ADAPTIVE CRITIC DESIGN TECHNIQUES FOR MOBILE TRANSMITTER PATH PLANNING

Grant Rivera (Student) and Kurt Kosbar (Advisor)
Telemetry Learning Center
Department of Electrical and Computer Engineering
Missouri University of Science and Technology

ABSTRACT

In geometrically complex indoor industrial environments, such as factories, health care facilities, or offices, it can be challenging to determine where each telemetry receiver needs to be located to collect data from one or more mobile transmitters. Accurately estimating the areas that each transmitter frequently travels, rarely travels, and quickly travels through, helps to simplify the telemetry system planning problem and establishes which areas may be acceptable to provide marginal coverage. This paper discusses how using A* (A-Star) for transmitter path planning can assist in the telemetry system planning problem.

Keywords: System and Mission Planning, Industrial Applications, Path Planning, A-Star, A*.

INTRODUCTION

In typical range applications, there are a reasonably small number of telemetry receivers. There typically are obstructions which can block a radio frequency (RF) signal, or reflectors which can generate multipath interference. A wide range of techniques and equipment have been developed to mitigate these challenges, in a way that minimizes the bandwidth, required transmitted power, and number of telemetry receivers.

The situation is substantially different for indoor telemetry applications. Health care facilities, manufacturing plants, office buildings, and even residential dwellings are now housing telemetry systems. The RF environment in these settings can be far more complex than that faced in range applications. The interior of a building is almost always geometrically complex. One way to reliably communicate in such an environment is to use frequencies that can penetrate multiple obstructions. However this may require the use of RF frequencies which are too low to provide adequate bandwidth, or require more transmitted power than can be easily supplied by the telemetry transmitter.

A common solution to this problem is to instrument the structure with a large number of telemetry receivers, or access points. Using simple design rules for placement of these receivers may result in some being asked to handle far more users than it can accommodate, while others are lightly used. Both issues can cause problems, since an overburdened telemetry node may be

forced to drop users, and lightly used nodes add cost to the system, and also may create unnecessary interference for the adjacent nodes.

To address this design challenge, one needs to understand both the RF environment where the system is operating, and the way in which telemetry transmitters move within the environment. There are a large number of RF modeling tools available, which can help address the first challenge. The focus of this paper is the second challenge, that of modeling how users may move within a structure. In particular, we are developing software tools which can model the aggregate behavior of individuals moving through an indoor environment. Part of this model involves path planning. Individuals often dwell within one room, or area, of a building for an extended period, and then select a path which will take them to another room/area, where they will again remain for an extended period of time. Accurately modeling the path they will take, will assist in the placement of telemetry receivers, and also allow us to estimate the number of users each receiver will need to accommodate.

A variety of algorithms may be suitable for this application [1-8]. Exhaustive search algorithms will always find the optimal path, but they tend to require a large amount of computational time. Greedy search algorithms are usually fast but will not always find an optimal solution. The A* algorithm chooses which node to go to, based on the steps taken to get to its current location which is a known cost and the distance from its current node to the desired goal which is an approximated cost [9-11]. By balancing these two costs, A* is neither an exhaustive or greedy search but acts like both. It does a thorough but directed search of the search space. This means an optimal path will always be found like in the exhaustive search but should take less time than an exhaustive search by using a heuristic like in a greedy search.

SEARCH SPACE

The search space is the first thing that needs to be created and will typically consist of obstacles which the path must avoid. There are a variety of ways to describe obstacles, but for simplicity this work will use a matrix to represent a regularly spaced grid, where each node is assigned a binary value. If a node has a value of one, then it is considered to be an obstacle which cannot be traversed. A node with a value of zero is a space free of obstacles and is allowed to be traveled. The number of nodes in the grid must be chosen carefully, since a small grid may not be sufficient to model a particular environment, while a large grid will result in very long simulation execution times.

Once the grid size is selected, and depending on if there are more free spaces or obstacles, the matrix should be initialized with either all ones or zeros respectively. The other will then be entered in specific locations to create the obstacles or free spaces of the search space. The outer perimeter of the terrain matrix should be set as an obstacle to prevent exploration outside of the search area. All desired starting and ending locations should also be initialized.

A* IMPLEMENTATION

Two lists need to be created when implementing A*, an open-list and a closed-list [10]. The open-list contains all of the surrounding nodes that can be traveled to. The closed-list contains all of the nodes already traveled to so they will not be traveled to again and should only be initialized with the starting node. As long as the open-list is not empty or the desired goal has not been reached, the A* algorithm will continue to search for the desired ending location.

A* checks all eight adjacent nodes from the current location. The adjacent node will be added to the open-list as long as it is free of obstacles and is not currently on the open or closed list. The movement cost will then be calculated for each node on the open-list. The movement cost is the cost to travel to that node from the original starting point. Thus, the starting position is given a movement cost of zero. The movement cost is calculated by adding the movement cost of the current node to the cost of moving from the current node to the new node. Since each node in the matrix are regularly spaced from each other and by using the Euclidean distance formula, all horizontal and vertical movements per node will have a cost of one and all diagonal movements to each individual node will have a cost of the square root of two [10].

$$G_n = G_c + \sqrt{(r_n - r_c)^2 + (c_n - c_c)^2}$$

Where:

- G_n is the movement cost of the new node.
- G_c is the movement cost of the current node.
- r_n is the row location for the new node.
- r_c is the row location for the current node.
- c_n is the column location for the new node.
- c_c is the column location for the current node.

If this cost has already been calculated previously, it should be compared to the previous calculation, and if the new cost is better (lower), then the current node will be considered the best way to get to the new location. The current node will be defined as the parent of all new nodes and nodes where the cost is lower. Following the parents of each node, will create the shortest path back to the beginning which is how the path is formed.

Next, the heuristic cost is calculated for every node on the open-list. The heuristic cost is an estimated value for the distance from the current node to the desired ending point. This value is calculated by using the Euclidean Distance formula [9].

$$H_n = \sqrt{(r_f - r_n)^2 + (c_f - c_n)^2}$$

Where:

- H_n is the heuristic cost of the new node.
- r_f is the row location for the desired ending node.
- r_n is the row location for the new node.
- c_f is the column location for the desired ending node.
- c_n is the column location for the new node.

This value needs to be calculated only if it has not already been done before since this value never changes. If a heuristic is not used in A*, it is known as Dijkstra's Algorithm.

With the movement and heuristic cost calculated, the fitness cost can be determined. The fitness cost is the sum of the cost to get to the current location plus the estimated cost to the desired final location.

$$F_n = G_n + H_n$$

This combination of costs directs the search and allows for a fast, optimal solution to be found. The node on the open-list with the smallest fitness cost will become the new current node. This new current node is removed from the open-list and placed on the closed-list. The entire process is repeated again until there are no longer any nodes to choose from the open-list or until the current node is the desired ending location.

SIMULATION RESULTS

In path planning, the most common algorithm test is a "u" or "n" shaped maze which can be seen from figure 1. Each Figure A shows the standard A* algorithm. The standard algorithm tends to cut corners and gets close to obstacles but in each Figure B, the diagonal movement cost is doubled to make all movement horizontal and vertical. Doubling the diagonal cost still allows for diagonal movement but only when necessary. If diagonal movement is never desired, the Manhattan distance formula should be used for both the movement cost and if a heuristic is being used, then for the heuristic too [9].

$$D = abs(r_c - r_n) + abs(c_c - c_n)$$

Where:

- D is the Manhattan distance from current node to either the new node (movement cost) or goal (heuristic cost) depending on which cost is being calculated. For the movement cost, G_c still needs to be added to this value just like in the Euclidean distance formula.
- abs is the absolute value.
- r_c is the row location for the current node.
- r_n is the row location for either the new node or goal depending on the cost function.

- c_c is the column location for the current node.
- c_n is the column location for either the new node or goal depending on the cost function.

Each Figure C and D uses the Dijkstra's algorithm, C being the standard algorithm and D having twice the diagonal movement cost. Dijkstra is the same as A* but without the heuristic information. Dijkstra's path takes longer which is expected since it is an exhaustive search, but the benefit from using Dijkstra's algorithm is that every node is searched and therefore, if multiple paths from the same starting position are desired, the algorithm doesn't need to be run again since every node in the search space points to a parent which will create the shortest path back to the beginning.

Figure 2 show a slightly more complicated maze. Figure 3 demonstrate that moving only in horizontal and vertical fashion might not always be beneficial. One important thing to keep in mind is that even though both of these algorithms with find an optimal path, there could possibly be more than one optimal path. Depending on how the algorithm is written determines which optimal path is chosen. This can be seen in figure 3C and 3D in which the algorithms each found an optimal path which were slightly different from each other.

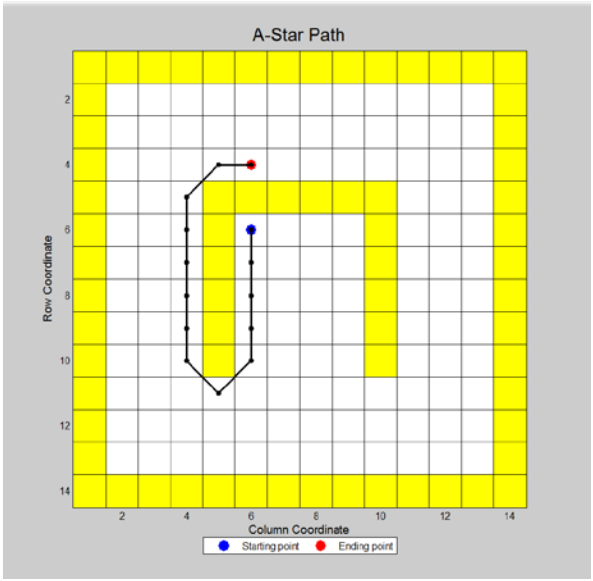

Figure 1A. Length of path: 14.2426
Computational time: 0.019 seconds

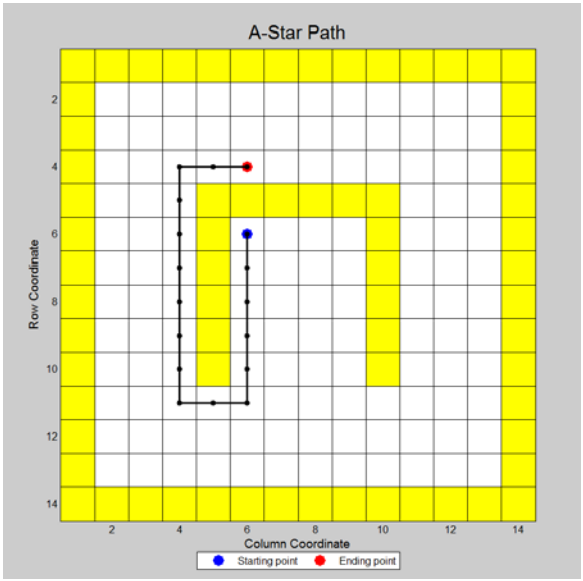

Figure 1B. Length of path: 16;
Computational time: 0.021 seconds

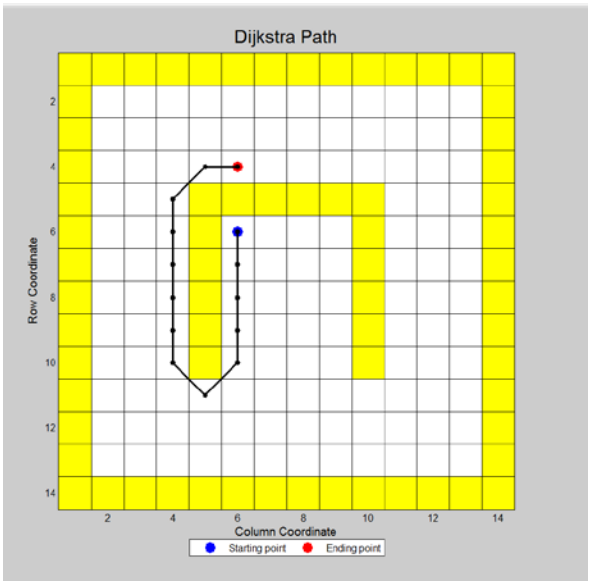

Figure 1C. Length of path: 14.2426; Computational time: 0.03 seconds

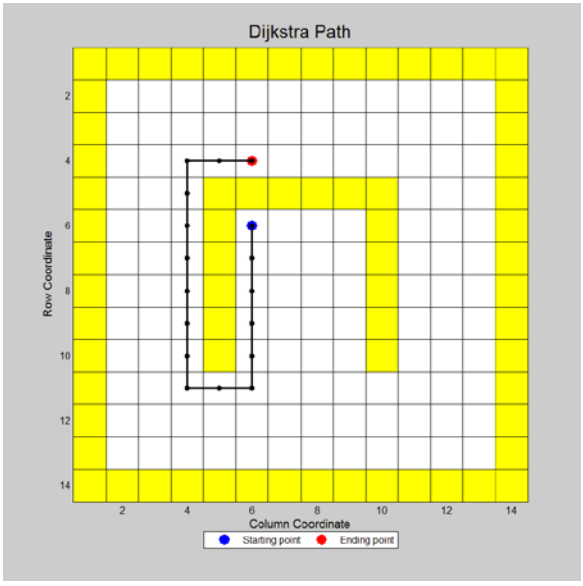

Figure 1D. Length of path: 16; Computational time: 0.025 seconds

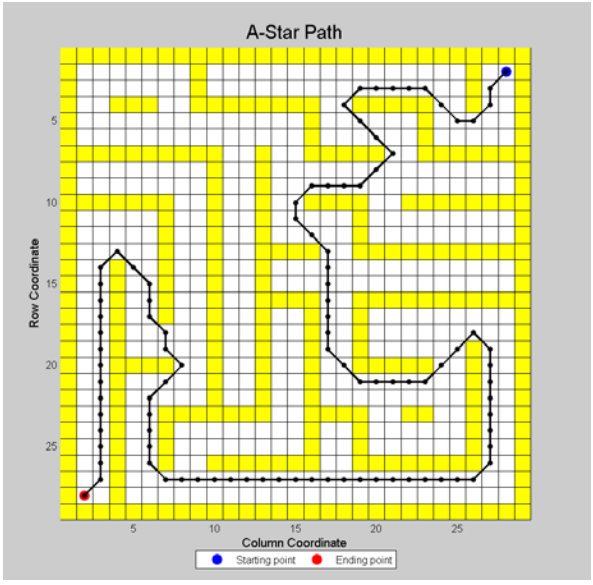

Figure 2A. Length of path: 107.012; Computational time: 0.142 seconds

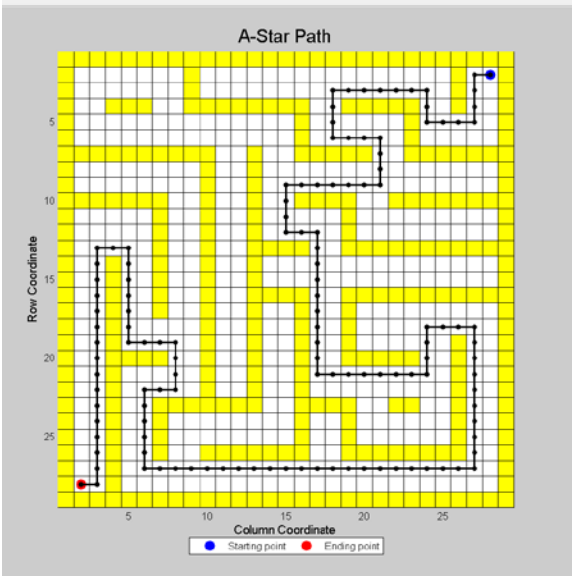

Figure 2B. Length of path: 124; Computational time: 0.145 seconds

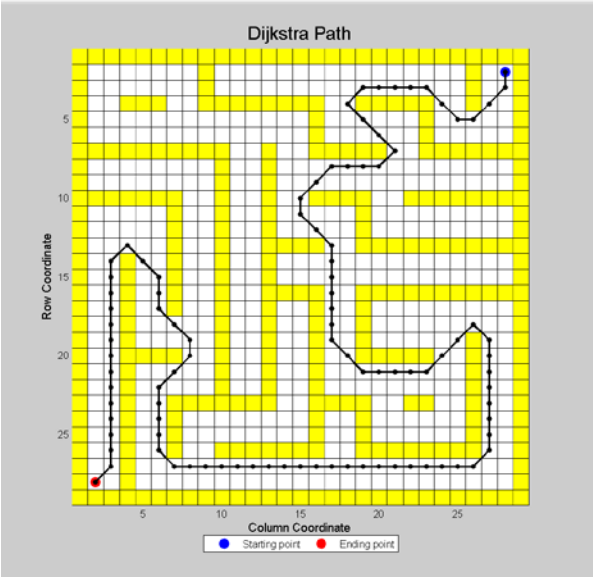

Figure 2C. Length of path: 107.012; Computational time: 0.163 seconds

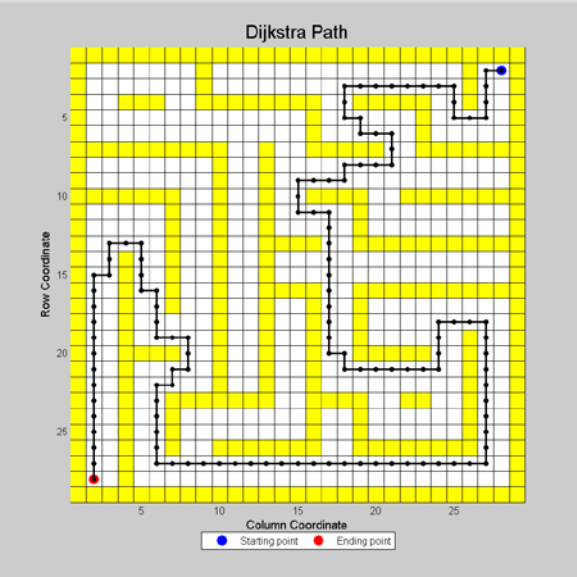

Figure 2D. Length of path: 124; Computational time: 0.145 seconds

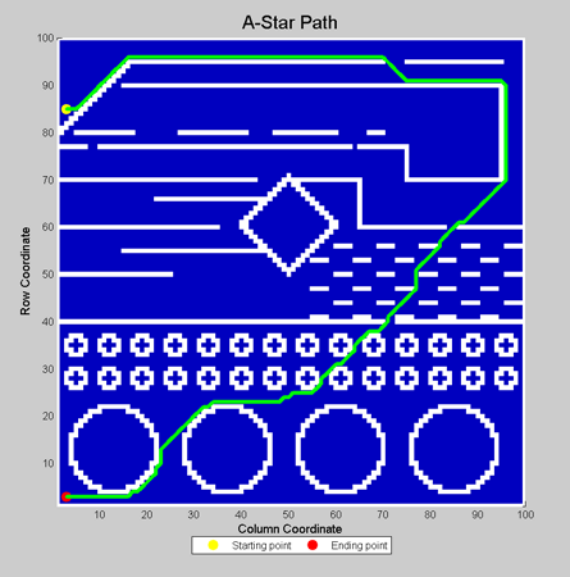

Figure 3A. Length of path: 247.823; Computational time: 6.203 seconds

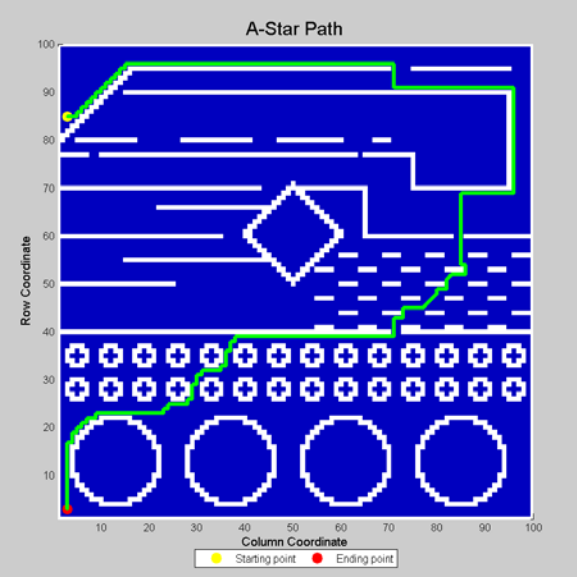

Figure 3B. Length of path: 289.657; Computational time: 11.232 seconds

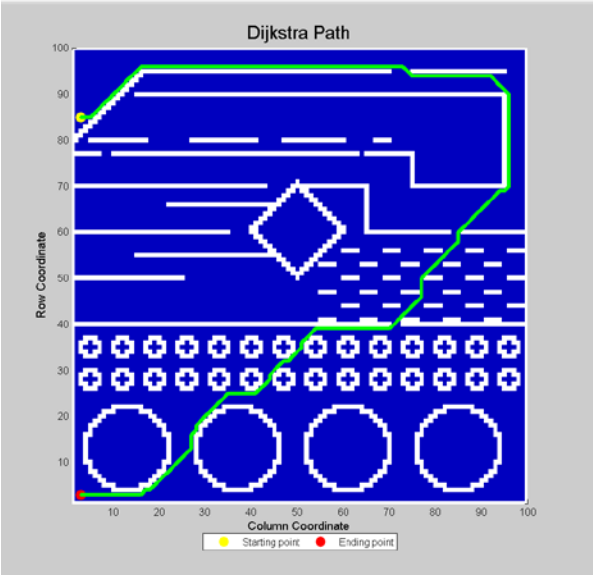

Figure 3C. Length of path: 247.823; Computational time: 14.792 seconds

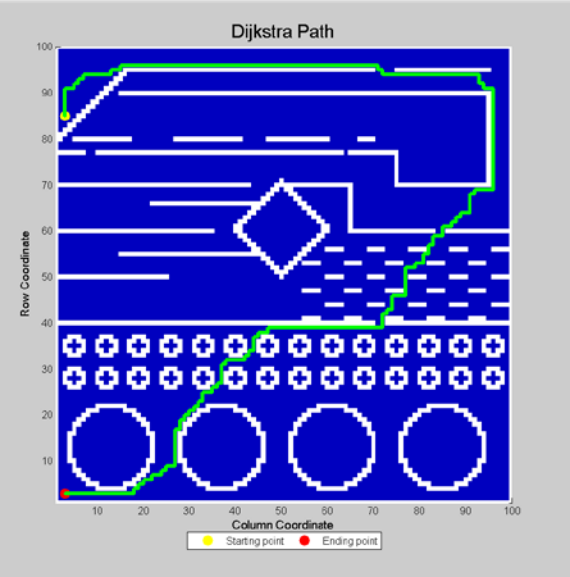

Figure 3D. Length of path: 290; Computational time: 14.846 seconds

CONCLUSION

The standard A* will always find the shortest path in terms of distance much like an exhaustive search but A* should be faster than an exhaustive search because of the use of heuristic information. Since A* is a modification of Dijkstra's algorithm, Dijkstra's algorithm was also compared. Even though Dijkstra is typically slightly slower when planning a single path, it could possibly be more beneficial when planning multiple paths since every node and parent remain stored which allows for instant path planning from any node to the original starting point. By using these algorithms, typical telemetry transmitters' paths can be predicted. These predictions can then be used to economically position telemetry receivers.

REFERENCES

[1] Y. Zhong, B. Shirinzadeh, Y. Tian, "A New Neural Network for Robot Path Planning," *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, Xi'an, China, July 2008.

[2] S. Yang, M. Meng, "Real-time Collision-free Path Planning of Robot Manipulators using Neural Network Approaches," *IEEE International Symposium on Computational Intelligence in Robotics and Automation*, 1999.

[3] X. Yang, M. Meng, "An Efficient Neural Network Model for Path Planning of Car-Like Robots in Dynamic Environment," *IEEE Canadian Conference on Electrical and Computer Engineering*, 1999.

[4] H. Chan, K. Tam, N. Leung, "A Neural Network Approach for Solving the Path Planning Problem," *IEEE International Symposium on Circuits and Systems*, 1993.

[5] B.A. Garro; H. Sossa; R.A. Vazquez; "Path Planning Optimzation Using Bio-Inspirited Algorithms," *IEEE Artificial Intelligence*, Nov. 2006.

[6] B.A. Garro; H. Sossa; Vazquez, R.A.; "Evolving Ant Colony System for Optimizing Path Planning in Mobile Robots," *IEEE Electronics, Robotics and Automotive Mechanics*, Sept. 2007.

[7] Joon-Woo Lee; Jeong-Jung Kim; Ju-Jang Lee; "Improved Ant Colony Optimization Algorithm by Path Crossover for Optimal Path Planning," *IEEE Industrial Electronics*, July 2009.

[8] Joon-Woo Lee; Jeong-Jung Kim; Byoung-suk Choi; Ju-Jang Lee; "Improved Ant Colony Optimization Algorithm by Potential Field Concept for Optimal Path Planning," *IEEE Humanoid Robots*, Dec. 2008.

[9] Amit Patel; "Amit's A* Pages," http://theory.stanford.edu/~amitp/GameProgramming/.

[10] Patrick Lester; "A* Pathfinding for Beginners," http://www.policyalmanac.org/games/aStarTutorial.htm, July 18, 2005.

[11] Marco Pinter; "Toward More Realistic Pathfinding," http://www.gamasutra.com/view/feature/3096/toward_more_realistic_pathfinding.php, March 14, 2001.