Optimal Path Planning with A* Search Algorithm
M. Michael Nourai
Department of Computer Science
University of Massachusetts Lowell
mnourai@cs.uml.edu

ABSTRACT
In this paper I describe my findings and implementation of an optimal path planning artificial intelligence program designed to help make commute time from Danvers to Salem consistent on day to day basis. The problem is formulated into graph with vertex and edges. The graph is transformed into a tree data structure so that it can be traversed. Then the A* Search algorithm is implemented to traverse the tree and find an optimal path from start to destination.

Author Keywords
Path planning, A* search

INTRODUCTION
I live in Danvers and work in Salem MA, which is about an eight mile drive from home to work. The time it takes to get to work varies on day to day basis and depends on several dynamic factors such as the weather condition and special events in and around Salem. On average, driving to work takes 40 minutes to an hour and half. Over the years this has become a daily challenge since I commute every day of the week to a very busy city and in addition my job requires that I have a consistent work schedule. My project addresses this real-world problem with AI path planning solution finding an optimal path using A* search algorithm. My work builds upon prior work done by Peter Norvig for finding an optimal path and it’s also based on Sebastian Thrun’s robot simulation using A* search algorithm for navigating through a maze.

PROJECT DESCRIPTION
There are no major highways to use to get to work; it’s all back roads and some major roads. Although the major roads cannot be totally avoided, the path from home to work can be optimized to reduce the overall commute time. This problem can be formulated using A* admissible heuristic search. The A* search algorithm is a simple and effective technique that can be used to compute the shortest path to a target location.

The A* search algorithm would be a good fit for this problem since the algorithm avoids expanding paths that are already expensive. Paths would be flagged as expensive which consists of being used for special city wide events, during heavy traffic time slots, constructions, and unfavorable weather conditions. The evaluation function $f(n) = g(n) + h(n)$ would be used to calculate $f(n)$, which is the estimated total cost of path through $n$ to goal. The $g(n)$ is the cost so far to reach $n$, and $h(n)$ is the estimated cost to goal from $n$. The up-to-date road conditions (if available) can be dynamically programmed into $g(n)$ for more accurate cost for reaching $n$. The solution will be using A* admissible heuristic search algorithm, therefore $h_{SLD}(n)$, i.e. the Straight-line distance for each node is needed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Figure 1 – Data flow diagram
There are several major components that needs to be fed into the A* Search algorithm (figure 1). The Earth Map component does all the mapping and figuring out cost for each road. The cost is required by the A* search algorithm and will be used for calculating the heuristics $f(n)$. The cost consists of distances between each road and distances
between each road and the destination, i.e. the components needed to calculate the heuristic function for each road. The Road Condition component prompts the user to specify which road is blocked, and then marks that road in the tree data structure as expensive, by removing its edge. The A* component is where all the artificial intelligence program logic resides. It takes all the available information, and plans an optimal path from start to destination. The Display component handles all the information that user need for planning their daily route. It provides step by step directions from the start to the destination.

**Project implementation**

The program is written in CSharp 4.0 with the .NET Framework 4.0 using Microsoft Visual Studio 2010 Ultimate edition on Windows 7 64-bit Professional. The project was implemented using advanced object-oriented programming techniques (figure 2), including encapsulation, inheritance, and polymorphism. The CSharp features such as Func, Delegate, and Lambda Expressions have also been used for more efficient coding.

The heuristic function calculation

To compute the cost for each road, the program utilizes the haversine formula. For the A* Search algorithm to work efficiently, h(n), the heuristic part of the f(n) function must be an admissible heuristic, i.e. it must not overestimate the distance to the goal. To satisfy the A* Search algorithm's admissibility requirement, the Haversine formula has been implemented for calculating h(n).

The haversine formula is an equation important in navigation, giving great-circle distances between two points on a sphere from their longitudes and latitudes (figure 3). It is a special case of a more general formula in spherical trigonometry, the law of haversines, relating the sides and angles of spherical "triangles".

![Figure 3 – The law of haversine](image)

The haversine formula

For any two points on a sphere:

\[
\text{haversin} \left( \frac{d}{r} \right) = \text{haversin}(\phi_2 - \phi_1) + \\
\cos(\phi_1) \cos(\phi_2) \text{haversin}(\psi_2 - \psi_1)
\]

Where

- \text{haversin} is the haversine function
- \text{haversin}(\theta) = \sin(\theta/2)^2 = \frac{1 - \cos(\theta)}{2}
- d is the distance between two points
- r is the radius of the sphere
- \phi_1, \phi_2 are latitude of point 1 and point 2
- \psi_1, \psi_2 are longitude of point 1 and point 2

On the left side of the equals sign, the argument to the haversine function is in radians. In degrees, \text{haversin}(d/R) in the formula would become \text{haversin}(180^\circ d/\pi R).
One can then solve for either by simply applying the inverse haversine (if available) or by using the arcsine (inverse sine) function:

\[ d = r \text{haversin}^{-1}(h) = 2r \arcsin \left( \sqrt{h} \right) \]

Where
- \( h \) is \( \text{haversine}(d/R) \)

\[ d = 2r \arcsin \sqrt{\text{haversin}(\phi_2 - \phi_1) + \cos(\phi_1) \cos(\phi_2) \text{haversin}(\psi_2 - \psi_1)} \]

Performing substitution results in the final form of haversine formula:

\[ d = 2r \arcsin \sqrt{\sin^2 \left( \frac{\phi_2 - \phi_1}{2} \right) + \cos(\phi_1) \cos(\phi_2) \sin^2 \left( \frac{\psi_2 - \psi_1}{2} \right)} \]

**Tree data structure**

The A* Search algorithm is this program works on a graph which is made up of nodes and edges. My project mapped out all the practical routes from Danvers to Salem in a form of a graph with nodes to create a search tree space for A* to traverse it (figure 4). The nodes in the tree represent the beginning of the street, and the edges represent connections between streets. Then the A* Search algorithm is used to efficiently traverse path between the nodes.

As A* traverses the graph, it follows a path of the lowest known cost, keeping a sorted priority queue of alternate path segments along the way. If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. This process continues until the goal is reached.

**Data set**

The data set for the project will consist of roads listed on maps. There are a total of 26 nodes and 32 edges. The cost of each nodes and edges have been calculated using haversine and saved in the tree structure. The distance-plus-cost heuristic function \( f(n) \) is used to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions \( g(n) \) and \( h(n) \). Where \( g(n) \) is the cost of reaching node \( n \) from the start state and \( h(n) \) is a heuristic estimate of the distance from node \( n \) to the nearest goal node.
ANALYSIS OF RESULTS

The program successfully finds the optimal path from the start point in Danvers to the destination point in Salem. The solution has been tested with simulated road conditions and has performed exceptionally well, i.e. the A* Search algorithm worked every time and was able to find the optimal path.

A menu driven interface has been designed and implemented to simulate the road blockage condition which can be due to heavy traffic, road construction, or bad weather condition. The calculated optimal path is then displayed on the console, giving step by step directions from the start to the destination.

With all roads active, the program successfully finds the optimal and shortest path (table 1). Other road conditions have been tested, and the A* search algorithm worked flawlessly to find the optimal path to destination.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Total Cost (miles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reservoir Drive</td>
<td>Route 35</td>
<td>1.015</td>
</tr>
<tr>
<td>Route 35</td>
<td>Water Street</td>
<td>3.452</td>
</tr>
<tr>
<td>Water Street</td>
<td>North Street</td>
<td>4.807</td>
</tr>
<tr>
<td>North Street</td>
<td>Norman Street</td>
<td>6.325</td>
</tr>
<tr>
<td>Norman Street</td>
<td>Lafayette Street</td>
<td>6.585</td>
</tr>
<tr>
<td>Lafayette Street</td>
<td>Salem State</td>
<td>7.6</td>
</tr>
</tbody>
</table>

Table 1 – A* search result for all active roads

Another example of program output with the successful result is shown in table 2, when the following roads were marked as blocked: Reservoir Drive to Route 35, Center Street to Endicott Street, and Route 114 to Gardner Street. Similarly, other roads were blocked in the simulation to test the A* search algorithm, and in all cases it was able to successfully find an optimal and shortest possible path to the destination.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Total Cost (miles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reservoir Drive</td>
<td>Route 1</td>
<td>0.297</td>
</tr>
<tr>
<td>Route 1</td>
<td>Lowell Street</td>
<td>3.953</td>
</tr>
<tr>
<td>Lowell Street</td>
<td>Highland Ave</td>
<td>8.452</td>
</tr>
<tr>
<td>Highland Ave</td>
<td>Wilson Street</td>
<td>9.128</td>
</tr>
<tr>
<td>Wilson Street</td>
<td>Jefferson</td>
<td>9.660</td>
</tr>
<tr>
<td>Jefferson Ave</td>
<td>Loring Ave</td>
<td>10.175</td>
</tr>
<tr>
<td>Loring Ave</td>
<td>Salem State</td>
<td>10.611</td>
</tr>
</tbody>
</table>

Table 2 – A* search result for several blocked roads

DISCUSSION

This project presented an interesting challenge in developing path planning software for a large area as big as 15 miles radius and dealing with the real-world dynamic situation. The biggest thing I’ve learned about my project was during the implementation of the A* Search algorithm and how to calculate the values for the heuristic function f(n). In particular calculating the values for the heuristic h(n) was the most challenging. I research extensively how to work with earth maps and work with great distances. I started with Google online map, and then tried a variety of mapping tools such as Microsoft Streets & Trips 2011. Through research I found a tool that GIS professionals use, and finding the g(n) and heuristic h(n) became very easy.

Researching and finding the haversine formula and its concept was a great learning experience for me. Before finding this concept, calculating h(n), the heuristic part of the A* search algorithm for a 15 miles map seemed like a big job. With haversine, no map problem is too big. I can map a very large area that stretches to thousands of miles with incredible accuracy. Using this technique also guarantees that the heuristic h(n) is admissible.

RELATED WORK

There are two related work on the path planning and finding the shortest path using A* Search algorithm which is similar to the concept I have used in my project. One work was done by Peter Norvig for a lecture material, mapping out cities and towns of Romania, and calculating cost of going from each city to the next (figure 6). Then using the A* Search algorithm to find the shortest path from start to the destination.

Figure 6 – Map of Romania used with A* Search

1 The haversine was discussed in great details on the second page of this document.
Another work was done by Sebastian Thrun for navigating a robotic car in a parking lot. In the video lecture materials, Thrun shows an actual result of applying a modified version of A* algorithm for a robotic car in simulation (figure 7). He created a graphical simulation in which he added obstacles which are laser scans of parked cars, and a target location. While the curve isn’t super smooth, the robot car in simulation was able to find a continuous and drivable curve to the parking location. In the simulation the tree gets expanded dynamically while robot is in that search. And every time it gets stuck, it does a new A* search. Simulation shows how the map is being acquired as the robot moves. In its state that’s in front of the robot, it not only considers the x, y and hidden direction but also allows the robot to go forward and backwards. Driving backwards is just a different state than going forwards. The robot car backs up, finds a new path, and it is an incomplete maze until it finally is able to reach the goal location through an actual opening. For this simulation Thrun used a complicated maze to test the algorithms. The nice thing is these algorithms work almost in real time. It takes less than a tenth of a second to build this entire search tree, and the robot is able to navigate the parking very efficiently.

Thrun applied the same algorithm to an actual parking lot example using a robot car named “Junior” (figure 8). The robot car successfully backs up, goes forward, and navigates the parking lot. The state space consisted of 4-dimensions. It comprises of x, y, hidden direction, and whether the car is going forward or backwards. There is a cost to changing directions, so it doesn’t change direction too often. The robot successfully navigates to its target location.

**CONCLUSION**

The system described in this paper was successful in test situations in being able to route an optimal shortest path from start to destination. To determine if the solution is successful in the real-world, a variety of driving conditions needs to be tested using this solution. I am planning to put the solution to test in the real-world for several months, while collecting and compiling its result. The solution is successful in the real-world when the average arriving time is consistently low for variety of commute conditions.

**ACKNOWLEDGMENTS**

The work described in this paper was conducted as part of a Fall 2011 Artificial Intelligence course, taught in the Computer Science department of the University of Massachusetts Lowell by Prof. Fred Martin.

**REFERENCES**

2. Haversine formula.  
3. ArcGIS Explorer Online.  
   http://explorer.arcgis.com
4. Digital Geography Lab (DGL), Salem State University  
   http://dgl.salemstate.edu
5. Microsoft CSharp programming reference library.  
   http://msdn.microsoft.com
6. The Stanford Artificial Intelligence online class lectures.  
   http://ai-class.com