Potential Error in the Reuse of Nilsson's A Algorithm for Path-finding in Military Simulations

Emmet Beeker
The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102
ebeeker@mitre.org

We were recently asked to evaluate the Footprint-to-Pathfinder[1] path-finding implementation for possible reuse in OneSAF Objective System. The Footprint-to-Pathfinder path-finding code implements a variation of an algorithm described by Nils Nilsson called the A Algorithm. The A Algorithm can guarantee the optimal, lowest cost path between two points when it uses an "admissible" evaluation function. It is then called the A* (A-Star) Algorithm. The Footprint-to-Pathfinder implementation uses a simplification to the A Algorithm that preserves optimality when a particular condition exists. The condition that enables the simplification—the monotone restriction—is neither well known nor explained with most references for the A Algorithm. In fact, most references imply that an admissible evaluation function is sufficient. However, there are admissible evaluation functions that will cause the simplified implementation to return a path that is sub-optimal. OneSAF Objective System does not restrict admissible evaluation functions as Footprint-to-Pathfinder does. Thus the simplified algorithm is inappropriate for the OneSAF Objective System. Because the monotone restriction is not well publicized, this paper describes it in the context of path finding for combat simulations.

Keywords: A-Star, algorithm, path-finding, routing, shortest path

1. Introduction

The A Algorithm as described by Nils Nilsson [1,2] forms the basis of the route planning function in many combat simulations, including OneSAF, Combat XXI, CCTT, and in many computer games such as Warcraft and Civilization. It is described on many game programming web sites [3,4], in game programming references [5], in mathematical graph theory, artificial intelligence, and computer science references. The key feature of the A Algorithm is an evaluation function, which will be described in the next section. A particular restriction on the evaluation function—admissibility—can guarantee that the algorithm gives the optimum solution: the lowest cost path between two points. When the algorithm uses an admissible evaluation function, it is called the A* (A-Star) Algorithm. Many of the game programming sites do not distinguish between admissible and non-admissible evaluation functions and call it the A* Algorithm in both cases.

Nilsson also describes the "monotone restriction" that allows the A* Algorithm to be slightly modified for better performance. This restriction is not mentioned in the majority of sources. In practice, the evaluation function that is used by most simulations and games meets the monotone restriction, but this is not explicitly stated. From the descriptions available, a reader could infer, incorrectly, that the admissible criterion is sufficient to support the modified algorithm.

Section 2 describes path-finding algorithms, including the A Algorithm, the A* Algorithm, and the Iterative Deepening A* Algorithm. It also describes admissible evaluation functions and the monotone restriction. Because information about the monotone restriction is not commonly available, this paper goes into significant detail. Section 3 describes how the A Algorithm and some of its variants are used in combat simulations: Combat XXI, Close Combat Tactical Trainer, and OneSAF Objective System. Section 4 provides conclusions.

2. Path-finding Algorithms

2.1 Mapping from Terrain to Graphs

The path-finding algorithms discussed in this paper operate on graphs. To create a graph, terrain is divided into cells, so that every possible position in the terrain is in some cell. Typically, the terrain is divided with a square grid so that each cell is adjacent to eight other cells. Each cell becomes a node in the graph that represents the terrain. The nodes of the graph are connected by arcs. Each arc represents the cost of moving from one cell to another cell. The cost could be as simple as the distance between the centers of the two

JDMS, Volume 1, Issue 2, April 2004 Page 91–97
© 2004 The Society for Modeling and Simulation International

[1]Footprint-to-Pathfinder project was developed as an initiative of the Military Operations in Urban Terrain (MOUT) Focus AreaCollaborative Team (FACT), sponsored by Army Modeling and Simulation Office, 400 Army Pentagon, Washington, D.C., 20310-0400

cells. The cost also could be measured in the time it takes to move from one cell to an adjoining cell. Costs need not be the same in both directions; movement uphill could be more difficult than movement downhill, for example. In combat simulations, cost sometimes includes penalties for areas under enemy surveillance or fields of fire. Path-finding algorithms attempt to find the set of arcs and nodes that form a path from a source node to a destination or goal node with the lowest possible total cost.

2.2 Dijkstra Algorithm

One of the earliest algorithms for finding the shortest path from a source node to a set of goal nodes is the Dijkstra algorithm [6]. The Dijkstra algorithm keeps a single list of nodes on the frontier of the searched region. Initially, the source node is placed on the list. Iteratively, the node with the lowest cost path is removed from the list and examined. For each other node that can be reached from the node being examined (i.e., the neighbor nodes), if that node has not itself already been examined, its cost is evaluated and the node is put on the list. When the goal node is removed from the list, the algorithm ends. When a node is removed for examination there can be no lower cost path to the node, else it already would have been found because the lowest cost path is always the first to be examined. Thus, the first path found is the lowest cost path. If the list is exhausted and the goal node has not been reached, then there is no path from the source to the goal. Unlike the following algorithms, the Dijkstra Algorithm can be used with multiple goal nodes. It also is used often to find a path from every node to a goal node.

2.3 The A Algorithm

The Dijkstra Algorithm works out from the source node in all directions. Nodes are removed from the list and examined according to how close they are to the source node. Therefore, many nodes are evaluated which are not on the shortest path to the goal. In order to reduce the number of nodes that must be examined, the A Algorithm adds an estimating function for reaching the goal. For every node that is examined, the algorithm estimates the cost for reaching the destination from that node. Because the actual cost is unknown, this estimate is called a heuristic. The use of the heuristic function directs the path search in the direction of the goal. If the heuristic function were exactly correct, then the path-finding algorithm would only expand nodes that were on the lowest cost path. It is assumed that the number of nodes examined is related to the efficiency of the algorithm and that examining fewer nodes to determine the lowest cost path is desirable.

We use Nilsson's notation in the following discussion. The cost of the minimum cost path from node i to node j is represented by $k(i,j)$. If there is no path from node i to node j, then $k(i,j)$ is undefined. The cost of the minimum cost path from the source node to a goal node through an arbitrary node n is represented by $f^*(n)$. $f^*(n)$ is equal to $k(s,n)$, the cost of the minimum cost path from the source to node n, plus $k(n,g)$, the cost of the minimum cost path from node n to the goal, g. We let $g^*(n) = k(s,n)$ represent the minimum cost for reaching node n from the source, and $h^*(n) = k(n,g)$ represent the minimum cost for reaching the goal from node n. Then the minimum cost for reaching the goal with a path through node n is $f^*(n) = g^*(n) + h^*(n)$.

The A Algorithm keeps a sorted list of nodes to be expanded. The nodes are sorted by the total estimated cost of a path from the source to the goal through that node. The A Algorithm picks the node that has the lowest estimated total cost to be expanded. We let $f(n)$ be the estimated total cost for reaching the goal from the source through node n. Note that $f(n)$ is not necessarily equal to $f^*(n)$. The estimate does not have to equal the actual cost. Let $g(n)$ be the cost of the path we have found from the source to node n. Because there may be more than one way to reach a node, it is not certain that the actual lowest cost has been found—i.e., $g(n)$ does not necessarily equal $g^*(n)$. Rather, $g(n)$ is the lowest cost found at this point in executing the algorithm. We let $h(n)$ be the estimate of how much it will cost to reach the goal from node n. Then $f(n) = g(n) + h(n)$. The A Algorithm will always take the node with the lowest $f(n)$ from the list to expand.

To expand node n, we examine all of the successor nodes, n', that can be immediately reached from node n. For each successor node n', $g(n') = g(n) + k(n, n')$. In other words, the cost for reaching node n' is equal to the cost for reaching node n plus cost of the path from node n to node n'. Because, in general, we cannot guarantee that we will not reach a node by a lower cost path later, nodes may be placed on the list to be expanded more than once. The A Algorithm keeps two lists: OPEN, the list of nodes that have been reached so far and have yet to be expanded (equivalent to the list in the Dijkstra Algorithm); and CLOSED, the list of nodes that already have been removed from the OPEN list and expanded. The CLOSED list is necessary in case we encounter a node a second time. If a node is reached with a lower cost then before, it must be reconsidered. The CLOSED list will store the already expanded nodes and the cost previously calculated for reaching them for comparison purposes.

The A Algorithm works as follows:

> The A Algorithm has two lists—OPEN and CLOSED—both initially empty.
>
> 1. The source node is placed on the OPEN list.
> 2. The node with the lowest evaluated cost, $f(n)$, is removed from the OPEN list and made the current node. If the current node is the destination node then the algorithm terminates with the path to the current node. If the OPEN list is empty, there is no path from the source to the goal.

3. The current node is expanded by examining all of the nodes directly reachable from the current node. For each examined node n', calculate its evaluated cost, $f(n') = g(n') + h(n')$, where $g(n')$ is the cost of the path to the current node plus the cost from the current node to the node being examined and $h(n')$ is the estimate for traveling from the examined node n' to the goal. If the examined node is not on the OPEN list or the CLOSED list, place on the OPEN list. If the examined node is on the OPEN list and the new evaluated cost is less than the previously evaluated cost, update the evaluated cost on the OPEN list. If the examined node is on the CLOSED list and the evaluated cost is less than the previously evaluated cost, then remove the node from the CLOSED list and place on the OPEN list with the new evaluated cost.
4. Place current node on CLOSED list.
5. Repeat step 2, 3, and 4 until the goal is reached or the OPEN list is empty.

The pseudocode in Figure 1 has been taken from Brian Stout's article in *Game Programming Gems* [5]. The parameter for agenttype is used because costs may vary by the type of entity doing the movement. Also note that the parentage of each node is kept to construct the path when the algorithm terminates.

```
Open: priorityqueue of searchnode
Closed: list of searchnode

AStarSearch( location StartLoc, location GoalLoc, agenttype Agent) {
    clear Open and Closed

    //initialize a start node
    StartNode.Loc = StartLoc
    StartNode.costFromStart = 0
    StartNode.CostToGoal = PathCostEstimate( StartLoc, GoalLoc, Agent )
    StartNode.TotalCost = StartNode.CostToGoal
    StartNode.Parent = null
    insert StartNode into Open

    while Open is not empty {
        pop Node from Open      // Node has lowest TotalCost

        // if at goal, we are done
        if (Node is goal node) {
            construct a path backward from Node to StartLoc
            return success
        } else {
            for each successor NewNode of Node {
                NewCost = Node.CostFromStart + TraverseCost( Node, NewNode, Agent )
                // ignore this node if it exists and no improvement
                if (NewNode is in Open or Closed) and (NewNode.CostFromStart <= NewCost) {
                    continue
                } else {     // store the new or improved information
                    NewNode.Parent = Node
                    NewNode.CostFromStart = NewCost
                    NewNode.CostToGoal = PathCostEstimate( NewNode.Loc, GoalLoc, Agent )
                    NewNode.TotalCost = NewNode.CostFromStart + NewNode.CostToGoal
                    if (NewNode is in Closed) {
                        remove NewNode from Closed
                    }
                    if (NewNode is in Open) {
                        adjust NewNode's location in Open
                    } else {
                        insert NewNode into Open
                    }
                }
            }   // now done with Node
        }
        insert Node into Closed
    }
    return failure   // no path found and Open is empty
}
```

Figure 1. Pseudocode for Algorithm

2.4 Role of the Heuristic Function

Performance of the A Algorithm is closely related to the heuristic function. If the heuristic function is a lower bound on the actual cost of reaching the goal for every node, $h(n) \le h^*(n)$ for all n, then the algorithm is called "admissible." An admissible search algorithm always terminates with the lowest cost solution, if it exists. When the heuristic function is a lower bound on the actual cost, Nilsson calls it the A* Algorithm, to show that it is admissible and that the cost of the solution is equal to $f^*(s)$, the lowest cost path from the source to the goal.

The zero function, $h(n) \equiv 0$, is a lower bound on $h^*(n)$, the actual cost of reaching the goal from node n. Therefore, using $h(n) \equiv 0$ for the heuristic function makes the A Algorithm admissible, and therefore an A* Algorithm. In this case, the A* Algorithm is the same as the Dijkstra Algorithm, since $f(n) = g(n)$. Given two heuristic functions, h_1 and h_2, that are both lower bounds on $h^*(n)$, it can be shown that if $h_1(n) \le h_2(n)$ for all n, then every node expanded by h_2 also will be expanded by h_1 [2]. Normally, h_2 will expand fewer nodes than h_1. This implies that using larger cost estimation will result in fewer nodes being examined.

If we allow the heuristic function to be larger than the actual cost, then the A Algorithm is no longer admissible, but it may find a solution faster. However, the solution is not guaranteed to be the actual lowest cost solution. For some applications, finding a solution quickly may be more important than finding the least cost solution. See any of the references [2,3,5] for details.

Processing the OPEN and CLOSED lists in the A Algorithm may be computationally intensive. Clearly, when the zero heuristic is used and the A Algorithm becomes the same as the Dijkstra Algorithm, the CLOSED list is not needed. Nilsson gives a constraint on the heuristic function that guarantees the first time a node is expanded, the cost for reaching the node from the source is the lowest possible cost, $g(n) = g^*(n)$. In this case, a node can never be reached a second time with lower cost and nodes on the CLOSED list do not need to be checked. Nilsson calls this the monotone restriction. This restriction is described in the next section.

2.5 Conditions for Reaching a Node on the CLOSED List with Lower Cost

In this proof, "evaluated cost" of a node, n, will mean the cost of reaching a node plus the estimated cost from the node to the goal—i.e., $f(n)$.

The CLOSED list contains nodes that have been expanded and removed from the OPEN list. For a node to be on the CLOSED list there must have been a path found to the node from the source. For a node on the CLOSED list to be re-examined, there must be another path to the node from the source whose cost is less than the cost using the previous path. In figure 2, assume that node p has been placed on the CLOSED list with cost $f_1(p) = g_1(p) + h(p)$. Assume that a second path to node p has been found. Label the node previous to node p on this path node q. The total cost for reaching node p via node q is the cost for reaching node q plus the cost between node q and node p, $g_2(p) = g(q) + k(q, p)$. If the cost of reaching node p via node q is greater than or equal to the previous cost for reaching node p, then we can disregard the path from node q to node p (an equal or less cost path has already been found through node p). However, if the cost of reaching node p via node q is less than the previous cost of reaching node p, then we have found a potentially shorter path to the goal and node p should be removed from the CLOSED List and placed on the OPEN list with the new cost. The following proof shows that reaching node p with a lower cost path after node p has been placed on the CLOSED list requires a particular condition of the heuristic estimation function.

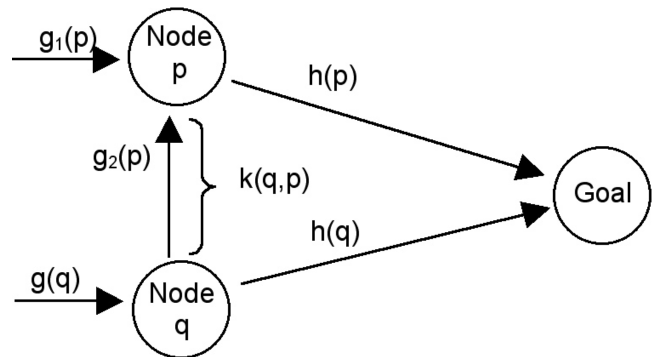

Figure 2. Second path through node p

In the following discussion, refer to Figure 3. In order for the new path through node q to node p to have a lower cost, $g_2(p)$, than the cost of the previous path to node p, $g_1(p)$, there must be some node (call it node r) on the new path that has an evaluated cost, $f(r)$, greater than or equal to the old evaluated cost for node p, $f_1(p)$. If there were no such node, then every node on the path to node q would have an evaluated cost less than the evaluated cost for node p and would, therefore, have been examined prior to examining node p. In order for node r to be unexamined, it must have had an evaluated cost, $f(r)$, greater than $f_1(p)$. Therefore:

Equation 1 $$f(r) \ge f_1(p)$$

The new cost to node p is less than the old cost, by assumption:

Equation 2 $$g_1(p) > g_2(p)$$

Adding $h(p)$ to both sides of Equation 2:

Equation 3 $$g_1(p) + h(p) > g_2(p) + h(p)$$

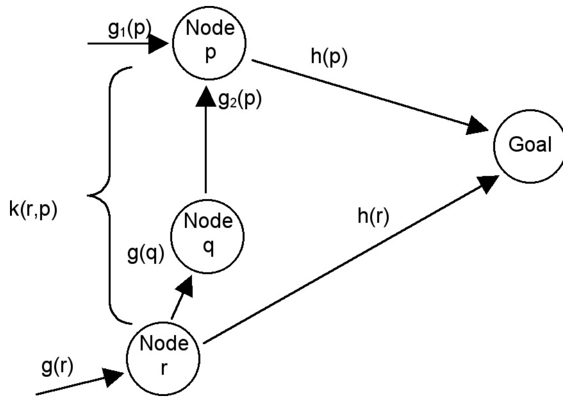

Figure 3. Derivation of monotone restriction

Since $f_1(p) = g_1(p) + h(p)$ and $f(r) \geq f_1(p)$ by Equation 1:

Equation 4 $$f(r) > g_2(p) + h(p)$$

The cost from the source to node p along the new path is equal to the cost from the source to node r, $g(r)$, plus the cost from node r through node q to node p, $k(r, p)$. Replacing $g_2(p)$ with $g(r) + k(r, p)$ in Equation 4:

Equation 5 $$f(r) > g(r) + k(r, p) + h(p)$$

The evaluated cost at node r is equal to the cost from the source to node r plus the estimated cost from node r to the goal. Replacing $f(r)$ with $g(r) + h(r)$:

Equation 6 $$g(r) + h(r) > g(r) + k(r, p) + h(p)$$

Subtracting $g(r)$ from both sides gives:

Equation 7 $$h(r) > k(r, p) + h(p)$$

This means that the estimated cost for reaching the goal from node r is more than the actual cost of traveling from node r to node p plus the estimated cost of reaching the goal from node p. Rearranging terms:

Equation 8 $$h(r) - h(p) > k(r, p)$$

Equation 8 says that the change in estimated cost to reach the goal from node r to the estimated cost for reaching the goal from node p is larger than the actual cost from node r to node p. Using the A* Algorithm, we have proved that for a node to be reached a second time with lower cost than previously, the actual cost between two nodes must be less than the change in the heuristic function at the nodes. In other words, the heuristic function over estimates the change in cost between two nodes.

Nilsson's monotone restriction is the contrary of this. A heuristic function, h, is said to satisfy the monotone restriction if for all nodes i and j, such that j is a successor of i, $h(i) – h(j) \leq k(i, j)$ with $h(goal) = 0$. If the heuristic function meets this condition, when the A* Algorithm selects a node for expansion, it has found an optimal path to that node. Therefore, when the heuristic function meets the monotone restriction, nodes on the CLOSED list will never be removed and placed on the OPEN list.

2.6 Iterative Deepening A Algorithm*

Both the Dijkstra and A Algorithms are breadth first algorithms. They examine all potential paths adjacent to the initial node and continue to expand all of the routes that are adjacent to a selected node. As a result, they keep large lists of potential routes to be examined. The list in the Dijkstra Algorithm and the OPEN list in the A* Algorithm can become quite large, and in some cases exceed the memory available. A depth-first algorithm only keeps one path at a time and iterates over all possible paths, trading memory space for processing time. Because the depth of the optimal path is unknown until it is found, the depth-first algorithm may generate paths deeper than the optimal, creating large workload. Also, any paths found to the goal cannot be guaranteed to be optimal until all paths of lesser cost are generated and examined. The Iterative Deepening A* Algorithm (IDA*) limits the depth of paths which are examined so that non-optimum long paths are not examined. IDA* Algorithm performance has been shown to be asymptotically optimal in terms of running time and space required for exponential tree searches [7]. Testing in one combat simulation has shown it to be no more than five times slower than the A* Algorithm [8].

The IDA* Algorithm iterates over path cost. In the first pass, IDA* generates all paths connected to the source node and estimates their costs. The subsequent iterations use the minimum cost path from the previous iteration as a threshold. It then generates all possible paths until they exceed the threshold. The minimum cost of the generated paths becomes the threshold for the next iteration. The process continues until a path reaching the goal has equal or less cost than the threshold. Figure 4 give pseudo code for the IDA* Algorithm.

Performance of the IDA* Algorithm can be improved by saving the path distance—$g(n)$—to each node for the current iteration. Then the algorithm can avoid expanding any node that already has had its successors examined when the node has been reached at equal or less cost. This is equivalent to the use of the CLOSED list for the A* Algorithm. If the IDA* Algorithm only notes that a node has been expanded during the current iteration, but does not save the cost, then it is equivalent to the modified A* Algorithm where nodes are not removed from the CLOSED list and placed on the OPEN list. In this case, the same error can be made in the IDA* Algorithm as the A* Algorithm if the heuristic function does not meet the monotone restriction: a non-optimal path can be returned.

3. Use of the A Algorithm in Combat Simulations

Variations of the A Algorithm are used in Close Combat Tactical Trainer, Combat XXI, and OneSAF Objective System. All three are entity level combat simulations where individual soldiers and vehicles are represented in the simulation. Through a control interface, entities often are given directives to move from their current location to a goal location. All three simulations use a variant of the A Algorithm to find the optimal path.

Close Combat Tactical Trainer uses the Iterative Deepening A* Algorithm [9]. Close Combat Tactical Trainer had severe memory requirements and it was considered reasonable to trade run-time for memory space. Notice that in the algorithm, the successful path is not returned until its cost is below the threshold, rather than when it is first encountered. It is possible that several paths could reach the goal during the same iteration, each having different costs. The lowest cost path to the goal would not necessarily be generated first. By waiting for the threshold to be increased to the lowest cost path to the goal, the optimal path is guaranteed to be returned. Close Combat Tactical Trainer uses the lowest possible traversal cost over the straight line distance from the node to the goal as the heuristic function. This guarantees that the algorithm is admissible. The heuristic function also meets the monotone restriction. This can be seen by taking any two nodes, i and j, and drawing the triangle consisting of the two nodes and the goal node, g. The heuristic function for nodes i and j are the lowest possible costs for the straight line distances from the nodes to the goal. The actual cost between nodes i and j cannot be less than the lowest possible cost for the straight line distance between the two nodes. By the triangle inequality, the lowest cost between the two nodes i and j must be greater than or equal to the difference, $h(i) - h(j)$. The actual cost is greater than or equal to the lowest cost, therefore $h(i) - h(j)$ is less than or equal to $k(i,j)$, which is the monotone restriction.

Combat XXI uses a modified A* Algorithm implementation provided by the Footprint-to-Pathfinder project [10]. Nodes are never removed from the CLOSED list. Like Close Combat Tactical Trainer, Combat XXI uses straight line distance at the lowest possible traversal cost for its heuristic function. This ensures that the heuristic is admissible and meets the monotone restriction. Because the heuristic function meets the monotone restriction, the Combat XXI implementation does not have to consider reaching a node with a subsequent path of lower cost than the first time. The first time a node is reached, it is guaranteed to be the lowest actual cost. Thus Combat XXI never looks at the cost of nodes on the CLOSED list, since it will never move them. Rather than keep a CLOSED list, Combat XXI simply marks the node as having been expanded—requiring a single bit in the node data structure.

OneSAF Objective System uses the environmental model from WARSIM [11]. The WARSIM environmental model implements the A Algorithm and allows the user to provide an evaluation function through a callback reference. The user does not have to use an admissible heuristic function nor does the heuristic function have to meet the monotone restriction. It is assumed that the user is knowledgeable and if a non-admissible heuristic function is used, the user intends it for performance reasons. Because the A Algorithm is implemented completely, the user is guaranteed to obtain an optimal route as long as the heuristic function is admissible, whether or not the heuristic function meets the monotone restriction.

OneSAF Objective System is written in JAVA and the WARSIM environmental model is written in C++. Calls from OneSAF Objective System to the environmental model have to transition through a wrapper. Because of this, the Footprint-to-Pathfinder implementation, written in Java, was suggested as a replacement for the WARSIM environmental model. It was expected that the avoidance of the wrapper would increase performance. What was not considered was the user's ability to provide the evaluation function. The user could provide an evaluation function that used an admissible heuristic function, but one that did not meet the monotone restriction. The user would have expected the path-finding algorithm to return the lowest cost route. But because the Footprint-to-Pathfinder implementation does not remove nodes from the CLOSED list, the path returned is not guaranteed to be lowest cost when the heuristic function does not meet the monotone restriction. As a result, a path exposing soldiers to enemy fire could be returned, where a safer path exists. The Footprint-to-Pathfinder implementation, which works perfectly well in Combat XXI, is not appropriate for OneSAF Objective System.

4. Conclusion

Path-finding is ubiquitous in computer-based simulations. Combat simulations use significant memory and processor time to calculate point-to-point paths. The A Algorithm and its variations are widely used to determine point-to-point paths. Matching the implementation of the algorithm with the desired performance characteristics can be critical. Fortunately there are large numbers of references in the computer science, graph theory, artificial intelligence, and game programming texts and Web pages. Unfortunately, few of the references that give implementations discuss the monotone restriction. Examples—even those that use the modified A Algorithm—mostly use the lowest traversal cost as the heuristic function, stating that this choice is admissible. However, they do not state that it also meets the monotone restriction. This neglect could cause a potential user to think that an implementation would work for all admissible heuristic functions, not just those that meet the monotone restriction. If the modified implementation were

used with an admissible heuristic function that does not meet the monotone restriction, the application would use routes that are not guaranteed lowest cost.

5. References

[1] Hart, P. E., N. J. Nilsson, B. Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems, Science, and Cybernetics,* 4(2).

[2] Nilsson, N. J. 1980. *Principles of Artificial Intelligence*. Burlington, MA: Morgan Kaufmann Publishers, 74-88.

[3] http://www.policyalmanac.org/games/aStarTutorial.htm.

[4] http://www.gamasutra.com/features/19970801/pathfinding.htm.

[5] Stout, B. 2000. *The Basics of A* for Path Planning.* Game Programming Gems, Charles River Media.

[6] Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269-271.

[7] Korf, R. E. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Proceedings of the 9th International Joint Conference on Artificial Intelligence,* Los Angeles, CA.

[8] Karr, C. R., S. Rajput, L. J. Breneman. 1995. Comparison of the A* and Iterative Deepening A* Graph Search Techniques. *Proceedings of the Fifth Conference on Computer Generated Forces and Behavioral Representation,* Orlando, FL, May 9-11, 443-449.

[9] Campbell, C., R. Hull, E. Root, L. Jackson. Route Planning in CCTT. *Proceedings of the Fifth Conference on Computer Generated Forces and Behavioral Representation,* Orlando, FL, May 9-11, 233-243.

[10] http://www.trac.army.mil/4-ModelsSims/.

[11] http://www.onesaf.org.