uses a bivariate function to track the evolution of ‘equal distance curves’ on a surface and from that derive a scheme to find shortest paths on many surfaces.

Another idea is to define some kind of 'cost surface' and follow the local gradient direction, i.e. a 'steepest descent' method. However, although the basic idea is quite simple, handling the case of local minimas is not so. Consider for example a river that naturally flows in the direction of the negative gradient of the terrain height. In nature, it will build up lakes and pools around local minimas. Methods for detecting and treating such special cases could possibly be developed and used in a practical pathfinder but is not further examined in this paper.

Different heuristic based approaches have also been suggested. See for example the crash-and-turn algorithm of [LONN96]. In practice these methods in themselves behave far from optimally on anything but very simple terrain, are prone to ‘lock up’ situations and can often not even be guaranteed to find a path at all! However, heuristics may be a very good idea for speeding up more ‘exhaustive’ methods such as those in 2.1 and 2.2; especially if the requirement of always guaranteeing optimality can be sacrificed.

3 Graph construction

The problem (1.2) is reduced to a weighted graph problem. An explicit edge cost function is developed that takes into consideration all of the desired parameters. A theoretical derivation is presented in 3.1. It is a flexible model that could easily be modified for other path finding problems. An implicit, memory space efficient graph representation and details on a sample implementation are then presented in 3.2.

3.1 Problem reduction

The approach taken in this work is to discretize the problem into essentially a weighted graph search problem. Many of the basic ideas, if perhaps not the particulars, have been inspired by the work presented in [STEF95] as well as several earlier works. Similar, often simpler versions of the technique have long been in use with computer game programmers as a method to move units on the much smaller maps used by various strategy and other computer games. See for example [WOOD97]. For an introduction to graphs and related theory, see [BUCK90], [CORM90] or any other good textbook on graph theory.

3.1.1 Limiting space

The first step in our reduction is to limit the number of possible locations and the number of possible movements between these locations.

3.1.1.1 Restricting the search region

The extent of the surface where our vehicle could possibly move is, for our purposes, practically infinite in all directions. Clearly, we must limit ourselves to some smaller region in order to implement it on a computer. Furthermore, we would like this region to be as small as possible without compromising the optimality of the path. Both because we want to reduce the memory usage, and because it will have the positive effect of reducing the number of possible paths that we potentially will have to examine when searching for the optimal one.

How can the minimum size of this region be determined without a priori knowledge of the actual optimal path? If there were no untraversable obstacles present, the ‘cost’ of the 'straight line' path between source and destination could possibly be used as an upper bound for the ‘costs’ of the paths that we have to consider. It could then be used, in some way, as a means to limit the search region. However, this is much easier said than done, especially in a practical implementation. Moreover, it will nevertheless fail if there are untraversable regions of beforehand unknown extent (e.g. a long river crossing the line between the source and the destination).

It may even be that there is no possible path at all but that this is unknown until we have actually tried, and failed to find one. Thus, it is necessary to resort to using some kind of heuristic to determine a good region size. Knowledge of the particular application will be very useful here. For example, the extent of the largest obstacle that we expect that we will want to be able to find a way around, could be a good

guiding point. Another is the distance between the source and the destination. Practical tests have shown that at small scales the first of these two considerations is much more important while at larger scales the second is dominant. A simple compromise heuristic might then be constructed as b= a+ (1+c)* d. Here *b* is the length of the diagonal of the bounding rectangle (of the source and the destination). *d* is the distance between source and destination. *c* is an appropriate scaling constant (around 1/3 seems to work well). Finally, *a* is approximately the size of the largest obstacle.

In some applications, it may instead be preferable to let the end user manually specify the search region and so avoiding the problem entirely (as far as we are concerned). As a last resort, it is always possible to enlarge the search region and restart the pathfinder in case it did not succeed in finding a valid path in the chosen search region.

3.1.1.2 Dividing space into cells

Let's proceed by restricting (discretizing) the infinite number of locations of the continuous space into a finite number of points. To facilitate the representation it is most convenient to assign these points in a regular mesh. A region of space called a cell cover each point. The point itself shall be called a vertex. All cells are disjoint and the union of all cells should cover the complete space. The motion of our vehicle is restricted to movement in ‘straight lines’ between vertices. The mesh can be of any kind suitable to the particular space and application under study. Two examples of planar meshes are given in figure 3 and 4.

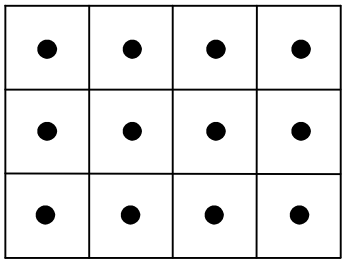


Figure 3 - Regular quadratic planar mesh

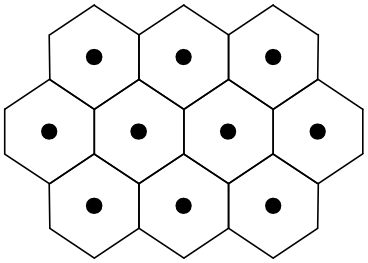


Figure 4 - Regular hexagonal planar mesh

For our application, the regular ‘quadratic grid' planar mesh shall be used. It is particularly suitable since we can then chose the cells to precisely coincide with the pixels of our raster data.

As mentioned in 3.1 we will reduce our problem to a weighted graph problem. We now have defined the graph vertices as the cell 'center points'.

3.1.1.3 Restricting movement to edges

The next step is to restrict the number of possible movements from a given cell. If it is possible to go from one vertex to another without passing through another vertex point, we shall say that they are connected by an (directed) edge. The vertex that the edge goes from is called the source. The vertex at the other end other is called the destination. Trivially, there could always be an edge from a given cell to itself; this case shall be neglected in the future as staying in one point is not a very useful

segment of an ‘optimal path’. The cells or vertices that are the destinations of all the edges of a given cell are called its neighbors. Which cells are considered as neighbors to a given cell, is up to us to decide. Ideally, all cells should be neighbors to all other cells (giving us a so-called 'complete' graph). However, this would mean having an incredible number of edges. Since each cell would be required to have as many edges as there are other cells we would for example get 10^{12} edges for a graph with one million cells. We will later use graph optimizing algorithms whose practical performance are highly dependent on the number of edges so this is clearly unacceptable. Instead we shall go toward the other extreme and only define the ‘physically’ adjacent cells as neighbors to a given cell. In the case of our quadratic mesh, every cell (that is not located on the boundary of the search region) has a layout as seen in figure 5. There is a choice of selecting either four or eighth adjacent cells as neighbors. Since we want maximum accuracy and selecting eight rather than four will not present much increased complexity; we will select the eight cells shown in medium gray in figure 5 as neighbors. We shall later see that this will allow us to use a highly efficient graph representation. What we do is to trade in accuracy for less complexity and better time performance. However, it is a very favorable trading, see further 3.1.2.6 and 6.2.2. The number of edges is now linear in the number of vertices.

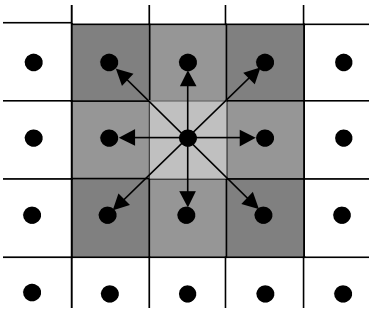


Figure 5 - A cell with edges to eight neigbours

To recapitulate: we have restricted the number of movements from each vertex to eight edges (or less at the borders) and finished defining a directed graph representation of space.

A point of interest is often the 'physical' length of an edge. The projection used to map the actual surface to the raster should be taken into account if we want to be pedantic. However, for the purpose of finding paths, the scale of the areas in question, are usually very small in comparison with the curvature of the earth. Then the raster can be taken as being a perfect rectangle and the projection can be ignored. The error in a single edge length will be very small for reasonably ‘regional’ scale maps. For maps covering several nations with a coarse grid, it may no longer be true but that falls outside the scope of our problem. Thus, we can simply use Pythagoras theorem in three dimensions to calculate the edge lengths. Yet another approximation that is very useful and has been made use of in our sample implementation is to ignore the height difference between the cells when calculating the edge length. I.e. only the length of the planar projection of the edge is used. The motivation for this is that typical vehicles cannot traverse very great slopes and that the height difference in all valid paths are relatively small in comparison to the total length of the edge. It may not be small enough to disregard completely though; moreover, moving vertically is for many vehicles much more time-consuming and ‘costly’ than moving horizontally. In the following sections, we shall see that the cost

of moving uphill or downhill can be effectively (and perhaps better) modeled by using a modifier of the cost function (see 3.1.2) instead of using the true distance (edge lengths).

The point of these ‘edge length approximations’ is that we now only have to check whether we have an axial or a diagonal edge in order to determine its (approximate) length. If furthermore the cells are square, only two lengths are possible, for axial and diagonal edges respectively, see figure 5. In a computer implementation, we can then easily pre-compute two constants for every expression involving the edge length and put them in appropriate tables.

3.1.2 Introducing the cost function

Now that we have constructed a directed graph model for movement through space, it is time for modeling the 'cost' to move through it. With cost can be meant many different things depending on if we want to find the shortest path; the fastest path, the path with least risk of enemy detection, et c. We shall here be mainly interested in a mix of the fastest and the least risky path problem. The modifications to other variants should be fairly obvious.

When using our path-finding algorithm, we need to treat map spaces as something other than binary (clear or blocked). We can move with different speeds though different types of terrain. For example, water, swamps and mountains may be difficult or impossible to pass (i.e. they are obstacles that must be circumnavigated) while grasslands and desert are relatively easy and roads are much faster still. In the absence of any enemies (to avoid) we thus want to find the fastest path possible through this terrain.

To accomplish this, a 'cost' value is assigned to every edge. These values can be seen as a function on the set of all edges and the term 'cost function' shall be used. A path is a ‘connected’ set of edges linking the source and the destination vertices. The cost of a path is the sum of the costs of all the edges in the path. A valid path is a path with finite cost sum.

Let us take the edge cost to be equal to the time it takes our vehicle to traverse the edge. Thus, the basic unit of the cost function is 'time delay'. By construction, it is now clear that the problem of finding the optimum path between two vertices, in the sense of minimizing the path's cost, is equivalent with the problem of finding the fastest path. The shortest path problem is easily seen to be the special case of the fastest path problem where all passable terrain types have the same ‘vehicle speed’.

As shall be seen in section 4, it will be advantageous to enforce the condition that all edge costs be greater than zero. This is also a physically valid assumption for earth bound vehicles – movement always have a cost.

But how can we model all the various parameters that we may want to affect the time it takes to move along an edge, from one vertex to another? And how can we model such restrictions as enemies, obstacles and other things that cannot be explicitly expressed as time delays? A ‘basic edge cost’ based on the terrain type shall be defined and various 'modifiers' shall then be used to take these, and other ‘constraints’ into account. A modifier is here defined as a function that in some way alters the basic edge costs.

With the definition of the cost function and various modifiers, we shall have completed the weighted graph reduction of the problem. Then it remains 'only' to actually find the optimum path in section 4.

3.1.2.1 Terrain

Let us base the cost function on the time it takes to pass through the terrain along the edge. Since the vehicle speed is taken as constant within each terrain type, every edge is divided into only two parts as seen blow-up in figure 6.

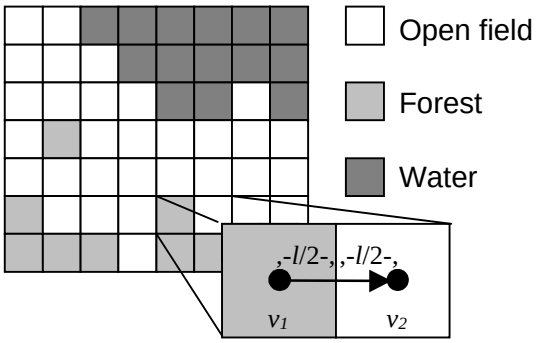


Figure 6 - The vehicle speed is constant within each cell

The basic edge cost is thus very simple: $cost_{edge} = (l_e/2)(1/v_1+1/v_2)$ when v1, v2 $\neq 0$ else ∞. l_e is the edge length as discussed in 3.1.1.3. v_1 and v_2 are the speeds (in length units per time unit) that the vehicle moves through the respective terrain types. A special case is if either v_1 or v_2 is 0 in which case the cost is set to ∞ thus effectively barring any paths leading into unpassable terrain, e.g. water for a (non-amphibious) land vehicle.

3.1.2.2 Roads

Movement along roads is generally considerably faster than through the surrounding terrain. We will use a modifier to replace the costs for edges between cells that are connected by a road by a lower cost. The new cost is calculated analogously to the one for the basic edge cost. It should be safe to assume that the road type is the same in the source and destination so that $ecost_{mod} = l_e/v$ where v is the speed we can move along the particular road type.

How do we determine if a road connects two cells? The vector-based nature of the road data is clearly not directly adaptable to our graph representation the way that the terrain data was. Let us start by 'rasterizing' the road vector data, i.e. translating it into a raster format corresponding to the cell mesh. We will keep one road raster 'pixel' corresponding to each cell. Rasterization can be done in several ways (e.g. see [CLAR90] or [FOLE90]). Basically there are two predominant ways: dominant pixel (thin line) rasterization, or intersected pixel (thick line) rasterization. The difference and the general principles should be clear from figure 7 and 8 respectively. Thick lines represent road vectors in the figures, squares represent raster pixels and the grayed pixels are the rasterized road. The really important thing here is that connectivity is preserved, i.e. there should be no 'gaps' in the rasterized roads.

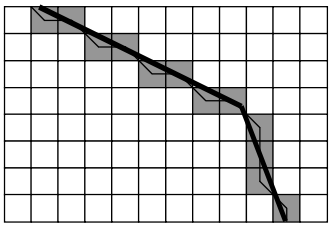


Figure 7 - Dominant pixel raster - thin lines

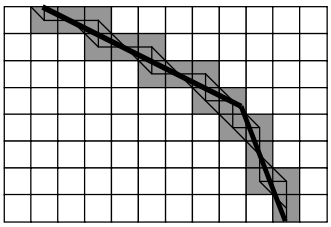


Figure 8 - Intersected pixel raster - thick lines

Given an edge, if the source and the destination (of the edge) both belong to the rasterized road we shall interpret that as that there is a road connecting them and we then 'activate' the road modifier. As seen in the figures, the 'thin line' rasterization has much better 'directional' characteristics so it is clearly the preferred type. With directional characteristics, I mean the property of having the 'road edges' pointing in approximately the same direction as the original road vector. They thus making up a single path when dominant pixel rasterization is used but not for intersected pixel rasterization, This is illustrated by the thin lines representing the 'interpreted' road edges in figure 7 and 8.

3.1.2.3 Slopes

Many vehicles have difficulties moving uphill or downhill. We could consider more or less advanced modifiers for taking this into account. Here is also a chance to add compensation for the vertical component. One version is to have a penalty function that gradually increases with the slope uphill and assuming negative values when going downhill. The details of such a function should be specific to each vehicle type. For most vehicles, it should probably be larger than what is motivated by an exact compensation for the edge length increase (of the 3D edge as compared to the 2D edge we only considered for basic terrain cost). With such a cost function, vehicles would try to avoid moving uphill and will try staying at constant altitude as much as possible.

In the model adopted for the test implementation, a simpler version has been used. Whenever the slope (the height difference between two cells divided by the edge length) is larger than some threshold value, it is considered as unpassable and a modifier is activated that sets the edge cost to ∞. With one exception, if the edge is a road in which case the cost is left unmodified. A motivation for this scheme is that land vehicles can very seldom move at more than very small slopes in most terrain types (most terrain being rather bad fairing). Thus, the vertical component becomes negligible. Roads on the other hand, are constructed so as to let vehicles pass through greater slopes (such as up and down hills) with as little effort as possible. This simpler scheme seems to work well unless possibly if you are working with terrain vehicles moving in very mountainous, road-less regions.

3.1.2.4 Obstacles

Obstacles are here interpreted as untraversable areas. They can appear either in the form of raster data (e.g. lakes) or of vector data (e.g. rivers). A wider interpretation of an obstacle allows not only 'absolute' obstacles but also obstacles of a mere delaying nature.

Instead of checking both for movement costs and for obstacles, we will simply assign ∞ cost to edges passing though an (absolute) obstacle. Later, in the search algorithm (see section 4) it will be possible to add a small check in order to directly discard any ‘proposed’ paths that would contain an edge with ∞ cost.

Raster data based obstacles are handled very easily: If the obstacle is a certain terrain type (e.g. or thick forest), then we just set the vehicle movement speed to zero for that terrain type. If we have other raster based obstacles (i.e. not already encoded in the terrain raster), then we can introduce a special ‘obstructed’ terrain type class with vehicle speed zero, and map all those obstacles to the normal terrain raster class. The mapping can be handled by replacing the original terrain type – since that pixel is ‘obstructed’ we will never be able to reach it anyhow and thus have no need for its true terrain type. Vector based obstacles (e.g. smaller rivers or fences) can be similarly handled by rasterizing them to the terrain raster using the same ‘obstructed’ terrain type. Unlike the rasterization of the road vectors, we now want to use a thick-line rasterization process since the thin-line variant gives rise to ‘gaps’ that would allow paths to pass through the obstacle, see figure 9.

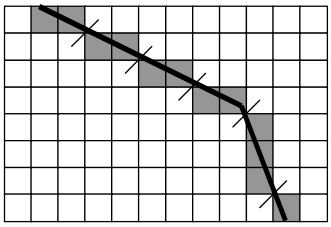


Figure 9 - Thin lines allow edges that pass through the obstacle

Thus, no cost function modifier is needed to handle obstacles.

3.1.2.5 Enemies

We can use yet another modifier to avoid trampling into enemy territory, i.e. regions where we can be detected and/or apprehended by enemies. A fixed 'penalty' cost is added to all edges whose destination is in enemy territory. Thus if we want to entirely avoid being detected by enemies we should add ∞. Exactly what criterion is used to 'activate' the modifier depends on the modeling of the enemy. This simple ‘avoidance’ modifier is enough for our purposes. A possibly more correct model would instead add a penalty both when moving in and out of cells (like the basic terrain cost). It could al be proportional to the time it takes to traverse the edge (i.e. proportional to the basic terrain cost plus the road modifier).

The temporal domain is ignored here, i.e. enemies are considered as static during the path determination. For a discussion of how the method could be extended to handle dynamic scenarios, see 6.2.1.

In the test implementation, visibility has here been used as the criterion for being detected by an enemy. To determine if the vehicle observed by an enemy, a simple form of ray tracing (see [CLAR90] pp.228) is used to trace the 'line of sight' between observers (the enemies) and the object (the vehicle). Draw a ray from the vehicle to each observer and check if the rays are intersected by terrain. For this, we use the height raster data to determine the ground height. To the ground height is added

specific 'visible heights' of different terrain types (e.g. water = 0m, light forest = 6m, et c.).

3.1.2.6 Error bounds

What are the errors inherent in our problem representation? Let us look at various possible error sources:

Rasterization error: What is the error of representing reality in a raster format? This is not a trivial question to answer. For example, consider the case of a very long and thin but very easily passable ‘squiggly’ path (i.e. in reality) as shown in figure 10.

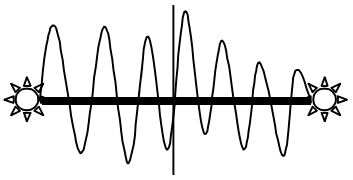


Figure 10 - Squiggly path between two vertices

Let all other space contain unpassable terrain and use a rasterization grid so that the path is contained in two neighboring cells. If the rasterization process were of an averaging nature, the thin squiggly path would disappear entirely out of our representation of reality and we might incorrectly be led to believe that there is no possible path. If, on the other hand, we use a rasterization process that selects the infinum value of the local terrain cost of the area represented by each pixel. Then any pixel containing the squiggly line would get terrain costs equal to that of moving along a corresponding length of the squiggly path. The difference between the computed best path cost (a straight line between the cell centers) and the actual one (the squiggly line) is thus proportional to the difference in distance between them. However, this difference can be made ever larger by making the thin path ever thinner, longer and more squiggly. The two cases presented here are of course very unlikely extremes. In the following, we will simply assume that the raster pixels are small enough to contain all, for our purposes, essential features of reality. Furthermore, it is assumed that they are classified in such a way that the average error in the answer becomes negligible. Moreover, it is assumed that, in a statistical sense, the expectancy of sum of the (signed) average errors of a large enough number of random paths is zero. Henceforth the ‘reality’ as far as our method is concerned shall be considered to consist of the square regions made up by the raster pixels and rasterization error are ignored.

Data errors: The various input data may have inherent errors. This is really something that should be treated outside our model and we shall consider all our input data to be exact. Nevertheless, we may note that the output path is a finite sum of edge costs. If then the edge cost function is a piecewise linear function of the input data (i.e. it *is* allowed to exhibit bounded ‘jumps’) then that must also be true for the error.

Cell discretization error: What is the error from restricting the path only to go between vertices? Let us consider the special case of uniform movement cost. The source and end points must each fall into a cell. The movement ‘positions’ are restricted to the vertices at the cell centers, see figure 11. Therefore, we have an absolute upper bound for the error in the supremum of the distances between all

possible placements within a cell and the center. This, naturally, is half the diagonal length. Therefore, a bound for the cell discretization error is one cell diagonal length. For the more complex case with variable movement cost, we can multiply with the highest movement cost of the edges along the path for an *a-posteriori* upper bound. We then need to know the optimum path to find the highest cost along it. We could use the highest cost of the entire graph instead for an *a-priori* estimate, but if we have obstacles, then this is ∞ which doesn't make for a very useful estimate.

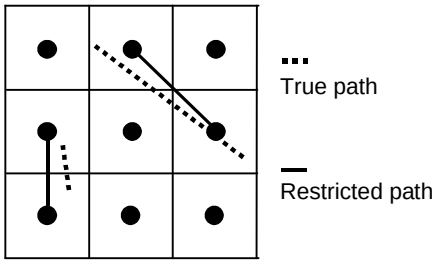


Figure 11 – Cell discretization error bounded by diagonal length

Edge restriction error: What is the error introduced when we restricted the movement between cells to neighboring cells only? Again considering the uniform movement cost case the error incurred is illustrated by figure 12 (note that the restricted path can be permuted without changing the cost, see further 4.2.4).

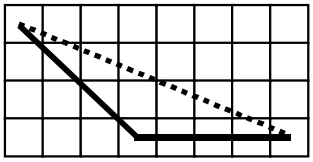


Figure 12 - Edge restriction error in uniform terrain

The worst case is obviously when the total length of the diagonal edges equal the total length of the axial edges. As the quotient between the length of a diagonal and an axial edge is $\sqrt{2}$, i.e. an irrational number, this worst case can only can occur in 'the limit' of an infinitely long path. If the true path have a length l_t, and our restricted path have length l_r, we get from some simple trigonometry:

$$\frac{l_r}{l_t} = \frac{1+1}{\sqrt{(\frac{1}{\sqrt{2}})^2 + (1+\frac{1}{\sqrt{2}})^2}} = \frac{2}{\sqrt{2+\sqrt{2}}} \approx 1.082392200$$

In other words, we have an upper bound of about 8% on the error due to this effect in a uniform graph. For non-uniform graphs, we can get an *a-posteriori* error bound by multiplying by the total cost of the computed path.

Cost function errors: The modeling of reality is based upon attempts to quantify various properties of reality. An incorrect model may yield substantial errors. Trying to absolutely measure these errors is very difficult due to the very intricate nature of reality. No such attempt is made here since it is so highly dependent on so many 'specifics'. Like detailed specifications of the vehicles, the terrain type classification, et c. et c. Perhaps the only really satisfactory way of measuring these errors is to actually perform field tests and compare with the computed results.

The conclusion that can be drawn from this discussion is that it is quite difficult to speak of any absolute error bound (especially an *a-priory* bound). On the other hand we have every reason to believe that the approximations made have the property of giving reasonably small ‘average’ errors. Since the approximations made are all most all of an ‘averaging’ nature, the stability should be good as well.

3.2 Efficient representation

Let us consider a traditional, ‘generic’ graph representation. Every cell has eight directed edges leading out from it to the eight neighboring cells. For every edge we would need at least 4 Bytes to explicitly store the edge cost and at least 2 Bytes to store an index to the destination cell. This makes for a minimum of 48 Bytes per cell. We want to be able to efficiently run our algorithm with typical instances of up to the order of one million cells. That would require at least 48Mbytes just to store the graph. To that comes all the memory needed for the actual path finding algorithm, not to mention that in the intended application for this work, it will only be part of a much larger program. Clearly a traditional generic graph representation simply requires about a power of ten too much memory to be feasible for our algorithm running on standard equipped personal computer today. Only constructing such a (relatively) huge graph would take significant computational time. Of course, in a handful of years, the memory and computer technology available in what is then considered a standard PC is very likely to have progressed to the point where this is no longer an issue. However, we do not have the luxury to wait until that happens and when that day comes, we will instead be able to solve even larger problems if we can only find a way to make this one practical today! Thus, I will here present a way to implicitly represent the very special graph needed for our problem in just 3 Bytes per cell.

The sample code snippets presented here use C++ syntax but should be understandable for most people with basic knowledge of programming languages.

3.2.1 Implicit graph representation

First, we would like to be able to treat all cells equally. Therefore, something has to be done about the 'missing' edges at the borders. This is conveniently handled by adding the missing edges leading to ‘nothing’ and setting their edge costs to ∞. A special modifier can be introduced in the cost function to take care of this.

All cells/vertices now have exactly the same structure, provided that care is taken never to 'walk' an edge with infinite edge cost. We know what edges there are without having to explicitly encode it for every cell (see figure 5). The cells in themselves have no real interest; it is only the edge costs that we care about when determining the total cost of a path. However, we saw in the previous sections that all the edge cost modifiers are functions of properties of the edge source and destination cells. Let us call those properties for cell 'attributes', store them in a `CellAttr` structure and define an array where we store all the cell attributes, one attribute for each cells vertex.

Every cell is referenced by a two-dimensional index, a `CellRef`, into this array. All that needed to be known in order to walk an edge from any one cell to another is how this (two-dimensional) index should be changed. Let us number the edge

directions clockwise from 0 to 7 (0 being North, 1 Northeast, 2 East, 3 Southeast, et c). Then it is easy to calculate the index change and store it in an look-up table, `crWalkEdgeDelta[8]`. To walk an edge, with direction `ed`, all we then have to do is 'add' `crWalkEdgeDelta[ed]` to the source cell's `CellRef`.

3.2.2 Computing the cost function

Rather than store all the edge costs we will express it as a function of the edge source and destination cell attributes. We use this function to compute any edge cost as needed. In section 4 we shall se that we actually only will need a certain edge once so no duplicate work will be performed as compared to pre-computing all the edge costs. Better still, a very large percentage of the edge costs will usually never have to be calculated at all. If all attributes for a vertex can be stored in roughly the same amount of memory as an edge cost, then much memory can be saved by storing attributes instead of explicit edge costs. This since the number of vertices is only one eighth of the number of edges.

A 32bit unsigned integer should be enough to store any path and edge cost in a fixed-point format. A special value, *infinity*, is used to represent ∞. It should be set at half maximum value of the data type so that it can safely be added to any cost value without any risk of overflow and so that we can clamp it to *infinity* again. This corresponds to the algebraic operation $\infty+x = \infty \ \forall x \notin \{\pm\infty\}$.

3.2.2.1 Terrain

The basic terrain cost is easily implement using a pre-calculated look-up table as: `dwCost = TerrainLut[`$ed \ \& \ 1$`][`t_1`] + TerrainLut[`$ed \ \& \ 1$`][`t_2`]`. `TerrainLut[i & 1][j]` contains the cost for traversing half an edge of direction `i` in terrain of type `j`. Since the edge directions were numbered clockwise from 0 to 7, we have `ed & 1` equal 0 for axial edges and 1 for diagonal edges. In the test implementation there is 16 terrain classes so we need 4bits to store them as an attribute, `cTerrainType`, for each vertex.

3.2.2.2 Roads

It is desirable to be able to compute the rasterization for all roads once only and that one single road raster can be used to represent all the roads in the map. We will store the road pixels as an attribute, `cRoadType`, for each vertex and let it contain either the road 'class' or a special ***noroad*** value. In the test implementation, there are only three road classes: small, medium or large, so only 2bits are needed to store this information. Given an edge, if the source and the destination both have road raster pixels differing from the ***noroad*** value, then it is assumed that there is a road connecting them. This could be a problem if there are roads that are so close together that their rasterized representation 'mingles'. With 25m pixels/cells this does not seem to be a problem in practice. Even should such mix-ups occur (e.g. in urban traffic knots), it is highly likely that there are actually interconnections between the roads in question in reality as well, so that the affect upon the optimum path solution should be negligible. Nevertheless, to decrease the likely-hood of such mix-ups, a 'thin line' rasterization is again preferred (see figure 7). The rasterization

algorithm of choice is the classic Brezenham's line drawing algorithm (see [FOLE90] for details). It is a nice algorithm in that is about as fast as you get, uses only integer operations, and is of the dominant pixel (thin line) kind.

3.2.2.3 Slopes

As we have a rather simple modifier for handling slopes, that also involves a test for road presence, it is easy to implement it by adding it as a small `else if` clause to the road modifier.

3.2.2.4 Obstacles

All the obstacles that occur in the test implementation are already incorporated in the raster data (e.g. the water terrain class represents lakes and larger rivers) so no special modifier is necessary to handle them.

3.2.2.5 Enemies

We want to trace the line of sight between the vehicle's and each enemies vertices in order to decide enemy detection. In order to avoid overt 'aliasing' ('jaggedness') effects, will need a sub-pixel accurate ray-tracing algorithm. Thus, a 3D version of the Brezenham's line drawing algorithm used to rasterize road vectors will not do. A fixed point (DDA) line drawing algorithm has instead been used. It is quite simple; the position in 3D space is stored as a fixed-point number and uses another fixed point 3D 'incrementor' to walk in small steps along the line. At each step, a check is performed to se if we are above or below the visible terrain level at that point. The terrain level is calculated using bilinear interpolation of the four neighboring height and terrain raster pixels as illustrated in figure 13. For best accuracy, the step length should be chosen the same as, or smaller than the pixel size.

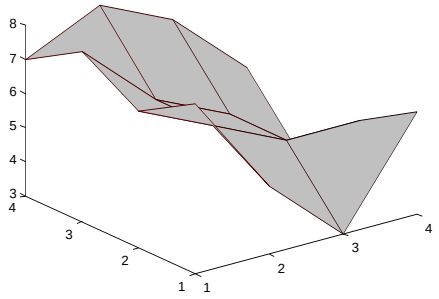


Figure 13 – Example of bilinear interpolation surface on a 4x4-vertex grid

This calculation can be one of the heaviest computational burdens in this algorithm. Therefore, we don't want to do this if it isn't absolutely necessary. Thus we delay the calculation as long as necessary and use a binary flag attribute, `bHaveVisCalc`, to check if we have previously calculated it for the particular vertex. If not, then we calculate it and store the result in another binary flag attribute, `bVisible`. This 'lazy' evaluation can been seen as form of memoization – see [CORM90] p312.

The ground height is stored as yet another attribute, `wHeight`. A 16bit value should be sufficient for our home planet (Mount Everest rise to 8848 m above the sea

level). To the ground height is added specific ‘visual heights’ depending on the terrain type (implemented as a look up table of course). The terrain type was already stored in the `cTerrainType` attribute.

3.2.3 Memory space

Scanning back though the sections of 3.2.2 and noting all the attributes used, we come up with the following set of vertex attributes for the test implementation:

```
typedef struct {
    BYTE  cTerrainType : 4;  // Terrain class, 4bit
    BYTE  cRoadType    : 2;  // Road class, incl. noroad, 2bit
    BYTE  bVisible     : 1;  // Are we visible?, 1bit
    BYTE  bHaveVisCalc : 1;  // Have we calculated bVisible?, 1bit
    SWORD wHeight;           // Height over sea level, 16bit
} CellAttr;
```

Altogether, we have three bytes per cell. If enemy and slope detection is not necessary, we could skip the height attribute and make do with one single byte per vertex! If we should need more terrain and/or road classes, an extra Byte might be needed.

3.2.4 Sample implementation

The following sample code brings together the various bits and pieces into an edge cost function for the problem instance defined by the test implementation:

```
DWORD SearchGraph::EdgeCost(
// Calculate the cost function
    CellRef &crDst,     // Out: destination cell for edge
    CellRef crSrc,      // In:  source cell
    EdgeDir ed          // In:  edge direction
)
{
    // Walk to the destination cell index
    crDst = crSrc + crWalkEdgeDelta[ed];    // 2d index + operator

    // Border cell modifier, return infinity if outside out graph
    if (IsCellRefValid(crDst)) return infinity;

    CellAttr &caSrc = AttributesFor(crSrc);
    CellAttr &caDst = AttributesFor(crDst);
    DWORD    dwCost;

    // Apply road modifier?
    if ((caSrc.cRoadType != noroad) && (caDst.cRoadType != noroad))
        dwCost = dwRoadEdgeCost[ed&1][caSrc.cRoadType];

    // Or Apply slope modifier?
    else if (abs(caSrc.wHeight-caDst.wHeight) > dwMaxHeightDiff[ed&1])
        return infinity;
```

```
    // Or Compute the basic terrain cost
    else dwCost = dwTerrainHalfEdgeCost[ed&1][caSrc.cTerrainType] +
                  dwTerrainHalfEdgeCost[ed&1][caDst.cTerrainType];

    // Apply enemy modifier
    if (!caDst.bHaveVisCalc) { // Lazy evaluation of the visibility
        caDst.bVisible = IsObserved(crDst); // Uses DDA trace alg.
        caDst.bHaveVisCalc = true;
    }
    if (caDst.bVisible) {
        dwCost += dwVisibilityCost;
        if (dwCost > MAXCOST) dwCost = MAXCOST;
    }

    // Done!
    return dwCost;
}
```

As it is heavily broken up into `if else` clauses the actual code traversed in a call to `EdgeCost()`, is quite short. The real performance bottleneck is in the `IsObserved()` call, i.e. determining whether an enemy can detect us in the destination cell or not. Thus, if enemy detection is not necessary we can save much in computational cost.

4 Search algorithms

We have reduced our problem into the more common problem of finding the minimum cost path in a weighted, directed graph. It is sometimes referred to as 'the shortest path' problem instead of as 'minimum cost', see [CORM90] and [CHER93]. Established methods exist for such problems, and a few approaches are briefly presented in the overview 4.1. The problem is that most methods either have very bad performance for larger or more complex graphs, or only produce approximate solutions with more or less uncertaincy. A practical algorithm for our problem is then examined in closer detail.

4.1 Overview of methods

A number of techniques exist for solving graph related minimization problems. Here, the applicability to the present problem is investigated for some of the most common approaches.

4.1.1 Exhaustive search

The naïve approach is to perform an exhaustive search, enumerating all possible paths and selecting the path(s) with the lowest cost. Any method used to construct these paths can be illustrated by a decision tree. We could for example perform the exhaustive search using a 'depth-first search' algorithm, see [CORM90] pp477..483 for details. Briefly: start by 'tracing' a decision path down to a leaf in the decision tree producing a first path. Then we backtracking to the first decision point higher up in the tree where we have not yet examined all the possible 'decisions' and trace down a new branch of the tree for a new path. To limit the number of paths to examine, we can directly discard any paths that would have 'loops' in them. Continue that way until the entire tree has been traversed. In other words, start at the source vertex, 'decide' upon one of the edges leading to another vertex (that is not already in the path), walk it and repeat until the destination vertex is reached.

At each decision point, we have a choice of eight directions. Call the number of vertices for N. Since any cell can be visited at most once (otherwise there would be a loop), we can state that every possible path has at most N-1 edges, i.e. the search tree has depth $O(N)$. With the number of branches increasing with (at most) an eight fold at each level we get at most 8^{N-1} possible paths, i.e. we have $O(8^N)$ paths to examine in the worst case. Thus, the naïve approach takes exponential time to find a solution. It is clearly not a very good. The algorithm is very dumb in that it does not distinguish between the vast majority of the possible paths that are not likely to be optimal and those very few that are likely candidates.

4.1.2 Relaxation methods

An alternative to the 'depth-first' algorithm for doing an exhaustive search is the so called 'breadth first' search algorithm, see [CORM90], pp469..477. It derives its name from the way it expands a 'search front' from a start point (in the decision tree). Nothing much is directly gained by using this alternative as-is instead of a depth-first-search. However, there exists a major class of graph search algorithms

based on a technique called ‘relaxation’, see [CORM90], that uses ideas very similar to breadth first search but with some crucial improvements. Most common of these that are also applicable to the current problem, are the textbook classics: Dijkstra’s and Bellman-Ford’s algorithms. They both determine all the optimum paths from a ‘single source’ to all the other vertices. Dijkstra’s algorithm is faster but requires that there are no negative edge costs; which we have made sure of in section three. The A* algorithm is an effective heuristic improvement of Dijkstra’s algorithm that yields a better average performance when you only need the optimum path between two vertices (and not from one vertex to all other). Because it provides good, predictable performance without compromising optimality, it is the algorithm chosen for our test implementation; see section 4.2.

4.1.3 Linear programming

Linear programming is often used to solve various network flow and resource allocation problems. It is possible to consider our graph problem as a linear programming problem by reducing it to, for example, a minimal cost flow problem or an equivalent assignment problem. For references see [LUEN84] and [CSEP95]. In practice, the problem becomes very ‘huge’ formulated in this way and the normal optimization methods used for such problems do not seem to be very efficient for so large problems. For example, the so-called simplex method is the classic text book method for solving linear programming problems. However, establishing convergence for the simplex method is non-trivial and the time it will take can not be established *a priori*. In addition, and as expected, the iteration matrix (see [LUEN84] becomes, although quite sparse, tremendously huge (square of the number of cells). For these reasons this approach has not been further evaluated here although there may be better, or at least more memory effective, though more complicated solvers available than what has been mentioned here.

4.1.4 Simulated annealing

This general minimization method can be very useful for approximately solving many 'hard' minimization problems by using a pseudo-random search. As an example the NPC problem known as 'the travelling salesman' can be effectively 'solved' by using simulated annealing, see [PRES92]. The method finds its inspiration in the way nature herself achieves minimas. The name comes from the process where hot metal slowly is allowed to cool and freeze into the minimum energy crystal lattice configuration.

The function, *f*, to be minimized is called the objective function. In our case, *f* is the path cost. The method starts with an initial 'guessed' solution. As an example, we could use the 'straight line' path between source and destination (it need not be a valid path). It is then iteratively improved by applying random perturbations of the path. A perturbed path is accepted as a new initial solution with probability: $p = exp(-\frac{\delta f}{T})$. Thus, there is a probability of accepting some perturbations that actually increases the objective function. This is necessary in order to avoid being 'caught' by local minimas. The system parameter *T*, often called the 'temperature', is used to control this probability and is lowered until an acceptable solution has been found. The difficult part is to construct a good perturbation function. It should have

the property of both introducing small, random, 'feasible' changes and at the same time allow all possible solutions to be reached. Often *T* is also used to directly control the 'scale' of the perturbations. Exactly how *T* is controlled and how to determine when to stop can be difficult. Some kind of heuristic or adaptive control logic is often employed. As a simple example we could use a fixed T until, say 100 iterations in a row, failed to produce a better solution. Then T is halved and we continue in the same fashion. We stop when the scale of the perturbations (as a function of T) is small enough to lie below the desired solution accuracy. This way, large-scale optimization is first performed (large T giving large perturbations) and then optimization is performed on continuously lower scales until we are finally satisfied with the 'local' optimization as well.

The iterative nature is both a big strength and a big weakness for this method. On the plus side, if we must find a path in a certain time, we can just stop iterating when we want to. On the minus side, we have no knowledge at all about how long it will take to reach a certain degree of accuracy, if it ever will (depending on how good the perturbation function is).

For further information on simulated annealing, see [PRES92] or chapter 4.1 of [CSEP95]

4.2 Algorithm

The A* algorithm add a heuristic to improve upon the classic Dijkstra's algorithm that will effectively let us find a least-cost optimal path in a directed graph with non-negative edge weights.

4.2.1 Dijkstra's algorithm

This algorithm was first introduced in [DIJK59]. A more modern interpretation and proof of its correctness can be found in [CORM90]. Here shall be given a short description of the algorithm together with some notes on its properties without presenting any further proofs or theory.

Input is a graph $G = (V, E)$ and the source vertex $s \in V$. Here V is the set of vertices and E the set of edges in G. Further we have an edge weight function $w(u,v) \geq 0$ $\forall (u,v) \in E$. The algorithm divides the vertices, V, into two *disjoint* sets, $V = R \cup Q$. The set, R, contains those vertices whose final shortest path weights have been determined and the set, $Q = V\text{-}R$. When a vertex is moved from Q to R, we say that it is 'retired'. Some descriptions of the algorithm calls Q the 'open' set and R the 'closed' set. For every vertex, v, we store a currently minimum shortest-path estimate value, $g[v]$. Since we are not really interested in finding the value of the least cost path, but rather the actual set of vertices that make up it, we will also store, for every vertex a predecessor edge, $ed[v]$, that is the edge from a neighbor to the vertex in the currently best path to the vertex in question. This can then be used to reconstruct the path by backtracking from the destination, see 4.2.3. In case there are more than one optimal path, only the most recently discovered one would be remembered with this scheme. In section 4.2.5 is described a way modify it to 'select' a more visually pleasing path in such cases of ambiguity.

The algorithm, in pseudo-code, proceeds as follows:

```
Dijkstra(G,w,s)
    R = Q = {}
    for all v∈V do
    g[v] = ∞
    Q.Insert(v)
    while (Q ≠ {}) do
    u = Q.ExtractMin()
    R = R ∪ {u}
    for v∈Neighbors{u} do
        if (g[v] > g[u] + w(u,v)) then
            Q.DecreaseKey(v, g[u] + w(u,v))
            ed[v] = (u,v)
```

The set Q should be implemented as a priority queue with the following operations: *Q.Insert*(*u*) – inserts a vertex into the queue, *Q.ExtractMin*() – removes the element $v \in Q$ with smallest *g*[*v*], *Q.DecreaseKey*(*v, a*) – decrease *g*[*v*] to *a,* and updates the queue.

The algorithm works by repeatedly extracting the vertex $v \in Q$ with the minimum shortest path estimate, *g*[*v*]. Since all the edge-costs are positive, we can't find any shorter path to it by extending a path from any of the other vertices. Thus we have determined the shortest path to *v* and can retire it to *R*. We then check if any of the extensions of this optimal path to its neighbors would be a shorter path than the current shortest path estimate for the respective neighbor. If it is, then we update (or 'relax') the shortest path estimate and the predecessor edge for the neighbor(s) accordingly. Because we repeatedly select the least costly vertex and 'relaxes' it, the algorithm belongs to the class of 'greedy' relaxation methods.

In a practical implementation, we can divide *Q* into two disjoint subsets, *U* and *PQ* where, to start with, *U* contains all vertices except *s* and *PQ* are used instead of *Q* above. Vertices are then moved from *U* into *PQ* when they are first visited. We can use some flag to denote if a vertex has been previously visited or not (i.e. if it is in *U*) and thus do not have to store *U* explicitly. Always keeping the priority queue *PQ* as small as possible can be very beneficial for performance; see 4.2.5.

4.2.2 The A* heuristic improvement

As we are only interested in the shortest path between two vertices, it seems like a waste to compute all the paths from one vertex to all the other vertices as Dijkstra's algorithm does. The most obvious improvement is to stop directly at the moment when the destination vertex, *t*, is retired. The knowledge of the position of the destination is not used at all during the search so it is still rather 'dumb' in this respect. The A* algorithm, introduced in [NILS82], takes care of this by adding a heuristic in order to bias the search toward the destination and thus hopefully finish faster. The only difference between the algorithms lies in which vertex is retired in each iteration step, i.e. how they are ordered in the priority queue. The basic idea is

that instead of using *g*[*v*] (the actual cost from the source to the current vertex) to order the queue as in Dijkstra's algorithm, a new entity, *f*[*v*] = *g*[*v*] + *h*[*v*] is used instead, where *h*[*v*] is the heuristic. It can be proved that if *h*[*v*] is an underestimate of (or at most equal to) the actual least cost path for moving from *v* to *t,* then A* will produce an optimal solution (just as Dijkstra's did). The rigorous proof is a little out of scope for this text; please refer to [NILS82]. As only the ordering of the priority queue is changed, the worst-case characteristics are the same as for Dijkstra but we can expect the average performance to be better much better for 'real world' graphs if we can provide a good heuristic, *h*[*v*].

Generally the '*distance from u to v*' time '*smallest terrain cost*' is the best estimate we can do for *h*[*v*] when an underestimate must be guaranteed. What is the optimal 'distance' estimate depends on the graph structure. The true Euclidean distance should serve, but it would often be lower than the actual edge lengths along the shortest path (see figure 12). Instead, since the graph structure is uniform we can actually calculate the exact, 'optimal' distance. If we'd only had orthogonal edges, it would be the so-called 'Manhattan' distance: $|\Delta x| + |\Delta y|$. In our case, with diagonal edges as well, it is easily seen to be '*diagonal edge length*'*min($|\Delta x|$,$|\Delta y|$) + '*axial edge length*'*$||\Delta x|-|\Delta y||$. An addition bonus compared to using the Euclidean distance is that no costly square root calculation is required.

We can delay computing *h*[*v*] for each vertex until it is actually needed, i.e. when it is first entered into the search queue. Thus the same flag as is used for keeping track of if a vertex is in *U* (see 4.2.1) will also tell if *h*[*v*] needs to be initialized. Since many vertices may never be visited at all, this may save us a bit of work.

Note that if the edge costs of unobstructed terrain varies widely (e.g. roads are very cheap while plain terrain types are costly), then adding a heuristic based on the '*smallest terrain cost*' generally provides a large underestimate. In those cases A* is only a minor improvement and will behave more like Dijkstra's algorithm. An alternative would be to use a '*typical terrain cost*' instead of the smallest cost. This would violate the assertion that the heuristic be an underestimate of the real cost and thus compromise the optimality guarantee. In some applications, this might be a reasonable trade-off for faster solutions. In fact, we could use *f*[*v*] = *g*[*v*] + *c***h*[*v*] where *c* is a scaling 'control' constant. For *c*=0 we have Dijkstra's algorithm, for *c*=1 we have A* for *c*>1 we have a modified 'approximate A*'. The larger the value of *c* the faster (on the average) but more 'head on target' the solution becomes. An good interactive Java tutorial that illustrates this difference can be found in [WAVE96]. With equal cost terrain, a heuristic *h*=0 gives for our special graph a search front that expands in a hexagonal shape, while A* with a true Euclidean heuristic gives a more pear-like shape.

An example of a quite different application of A* can be found in [SONK93].

4.2.3 Reconstructing the path

The result of a typical search is illustrated in figure 14. The circle is the path's destination. Gray cells never needed to be visited during the algorithm execution. Dark gray is an obstacle. The arrows show the edges of 'currently best known' paths to all examined cells, i.e. the *ed*[*v*] values for the vertices at the arrow heads. For the retired cells (not marked) this is also the optimum path to those cells. The thicker dark arrows show the least cost path to the destination.

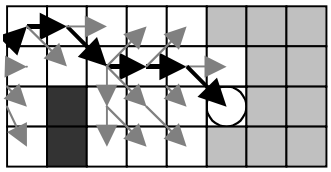
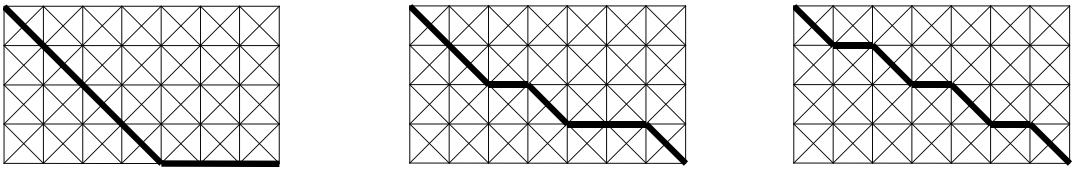


Figure 14 - A small A search tree*

If we were only interested in finding the cost of the minimal path then we could have skipped saving *ed*[*v*] in the algorithm (thus saving a Byte for each node). Now we can easily use it for backtracking the optimum path from the destination vertex. We start in the destination cell and then walk in the opposite direction to the edge direction stored in the current cell until we reach the source cell. It is then a simple thing to reverse the sequence of the cells visited during the backtracking to produce a 'forward' path from source to destination.

4.2.4 Avoiding directional biasing

A straightforward implementation of A* will yield one, of perhaps many optimal paths. However, this may not always be the one most visually pleasing path. It will give biasing (preference) to certain directions through the order that neighbors are examined and the order that equal cost vertices are removed from the priority queue. This may produce visually less than pleasing paths in areas of uniform edge costs. See figure 15. Exactly what directions are preferred and in what order are of course highly implementation dependent.



Biased path *Randomized path* *Visually most pleasing path*

Figure 15 - Three equal cost paths in a graph with uniform edge costs

The most intuitive way to avoid biasing seemed to me to be to regularly, or even randomly, permutate the order in which path extensions to neighboring cells are examined. However, in practical tests, this produced less than satisfactory improvements. A probable explanation is that this scheme only introduced a new order of preference that is relatively static over the graph. As an alternative, introducing a random element was tried. With 0.5 probability, accept new paths that would not improve the current best path but neither worsen it (i.e. new paths with cost equal to the current best cost). This proved to be a satisfactory (if not perfect) as seen in the example in figure 15. The number 0.5 was chosen because the probability of having two equally good extensions of a given path in a real-world map is (heuristically if you will) so much larger than that of having three or more. Thus when we come to a point where we have to decide between two equally good paths we can use coin tossing disregarding the possibility of a third equal cost path and still expect an unbiased result.

4.2.5 Memory space and time complexity

When implementing the A* algorithm, the most important factor (for both memory usage and computation time) is how the priority queue is implemented. Let us denote the number of cells by *N*. During each pass of the main loop, *PQ.ExtractMin* is called exactly once. When a node has been extracted, it will not be entered again so the loop runs at the most N times. The only operations, except PQ.ExtractMin, inside the loop that does not run in constant time is found inside the neighborhood scan. The neighbor scan loop itself runs at most eight times, i.e. it do not increase the time complexity other than a 'constant' amount of the operations inside it. Because of the special nature of our graph representation, we can easily ‘find’ the neighbors in constant time. The non-constant time operations possible inside the neighbor scan loop are *PQ:Insert* and *PQ.DecreaseKey*. So the entire algorithm takes *O(N)*min{8*PQ.Insert, 8*PQ.DecreaseKey, PQ.ExtractMin}* time.

The simplest method that is often used for small problems is to use a sorted linked list as a priority queue. However, while the *PQ.ExtractMin* operation for that takes *O(1)* time, the *PQ.Insert* and *PQ.DecreaseKey* operations takes *O(N)* time. Thus, the whole algorithm would go in $O(N^2)$ time. The, for this purpose, best-known priority queue is the so-called Fibonacci heap. As the theory and implementation of these heaps are somewhat complex, it is here left to text books like [CORM90] pp420..439. A sample implementation can be found in [BOYE97]. The Fibonacci heap has operations that take different time depending on internal state variables. As an example, inserts may go very fast until, after a few inserts, it decides it is time to do some 'house cleaning' which can take considerably more time. Thus 'amortized' costs are used instead where the time required to perform a sequence of operations is averaged over all the operations performed during the algorithm; see [CORM90] ch.18 for further details. Suffice to note here, that the amortized costs for a Fibonacci heap are: *O(1)* for *PQ.Insert, O(1)* for *PQ.DecreaseKey* and *O(log(N))* for *PQ.ExtractMin*. Thus the best known implementation of A* (as well as Dijkstra's algorithm) runs in $O(N \cdot log(N))$ time. The constants hidden inside the *O* notation is quite large for Fibonacci heaps though, so they are not that useful for very small heaps.

What about space complexity? To start with, every vertex, *v*, needs a value for *g*[*v*] and *h*[*v*], which thus requires *O*(*N*) space. Furthermore, every element in the priority queue takes up some space. A trivial upper bound of the maximum number of items in the queue is *N*, so the total space complexity has an upper bound of *O(N)*. In practice, the maximum queue size tends to be considerably less than N, see for example figure 23 in section 5.4 (where vertices in the queue have been marked with white). On the other hand, in a Fibonacci heap implementation, each element stored in the queue eats up quite a lot (although linear) amount of space.

For our application we may be more interested in the time complexity as the function of the distance, *d*, between the source and the destination. Some models for the dependence of *N* upon *d* was discussed in 3.1.1.1. In most cases *N* grows quadratic with *d* so that the algorithm runs in $O(d^2 \cdot log(d))$ time. Thus on every 'real' computer there is bound to be some distance, d_{max}, after which we soon run out of memory space. In section 4.3 is presented a method to increase d_{max}.

4.2.6 Sample implementation

An example of how the A* algorithm could be implemented in C++ is given here. As the weight function *w*(*u*,*v*), the `EdgeCost` function defined in 3.2.2 is used. The A* heuristic is implemented by:

```
DWORD SearchGraph::CostNorm(CellRef crSrc, CellRef crDst)
// Returns an underestimate of the path cost from crSrc to crDst
// dwMinEdgeCost holds the infinum of all possible edge costs
{
    int iDX = abs(crSrc.X()-crDst.X());
    int iDY = abs(crSrc.Y()-crDst.Y());
    return  dwMinEdgeCost[edDiagMod]*min(iDX,iDY) +
            dwMinEdgeCost[edAxisMod]*abs(iDX-iDY);
}
```

The `FibHeap` class is a standard Fibonacci heap of `FibHeapNode`'s. Each such 'node' stores a vertex and its associated *g* and *h* values. The Fibonacci heap is constructed after increasing order of the value of *f* = *g* + *h*. In addition to the (quite large) data structures required for maintaining the Fibonacci heap, a `FibHeapNode` also contains the following data fields:

cr	Store a reference to the cell represented by this node
g	The cost of the currently best path to cr
h	The min. cost estimate of the path from cr to the destination

The `Nodes[]` array is a look-up table used to find either a pointer to the `FibHeapNode` for vertices that is currently in *PQ*, or the special values ***retired*** or ***notvisited***. When set to ***notvisited*** it means that the vertex is in *U*, when set to ***retired*** it is in *R* (see 4.2.1), else it is in *PQ*. Memory for a `FibHeapNode` is allocated first when it is entered into *PQ* and immediately released when it is retired. Every `FibHeapNode` requires much more memory than an entry in `Nodes[]` (34 versus 4 Bytes in our implementation). The number of vertices in *PQ* at any time is on the average much smaller then the total number of vertices (our tests display an almost logarithmic behavior, though the worst scenario is linear, it is very unlikely to occur). Thus it may save a lot of memory doing it this way instead of just allocating a `FibHeapNode` for every vertex in one large block. To keep down the memory management overhead, the last retired `FibHeapNode` from *PQ* is cached for reuse the next time a new node needs to be inserted for the first time, thus saving many `new` and `delete` operations.

```
DWORD SearchGraph::AStarPath(
    EdgeDir *&pedEdges,  // Out: A list of edge directions
    int &iNumEdges,      // Out: The number of edges in pedEdges
    CellRef crBeg,       // In:  The source cell
    CellRef crEnd        // In:  The destination cell
)
{
    PFibHeapNode *Nodes = new PFibHeapNode[dwTotalCells];
    EdgeDir      *ed = new EdgeDir[dwTotalCells];
    PFibHeapNode pU, pV, pReuse = NULL;
```

```
FibHeap      PQ;
CellRef      crU, crV;
DWORD        dwCost;
int          iV, i;

// Set all node ptrs to notvisited
for (i = 0; i < dwTotalCells; i++)
    Nodes[i] = notvisited;

// Insert the original node
Nodes[IndexOf(crBeg)] = pU = new FibHeapNode;
pU->cr = crBeg;
pU->g = 0;
pU->h = CostNorm(crBeg, crEnd);
PQ.Insert(pU);

// While there are still nodes to visit, visit them!

while ((pU = PQ.ExtractMin()) != NULL) {

    // If we're at the destination, then exit the while loop
    crU = pU->cr;

    if (crU == crEnd) {
        dwCost = pU->g;                  // Final path cost!
        delete pU;
        break;
    }

    // Examine possible paths from U to its neighbors

    for (EdgeDir edp = 0; edp < NUMEDGEDIRECTIONS; edp++) {

        // Compute new least cost path candidate
        dwCost = pU->g + EdgeCost(crV, crU, edp);

        // If it's off limits (e.g. outside the map), then skip it
        if (dwCost >= infinity) continue;

        iV = IndexOf(crV);
        pV = Nodes[iV];

        if (pV == retired) continue;     // Can't improve it!

        if (pV == notvisited) {          // First time,add to queue

            if (pReuse != NULL) {
                pV = pReuse;
                pReuse = NULL;
            } else {
                pV = new FibHeapNode;
            }

            Nodes[iV] = pV;
```

```
                ed[iV] = edp;
                pV->cr = crV;
                pV->g = dwCost;
                pV->h = CostNorm(crV, crEnd);
                PQ.Insert(pV);

            } else if (dwCost <= pV->g) {    // Can be improved!?

                if (dwCost == pV->g) {
                    if (rand() <= RAND_MAX/2)// To avoid biasing...
                        ed[iV] = edp;
                    continue;
                }

                ed[iV] = edp;                // Update neighbor
                pV->g = dwCost;
                PQ.DecreaseKey(pV);
            }
        }
    }

    if (crU == crEnd) {                      // We did find a path!

        // Count the number of edges by backtracking the path
        for (iNumEdges = 0; crU != crBeg; iNumEdges++)
            WalkEdge(crU, crU, OppositeEdge(ed[IndexOf(crU)]));

        // Construct the edge path by backtracking & reversing...
        pedEdges = new EdgeDir[iNumEdges];
        EdgeDir *ped = pedEdges + iNumEdges;

        for (crU = crEnd; crU != crBeg; )
            WalkEdge(crU, crU,
                OppositeEdge(*(--ped) = ed[IndexOf(crU)]));

    } else {                                 // No path found, return 0
        delete[] ed;
        pedEdges = NULL;
        dwCost = 0;
    }

    // Free up some memory (PQ frees up its queued nodes itself)
    if (pReuse != NULL) delete pReuse;
    delete[] Nodes;
    delete[] ed;

    return dwCost;
}
```

4.3 Progressive approximation

For very large graph constructs, we inevitably run out of physical memory when venturing beyond a few million vertices on today's standard personal computers. Using the hard disk for virtual memory helps but slows things down considerably. Under these circumstances, can the A* search be speeded up by decreasing its memory usage? Here, a progressive method has been examined where first a coarse, approximate solution is computed which is then extended to a more accurate solution.

The coarse solution is found by running the usual A* algorithm on a 'subsampled' graph. The subsampled graph is constructed from the original one by 'clumping together' square regions of neighboring vertices into a single vertex in order to get a smaller graph that still represent the original. For a subsampling factor of 4, meaning that 4x4 vertices are collected into each new one, we have a 16-fold reduction of both the number of edges and of vertices as illustrated in figure 16. The memory used by the A* algorithm to find the coarse solution on the subsampled graph is correspondingly reduced. How should we define the cost function for the subsampled graph? Several approaches are plausible. One is to select the shortest path values between the vertices in the original graph that coincide with the ones in the subsampled graph (see figure 16). Another would be to, in some way, 'average' all the possible paths that can be constructed from the edges that are clumped together. Both these approaches present a number of difficulties and would take non-negligible time to compute. Instead, it was chosen to construct a new set of attributes for subsampled graph and 'reuse' the 'original' cost function. If the 'scale' is reduced in all the spatial dimensions, the edge-length dependent values used by the cost function do not necessarily have to be recomputed; the optimum path will be the same although the total path cost will not.

How should the new attributes be selected? The height attribute do not present much of a philosophical problem, it is quite natural to select the (integrated) mean value of all the constituting 'unsubsampled' values. For the road attribute, it is instead beneficial to select the least cost value (i.e. the 'fastest' road attribute of the ones that are clumped together) in order to preserve the connectedness of the roads. If two roads happen to be clumped together, it should be natural that the fastest one is selected in this way. The terrain type presents a tougher problem though. It cannot simply be 'averaged' since it consists of distinct, non-mergeable categories. Here we have chosen to select the most 'populous' terrain type, and in case ambiguity the 'least costly'. This is by no means an obvious choice and may certainly be debated and/or improved. Nevertheless, for most applications, it is the roads that are crucial and the relevant information about their extent is preserved in the present scheme. It must be stressed however, that there may very well be applications where this is not the case and where this approach of 'subsampling' the graph may not be feasible.

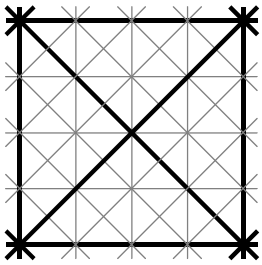


Figure 16 –Subsampled edges, factor 4

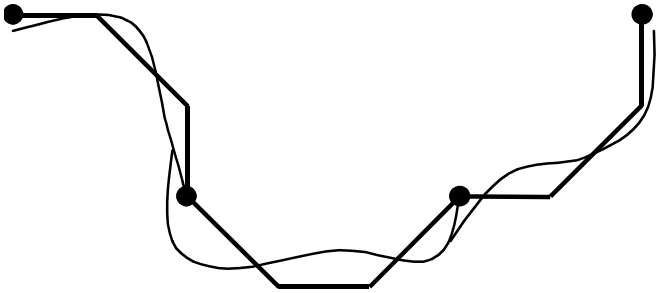


Figure 17 –Path segmentation (exaggerated)

Next, the coarse solution (found on the subsampled graph) is divided into ‘segments’ as is somewhat exaggeratedly illustrated in figure 17, where a small nine-edge path has been divided into three segments of three edges each. Then, the optimum path between the first segment’s end points (marked by black circles in the figure) is found using the A* algorithm on the original, ‘fine’ graph. This is repeated for the other segments and all the ‘fine’ segment solutions is finally pasted together into a complete solution. There is nothing that says that the segment end points should lie on the optimum path however. Thus, this scheme is improved, firstly by using the actual source vertex for the first of the fine segment paths and the destination vertex for the last. Further, at each new segment (after the first one) the previous ‘fine’ path is ‘backtracked’ a few edges. This location is then taken as the source vertex for the A* search on the new segment instead of the ‘coarse’ segment start point; see figure 17. It might also be a good idea to add a check to verify that the vertices used as the source and destination in each fine segment solution doesn’t fall inside e.g. water or other inaccessible areas. Since each segment can be made much shorter than the total path length, the A* search queue will become much smaller. There is a certain overhead associated in finding the path using this procedure so it will not normally be faster. Nevertheless, the much smaller search queue may well be the difference between fitting the whole problem into memory and having to rely on the hard disk for virtual memory. Thus, on large graphs, we can actually expect to save a lot of time. This has been verified in practical tests, see further section 5.5

This process could theoretically proceed in several steps, starting with a very coarse solution and going towards ever finer solution grids/graphs. However, the subsampling may ‘loose’ information that is vital to finding optimum paths, e.g. narrow land bridges, unpassable rivers, et c. This means that if we want to remain ‘close to perfect’, the coarsest graph should still encompass all the ‘essential’ features of the map. This provides a practical upper bound for the subsampling factor. Practical tests show that a factor 6 is about as far as you can go without too apparent side effects (especially in urban areas). Furthermore, there is not much use in having a graph that is finer than the original raster data so the two-step process turns out to be the only one that makes sense. Another issue is the segment length. For maximum speed it should be as large as possible up to a limit value, going beyond which, the A* algorithm starts using too much memory. The actual limit value is dependent on a great many things, both in the actual implementation as well as in the specific environment in which it is run. Trial-and-error analysis of a particular application coupled with and a good ‘safety margin’ should be able to provide a decent segment length number.

5 Examples

How well does the method presented here work out in a real-world application? A number of test cases have been examined, some of them are presented here.

In these figures, the gray level corresponds to the cost of moving through the terrain. The lighter areas are cheaper and the darker are more expensive. Water is black. Roads have been drawn in white. Thick roads have small movement cost while thin roads have larger cost (but generally still less than the typical terrain cost). Paths are drawn in dark gray. The search region is bounded by a black rectangle in case it falls inside the figure, e.g. see figure 18.

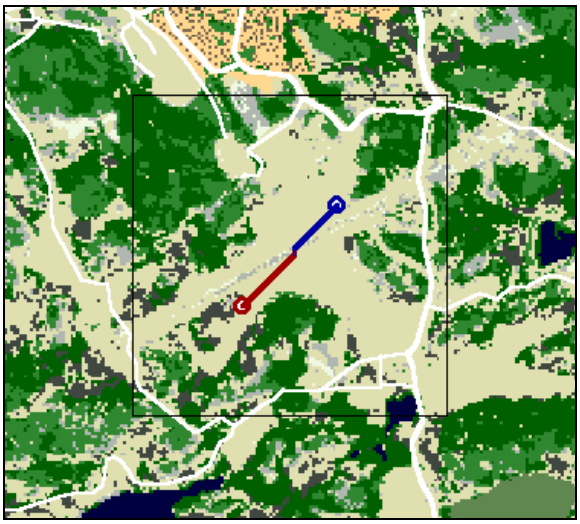


Figure 18 –Some typical terrain, roads, search region and a path.

5.1 Avoiding obstacles

Obstacles in the form of lakes and 'bad' terrain are correctly circumnavigated. In figure 19, the dark gray is forest (very low speed traversal) and black is water.

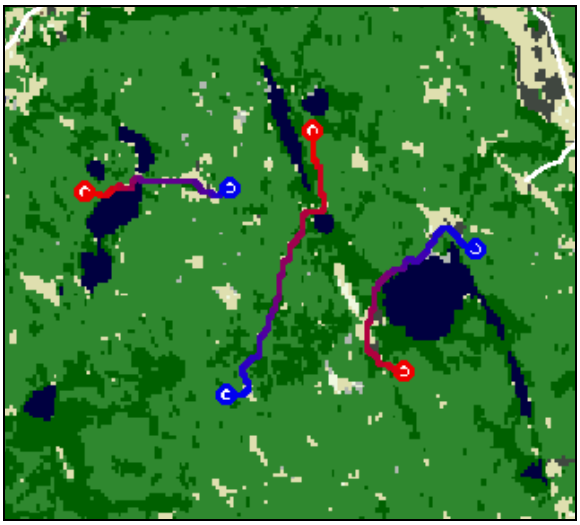


Figure 19 – Three short paths through terrain with some forest and lakes.

5.2 Avoiding enemies

Here is an example where an enemy observer has been strategically placed somewhere between the source and the destination so that it is overlooking several of the main roads. The 'enemy territory' has been shaded by black dots. In figure 20, the penalty for intruding upon it has been set to ∞, i.e. it should be entirely avoided.

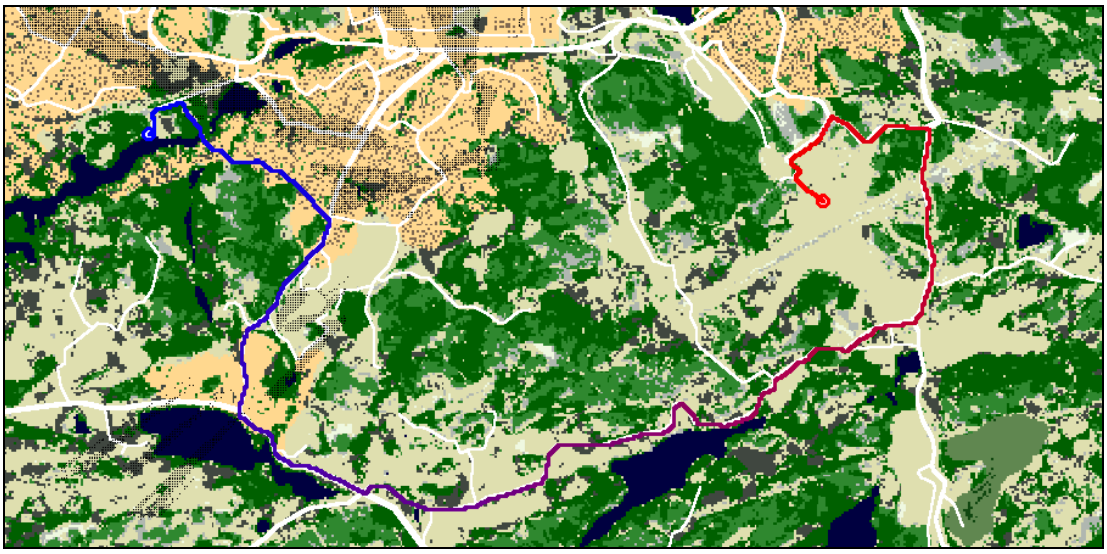


Figure 20 –A roundabout route is taken in order to avoid enemy territory.

In figure 21 the 'detection' penalty has instead been set to a finite value. We see that this allows it to cross over a bridge that is under enemy surveillance since this is the only possible way to get to the destination (because of its isolation by water). However, the penalty is large enough that the number of cells were it is seen by the enemy is kept to a minimum.

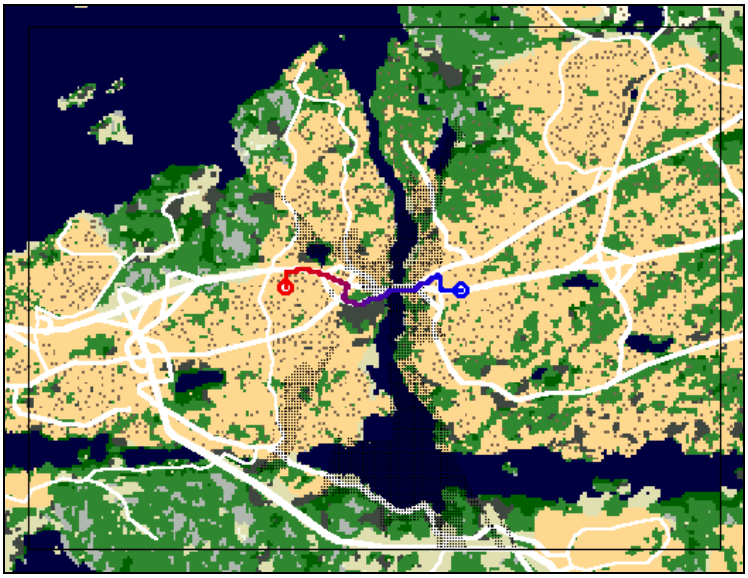


Figure 21 – It may be more important to reach the destination than to completely avoid the enemy.

5.3 Following roads

If we set the road traversal cost very low compared to the cost of moving through terrain, then we should expect the path to follow the roads very closely. Indeed, in figure 22, the road costs are approximately a factor 200 lower than the typical terrain cost. The path goes from the source or destination straight to the nearest road. It then follows the fastest road path between them. Notice that the length of the fastest path is in this case considerably longer than the shortest path distance (through the terrain). Also, notice the path segment at the left, where the slightly longer but faster 'thick road' is chosen instead of the nearer road just to the right of it.

Figure 22 – Roads are highly favored in this example

5.4 A* search area

To accurately demonstrate how the A* search algorithm ‘spreads out’ from the source would really need animation. In deference to the printed media, we will here have to make do with a still picture illustration. Figure 23 illustrates the situation at the moment when the optimum path to the destination is found and the algorithm terminates. Cells that are in the search queue (i.e. on the search ‘front’) have been marked with white dots. All the cells inside of this ‘region’ has been retired from the search queue (i.e. the optimum paths to them have been determined). The cells outside it did not need to be visited (entered into the search queue) during the search. Note that the number of cells in the queue is only a small fraction of all the cells in the search region (delimited by the black rectangle). Further, the total number of cells processed (throughout the search) is something like a fourth of all the cells. In case we have a longer path, more roads and less forest, this fraction is typically more like one half. We see, even in this example, the tendency for roads to enlarge the search area by extending long ‘tendrils’ to remote regions.

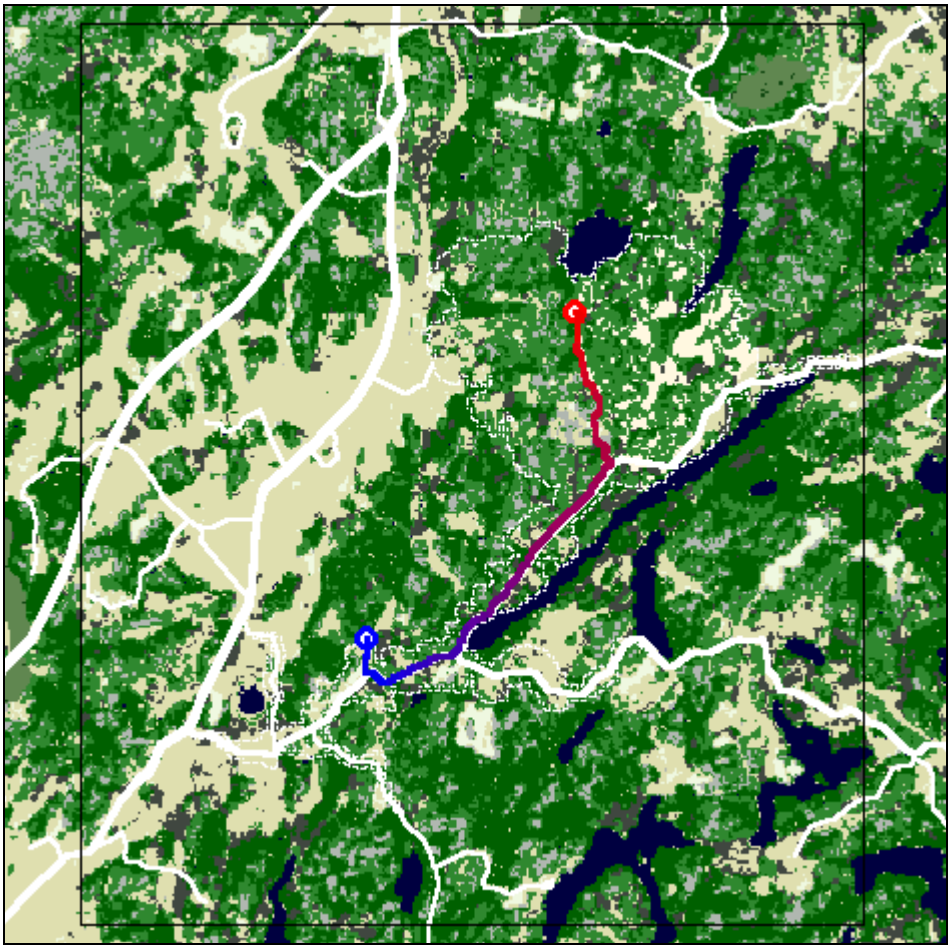


Figure 23 – The search front at the moment when the destination is reached.

5.5 *Progressive approximation*

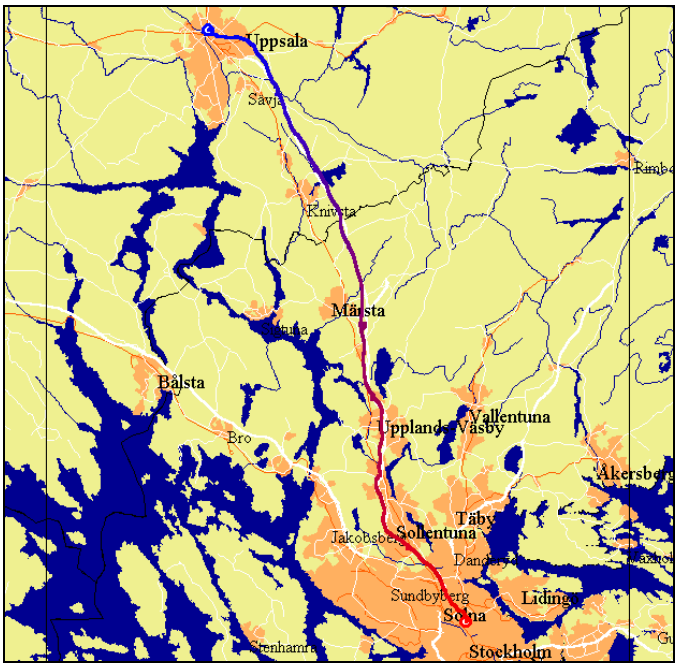


Figure 24 – A larger search graph, 8.5 million vertices

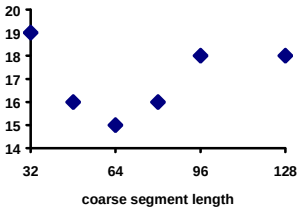


Figure 25 – Time vs. segment length

In order to bring out the details better, the hitherto presented examples have all used a relatively small search region. They all take at most a few seconds to compute. To get an idea of the performance on larger graphs let's look at figure 24 which shows a path where the distance between source and destination is approximately 64km (the cell size is 25m). For clarity, the figure only indicates major roads and only differentiates the terrain type into water, urban or other. The actual search graph data however, is just as complex as in the previous examples, just so much larger that it is not useful to print all the details. The complete search graph contained approximately 8.5 million vertices. No enemies were present. Using the normal algorithm, it took 48s to compute on a standard Pentium 133 with 18Mb of free memory. The time it takes to fetch the raster data has not been counted. Most of the time was taken up by the virtual memory management thrashing the hard disk. As a comparison, computing a 'coarse' solution on a factor 6 subsampled graph took only 2.5s! How well does the progressive method fare on the same problem? It depends a little of the segment length chosen; the more segments, the more overhead there is; so they should be as few (i.e. as long) as possible. On the other hand, if they are too long we again run out of memory and disk swapping starts consuming much time. Figure 25 show the time plotted against the segment length for a few test runs. All these tests used a 6x subsampled approximate solution that were then divided into segments of various lengths. The segment length is expressed as the number of edges in the coarse solution graph (i.e. $1/6^{th}$ of the number of edges in the fine graph). In all the mentioned cases (even the 6x subsampled one) the result actually was visually indistinguishable when printed as in figure 24 due to the fact that the cell size was much smaller than a screen pixel.

5.6 *Miscellaneous*

We will only find paths that fit inside the designated search region. Figures 26 and 27 illustrate this with a lake as an obstacle and two different sized search regions.

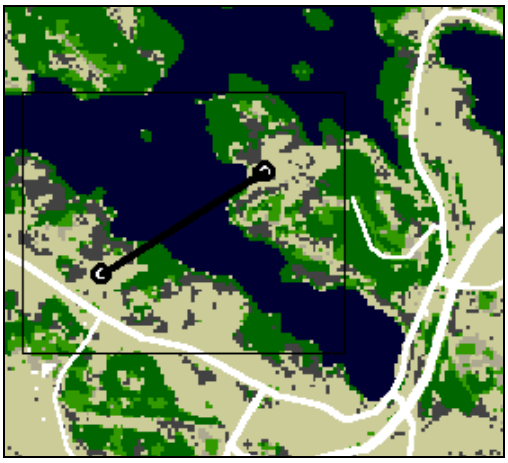


Figure 26 – To small search region – no path found

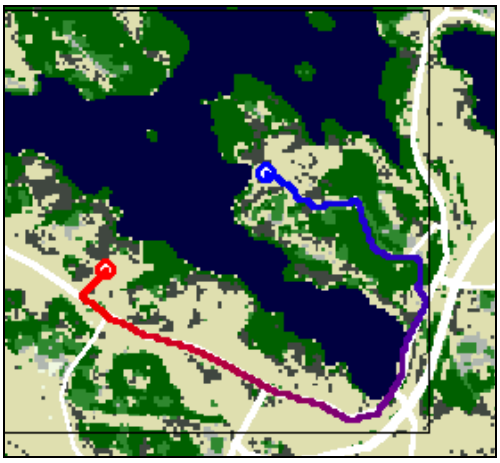


Figure 27 - With slightly larger search region

There is nothing that says that we necessarily have to restrict ourselves to land vehicles. Here's an example simulating a boat; all road and terrain costs are set to ∞ except for water, which is dark in figure 28. Unlike in the previous figures, the colors in this figure do not correspond to the movement cost. Note that if we know what to look for, we can clearly see hints of the fact that there are only eight possible directions of movement. For example in the upper left corner of the path, there is a clear example of directional biasing.

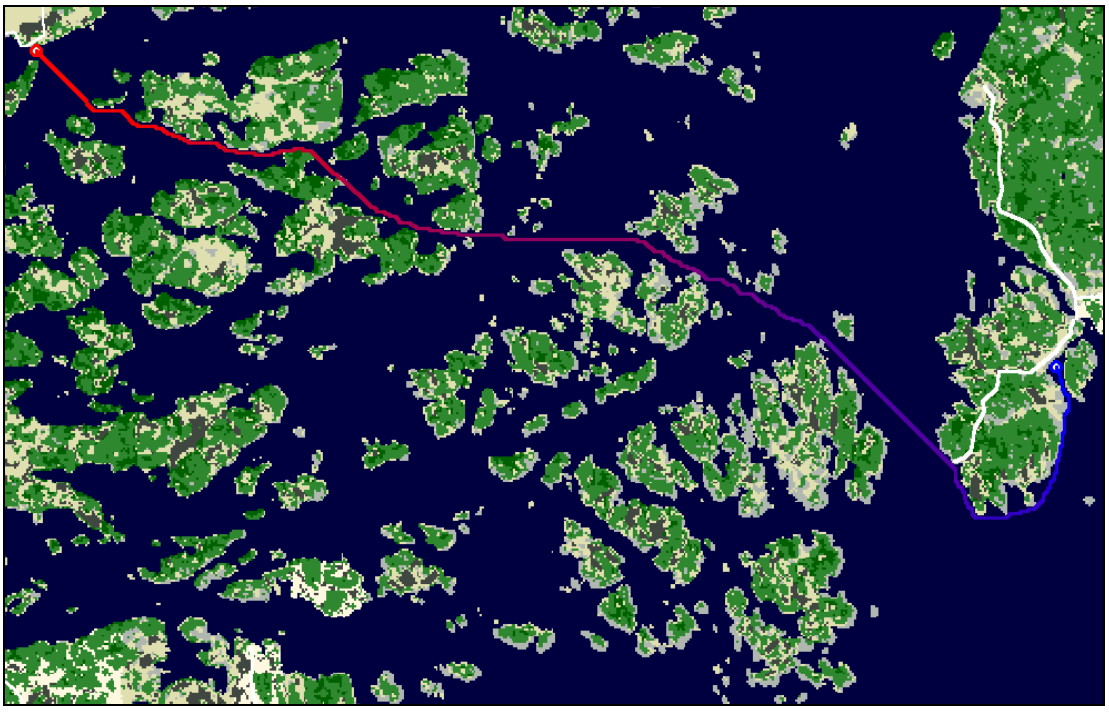


Figure 28 - A boat navigating though a part of the archipelago of Stockholm.

6 Conclusions

A solution to the problem 1.2 has been found. Evaluation of a test implementation has shown that it satisfies the objectives. As always, there are a number of issues that could be further improved upon and avenues of thought left for the curious of mind to explore.

6.1 General conclusions

During the various stages of the method, many approximations have been made. It would seem that worst case accuracy of the method could be put into serious doubt. However, the approximations have always been of a statistically averaging nature and the stability (error propagation) seems to be good. The real world is usually very far removed from the worst case behavior and practical results seem to be very satisfactory. The memory usage and speed objectives have been met and implementation complexity is relatively low.

6.2 Fields for future work

Here are a few thoughts that it might be worthwhile to examine in greater depth:

6.2.1 Dynamic scenes

The presented method inflects a heavy penalty if the ‘scene’, i.e. the cost function, changes. If any input parameter changes (e.g. if an ‘enemy’ moves), a complete new search must be done. Thus, dynamics are not handled very well. A possible solution would be a variant of the progressive scheme presented. Calculate a coarse solution and assume that the scene does not change enough to cause the course solution to be significantly altered. Then calculate only one sub-segment of the path at a time. Take care of any new changes in input parameters only when switching to the next segment. Some kind of threshold detector would be needed to somehow detect large changes in input parameters (movements by the enemies) and recalculate the coarse solution only if the change is large enough. A big question here would be to find an optimum size (division) of the sub-segments. Probably this would have to be determined from heuristics appropriate for each specific application.

Another alternative would be to keep track of a time parameter along with the ‘currently best path’ parameter for every node during the A* search. This parameter could then be used together with the spatial position as an input to the enemy and other modifiers (see 3.1.2.5). This has big the advantage (as opposed to the previous suggestion) of only giving a change in the ‘constant’ of time and space complexity. It also has the advantage of working equally well even if there are large changes in the enemy locations et c. On the other hand, it induces a lot of work, not only for keeping track of this ‘current time’ parameter during the search, but it would also prevent us from using the ‘lazy’ visibility computation as visibility is no longer static during the search.

6.2.2 Extending the number of edges

By extending the number of cells considered as neighbors to a given cell, we can increase the number of edges in the graph. Taking this to its extreme and considering all other cells as neighbors gives us a so-called complete graph. As the A* algorithm with a good heuristic really gives the optimum solution for the graph, this should give us a perfect solution to the ‘rasterized’ problem. But, then the time complexity is $N\log(N)$ found in 4.2.5 is no longer true (N is the number of vertices) since the number of edges, M, is no longer static as was assumed. Instead we have $M=N^2$ number of neighbors to examine in each iteration giving an $O(N\log N + NM) = O(N^3)$ time complexity. Therefore, the cost for perfecting the accuracy for all sizes of graphs is a higher degree polynomial time and memory consumption. Nevertheless, what is the optimum number of neighbors? I.e. what is the ideal compromise between error and performance? The decision to use eight neighboring cells was based upon convenience since that is the maximum number for which the edge cost function is easily defined and computed.

6.2.3 Faster ways of determining visibility

Alternate methods of determining enemy detection/visibility have not been researched for this work. Since it is a rather heavy burden, it is probably the one area where a more effective algorithm could be most advantageous for performance.

6.2.4 Approximate paths using non-optimal A* heuristics

In section 4.2.2, it was observed that an optimal A* heuristic function h, could be scaled as ch so that it is reduced to Dijkstra's method for $c=0$, or giving approximate results for $c>1$. The higher c is, the more the search is 'directed' towards the goal and the lower the expected average running time but also the ‘dumber’ it becomes about walking around obstacles. The worst case behavior (see 4.2.5.) is not improved though. Perhaps better approximating heuristics can be found? Or perhaps we could construct a heuristic that tries to mimic the decision of a human that ‘acts’ upon the 'local' appearance of the surrounding terrain?

6.2.5 Other graph search methods

Alternatives to A* could be examined for finding approximate paths. Simulated annealing, mentioned in 4.1.4, could be an interesting alternative. It has a certain appeal since it gives an absolute control over the solution time; it can stopped whenever we run out of time or when we think the solution is good enough. The longer it is run it the higher accuracy can be expected. However, constructing a good perturbation function seems to be far from trivial. To get a high degree of accuracy in the solution, it might be very hard to get the method to even come close to the performance of the A* method. A detailed comparison of the methods should be interesting. An idea would be to use simulated annealing for a coarse solution and then use A* to improve it locally, similar to the progressive method described in section 4.3.

7 Bibliography

[BOYE97] J. Boyer, *The Fibonacci Heap*, Dr. Dobb's Journal, #261 January 1997.

[BUCK90] F. Buckley, F. Harary, *Distance in graphs*, Addison-Wesley, 1990, pp. 270-272.

[CHER93] B. V. Cherkassky, A.V. Goldberg, and T. Radzik, *Shortest Paths Algorithms: Theory and Practice,* Technical Report, STAN-CS-93-1480, Computer Science Department, Stanford University, Stanford, CA, 1993.

[CLAR90] K. C. Clarke, *Analytical and Computer Cartography*, Prentice Hall, 1990.

[CORM90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1990.

[CSEP95] The Computational Science Education Project, *Mathematical Optimization*, Internet, http://aquarius.u-aizu.ac.jp/CSEP/MO/ MO.html, 1995.

[DIJK59] E. W. Dijkstra, *A note on two problems in connection with graphs*, Numerische Mathematik, 1959.

[ELGR92] R. Elg, *Sökgraf för kortaste vägen i planet*, FOA, 1992.

[FOLE90] J. Foley, A. van Dam, S. Feiner, J. Hughes, *Computer Graphics, Principles and Practice, 2nd ed.*, Addison-Wesley, 1990.

[HOLM92] P. D. Holmes and E.R.A. Jungert, *Symbolic and geometric connectivity graph methods for route planning in digitized maps*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 14, no. 5, 1992, pp549-565.

[KIMM95] R. Kimmel, A. Amir, and A.M. Bruckstein, *Finding shortest paths on surfaces using level sets propagation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 17, no. 6, 1995, pp635-640.

[LONN96] J. C. Lonningdal, *Smart unit navigation*, Internet, http://www.lis.pitt.edu/~john/shorpath.htm, 1996.

[LUEN84] D. G. Luenberger, *Linear and Nonlinear programming*, 2nd ed, Addison Wesley, 1984.

[MONT87] M. Montgomery et al., *Navigation algorithm for a nested hierarchical system of robot path planning among polyhedral obstacles*, Proceedings IEEE International conference on Robotics and Automation, pp. 1616-1622, 1987.

[NILS82] N J Nilsson, *Principles of Artificial Intelligence*, Springer Verlag. Berlin, 1982.

[PATE97] A. Patel (editor), *Game Programming*, Internet, http://www-cs-students.stanford.edu/~amitp/gameprog.html, 1997

[PRES92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C, 2nd ed.*, Cambridge University Press, New York, 1992, pp. 444-455.

[SONK93] M. Sonka, V. Hlavac, R. Boyle, *Image Processing, Analysis and Machine Vision*, Chapman & Hall, London, 1993.

[STEF95] E. Stefanakis, M. Kavouras, *On the determination of the Optimum Path in Space* from A.U. Frank, W. Kuhn (editors) *Spatial Information Theory, A theoretical basis for GIS*, COSIT95 Proceedings, LNCS 988, 1995.

[WAVE96] B. Wavell, *Exploring shortest path algorithms: Dijkstra and A*,* Internet, http://ugrad-www.cs.colorado.edu/~karl/ClassNotes+/Topics/ShortestPath/, 1996.

[WOOD97] S. M. Woodcock (editor). *Artificial Intelligence in Games*, Internet, http://www.cris.com/~swoodcoc/software.html, 1997.