Finding Good Paths: Applications of Least Cost Caloric Path Computations

Zoë Wood, Greg Hoffman and Mark Wazny
Computer Science Department
California Polytechnic State University
San Luis Obispo, CA 93407

Abstract

As you walk around during your daily life, you commonly make path finding decisions based on the world around you. For example, when you are walking in the outdoors, you take the path of least resistance over a terrain. We present two applications which demonstrate the importance of using the least cost caloric cost path computation in two different domain settings. One application uses the popular Google Earth API to explore the use of least cost caloric path computations to create an interactive path-finding tool. The second tool uses least cost caloric path computations to enhance crowd simulations in the 3D modeling and rendering application, Maya. Both applications show that considering the cost of travel from a human centered perspective can produce better results for computing good walking paths for crowds and individuals.

1 Introduction

Humans have been traveling by foot for thousands of years and the task of finding good foot-paths to travel from point A to point B is something we all think about. Recent work [1] [2] on computing a least cost caloric path to help archeologists learn about ancient human's travel patterns have revealed the importance of using a human centered path computation instead of the traditional distance metrics. We continue this work in two individual application projects. One, which builds a plug-in for the popular computer modeling program, Maya, allows for the creation of crowd path computations that consider the terrain as one factor in agent path computations. The second application builds on the popular Google Earth API to provide a tool for users to compute the least cost caloric path from a starting point to an ending point. Both of these projects show the importance of finding 'good paths' by including a human centered cost metric, namely caloric expense, in their computations.

1.1 The Google Earth Application - Individual Path Finding

Mapping software has been used in combination with GPS units and on personal computers to find the most efficient path between a starting and an ending point for about ten years now. Most of these implementations have calculated the most efficient routes for automobiles on city streets and highways in terms of time. In addition, some have been able to calculate routes along trails for hikers to follow. However, one possibility that has been largely overlooked for personal-use route planning has been that of calculating the most energetically efficient freeform path for a human to walk. This type of path calculation allows people to plan out the least tiring path from one point to another without worrying about pre-existent paths, trails or roads. This information could be useful in such applications as planning out accessible paths for individuals with disabilities, an individual planning out his or her path in a freeform race or even in archeology, to model the movement of ancient people since they would tend to choose the most energetically efficient path in their travels.

The goal of this project was to design and implement an application using the Google Earth API. Given the latitude and longitude of a starting and ending point, the application finds the most energetically efficient path between them for walking. This requires taking into account the effects of slope on the energy required for humans to walk, along with issues with the curvature of Earth. When finished with the computation, the application displays the path overlaid on the Google Earth user interface. Development on this application was done in C# using Visual Studio 2008 and the Google Earth COM API, both of which had their strengths and their shortcomings.

Assuming that we can represent any terrain that we would like to traverse as a bidirectional connected graph (see Figure 2), we can use popular shortest path algorithms to find shortest paths across the terrain, such as Dijkstras shortest path algorithm [3]. Of

Figure 1: [Path Finding]The yellow line indicates the shortest path, assuming that the cost of travel is in calories.

course, the question becomes, what is the cost of travel across a given terrain. Traditional metrics such as Euclidean distance do not capture the human desire to not climb large mountains. Previous work [1] [2], has shown that the use of the caloric cost equation is the appropriate metric to consider human travel, i.e. walking. The caloric cost equation is used in kinesiology to find the amount of calories used to travel at a given speed by a given person up or down a given slope. This equation defines the power usage of traveling downhill as MR and the power usage of traveling uphill as M in the set of equations:

$$MR_{uphill} = M \tag{1}$$

$$MR_{downhill} = M - C \tag{2}$$

$$M = 1.5w + 2.0\,(w+l)\left(\frac{l}{w}\right)^2 + n\,(w+l)\left(1.5v^2 + 0.35vg\right) \tag{3}$$

$$C = n\left(\frac{g\,(w+l)\,v}{3.5} - \frac{(w+l)\,(g+6)^2}{w} + 25 - v^2\right) \tag{4}$$

Where:

MR	is the metabolic rate in watts
w	is the person's weight in kilograms
l	is the load carried in kilograms
v	is the velocity in meters per second
g	is the percent grade
n	is the terrain factor

Terrain factors are:

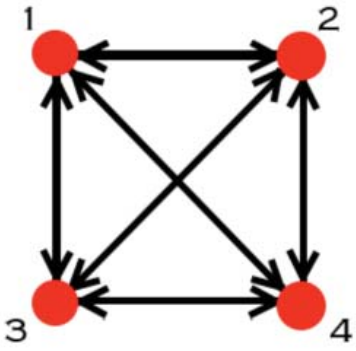

Figure 2: [Graph] An example of one grid of the bi-directional graph used in both applications.

1.0	Black Top Road / Treadmill
1.1	Dirt Road
1.2	Light Brush
1.5	Heavy Brush
1.8	Swampy Bog
2.1	Loose Sand
1.3+0.082*D	Snow, where
	D = depression depth in cm

However, to calculate the actual amount of calories used, the number calculated by this equation is multiplied by the amount of distance between the two points since power does not take into account how long the person must travel and, with a constant speed, the distance travelled is a good approximation of the time it will take. The distance also had to be calculated from the starting and ending latitude and longitude since Google Earth only gives latitude and longitude rather than direct distance values. A cartesian distance was computed by multiplying the circumference of Earth by the change in degrees divided by 360 for latitudinal distance and by multiplying it by the change in degrees times the cosine of the latitude for longitudinal distance. The reason for the cosine of the latitude is that the earths circumference changes according to the latitude ranging along a cosine curve from its maximum at the equator to zero at the poles.

Once this cost is set as the weights for the edges, the application finds the shortest path in terms of caloric cost; in other words: the most energetically efficient path. In addition to Dijkstra's shortest path algorithm, there are various methods to approximate shortest paths, for example A* informed search. We implemented this search metric to try to speed up the path computation time. To do this, the existing Dijkstras algorithm was modified to also factor in the cost from the current nodes to the end point (by adding the caloric cost of traveling the Euclidean distance from each node directly to the endpoint) This algorithm

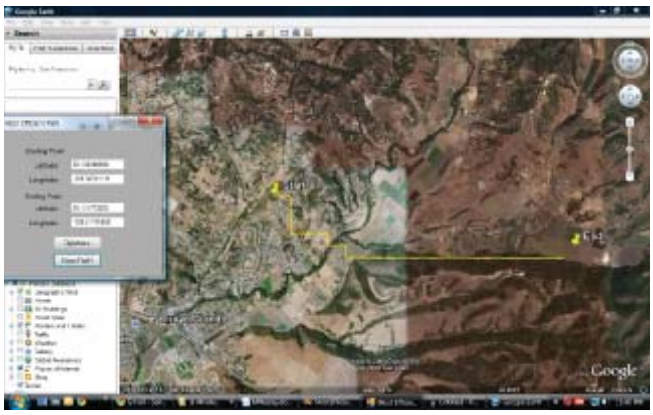

Figure 3: [Path Finding using Dijkstra's] The yellow line indicates the shortest path computed using Dijkstra's shortest path algorithm from (35.138866666, -120.56311111) to (35.131772222, -120.51175555).

Figure 4: [Path Finding using A*]The yellow line indicates the shortest path computed using A* path finding algorithm from (35.138866666, -120.56311111) to (35.131772222, -120.51175555).

eliminates many more potential paths earlier in the search.

1.1.1 Google Earth Application Results

This application successfully uses either Dijkstras shortest path algorithm or the A* informed search algorithm to find the most energetically efficient path between two points on Earth. All testing has shown that it does so fairly accurately and, in the case of A*, in a reasonable amount of time. Some problems were encountered, which hinder the application. Namely, a major limitation was in the Google Earth COM API. While this API provided many of the functions needed for this project, it was difficult to get altitudes (needed to compute slope and caloric cost) in an efficient way. The API provides the method *GetPointOnTerrainFromScreenCoords([in] double screen_x,[in] double screen_y)* which returns the latitude, longitude, and altitude of that point encapsulated in an IPointOnTerrainGE object. It does not, however, provide any means with which to request this information for more than one point at a time. Even with caching the altitudes of previously visited points, these point by point queries were very slow.

In terms of results, the below tables demonstrate that A* was always significantly faster, partially caused by the previously mentioned limitation of the Google Earth COM API which causes a massive slowdown when Dijkstras shortest path algorithm checks over two-thousand locations. The testing proved that A* is far and away more efficient than Dijkstras for this scenario. Also, while Dijkstras proved to produce paths that were more energetically efficient, the difference was very slight between the two algorithms. The reasons why the two algorithms provided different paths is likely because A* attempts to go towards the end point, causing it to produce more direct paths, even though they may be slightly less efficient in terms of calories. See Figures 3 and 4 for a comparison of the two different paths created using the two different search algorithms on the same data set. In summary, this application shows the utility of including a human centered cost in the creation of paths.

The following tables give the search times for specific latitude and longitude start and stop positions on 100x100 resolution Grids:

Test 1	(35.298133, -120.655797) to (35.300775, -120.661711)
Dijkstras **nodes**	4.482 seconds 4371 tested locations
A* **nodes**	0.174 seconds 184 tested locations
summary	A* 25.759 times as fast tests less than 1/23 the nodes

Test 2	(35.138866666, -120.56311111)) to (35.131772222, -120.51175555)
Dijkstras **nodes**	43 minutes 24.612 seconds 5129 tested location
A* **nodes**	0.155 seconds 132 tested locations
summary	A* 16,803.948 times as fast tests less than 1/38 the nodes

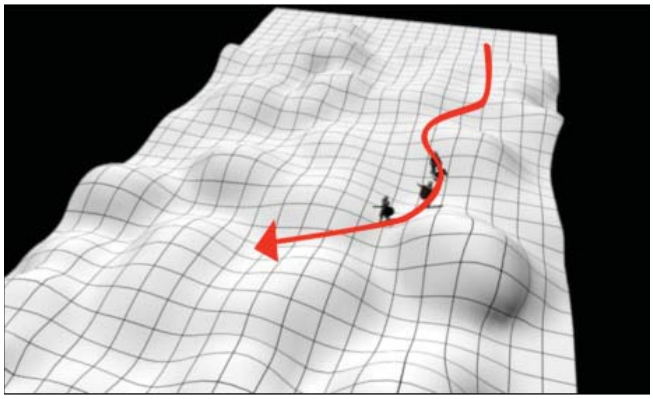

Figure 5: [Following the Terrain] The agents correctly follow the terrain instead of walking straight across the terrain and over the hills, a very unnatural human path.

Test 2	(35.160444444, -120.5307027777) to (35.131772222, -120.51175555)
Dijkstras nodes	1.622 seconds 1756 tested locations
A* nodes	0.113 seconds 127 tested locations
summary	A* 14.354 times as fast tests less than 1/13 the nodes

1.2 The Maya plugin - Agent and Crowd Path Finding

As you walk around during your daily life, you commonly make path finding decisions based on the world around you: when you are driving on the freeway you avoid lanes compacted with traffic and when you are walking in the outdoors, you take the path of least resistance over a terrain. These examples, while second nature to us, are not something that a computational path algorithm may consider. When simulating a large number of realistic computer generated characters, they should model this same behavior. Commonly, a simulated computer generated character is referred to as an agent and a grouping of agents is referred to as a crowd. There are many applications of agent simulations. They are often employed within the film and game industry [5] because it is more cost effective to let a computer control the movement of hundreds of characters instead of letting an animator animate each one by hand.

Behavioral solutions have been developed to solve basic agent interaction problems. Simple behaviors can be set up to cause agents to seek towards positions or to avoid other agents around them. Behavioral solutions, however, only get you so far. Once you start working with more complex environments other issues develop. If you are, for example, trying to realistically navigate a hilly environment, the agents would have no idea about human strengths and weaknesses. The agent would have no idea that walking up and over a hill takes just as much if not more energy as walking around it. Thus, we present an implementation of agent path finding where the surrounding terrain can be taken into account and the actual traversal cost calculated. More complex path finding issues also occur in regard to group dynamics. For example, when you are driving down a freeway or a city street you tend to follow the path of least resistance. If one lane is clogged with traffic, you switch to another lane in order to avoid that traffic. In much the same way, agents need to account for congestion for more realistic traversal of terrain. If the route an agent is traveling has other agents in front of it that are causing a slowdown, it should seek alternative routes.

This project implements a plug-in for Maya that simulates agent path finding over a terrain. The project also uses an influence map for advanced path planning. The application of this project is geared towards the film and visual effects industry. For this reason, the simulation is biased towards accurate results at the expense of speed.

1.2.1 Terrain Navigation

When you see a hill in front of you and you need to get to the other side, it is not cost effective to simply walk over the hill. If the hill is high enough, you can use far less energy by walking around. Thus, we implemented the plug-in to create a path finding solution that would take into account the energy needed to traverse a terrain. The terrain can be created directly from the 3D program, with the terrain representation used for collision detection as well as to build the graph. The graph, like the previous application, is a bidirectional graph (see Figure 2). Both nodes and edges can have some sort of weight associated with them. For this simulation the weight on each node represented the density of agents, or the agents influence in the surrounding area, and the weight on the edges represented the cost of traveling along that edge. Edges are weighted using the same caloric cost to traverse that edge as the previous application (see Equation 3).

In order to find a path between two points through the graph, Dijkstra algorithm was used. Dijkstras algorithm finds the least cost path between two points. Dijkstra's is not as efficient as A* in finding a path, however for this application we decided that the approximations introduced by A* did not warrant the speed increase. Since the agent computations are typically done offline and not intended to be real-time, we took the time to use the more accurate Dijkstra's

shortest path finding algorithm.

A unique aspect of this application is determining the agents location with respect to the graph data structure. Using a linear search technique would require comparing the agents position with that of every node in the graph. With a graph of thousands of nodes, this takes excessive amounts of time. As a result, a data structure called a KdTree is used to mitigate this. A KdTree is a way of spatially organizing information which is optimized for nearest neighbor searches. KdTrees function by parsing space in N dimensions, in this case it uses a 3 dimensional tree, cycling between x, y, and z. At each node, the branch on the left is all nodes that are less than the current nodes value for the given dimension. Conversely, all nodes in the right branch are greater in the current nodes dimension. Searches take place by comparing the search position with the current node of the tree.

As shown in Figure 5, the agents correctly follow the natural valleys in the terrain instead of walking straight over the hills, which would be a very unnatural human path.

1.2.2 Agent Avoidance

Another concern with agent path finding is that the agents themselves can become obstacles. In a static situation, with unmoving agents, agents should choose paths around those obstacles. This, however, is not the case for a behavioral path finding solution. In a behavioral solution, the agent would run into the agents in front of it and attempt to push them aside or, the agent would stop because it recognized agents in front of it and become stuck. The implemented solution added extra information to the graph used for the terrain navigation. This, in effect, layered an influence map on top of the graph. An influence map is commonly used in AI to keep track of the influence an agent or group of agents has over an area [6]. By doing this it keeps all the information for an agent to navigate an environment in one place. It also allows us to leverage the existing algorithms. Every frame, each agent's influence is calculated and added into each node in the graph. The influence is calculated by propagating a constant value across the nodes an agent is close to. Half this value is then propagated to any surrounding nodes. When Dijkstra's algorithm is running, any node it passes through has this extra influence value added in to the total cost. An option for the agents to not take into account the influence map when calculating paths is provided as well.

In order to provide a more realistic simulation of agents picking paths through an environment you need to allow agents to reevaluates their current path based on new information. The first approach tried was to have agents revaluate their pathing decisions every frame. This, however, proved to be problematic. It would force agents to scatter whenever they were grouped up due to the influence of the surrounding agents on the graph. Also, calculating pathing every frame resulted in very slow runtimes. While a number of different approaches were tried, the implementation that was the most successful was when an agents reevaluates their pathing only when they were actually impeded. As an agent reached a point along its path it would calculate the approximate time it will take to get to the next point. If the time it takes to travel to that next point is longer than that calculated value, plus some constant to allow for a small amount of error, then the agent reevaluates its path.

As shown in Figure 6, you can see a group of agents breaking off from the main group and traveling along a different path to avoid the bottleneck between the two hills. This mimics a more natural way of moving about an environment. Two variables control the likelihood of agents breaking off from the main path. The first is the constant value that accounts for error. The lower that is, the more likely the agents will recalculate their paths. The other factor for forcing agents to actually choose another path when they reevaluate is the influence amount each agent adds to the graph. Larger values will allow agents to choose successfully longer paths. This can be seen in Figure 7. The overhead path that the highlighted agent group is following is a significant detour from the primary path. By making the influence agents add to the graph much higher you are able to force them to take longer paths.

1.2.3 Maya plugin Results

The simulation itself worked correctly. It would find paths for any number of agents across the terrain. By controlling the amount of influence each agent added to the influence map and the tolerance they had for being delayed, you could easily control the path agents took. They could find their way around any type of terrain no matter the shape or size in a way that made sense. Although different terrains did require tweaking the variables of an agent's influence and the error term in the estimated path computation. There were, however, two implementation limitations. For one, the way in which the agents follow a path is very basic. As a result, there would be times they get off the path and circle around a little until they can find their way back on. This movement was very easy to spot and looked very unnatural. Also, there were issues in regards to the physics simulation used to control agent movements. When agents went down a terrain slope the simple physics model used in this implementation resulted in a stepped type motion, instead of smoothly

Figure 6: [Going around the hill] The agents are correctly breaking off and going around the hill instead of being trapped in the bottleneck of the gap between the hills.

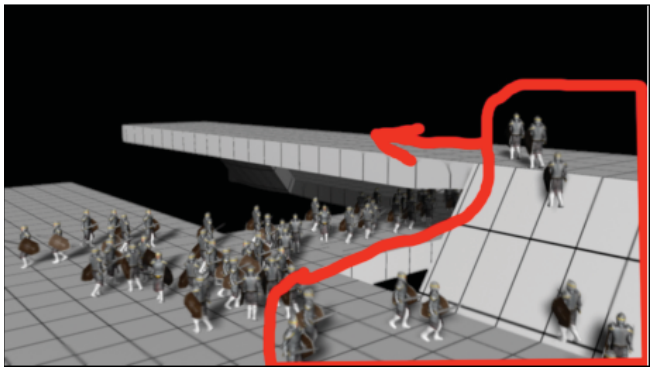

Figure 7: [Re-routing]The agents are correctly breaking off and traversing over a 'bridge' because the lower path is blocked by too many other agents.

moving down a hill. This is in part because of the very basic way physics is implemented in this simulator and could be improved in future implementations.

2 Summary

Both of these applications show the importance and utility of including a more human centered path computation in applications which try to find good paths. The first application presents an interactive application built using the Google Earth API that allows the user to select a starting and ending point. The application demonstrates that using the caloric cost metric in conjunction with the A* algorithm for path finding , computes paths efficiently which can be displayed overlaid on the popular Google Earth user interface. A* was the path computation of choice for this application, since interactivity was one of the goal of the project. The second application, built as a plugin for Maya, shows that agent simulations can incorporate a human centered path computation to create more realistic behavior over varying terrain. This application uses the caloric cost metric as the edge weights for Dijkstra's shortest path algorithm in order to find paths for agents that conform well to the given terrain. In this setting the more accurate paths computed using Dijkstra's were used.

References

[1] B. Wood and Z. Wood, "Energetically optimal travel across terrain: visualizations and a new metric of geographic distance with anthropological applications," *SPIE Visualization and Data Analysis 2006.*

[2] Andrew Tsui and Z. Wood, "Energetic Path Finding Across Massive Terrain Data," *Proceedings of ISVC, 2009.*

[3] Edsger W. Dijkstra, "A note on two problems in connexion with graphs", *Numerische Mathematik*,1959.

[4] Jason Rickwald, "Continuous Energetically Optimal Paths across Large Digital Elevation Data Sets", *Cal Poly Master's Thesis.*

[5] http://www.massivesoftware.com/

[6] Ian Millington, "Artificial Intelligence for Games", *United States of America: Morgan Kaufmann.*

[7] Mat Buckland, "Programming Game AI by Example", *United States of America: Wordware Publishing Inc.*

[8] Rick Parent, "Computer Animation Algorithms and Techniques", *United States of America: Morgan Kaufmann*

[9] Joseph Knapik, "Physiological, Biomechanical and Medical Aspects of Soldier Load Carriage", *RTO MP-056*