Dynamic Routing

E90 Project
Disha Katharani
Jonathan Sarmiento

Advisors: Professor Erik Cheever
May 4, 2006

Table of Contents

Abstract	3
Introduction	3
TransCAD Information	4
GPS Information	5
Modeling the Map as a Graph	6
The A* Algorithm	7
Accelerating the A*	15
Dynamic Route Updates	18
Acquisition and Application of Traffic Data	19
Tests and Results	20
Future Extensions	25
Acknowledgements	29
Bibliography	30

Abstract

For our project, we developed an intelligent vehicle routing system supported by a GPS receiver and a laptop interface. The GPS device is used to determine the current location of the vehicle relative to a user specified final destination. The A* network optimization algorithm is used to dynamically choose the best route to reach the destination based on distance or time. The vehicle location is then tracked as it follows the specified route and it is re-routed when found to be off course. The algorithm also incorporates traffic conditions in order to establish the optimal route.

Introduction

In-car navigation systems are widely gaining popularity as a source of information for safer and more comfortable driving. Not only does it provide the optimal route between any origin-destination pair but also guides the driver throughout the route with turn – by – turn directions and constantly re-routes the vehicle the instant it goes off-course. This allows the driver to easily navigate through unknown road networks and reach a desired destination in the shortest possible time.

In combinatorial optimization computing the shortest path is regarded as one of the most fundamental problems. Combinatorial models of real world scenarios often reduce to or contain shortest path computations. Much research has been done on shortest path problems and a variety of different algorithms for computing shortest paths efficiently for a given network currently exist. However in today's resource constrained and increasingly demanding landscape it is necessary to continue to optimize them further and customize them to various applications.

A natural application for the shortest path problem is vehicle routing. The problem of vehicle routing gets particularly challenging because of two main factors. Firstly, it has to be solved in real time in order to dynamically re-route a vehicle as soon as it goes off-course. Secondly, the size of the street network is very large as it contains millions of intersections and the memory space available to most navigation systems is fairly limited and this restricts pre-processing the network to a large extent. These aspects, while making path-finding challenging also make it particularly interesting as they provide a large scope for optimization.

A core piece of the project is the map system. A database of maps, along with a program capable of interpreting them and interfacing smoothly with custom network optimization algorithm was required. Another key preliminary task was to acquire and establish the functionality of a GPS and ensure that it could connect and communicate with the mapping software effectively. These aspects form the platform for the shortest path algorithm.

The routing system developed in this project uses the A* algorithm to compute the shortest path. Although A* is similar to Dijkstra's algorithm, it is biased towards the direction of the destination and gives priority to exploring that section of the road network. While this leads to A* having a shorter run time than Dijkstra's algorithm there are number of properties of the algorithm and the road network, such as highway hierarchies, were exploited to further accelerate path-finding.

Once the run time had been sufficiently reduced in the static domain, the algorithm was applied in a dynamic realm to instantly re-route the vehicle when it went off-course. While re-routing, rather than re-computing the entire route, the algorithm stopped re-computing the route as soon as it added a street to the path that was already on the previously computed optimal path.

With road traffic getting busier and busier each year, traffic congestion especially during rush hour is a great concern to commuters. The route assistance offered by an in-car navigation system can be increased by incorporating traffic updates and avoiding congestion spots. Ideally, these would be live traffic updates, but in the absence of live data, static traffic volume data was used to identify congestion spots around Swarthmore. A congestion factor was developed for the identified spots and used in re-routing vehicles to avoid congestion.

The algorithm was tested at multiple stages during the progress of the project. Maps and algorithms for various stages have been included in the section, *Tests and Results*. The research and project development inspired a number of extensions to the project that could not be implemented due to time and resource constraints; these are mentioned in the section, *Future Extensions*.

TransCAD Information

One major preliminary requirement that had to be taken care of before work could commence on the algorithm, was to acquire the necessary map databases, and a method of interfacing with the data presented in them. In order to allow us to focus our efforts on creating and improving the algorithm, a software package capable of handling this side of the project was acquired. The package selected was TransCAD, a transportation GIS (Geographic Information System) software package produced by Caliper with all of the capabilities required.

One of the particular attractions of TransCAD is its use of the programming language GISDK (Geographic Information System Developer's Kit), a language designed specifically to interface with the functions available from TransCAD with the optimum efficiency. The GISDK works by allowing you to create functions called macros that operate using similar commands to such standard programming languages as C and Visual Basic. Many pre-made macros are provided with the software package that may be called to perform certain standard actions, such as reading information from a GPS (Global Positioning System) unit, or creating a network from a selection of roads. Once a program has been written in the GISDK it must be compiled and run from within TransCAD.

GPS Information

Another major preliminary requirement for the project was the acquisition and setup of a GPS unit to track the current position of the vehicle running the algorithm. The GPS, or Geographic Positioning System, is a satellite navigation system designed by the US military. There are many devices available to the public that are capable of interacting with the GPS satellites currently in orbit. These devices operate by comparing the time signal transmission from four or more of these satellites to find their precise location. The USA also has a system referred to as WAAS (Wide Area Augmentation System), which consists of several ground stations that send a correction signal to WAAS enabled GPS receivers by communicating with satellites that might otherwise be out of range of the receiver, further improving their accuracy. The standard signal format for a GPS unit is NMEA 0183, a specification created by the National Marine Electronics Association.

For this project, a Garmin GPS16-LVS was used. The GPS16-LVS is a WAAS enabled GPS receiver, which consists of an antenna housed in a small package with an RJ-45 cable protruding to transmit any data received to an appropriate receiver using standard RS-232 signals. In order to transmit this data to a computer, the RJ-45 cable had to be converted to a DB9 connection. This RS-232 signal then had to be converted to USB for use with the laptop used for testing. The RS-232 to USB connection was accomplished by simply purchasing an appropriate converter. The RJ-45 to DB9 connected was created manually and has the appropriate wiring information shown below, where the power source must provide from 3.3 VDC – 6 VDC.

RJ-45 Pin Number	Wire Color	Signal Name
1	Red	Power
2	Black	Ground
3	Yellow	Remote Power On/Off
4	Blue	Port 1 Data In
5	White	Port 1 Data Out
6	Gray	One-Pulse-Per-Second Output
7	Green	Port 2 Data In
8	Violet	Port 2 Data Out

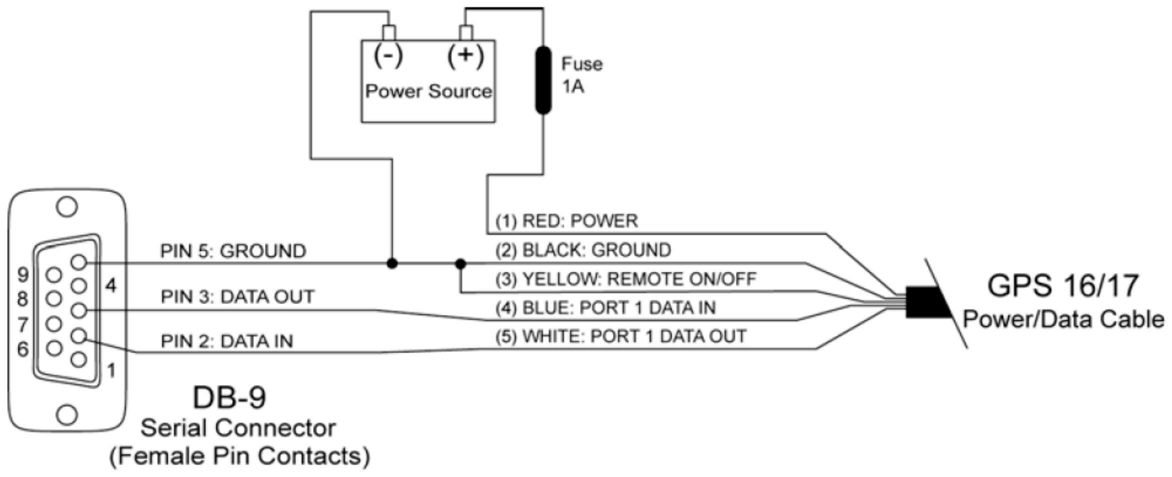


Figure 1: Computer Serial Port Interconnection

Once properly connected to the computer it was relatively straight forward to read the required data from the GPS using the functions provided for the purpose by TransCAD and the GISDK.

Modeling the Map as a Graph

Mathematically, a graph is a collection of vertices connected by a number of edges. The street layer of the map forms a graph, each intersection is a node on the graph and the section of a road between two consecutive intersections is a link. Since the road network includes one-way streets the graph is a directed one. The graph is weighted as there is a cost associated with traversing each link. This cost could be the length of the link, the time taken to traverse it or a combination of factors normalized consistently over the entire network. It is important to note that this graph is connected and does not contain any self-loops or links with zero-cost.

A common variant of the shortest path problem is the *point to point* shortest path problem, where the shortest path between two nodes in a weighted directed graph needs to be determined. The vehicle routing problem has the attributes of the point to point shortest path problem and thus can be modeled and solved using algorithms designed for the point to point shortest path problem.

A lot of research has been done in the field of network optimization that can be applied to vehicle routing once the map is modeled as a graph. Of the numerous path-finding algorithms that are commonly used in network optimization, Dijkstra's algorithm[1] is the most popular. Dijkstra's algorithm works by visiting vertices in the graph starting with the object's starting point. It then repeatedly examines the closest not-yet-examined vertex,

[1] For an explanation of Dijkstra's algorithm, please refer to: http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

adding its neighboring vertices to the set of vertices to be examined. It expands outwards from the starting point until it reaches the goal. Dijkstra's algorithm is always guaranteed to find the shortest path between any two nodes in a network but as it expands equally in all directions, it ends up examining a very large number of nodes in the process. The number of nodes examined can be reduced significantly by biasing the node exploration towards the direction of the destination node. This feature of the A* algorithm makes it more suitable for use in path-finding.

The A* Algorithm

The A* algorithm is a graph search algorithm in first described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael[2] and is commonly used for finding the path from a given start node to a specified end node in a variety of artificial intelligence applications. The algorithm incrementally builds all routes leading from the start node till it reaches the goal node, while functioning as an informed search algorithm and giving priority to building the routes that *appear* to lead towards the goal. To identify the routes that are likely to lead to the destination, it employs a *heuristic estimate*[3] of the distance from any given point to the goal. In the case of a road network, this may be the Euclidean distance.

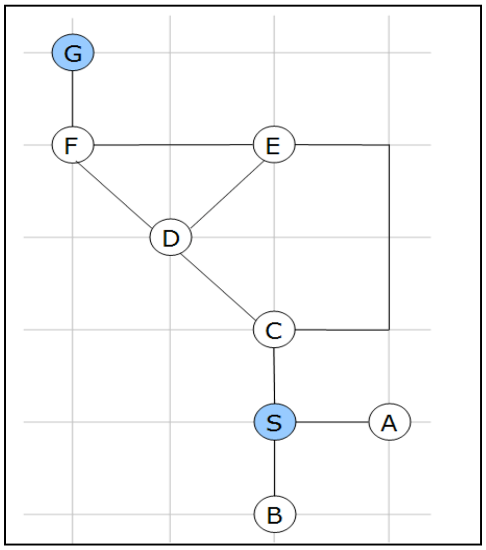


Figure 2.1: Initial network for the A* example

Consider the network shown in figure 2.1, where a vehicle needs to be routed from its start node S, to its goal node G along a path on the given network. The grid in the background helps estimate the distance between nodes. Let the length of a horizontal or vertical edge be 1 mile and the length of the diagonal edge be 1.5[4] miles.

Starting the search
Once the start node has been identified, the algorithm maintains two lists for storing visited nodes and the immediate neighbors of the visited nodes. When a node is visited it gets added to the *closed list* and all nodes adjacent to it are added to the *open list*. So

[2] Nilsson, Nils J. Problem-Solving Methods in Artificial Intelligence. New York: McGraw-Hill, Inc., 1997.

[3]A heuristic is a technique designed to solve a problem that focuses on finding a good feasible solution for the problem in a short span of time, while possibly sacrificing accuracy or precision. Optimization algorithms might exist to solve the same problem that are guaranteed to produce the optimal result but frequently take too long to solve real world problems where data sets are fairly large.

[4] The true length of the diagonal of a unit square is 1.414 but for simplicity the length of a diagonal path is modeled to be 1.5

in the first run through the algorithm, the start node, S is added to the closed list and all its neighboring nodes, A, B and C are added to the open list.

Path Scoring
A* functions similarly to Dijkstra's algorithm except that at each step it selects a node with the least cost based on the following equation:

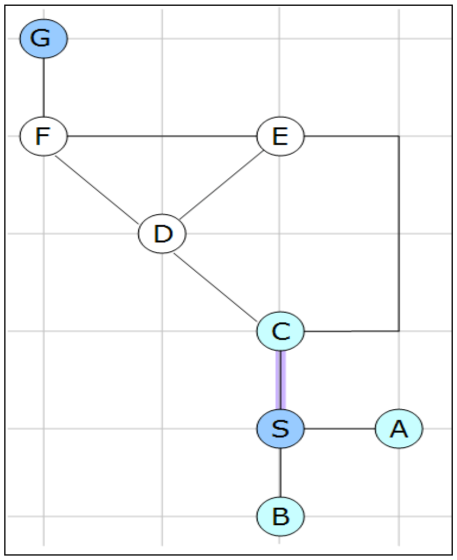
$$\boxed{f(n) = g(n) + h(n)}$$

- g(n) is the actual cost of reaching the current node from the start node
- h(n) is the heuristic cost of reaching the goal node from the current node
- f(n) is the total cost

h(n) can be estimated in a variety of ways. The method for estimating h(n) depends mainly on the type of map, available information about the links and the network and the possibility and extent of pre-processing. There are two main points to bear in mind while selecting a method for computing h(n). Firstly, in order to guarantee an optimal result, h(n) must be an underestimate of the actual distance between the current node and the goal node. Secondly, the closer h(n) is to the actual distance between the current node and the goal node, the faster the algorithm will run. More details regarding the heuristic are discussed later in the section titled *A* and the Heuristic*.

For road networks, the most convenient heuristic to use is the Euclidean distance between the two points. This is the heuristic used in this example and in the actual algorithm used for the project.

Every time a node is added to the open list, information about its parent node and cost function are also recorded. So, for the first run through the algorithm, the nodes A, B and C and information regarding them are stored in the open list as follows.



Current Node: S

Closed List

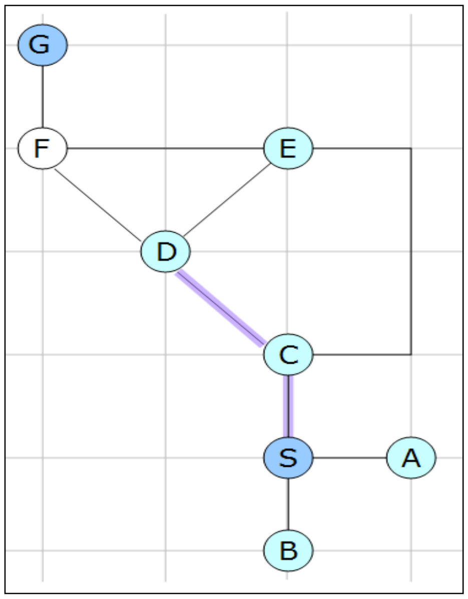
Node	Parent Node	g(n)	h(n)	f(n)
S	-	0	4.47	4.47

Open List

Node	Parent Node	g(n)	h(n)	f(n)
A	S	1	5	6
B	S	1	5.39	6.39
C	S	1	3.61	4.61

Figure 2.2: Iteration 1: Network, Closed list and Open list.

Continuing the search
Once the open list has been populated, the node on the open list with the least total cost (f(n)) is selected as the new current node. This node is now transferred to the closed list, its neighbors are found and added to the open list and their corresponding costs are calculated[5]. This is demonstrated in the second iteration of the algorithm shown in figure 2.3. It is clear that D is the node on the open list with the least total cost and is thus chosen as the next current node.



Current Node: C

Closed List

Node	Parent Node	g(n)	h(n)	f(n)
S	-	0	4.47	4.47
C	S	1	3.61	4.61

Open List

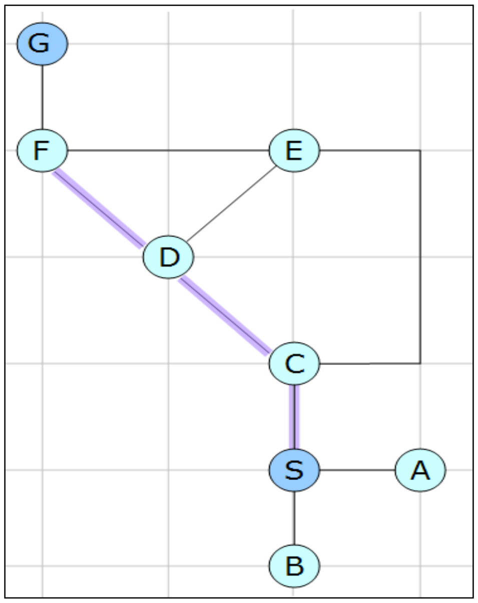
Node	Parent Node	g(n)	h(n)	f(n)
A	S	1	5	6
B	S	1	5.39	6.39
D	C	2.5	2.24	4.74
E	C	5	2.24	7.24

Figure 2.3: Iteration 2: Network, Closed list and Open list.

Before adding a node to the open list, it is essential to check if it was already present on the open list. In that case, the new actual cost, g(n) of reaching that node should be compared to the cost that had been previously recorded. If the new cost is smaller, the information regarding this node should be updated on the open list. If the new cost is larger, the new route to reach that node is ignored.

The case where the information regarding a node is updated on the open list can be seen in iteration 3 of the algorithm. Node E had been added to the open list in iteration 2 with an associated g(n) of 5 miles and node C as its parent node. In iteration 3, an alternate route of reaching node E is discovered (S – C – D – E) with a g(n) of 4. Since this is a shorter route, the information regarding node E is updated on the open list. However, node F has the least total cost and is chosen as the next current node.

[5] Although the diagonal length of a single grid square has been taken to equal 1.5 miles for the sake of simplicity in the calculation of g(n), while calculating the value of h(n) the actual Pythagorean edge length rounded to the hundredth's is considered.



Current Node: D

Closed List

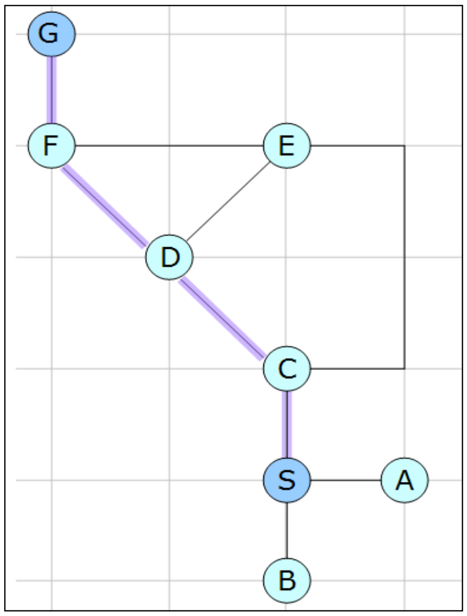
Node	Parent Node	g(n)	h(n)	f(n)
S	-	0	4.47	4.47
C	S	1	3.61	4.61
D	C	2.5	2.24	4.74

Open List

Node	Parent Node	g(n)	h(n)	f(n)
A	S	1	5	6
B	S	1	5.39	6.39
E	D	4	2.24	6.24
F	D	4	1	5

Figure 2.4: Iteration 3: Network, Closed list and Open list.

This process of finding the node on the open list with the least total cost (f(n)), transferring it to the closed list, finding its adjacent nodes and updating the open list is repeated until the goal node is reached. For the above example the goal node is reached in the next iteration. Another interesting point to notice in this iteration is that a third alternate route (S – C – D – F – E) of reaching node E is found. However, this route has a higher g(n) (6 miles) than the existing route (4 miles) and so the information regarding node E on the open list remains unchanged.



Current Node: F

Closed List

Node	Parent Node	g(n)	h(n)	f(n)
S	-	0	4.47	4.47
C	S	1	3.61	4.61
D	C	2.5	2.24	4.74
F	D	4	1	5

Open List

Node	Parent Node	g(n)	h(n)	f(n)
A	S	1	5	6
B	S	1	5.39	6.39
E	D	4	2.24	6.24
G	F	5	0	5

Figure 2.5: Iteration 4: Network, Closed list and Open list.

Once the goal node has been reached the path can be traced by following the parent nodes on the closed list all the way to the start node. For the above example the optimal path found is: S – C – D – F – G with a total cost of 5 miles.

Although the above example is fairly straight forward, applying A* to a larger, more complex algorithm just involves performing the same steps multiple number of times. The algorithm can be summarized by the following schematic.

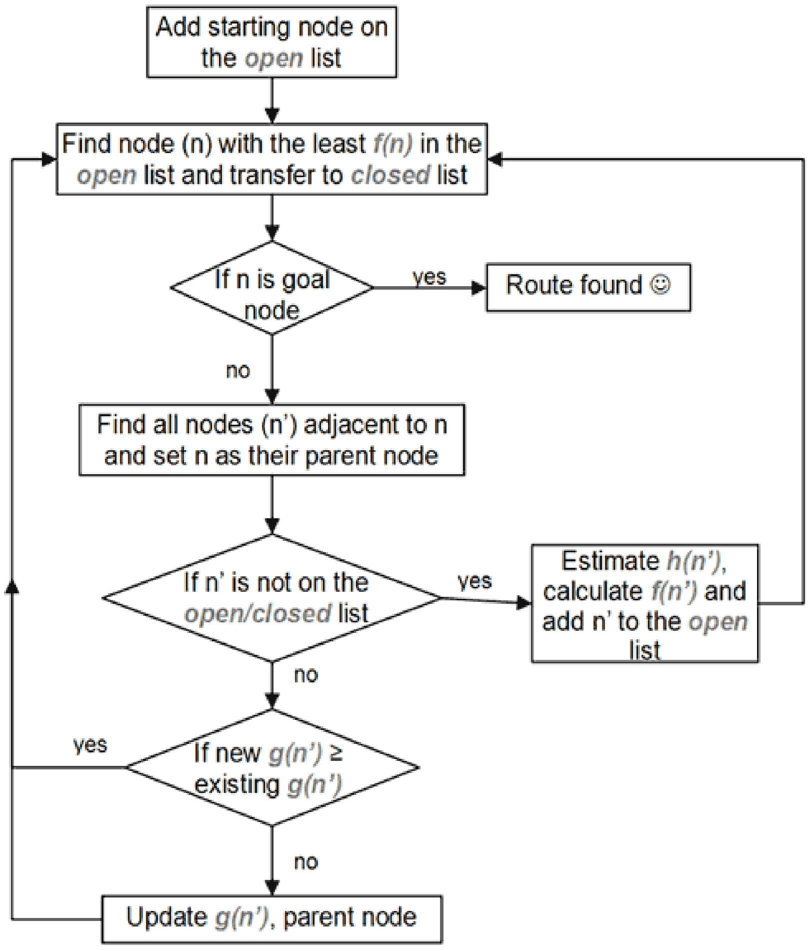

Add starting node on the *open* list
Find node (n) with the least *f(n)* in the *open* list and transfer to *closed* list
If n is goal node
yes
Route found ☺
no
Find all nodes (n') adjacent to n and set n as their parent node
If n' is not on the *open/closed* list
yes
Estimate *h(n')*, calculate *f(n')* and add n' to the *open* list
no
If new *g(n')* ≥ existing *g(n')*
yes
no
Update *g(n')*, parent node

Figure 3: Schematic of the A* algorithm.

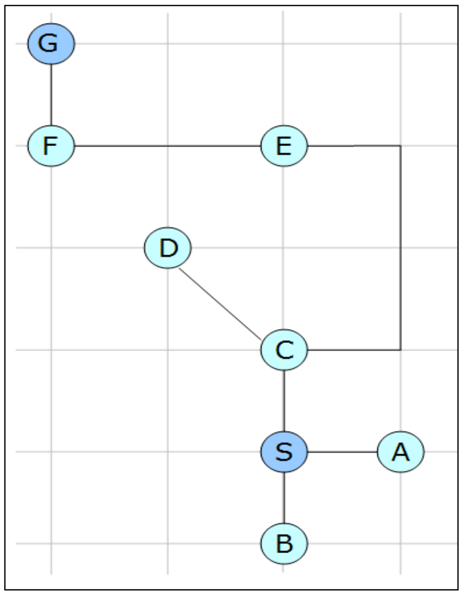


Figure 4.1: Modified network with dead-end

Treatment of Dead-ends

Even while aggressively pursuing a path in a particular direction, the list of nodes on the open list is maintained (without deletion) primarily because in case the path in the pursued direction proves to be going off at a tangent or reaches a dead-end then it can backtrack and find the next best node on the open list. This can be demonstrated using a modified version of the network used in the previous example as shown in figure 4.1.

Notice that links DE and DF that were present in the previous example are missing here and hence node D is a dead-end. Applying A* to this network, the first two iterations are exactly the same as that for the previous example. At the third iteration, node D is transferred to the closed list and it is observed that there are no new nodes to add to the open list as well as no nodes that need to be updated. In fact, the only adjacent node to node D is its parent node, node C and thus node D is identified to be a dead-end. At this stage, the algorithm identifies the next best node on the open list, which is node A. But then realizes that this node is a dead end as well. The next node added to the closed list is node B, which is also a dead-end. Finally the only node remaining on the open list is node E and so node E is chosen as the nest best node. The fifth iteration for this modified network is given in figure 4.2

Current Node: D

Closed List

Node	Parent Node	g(n)	h(n)	f(n)
S	-	0	4.47	4.47
C	S	1	3.61	4.61
D	C	2.5	2.24	4.74

Open List

Node	Parent Node	g(n)	h(n)	f(n)
A	S	1	5	6
B	S	1	5.39	6.39
E	C	5	2.24	7.24

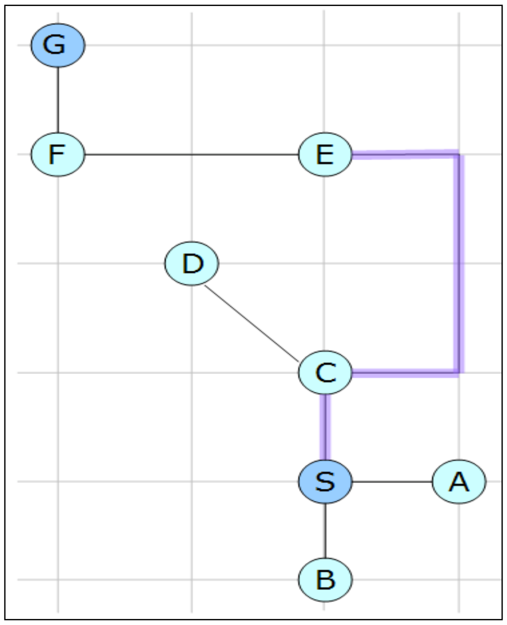
Current Node: A

Closed List

Node	Parent Node	g(n)	h(n)	f(n)
S	-	0	4.47	4.47
C	S	1	3.61	4.61
D	C	2.5	2.24	4.74
A	S	1	5	6

Open List

Node	Parent Node	g(n)	h(n)	f(n)
B	S	1	5.39	6.39
E	C	5	2.24	7.24



Current Node: B

Closed List

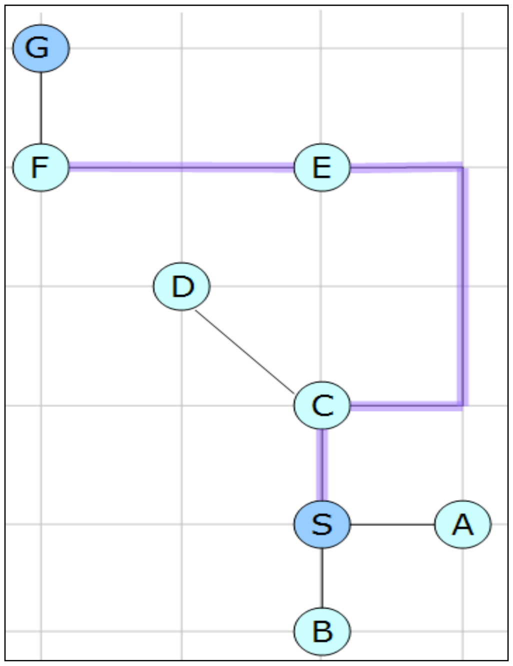
Node	Parent Node	g(n)	h(n)	f(n)
S	-	0	4.47	4.47
C	S	1	3.61	4.61
D	C	2.5	2.24	4.74
A	S	1	5	6
B	S	1	5.39	6.39

Open List

Node	Parent Node	g(n)	h(n)	f(n)
E	C	5	2.24	7.24

Figure 4.2: Iteration 3, 4 (Modified network with dead-end): Closed list and Open list. Iteration 5: Network, Closed list and Open list.

The remainder of the algorithm is fairly straight forward and it can be seen that the optimal path is S – C – E – F – G with a total cost of 8 miles. The open and closed list for both following iterations and the respective network diagrams are shown in figure 4.3



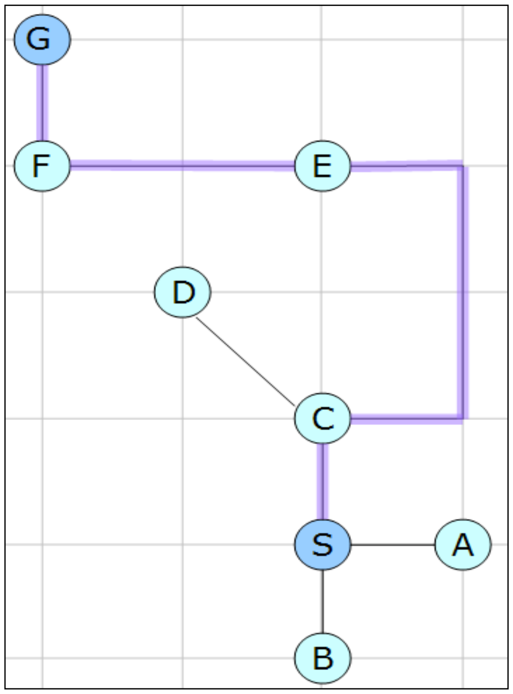
Current Node: E

Closed List

Node	Parent Node	g(n)	h(n)	f(n)
S	-	0	4.47	4.47
C	S	1	3.61	4.61
D	C	2.5	2.24	4.74
A	S	1	5	6
B	S	1	5.39	6.39
E	C	5	2.24	7.24

Open List

Node	Parent Node	g(n)	h(n)	f(n)
F	E	7	1	8



Current Node: F

Closed List

Node	Parent Node	g(n)	h(n)	f(n)
S	-	0	4.47	4.47
C	S	1	3.61	4.61
D	C	2.5	2.24	4.74
A	S	1	5	6
B	S	1	5.39	6.39
E	C	5	2.24	7.24
F	E	7	1	8

Open List

Node	Parent Node	g(n)	h(n)	f(n)
G	F	8	0	8

Figure 4.3: Iterations 6 and 7 (Modified network with dead-end): Network, Closed list and Open list.

This example also demonstrates that being on the closed list does not imply that the node is on the optimal path. In this case, node D is on the closed list but is not on the optimal path.

Real world road networks are very large and in these, the open list can become fairly large even for paths that are relatively small. Searching these large open list to check whether a particular node is already on the list can be very time expensive and can slow down the algorithm (Detailed description of this and possible solutions are presented in section *Accelerating the A* algorithm*). It might be tempting to consider cropping the open list in order to reduce its size. However, this must be done extremely carefully to ensure that if, at any stage, a dead end is reach, there are enough nodes still present on the open list (after cropping) to ensure that the algorithm can find the next best route. The open list was not cropped in any way in this project.

One – way streets are treated in different ways by the algorithm, based on how they can be recognized in the map network. Whenever the traversal of a link is considered in A* the direction of the traversal is know and thus most networks do not recognize one – way streets as a possible link if it were to be traversed in the wrong direction. With a basic familiarity of the manner in which one – way streets are recognized in a road network and a small amount of pre-processing, the possibility of traversing a one – way street in the wrong direction can be ruled out.

A and the Heuristic*

The key characteristic that differentiates A* as an informed search algorithm, from other path-finding algorithms that expand blindly in all directions is its heuristic component. A* with a heuristic of zero is exactly the same as Dijkstra's algorithm. The purpose of the heuristic is to drive the algorithm aggressively in the direction that *appears* to lead to the destination. Thus the choice of the heuristic is a crucial component in establishing the efficiency of the algorithm.

As mentioned earlier, there are two main points to bear in mind while selecting a method to calculate the heuristic cost. Firstly, the heuristic estimate of the cost must always be an underestimate of the actual cost to ensure that the A* search is *admissible*[6]. An admissible search algorithm is one that always terminates in an optimal path to a goal whenever a path exists.

Secondly, the closer the heuristic gets to estimating the actual cost, the fewer the nodes that the algorithm expands[7] and the shorter the run time. But it is important to note that the computational effort required in calculating a more accurate heuristic should not be large enough to nullify the time saved by the use of this more accurate heuristic in the algorithm.

Fastest Path Vs. Shortest Path

In road networks, the route with the shortest path length is not always the fastest route. In fact, the route with the shortest path length is primarily composed of minor roads with a large number of turns, stop lights and low speed limits that cause it to be an inconvenient and inefficient route to use. Thus in the case of vehicle routing in a road

[6] For a detailed proof please refer to page 78 in: Pearl, Judea. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley Publishing Company, Inc., October 1985.
[7] For a detailed proof please refer to page 81 in: Pearl, Judea. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley Publishing Company, Inc., October 1985.

network it is more useful to find the fastest route rather than the route with the shortest path length.

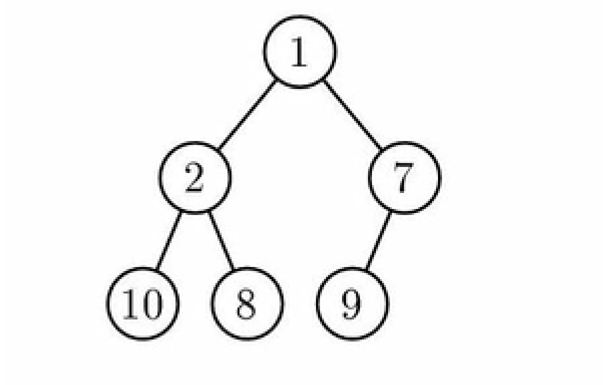
This involves changing the cost for each link from the length of the link to the time taken to traverse the link. If the speed limits on the links are known, the time taken to traverse the link can easily be calculated by dividing the length of the link by the speed limit of the link. It is very important to ensure that the heuristic is changed to be a time estimate as well. In this project this step was performed in conjunction with layering the map and its details can be found in the section titled *Accelerating the A* algorithm*.

Accelerating the A* Algorithm

The A* is by its nature an efficient algorithm for solving a shortest path problem on a large network, but there are certain tasks that it has to perform which can be optimized in order to produce the best results possible. We focused our efforts on two specific methods of speeding up the A* algorithm.

One of the primary processes that slows down the A* algorithm is the processing of the open list. On each pass of the algorithm the node on the open list with the smallest cost must be found, and when thousands, or tens of thousands of nodes are being dealt with this process can be quite time consuming. Therefore, making this process as fast as possible was of great importance to us. It was therefore decided to sort the nodes as they were added to the open list.

Since the node of most concern to us with each pass is the node with the smallest cost, it was decided to use a binary heap to store the nodes in the open list. A binary heap sorts its contents into a heap, where the nodes in each layer have a higher value then the nodes connected to it on the preceding layer. A typical binary heap might take the form of the one shown below in Figure 5.1, while an unsorted list of the same data stored in an array might take the form of the array shown in Figure 5.2. While it would be necessary to check each item in the array to find the node with the minimum cost, with the binary heap it is simply the first node.



1	2	7	10	8	9

Figure 5.1: Binary Heap

Figure 5.2: Array

The sacrifice comes with the fact that additional calculations must be performed each time a new node is added to the heap. These calculations, however, will in logarithmic time instead of linear time. As can be seen Figure 5.1, if a new node were inserted in the binary heap it would be possible to do so by simply placing it in the empty slot at the bottom of the heap, then swapping it with the 7 if it were smaller, and once again with the 1 if it were smaller then the 1, thereby bypassing most of the list. For relatively small lists this difference is insignificant, but once you start dealing with large numbers of nodes, such as the road networks that we are working with, the difference is much more tangible.[8]

Another method of speeding up the algorithm that was pursued was to treat the map as having several layers defined by road speed. Using this method, major roads are placed on a different layer then minor roads. Thereby producing a much smaller network where the majority of the calculation can be performed. The grids in Figure 6.1 illustrate how this might appear on a much smaller scale then that which we are dealing with.

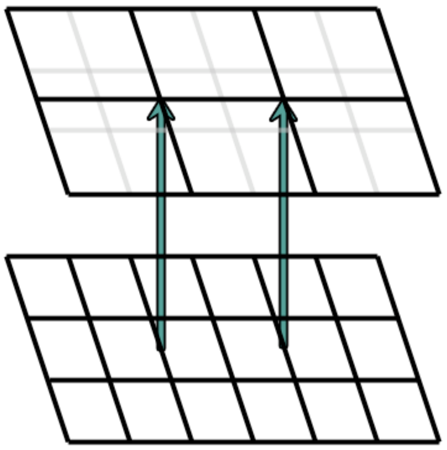


Figure 6.1: Layering

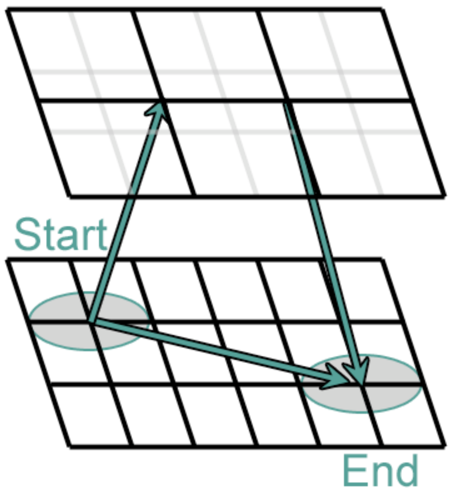


Figure 6.2: Layered calculation

The algorithm starts by searching for major road entrances in the vicinity of the start point of the trip, and major road exits in the vicinity of the end point of the trip. A network is then created using heuristic approximates to connect the start node to the nearby major road entrances, and the end node to the nearby major road exits. An additional approximate is then made connecting the start and end nodes. These heuristics are weighted heavily in favor of the major roads. The route between the major road entrances and the major road exits is calculated in the major road layer. The shortest path calculation is performed on this network. A basic illustration of how the two layers might be connected in this manner is shown in Figure 6.2. It is important to note that when using this approximation we are no longer guaranteed to get the shortest path. Even slight variation in the weighting of the heuristics can produce different results

[8] For more detailed information about using heaps see http://www.policyalmanac.org/games/binaryHeaps.htm

If the heuristic connecting the start and end node is the shortest path, the calculation is performed exclusively in the minor road layer. If one of the major road paths produces the shortest route, the portion of the route on the major roads becomes the middle component of the route, and beginning and end components connecting the start and end nodes to the major road entrance and exit, respectively, are calculated in the minor road layer, making up the beginning and end of the trip. Using this method, the calculation for the bulk of the trip is done on a much smaller network, significantly speeding up the time that the algorithm takes to find the shortest route.

One severe limitation posed to our ability to speed up the algorithm was in fact the GISDK itself. In order to fully understand these limitations, it is first necessary to have a basic understanding of the way that TransCAD handles maps. In TransCAD, maps are stored in several layers, the streets are on one layer, the intersections are on another layer, the railroads, railroad intersections, and water areas each have their own layers too. It is therefore necessary to switch between layers to acquire the data associated with any of them.

The TransCAD network is a proprietary structure that can be used to simplify this process by creating a network that incorporates the link and node layers of the specified type. The links connecting the nodes in such a network are weighted using a specified piece of data stored in the corresponding layer. So if for example, a network were created for the street layer, you could use the length of the streets, or some derivative of their length, such as length*2 as the cost of the links. Unfortunately access to the network is limited, making it impossible for a normal user to make any significant modifications to a network once it has been created. It is therefore necessary to switch from the network back to the associated link on its accompanying layer if any additional data is sought.

TransCAD has a built in shortest path method that differs slightly from our own, and is designed to make optimal use of the road networks created by TransCAD. However, since information for direct viewing and manipulation of these networks is limited for us, several roundabout methods had to be used to make our calculations on the network. These methods included, but were not limited to, frequent switching back and forth between the map and network layers in TransCAD. Another handicap posed was an inability to add any sort of flag or the like to nodes in the TransCAD network, to determine whether or not a node had already been visited. However, these factors were alleviated to a great extent after a discussion with the Caliper technical support staff that resulted in the addition of several functions to the GISDK, allowing for a greater level of control over the network and network flags.

Four new functions were added to the GISDK after the discussion. One of the functions introduced by Caliper allowed for a much greater portion of the calculation to be done exclusively in the network layer. This resulted in a significant speedup of our algorithm, as it allowed for several time consuming switches between the different information structures in TransCAD to be cut out of the algorithm. The remaining three functions allowed the flags built into the TransCAD network to be manipulated by users of the GISDK. By using two of these flags at each node, it was possible to keep track of whether a node was on the open list, on the closed list, or had not been visited. This allowed the time consuming process of searching these lists to make sure a node had not been used before being added to the open list to be bypassed.

The one limitation still posed is that each time a node is found on the open list, the open list still has to be searched for the node in order to check whether the new method of getting there is faster then the old. This process is still fairly time consuming, because the data in the open list is sorted in terms of the node cost, and not the node ID. As a result a linear search has to be performed to find the appropriate node in the open list. While it would be possible to make a compromise between sorting the nodes by the node cost and the node ID, it was deemed more important to minimize the time required to find the node with the minimum cost, as it is required that this node be found on each pass of the algorithm, whereas a node already on the open list is only encountered for a small portion of the calculations done. This once again might be possible to solve if sufficient control over the network were provided for the cost of each node to be stored at the node, preventing the need to search the open list.

Dynamic Route Updates

To make an efficient in car navigation system, the A* algorithm cannot be run every time the car moves. The A* algorithm is a static route calculation algorithm, and while efficient, still requires time and processor power to run. It is therefore necessary to design the navigation system so that the A* is run as little, and as efficiently as possible. The dynamic aspect of the program has therefore been divided into three primary phases.

The first stage occurs when the user specifies their desired destination. After a destination has been specified, an appropriate network for performing the calculation is chosen. An initial route calculation is then performed using the A*, and a timer is started that checks the car's current location every few seconds. If the car is not already on the route chosen by the A*, the program requests that the use find their way to the specified route. Once they have reached the first link of the specified route, the second stage of the dynamic aspect of the program begins.

In the second stage, there are two primary states that exist. The first occurs if the user is found to be on route. If this is the case, the location of their car is updated on the map, and, if the user has just passed an intersection, the appropriate intersection is removed from the path. If the user is found to be off the specified route, they must be re-routed. In order to re-route a user, the A* algorithm commences again from the user's current location. The calculation is performed until the new path reaches the destination or intersects the old path. Therefore if the user is headed in completely the wrong direction, the form of the new guidance will probably simply be to tell the user to turn back and take the same route as specified before. If the user were to turn onto a road that ran parallel to a road that was part of the specified path however, the new guidance might ask the driver to proceed in the same direction and take the next turn.

The final stage occurs when the user reaches their specified destination. When this occurs, a message is displayed informing the user that they have reached their destination. After this message appears, the timer keeping track of the car's status is shutdown, and the program ceases to update until the user specifies a new destination.

While the dynamic aspect of the program is currently working, there were several limitations to its implementation. One of these limitations was that we were only able to implement dynamic routing for the minor road layer of the program. This problem occurred because the minor and major road layers in TransCAD did not line up exactly. It was therefore extremely difficult to meaningfully translate between the two layers.

Another major limitation encountered was that the street coordinates specified in TransCAD did not line up exactly with the coordinates given by our GPS receiver. As a result, the user's current position had to be approximated by snapping their current location to the nearest link. This issue, while it did not pose any serious problems on our test runs, could cause issues with tightly packed roads and intersections.

One major improvement that could be incorporated to help handle this issue would be to implement error checking to make sure that user's location did not suddenly jump from one road to another, and otherwise followed a reasonable pattern. Another improvement that could be incorporated would be to develop a method of keeping track of a user's proximity to an intersection. If the user were not near an intersection, it would be possible to simply update their current location. When they were approaching an intersection would be the only time that the program would have to worry whether the user were on route or not. Implementing this would also make it possible to warn the user about any upcoming turns they need to take.

Acquisition and Application of Traffic Data

One of the aspects of our project that makes it different from other shortest path algorithms currently in existence is that it incorporates traffic data into the best route calculation in order to ensure that the shortest route is chosen. We had an E14 group that helped us with the first step of this process. Their task was to locate an appropriate source of traffic data and find a way to convert it into a useful format.

After an extensive search it was determined that no live traffic data was available for the Philadelphia area. While various news and radio channels provide live traffic updates by flying helicopters over the area, none of these sources had the information available in a format useful to us. With no live traffic data available for the Philadelphia area, the traffic conditions in the area had to be simulated. It was found that the Delaware Valley Regional Planning Commission had static traffic volume data from the past several years for the local roads in much of the Philadelphia area and the surrounding counties.

Unfortunately there was insufficient data for an in depth traffic analysis. Only one or two day's worth of data was available for most road segments. Therefore it was decided to handle the data by attempting to identify local streets with heavy traffic using the data available. These streets were then assigned a congestion factor based on the traffic volume data available for the link. The congestion factor was based upon the traffic flow on a road in a given hour, and the speed limit for the road. This data was then stored so that for a given 24-hour period, each link identified as having heavy traffic was identified by its corresponding link ID, followed by a congestion factor for each hour. This data was then stored in an appropriately formatted text file.

When the program initializes itself it imports the traffic data in the form of an array containing each link ID identified as containing heavy traffic, accompanied by the appropriate set of 24-hour congestion factors. Each time a node is added to the open list, the ID of the link just taken is compared with the IDs for which traffic data is available. If the link ID is found, the appropriate congestion factor is chosen based on the current time of day, and is applied to the road. Congestion factors take the form of a simple number from 1-1.75. To apply this congestion factor to a link, the cost of the link is simply multiplied by the congestion factor.

While it would be possible to incorporate data for every link in a network using the method described above, the process of searching through the traffic data each time a new link is added to the open list is quite time consuming. This process could be sped up significantly if it were possible to store this data in the network nodes when the program was initialized, and then simply update the nodes as traffic data became available. Using this method, the data would be immediately available when a node was visited, rather then searching an array containing all of the traffic data each time a node was added to the open list.

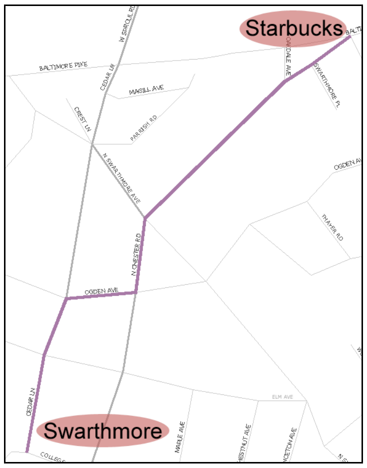
If live data were used, the network could be updated periodically, and scanned for any major changes in link weights. If anything that directly affected the current route were found, the route could then be correspondingly updated to display a better route to the destination. Unfortunately the simulated data changes with each hour rather then constantly updating, and while it may provide a good approximate of traffic flow, it will not provide the actual conditions that exist for any given day. However, when used appropriately it is still possible to produce a reasonable approximation of what the traffic conditions for a given day might look like.

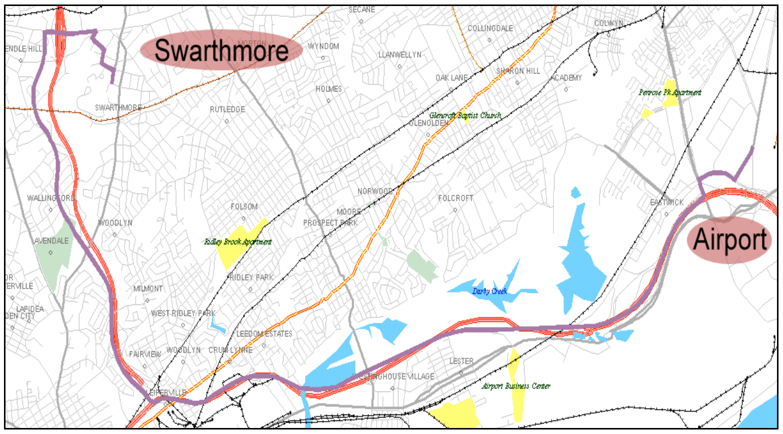
Tests and Results

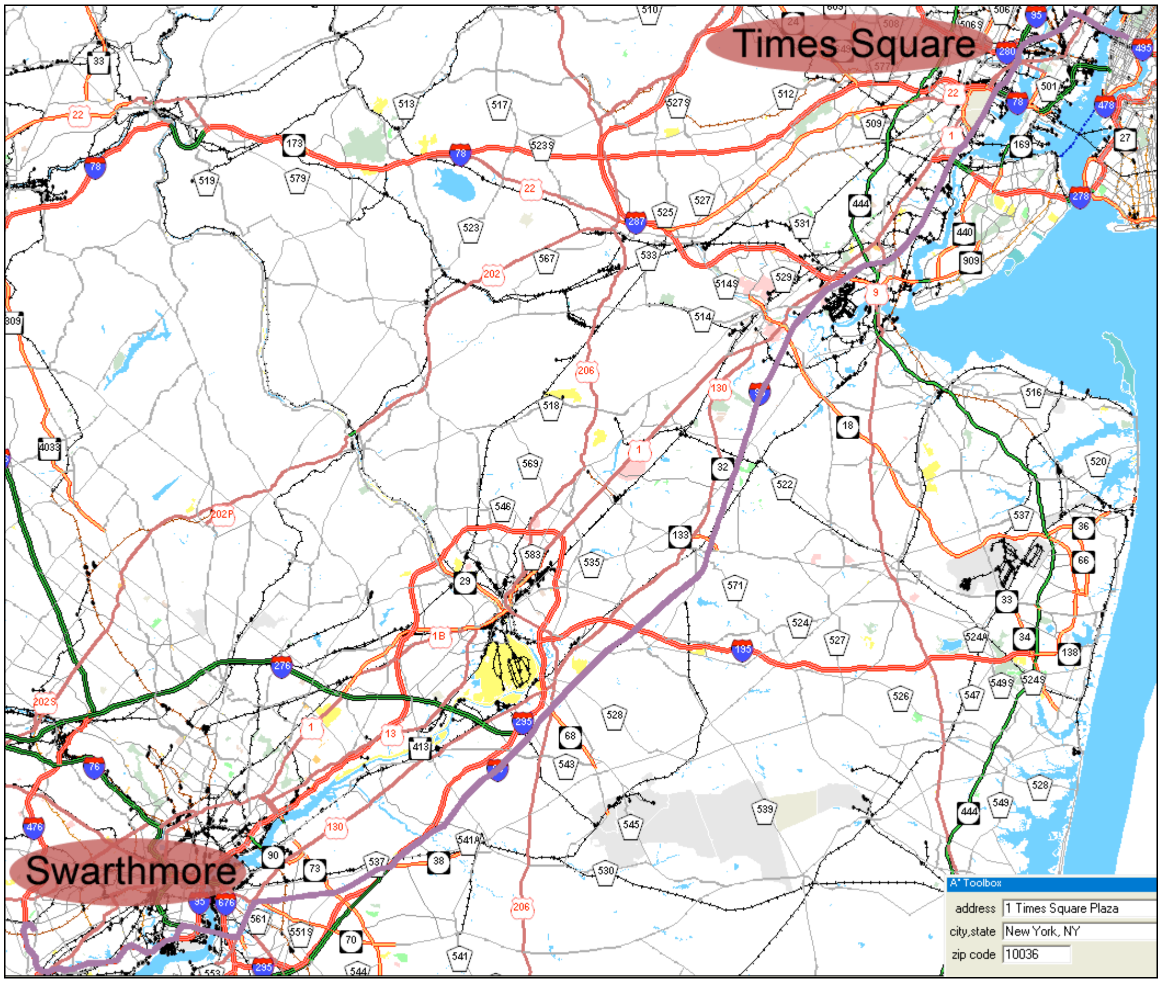
Two main measures of algorithm performance were used to establish the efficiency of the A* algorithm at the various stages of acceleration. The first is a simple run time test to observe how the run time of the algorithm for a particular route decreases through the various stages. The next is a more output sensitive machine, independent measure of efficiency based on the number of nodes explored by the algorithm relative to the number of nodes actually on the route. A number of maps showing routes for theses tests, produced in TransCAD, using the specified code are presented in this section to give a better idea of the capabilities and limitations of the software and algorithm system.

For all the above tests, the results are demonstrated through three routes of varying length. All routes start from Swarthmore College[9]. The first is a short route, less than a mile and simple, consisting of only two turns, to the local Starbuck Coffee Shop. The second is a longer one, over 10 miles, to the Philadelphia International Airport and the third, is fairly long, over 150 miles, to Times Square in New York. A map of the route (as established using the final algorithm), along with the address used for each of the above routes is given in figure 7.

[9] Swarthmore College: 500 College Avenue, Swarthmore, PA 19081







1 Times Square Plaza
New York, NY
10036

Figure 7 (top left to bottom): Route to Starbucks Coffee Shop: 1100 Baltimore Pike, Springfield, PA 19064. Route to Philadelphia International Airport: 8000 Essington Ave, Philadelphia, PA 19153. Route to Times Square, New York: 1 Times Square, New York, NY 10036

Run Time for the Algorithm

All tests to measure the run time of the algorithm through various stages of programming were performed on a 2.8 GHz Windows machine with 512 Mb of RAM. The same version of the algorithm was used across all. The results of these tests are summarized in table 1.

Table 1: Algorithm run times in seconds for three routes and four versions of the algorithm

Destination	***Route Length***	***Basic A****	***A* with Binary Heaps***	***A* with Layering***	***A* with New Functions***
Starbucks	0.71	0.594	0.594	0.594	0.047
Airport	8.076	681.887	607.639	5.078 *(12.67)+*	0.383 *(12.67)+*
Times Sq	116.36	-	-	-	8.062

$^{+}$: route length for alternate route using layering
All route lengths are in miles
All run times are in seconds

The first route is short and simple enough to achieve a good run time results within the basic A* algorithm. Since it is so short, the associated open and closed lists are small as well and so adding binary heaps does not decrease the run time. Also the route can be calculated easily within the minor road layer itself and so adding layering does not speed up the algorithm either. Adding the new functions definitely helps, as this allows easier access to the network. The route to the airport is the one that shows consistent acceleration throughout the programming. It starts off with a run time of over 11 minutes and reduces to that of less than 0.5 seconds. For this route, the first two versions of the algorithm establish the route along the shortest distance links, with a total route length of 8.1 miles, while the latter two versions use the major road network as well, and then the route length increases to 12.7 miles. The third route was just established using the final method, since the results of the second indicate that it would take a considerably long time to use any other version.

Table 2: Run time comparison of A* to Dijkstra

Destination	***Dijkstra's Algorithm***	***A* Algorithm***
Starbucks	0.25	0.031
Airport	80.456	11.309

One of the main reasons for choosing the A* algorithm over Dijkstra's algorithm was that the former is an informed search algorithm and so would be faster than later. The run times shown in table 2 illustrate that this is indeed the case. The version of the A* algorithm used here is the final one, with the new macros, but it only uses the minor node layer. This was done, to ensure easy and valid comparison with the results of Dijkstra's algorithm. This also implies that the numbers in table 2 will not coincide with the numbers in table 1 since a version with binary heaps and the new functions but without layering was not one of the test cases included in table 1. For both routes, A* is at least 7 times faster than Dijkstra.

Nodes Expanded

Consider the following expression to measure the efficiency of a point to point path finding algorithm:

$$\text{Efficiency} = \frac{\text{Nodes on Route}}{\text{Nodes on Open and Closed Lists}} \times 100$$

Since the size of the open/closed lists was a major concern targeted to be reduced through various acceleration methods, the above expression can also be used to examine the extent to which this was done. This measure is also interesting for two other reasons: it is output sensitive and machine independent.

Table 3: Efficiency comparison for Dijkstra, Basic A* and A* with layering

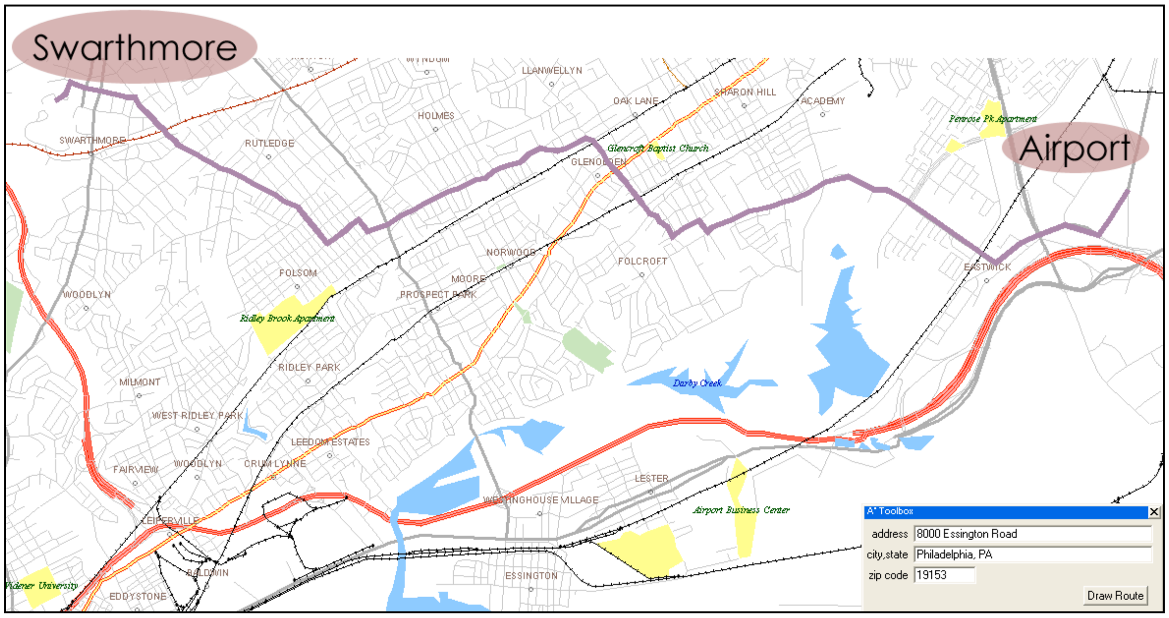
		Dijkstra's Algorithm			***Basic A* Algorithm***			***A* with layering***		
	Route	*Nodes on*			*Nodes on*			*Nodes on*		
Destination	***Length***	*Route*	*Lists*	*Efficiency*	*Route*	*Lists*	*Efficiency*	*Route*	*Lists*	*Efficiency*
Starbucks	0.71	8	126	6.35	8	25	32	8	25	32
Airport	8.076	126	16343	0.77	126	2995	4.21	42	207	20.29

The three particular algorithms compared here were, Dijkstra, the Basic A* and A* with layering. Other accelerations of the A*, such as A* with binary heaps and A* with new functions were not included since they would not affect the number of nodes. Both the A* algorithms are certainly better than Dijkstra's algorithm, since they both examine fewer nodes than Dijkstra. It is interesting to observe that the number of nodes examined by Dijkstra were considerably larger than those by A* even when they both had the same number of node on the route, indicating that they were both finding the same path. For the Starbucks route, the results for Basic A* and A* with layering are the same since, as discussed in the previous section, this route so short that it is unaffected by layering. For the airport route, there is a visible difference between the Basic A* and A* with layering as the later views a lot fewer nodes and the number of nodes on the route decreases as well.

Pre-layering and Post-layering maps

Incorporating layering in the A* algorithm produces two main effects. Firstly, as documented above, it speeds up the algorithm by forcing it to find the route while viewing fewer nodes. Secondly, in the major road layer, the cost factor used was travel time rather than link length, and this forced the algorithm to prefer routes with major roads, such as highways to routes that just went along minor roads, especially for longer distances.

Figure 8 shows two maps of the route to the airport. This first map has the route along the shortest distance path highlighted in purple. The second map adds another path highlighted in blue, this is the path along the shortest travel time found using layering. Notice that the route highlighted in blue does not completely line up with certain sections of the major roads. This is the problem referred to earlier as well, that the minor road layer and major road do not always match up.

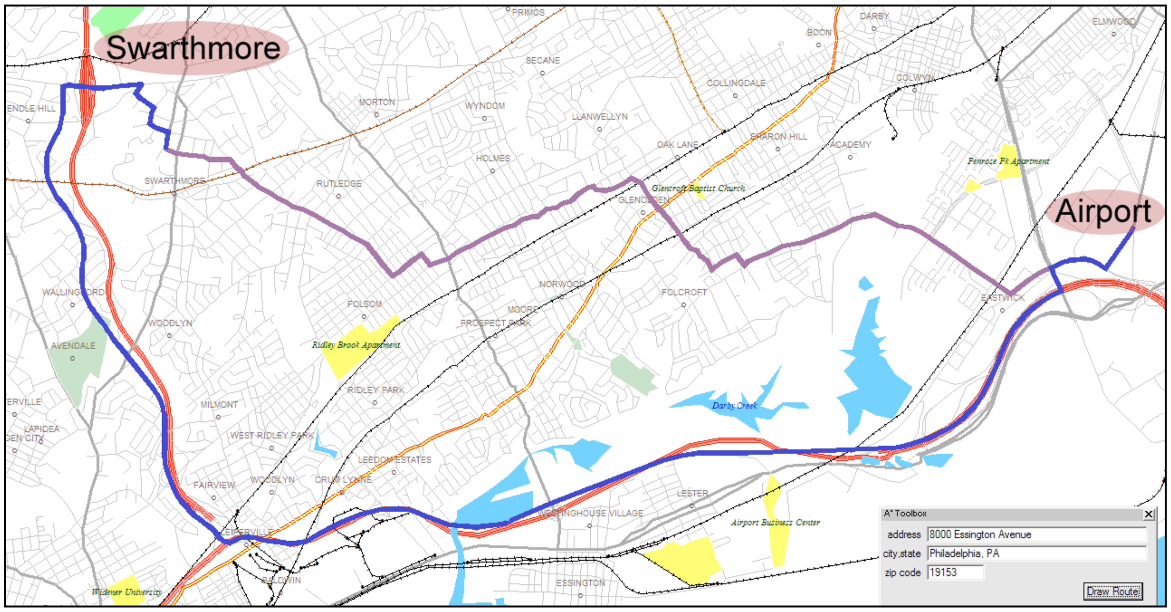


A* Toolbox

address 8000 Essington Road

city,state Philadelphia, PA

zip code 19153



A* Toolbox

address 8000 Essington Avenue

city,state Philadelphia, PA

zip code 19153

Figure 8: (Top to Bottom) Map showing shortest distance path found without using layering. Map showing both, the shortest distance path (purple) and major road path (blue), the later, found using layering.

Dynamic Re-routing and Traffic Incorporation

On the *dynamic platform*, a number of tests were performed to establish the route tracking and re-routing features of the system for local routes. The system successfully started by asking the driver to enter the postal address of the desired destination and obtained the current location of the vehicle using the GPS receiver. It then calculated the optimal route on the static platform and asked the driver to proceed to the highlighted rout, in case the search had started out in a parking lot, for example. Each time a new route was calculated, the system reminded the driver to proceed to the highlighted route every 10 seconds till he reached the first link on the route. From that point on, the system successfully tracked the position of the vehicle and updated it in the image on the navigation screen. It was observed that, for a number of roads co-ordinates from the tracking system did not coincide with the GPS co-ordinates from the mapping software. This problem was successfully resolved for the minor road layer and the route almost always snapped to the correct route.

The re-routing capabilities of the algorithm were also tested successfully for local roads. The system was accurate and efficient in detecting that the vehicle had gone off course and then re-routing it to reach the desired location. Having incorporated the traffic data, alternate recommended optimal routes were observed based on time of day and availability of traffic data for links along the route. On the re-routing platform, at the change of the hour, if the traffic pattern for certain links on the current route, or alternatives to current route changed, a new optimal route to the desired destination was recommended from the current location of the vehicle.

Future Extensions

Although the current version of the algorithm has been enhanced by a number of accelerations beyond the Basic A*, there are a number of other techniques that can be used to further optimize the working of the algorithm. Two that were research actively but could not be implemented due to lack of time and accessibility to appropriate network functions were the Bi-directional A* and the ALT algorithm described below. Another possible extension to the project can be to develop a simulation to predict traffic flow using the Ant Based Control model rather than relying in static or live traffic data. Also, the possibility of developing a comprehensive route guidance interface for the system could prove to be a very interesting project and help increase the utility of the system.

Bi-directional A*

One of the primary factors that slowed down the algorithm, especially for long routes through dense networks was the search through the open list. We were able to use binary heaps and efficiently find the node on the open list with the least total cost, but still had to scan the list for each neighbor of each current node to determine whether it was already present on the open list. Thus the size of the open list is a large factor in determining the run time of the algorithm.

The size of the open list can be considerable reduced by employing a bi-directional search: one starting from the start node and the other starting from the goal node simultaneously. This might seem trivial and easily realizable by using $h_s(v)$ as the heuristic estimate of the distance from any node v on the forward search to the goal node and $h_g(v)$ as the heuristic estimate of the distance from any node v on the backward search to the start node. But the algorithm is complex and cannot specify the shortest path when the searched areas from both sides overlap each other. This is due to the fact that the algorithm utilizes independent estimators for the two searches.

This problem can be overcome in two ways: by the *consistent approach*, that uses consistent heuristic functions for the two searches, or the *symmetric approach* which uses a new termination condition. The former restricts the choice of the heuristic to a large extent and can terminate when the two searches meet while the latter can use the best available heuristic but cannot terminate when the two searches meet.

The challenging aspect of the consistent approach is selecting a function that is consistent. One way to approach this is by modifying the link length definitions. Instead of using the actual length of the link as the cost, the following functions can be used;

Forward search: $l'(u, v) = l(u, v) + h_s(v) - h_s(u)$

Backward search: $l'(u, v) = l(u, v) + h_g(u) - h_g(v)$

Where: $l(u, v)$ is the actual length of a link from node u to node v,

$h_s(v)$ is the heuristic estimate of the distance from the node v to the goal node
$h_s(u)$ is the heuristic estimate of the distance from the node u to the goal node
$h_g(u)$ is the heuristic estimate of the distance from the node u to the start node
$h_g(v)$ is the heuristic estimate of the distance from the node v to the start node

These functions are equivalent to the original A* algorithm and so the rest of the properties of A* still hold[10].

The symmetric approach runs the forward and the backward searches in an alternating manner. Every time the forward search scans a link (u, v) such that v has already been scanned by the backward search, the paths s-u and v-g are concatenated and this new s-g path is compared with the best s-g path found so far. If it is shorter, it is recorded as the new best s-g path. The search terminates when either search is about to scan a node for which the f(n)[11] is smaller than the shortest path found so far, or when both searches have scanned all the nodes on each open list. This algorithm is correct because by the time the algorithm reaches either of the two termination conditions, it is guaranteed to have found the optimal route[12].

ALT Algorithms

Another effective way of accelerating the A* algorithm is by using a better heuristic function that gives an estimate of the distance between the current node and the goal node which is closer to the actual distance. The ALT is a preprocessing based algorithm that computes better distance bounds. The preprocessing consists of carefully choosing a fixed number of *landmarks* (nodes on the given network), computing and storing the shortest path distances between all the vertices and each of these landmarks. This algorithm is called ALT because it is based on the A* algorithm, landmarks and the triangle inequality.

The most crucial aspect of this algorithm is landmark selection. Firstly, the number of landmarks to be selected needs to be determined. It has been observed that the larger the number of landmarks, the shorter the runtime of the algorithm. The simplest way to select landmark nodes is to just select the specific number of nodes in the network at random and denote them to be the landmarks. However, having a landmark in close proximity of another does not convey much additional information. For road networks, a simple landmark selection process works as follows. Identify a node, c, as close to the center of the network as possible, and then divide the network into k (pre-determined number of landmarks to be selected) pie-sectors centered at c and contained roughly an

[10] For a proof and more details about the consistent approach, please refer to:
Ikeda, T, et al. "A Fast Algorithm for Finding Better Routes by AI Search Techniques." Proc. Vehicle Navigation and Information Systems Conference. IEEE (1994)
[11] As defined in the original description of the A* algorithm.
[12] For details about the symmetric approach, please refer to:
Pohl, I. "Bi-directional Search." Machine Intelligence (6). Edinburgh: Edinburgh University Press, 1971. 124 – 140

equal number of nodes. Then, pick a vertex in each sector that is as far away from the center as possible[13].

Ant-Based Control Algorithm
The key aspect of dynamic vehicle routing based on current road conditions is the frequency and accuracy of the live traffic updates. The Ant-Based Control Algorithm, popularly known as the ABC algorithm, is based on the natural behavior of ants. In order to find the shortest route from their nest to a food source, ants lay a chemical track (pheromones) along the path they travel. The ants that follow track the pheromone level and follow the path with the highest concentration. The idea is that the routing units within cars can function as mobile agents to observe and communicate traffic conditions along their routes to a control center. This control center collects data from all agents, processes it and transmits it to the relevant vehicles. The individual routing units in the vehicles receive these updates and re-route the vehicles when applicable.

Such an intercommunicating car navigation system is being heavily researched, but is not being actively used commercial as yet. Of the multiple approaches that are being researched, the following two seem to be the most popular ones. The first is one that focuses on inter-vehicular communication using wireless LAN and does not require the service of a central control unit. Here the route is predetermined using a vehicle routing algorithm and while the vehicle is following the route, it constantly receives signals from other vehicles that have traveled the same route or possible alternate routes. It also sends its own travel time and speed to relevant vehicles[14].

The second approach is one where the actual route is determined on a node by node basis based on pheromone deposits for the particular node. This approach is targeted for vehicle routing within a city. As vehicles travel through the network, the pheromone they deposit is a function of their travel time. That time is influenced by the congestion encountered on their journey. Based on the pheromone deposits of multiple vehicles for a particular node, the probabilities of reaching neighboring nodes in the network from that node are calculated. When a vehicle arrives at a node, these probabilities influence its selection for the next node[15].

Route Guidance Interface
The goal of good route guidance is to pilot the driver along a path with advice that is clear, informative and does not distract the driver and ensures a safe and efficient journey. Ideally this would be done through a series of audio and visual advices to the driver. The voice messages should be clear and concise, such that their interpretation is simple. To ensure that the driver spends as little time viewing the LCD as possible, the visual directions should be pictorial images of approaching turns with any text (such as street names) in bold, easily legible font. The timing of these presentations is crucial

[13] For more details about ALT algorithms, please refer to:
Goldberg, Andrew V. and Harrelson, Chris. “Computing the Shortest Path: A* search meets Graph Theory.” Proc. 16th ACM-SIAM Symposium on Discrete Algorithms (2005): 156 – 65
[14] For more information please refer to:
Hiraishi, H, Mizoguchl, F. and Ohwada, H. “Intercommunicating car navigation system with dynamic route finding.” IEEE (1999): 284 – 9
[15] For more information please refer to:
Kroon, Ronald and Rothkrantz, Leon J.M. “Dynamic vehicle routing using an ABC-algorithm.” Proc. of the 15th Belgium-Netherlands conference on Artificial Intelligence (2003): 211 – 8

such as to allow the driver sufficient time to slow down before turns as well as avoid taking the wrong turns.

Research is also being done regarding the effectiveness of advice based on landmarks rather than just street names. The problem faced by a number of drivers is that the street names are not legible early enough to allow sufficient time to prepare to take a safe turn. Instead if the audio advice received is of the form “Turn left the approaching intersection with the Exxon gas station on the left” the landmark is visible a great deal earlier and the driver has a lot more time to prepare for the turn.

Acknowledgements

- Our advisor Professor Erik Cheever, for his invaluable guidance throughout the project
- Edmond Jaoudi, for his help in making the necessary connection modifications with the GPS receiver
- E14 students: Omer Corluhan, Seth Hara and Patrick Lindsey for acquiring and pre-processing static traffic data
- Patrick Lester and Amit Patel, for their correspondence with us regarding the implementation of A* in road networks
- TransCAD, for providing us with licensed copies of the software, tirelessly answering our questions and providing us with four new functions
- The Delaware Valley Regional Planning Commission for providing us static traffic data

Bibliography

Arai, H, et al. “Automotive navigation display systems.” SID – International Symposium – Digest of Technical Papers (1998): 325 – 8

Cain, Timothy. “Practical Optimizations for A* Path Generation.” AI Game Programming Wisdom. Ed. Steve Rabin. Hingham, Massachusetts: Charles River Media, Inc, 2002. 146 – 52

Chakraborty, B, Chakraborty, G and Maeda, T. “Multi-objective route selection for car navigation system using genetic algorithm.” IEEE (2005): 190 – 5

Chakraborty, B. “GA-based multiple route selection for car navigation.” IEEE (2004): 76-83

Chua, H.C., et al. “Prototype design and implementation for urban area in-car navigation system.” IEEE (2002): 517 – 21

Dibowski, H, Rothkrantz, L and Tatomir, B. “Hierarchical routing in traffic networks.” 2004. <http://www.ai.rug.nl/conf/bnaic2004/ap/a26new.pdf>

Emmerink, C and Schulte, H. “Route planning and route guidance in the Philips in-car navigation system “CARiN”.” Towards an Intelligent Transport System (1995): 240 – 7

Goldberg, Andrew V. and Harrelson, Chris. “Computing the Shortest Path: A* search meets Graph Theory.” Proc. 16th ACM-SIAM Symposium on Discrete Algorithms (2005): 156 – 65

Goldberg, Andrew V., Kaplan, Haim and Werneck, Renato F. “Reach for A*: Efficient Point-to-Point Shortest Path Algorithms.” Technical Report MSR-TR-2005-132, Microsoft Research, 2005.

Guzolek, J and Koch, E. “Real-time route planning in road networks” IEEE (1989): 165 – 9

Higgins, Dan. “Generic A* Pathfinding.” AI Game Programming Wisdom. Ed. Steve Rabin. Hingham, Massachusetts: Charles River Media, Inc, 2002. 114 – 21

Higgins, Dan. “Pathfinding Design Architechture.” AI Game Programming Wisdom. Ed. Steve Rabin. Hingham, Massachusetts: Charles River Media, Inc, 2002. 122 – 31

Hiraishi, H, Mizoguchl, F. and Ohwada, H. “Intercommunicating car navigation system with dynamic route finding.” IEEE (1999): 284 – 9

Holte, R.C. et al. “Hierarchical A*: Searching Abstraction Hierarchies Efficiently.” National Conference on Artificial Intelligence (1996)

Ikeda, T, et al. “A Fast Algorithm for Finding Better Routes by AI Search Techniques.” Proc. Vehicle Navigation and Information Systems Conference. IEEE (1994)

Jönsson, Markus F. “An optimal pathfinder for vehicles in the real-world digital terrain maps.” <http://www.student.nada.kth.se/~f93-maj/pathfinder/index.html>

Köhler, Ekkehard, Möhring, Rolf H. and Schilling, Heiko. “Acceleration of Shortest Path and Constrained Shortest Path Computation.” 2004. Technische Universität Berlin. <ftp://ftp.math.tu-berlin.de/pub/Preprints/combi/Report-042-2004.pdf>

Kroon, Ronald and Rothkrantz, Leon J.M. “Dynamic vehicle routing using an ABC-algorithm.” Proc. of the 15th Belgium-Netherlands conference on Artificial Intelligence (2003): 211 – 8

Lester, Patrick. “A* Pathfinding forBeginners.” July 18th, 2005. <http://www.policyalmanac.org/games/aStarTutorial.htm>

Matthews, James. “Basic A* Pathfinding Made Simple.” AI Game Programming Wisdom. Ed. Steve Rabin. Hingham, Massachusetts: Charles River Media, Inc, 2002. 105 – 13

Möhring, Rolf H, et al. “Partitioning Graphs to Speed up Dijkstra’s Algorithm.” 2005. Technische Universität Berlin. <ftp://ftp.math.tu-berlin.de/pub/Preprints/combi/Report-011-2005.pdf>

Nilsson, Nils J. Problem-Solving Methods in Artificial Intelligence. New York: McGraw-Hill, Inc., 1997.

Pang, K.H., et al. “Adaptive Route Selection for Dynamic Route Guidance System Based in Fuzzy – Neutral Approaches.” IEEE Transactions on Vehicular Technology 48.6 (November, 1999): 2028 – 41

Patel, Amit. “Amit’s Thoughts on Path-Finding and A-Star.” February 25th, 2006. <http://theory.stanford.edu/~amitp/GameProgramming/>

Pearl, Judea. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley Publishing Company, Inc., October 1985.

Pohl, I. “Bi-directional Search.” Machine Intelligence (6). Edinburgh: Edinburgh University Press, 1971. 124 – 140

Sanders, Peter and Schultes, Dominik. “Highway Hierarchies Hasten Exact Shortest Path Queries.” October 2005. Universität Karlsruhe. <http://www.dominik-schultes.de/hwy/>

Wagner, Dorothea and Willhalm, Thomas. “Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs.” January 2003. Universität Konstanz. <http://www.inf.uni-konstanz.de/Preprints/papers/2003/preprint-183.pdf>