A Comparative Analysis of the A\* Pathfinding Algorithm

Jeramey Tyler and Suranga Hettiarachchi

Computer Science Department

Indiana University Southeast

New Albany, IN 47150

812-901-2698

jeatyler@umail.iu.edu, suhettia@ius.edu

ABSTRACT

( I copied some from your C458 term paper, I recommend that you use the term paper and extract as much info. as possible from that paper to this)

Our goal was to use simulation to quantify the abilities of the A\* pathfinding algorithm. We will show that a robot using an implementation of A\* can, under most conditions, outperform a randomly wandering robot, as well as a robot utilizing an algorithm of our own design, in the areas of uniformity of coverage and quantity of movements required to achieve said coverage.

Three algorithms were compared to determine the highest level of efficacy under different combinations of experiment parameters. As previously mentioned, the A\* pathfinding algorithm was the primary focus. For a baseline, a randomly wandering algorithm was designed. In this algorithm each robot randomly decides what its next movement will be. The third algorithm is an algorithm of our design called Primary Movement. In the Primary Movement algorithm a robot determines its next move based on the robot’s ability to unveil more of the map.

In each algorithm, every robot shares a series of communal maps. These maps are modified by sensor data received through range sensors positioned at 0°, 45°, and 315° from the forward facing direction of the robot. The sensor data is used to update the map as the robots traverse the environment. This continues until either the environment has been completely mapped or a robot has determined there are no acceptable movements available.

(I think we should remove references to real robots. Tell the work is done in simulation, tell that there are three algorithms, one being the random that is used as a base line, tell that results be presented at the end with an analysis.) .

INTRODUCTION

There often exist environments that are dangerous for humans to navigate. Natural and manmade disasters can leave areas strewn with hazardous debris, gas leakages, fires, and radiation. Is it possible to use autonomous robots to navigate these environments and perform tasks such as search and rescue, thereby reducing the risk to human life? Many navigation algorithms already exist, but what separates one from another? In which situation should which algorithm be used? Can an algorithm be created that can out-perform a mature algorithm? These are some of the questions spurred us to start this research.

We have chosen one mature algorithm to compare against and algorithm of our creation, an algorithm we call Primary Movement, and a baseline. The three algorithms were tested in a simulation of our design that allows the user to  specify many of the experiment’s parameters before the experiment starts. After the parameters are chosen, an environment is randomly generated. Each algorithm is then tested using the same environment. Two metrics were chosen to be measured for each experiment, percentage of the environment viewed and number of physical movements taken.

(We need to beef up this part with bit more detail, reflect what to come in background, tell that there are three algorithms being discussed, introduce them without much of detail.)

Background

Our works relies heavily on the A\*(A-Star) search algorithm. Given a starting point and an ending point, the A\* algorithm will determine a least cost path to travel from start to goal. (If you are explaining any modification, specific to what you implemented, to original A\* here, that should be done in METHODOLOGY section rather than background section, this should only include a brief description of original A\*)

The A\* algorithm was first published in 1968 by Hart, Nilsson and Raphael [2]. It was modeled off of the algorithm that Edsger Dijkstra described in his 1959 “A note on two problems in connexion with graphs” [1].

In the A\* search algorithm, every vertex in a grid is treated as a node. A node is a structure that consists of a parent node, and an F, G, and H value. The parent node is the node in the path that preceded the current node. The G value is the calculated cost to travel from the starting node to the current node. The H value is a heuristic estimate of the cost to travel from the current node to the goal node. The F value is the sum of these two costs and is used as the basis for determining which node will be assessed next.

In addition to the values that each node has, a pair of lists must be maintained. The two lists are the Open list and the Closed list. The Open list contains all of the nodes that have yet to be evaluated but have been determined to be adjacent to some node that has already been evaluated. The closed list contains all of the nodes that have been evaluated.

The A\* algorithm works by first adding the starting node to the open list. The starting node is given a G value of 0, as there is no cost to travel from itself to itself, an H value calculated by the heuristic function, and the sum of those two values is assigned as the F value. While the Open list is not empty, the node with the lowest F value is chosen to be evaluated next as the current node. If the current node is the same node as the goal node, then a path has been found. Otherwise, the current node is removed from the Open list and added to the Closed list.

For every node that is adjacent to the current node, a tentative G value is calculated by summing the G value of the current node and the distance between the current node and the adjacent node. If the adjacent node is already in the Closed list and the tentative G value is greater than or equal to the G value of the adjacent, meaning that the adjacent node has already been evaluated and the path that it lies on is a lesser cost path than the current path, continue to the next adjacent node. Otherwise, if the adjacent node is not in the Open list, or if its G Value is greater than the tentative G value, the parent node of the adjacent node is set to the current node. The G value is set to the tentative G value. The H value is calculated and set, and the F value is set as the sum of the G and H values. If the adjacent is not in the Open set it is added at this time.

The process is repeated until the current node is the same node as the goal node. The parent of the goal node is the vertex in the grid that would be traversed immediately before the goal. By working backwards from the parent of each node starting with goal node, a shortest cost path is created in reverse order.  (change the citations as below, give a background info of A\*, and your other two algorithms, why you picked them, what motivated you to use them. )

( NEED TO DO CITATIONS LIKE THIS ---- Peterson, et al., [5] demonstrated.... Please update the citation list at the end.)

Project Goals

The goal of this project is to design an algorithm that will allow a simulated robot to autonomously map an unknown environment while avoiding obstacles. To create the map the robot must be able to determine which portions of the map have not been viewed yet and navigate a path around obstacles to those sections. The map that is created should be accurate and as complete as possible.

(Can present as a summary of what you have discussed above, but keep a focus on goals and objectives of the project, nothing wrong with restating what you have already stated above)

METHODOLOGY

In order to perform the experiments with variable parameters a dynamic environment was required. We created an environment that allows us to specify before the beginning of the experiment what variables are to be used. The variables that can be chosen are quantity of robots, sensor distance, density of obstacles in the environment, environment width, environment height, the seed used to randomly generate the environment, and an optional time limit.

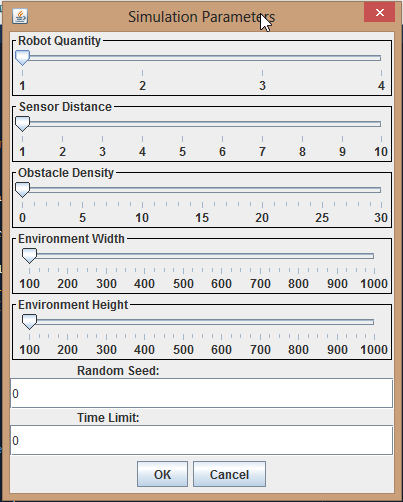


Figure : The parameter selection window

Using the Random Seed supplied by the user, the simulation creates the random environment that will be used for each algorithm for the current experiment. The environment is generated by selecting two random numbers between one and four inclusive to represent the width and height of a rectangle. Next a random number is drawn between one and the width of the environment exclusive representing the starting x coordinate of the rectangle. Similarly, a random number is drawn between one and the height of environment exclusive to represent the starting y coordinate of the rectangle. The environment is represented in memory as a Boolean matrix. When the starting x and y coordinates and the width and height of the rectangle are determined, the corresponding values in the matrix are marked as true, meaning they are occupied. The total number of occupied elements in the matrix is totaled. This process is repeated until the environment contains at least as many elements as is required to achieve the appropriate obstacle density. Using the smallest possible environment, 100 elements wide and 100 elements high, it is possible to have up 15 elements more than is required. This would create and environment with 0.0015% greater obstacle density than specified by the user.



Figure : Example of a randomly generated map with 30% obstacle density. The darker grey area represents the confines of the experiment. The black objects represent obstacles in the environment.

The simulation compares each of the three algorithms simultaneously. A window opens with each algorithm appearing in its own tabbed pane. Initially, a timer is set to perform a robots operations every 10 milliseconds. If the user desires to speed up the simulation, a ‘Fast Forward’ is present that reduces the delay between operations to 1 millisecond. A ‘Stop’ button is present to allow the user to pause the experiment. The ‘Stop’ button also has the added functionality of generating a report when pressed that contains a listing of each of the experiment’s parameters as well as the percentage viewed and movements taken by each algorithm.

Each algorithm run independently of each other algorithm but there are some aspects of the implementation that are constant among them. Every algorithm maintains two separate Boolean matrices, the viewed matrix and the occupied matrix. In the viewed matrix a value of true means that that particular element of the environment has been viewed and in the occupied matrix a value of true means that that particular element of the environment is occupied. An element of the viewed matrix will always be false until a sensor on a robot has occupied the corresponding element of the environment. At this time the element in the viewed matrix will be changed to true. The occupied matrix is the actual map that was generated at the outset of the experiment.

Another aspect that is constant among each algorithm’s implement is the way in which a robot’s bearing is handled. Our environment is designed such that the robot will only ever move in one eight directions. Each of these eight directions is represented as an integer value ranging from 0 to 7. When the robot is at the 0 bearing, it is facing toward increasing values of the x axis of the environment. As the bearing value is incremented, the degrees that the robot has rotated from 0 are incremented by 45. So if the robot is at a current bearing of 3, it is facing the direction 135°.

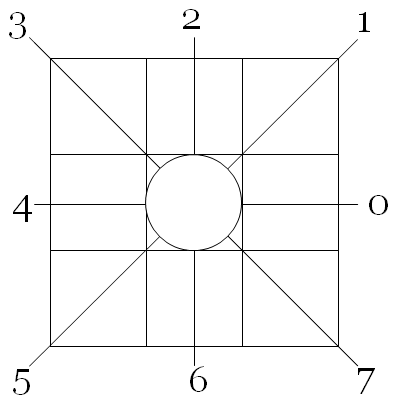


Figure : The Bearing system is represented as a series of integer values ranging from 0 to 7. Each integer value corresponds to a 45° increment from the increasing values of the x axis.

In the first tabbed pane is the Random Wanderer algorithm. At any given moment a robot has at most three possible physical movements. It may rotate 45° left, rotate 45° right, or move one unit forward. A Random Wanderer robot generates a random number between 0 and 3 exclusive. If the number is 0, the robot rotates left. If the number is 1 and the space directly in front of the robot is unoccupied, it moves forward. If the number is 1 and the space is occupied, a new number is drawn. If the number is a 2, it rotates right. This process is repeated until the percentage of the environment viewed is equal to 100%.

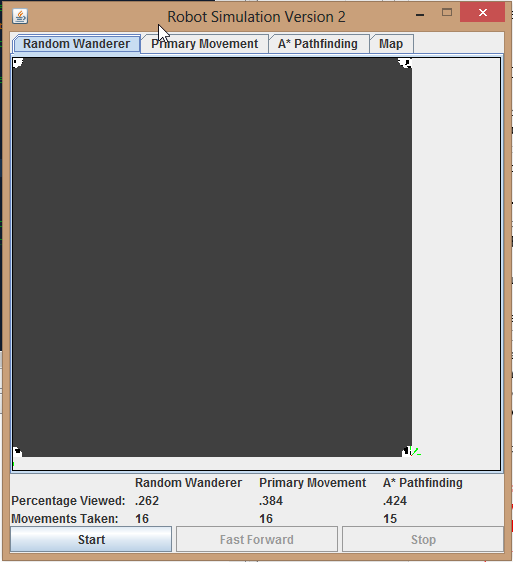


Figure : Each pixel represents one unit of the environment. This experiment contained four robots. Robots are represented as a single red pixel and their sensors are represented as green wedges. The area that has not been viewed is represented as dark grey while area that is unoccupied and has been viewed is light grey and area that is occupied and has been viewed is black.

The second tab in the simulation window displays the Primary Movement algorithm. The Primary Movement algorithm contains no mechanism to acquire a goal or to find a path to it. Instead, a robot will move forward as long a series of conditions are met. If the space directly in front of the robot is unoccupied and moving to that unit will allow the robot to unveil more of the environment, the robot will move forward. If either of these conditions is not met, the robot will rotate 45° counterclockwise and reevaluate. If a robot has achieved a full 360° rotation and no movement meets the requirements, the robot determines that it is finished and ceases to search for appropriate movements. If the conditions are continually met, the robot will continue to search until the percentage of the environment viewed is equal to 100%.

The final tab in the simulation contains the A\* pathfinding algorithm. Our implementation of the algorithm does not deviate greatly from the description in a previous section. The primary difference is in how the Neighbor list is populated. For a node to be inserted into the Neighbor list it must meet all of the original criteria, i.e. it must be adjacent to the current node, but it must also be unoccupied and it must have been viewed at some time previous.

When making a determination as to which area of the environment to evaluated next, our aim was to choose the nearest point to the robot that had not been already evaluated, but using a standard method of nested loops to iterate through a matrix would not always accomplish this. This method would return the node nearest the origin that matched the criteria.

To find the nearest node to the robot, we were required to design an algorithm that would work in a similar manner to a set of nested loops, with the first node evaluated being the node the robot currently occupies and iterating outwards in a spiral manner. This would return a node that is the closest to the robot.

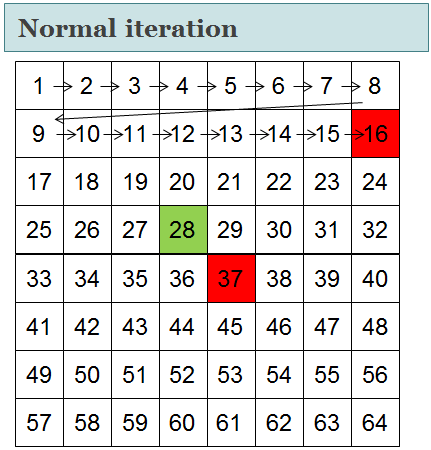


Figure : Using the standard method of nested loop iteration, a matrix is traversed through rows and columns starting at the origin. If the green cell represents the robot and red cells represent valid goals, the goal at 16 would be found before the goal at 37.

The goal acquisition algorithm we designed accomplishes spiral iteration by maintaining the current x and y coordinates to be evaluated in the matrix, the direction through which we are iterating in the matrix, and the width or height of the current row or column being evaluated in the matrix. Instead of a loop corresponding to the size of the row nested in a loop corresponding to the size of the column, we use one loop that corresponds to the maximum amount of cells the matrix could contain. When the loop is entered, the column height or row width is incremented according to the direction to be traversed. Next, a loop is entered that is the size of the row or column to be traversed. If the cell in the viewed matrix at the current x and coordinates has not been viewed, set that cell’s x and y coordinates as our goal. Otherwise increment the x or y coordinates accordingly. If no goal is found in a pass through the inner loop, increment the direction. If no goal is found at all, return failure.

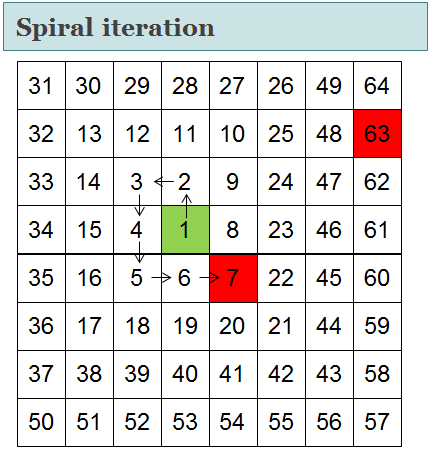


Figure : By designing an algorithm that iterates through a matrix by spiraling outward from a given node, we are able to locate the closest valid goal to the robot.

(Present the implementation details of the three algorithms, explain each individual algorithm, and explain your motivation to use three algorithms, the implementation difficulties of the algorithms (software perspective), finally explain your environment. Use some diagrams/pictures, etc… as you did in your term paper)

RESULTS

Analysis of the computed paths of the TSP reveal that.... (I am working on this part, I’ll write most of this later tonight and tomorrow)

CONCLUSIONS

(Will do this tomorrow after everything is done)

REFERENCES (Please update this list with our actual references. Do we have web references?? If so, see Manuscript\_Formatting.pdf)

[1] Dijkstra, E., A Note on Two Problems in Connexion with Graphs. In Numerische Mathematik 1, 269-271, 1959

[2] Hart, P. E., A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 100-107, July 1968

[3] Lester, P., A\* Pathfinding for Beginners. Retrieved April 24, 2013, from policyalmance.org:<http://www.policyalmanac.org/games/aStarTutorial.htm>, July 18, 2005