

Implementation of A* Artificial Intelligence Algorithm

A Gamebot Report
For the course on
Introduction to Intelligent Systems
(INTESYS)

Submitted by

Cansana, Jose Lorenzo M.
Jamalul, Gabriel Rasheed H.
Loyola, Leanne Marie C.
Marasigan, Giann Jericho Mari F.

Ms. Joanna Pauline C. Rivera
Teacher

October 27, 2019

I. Introduction

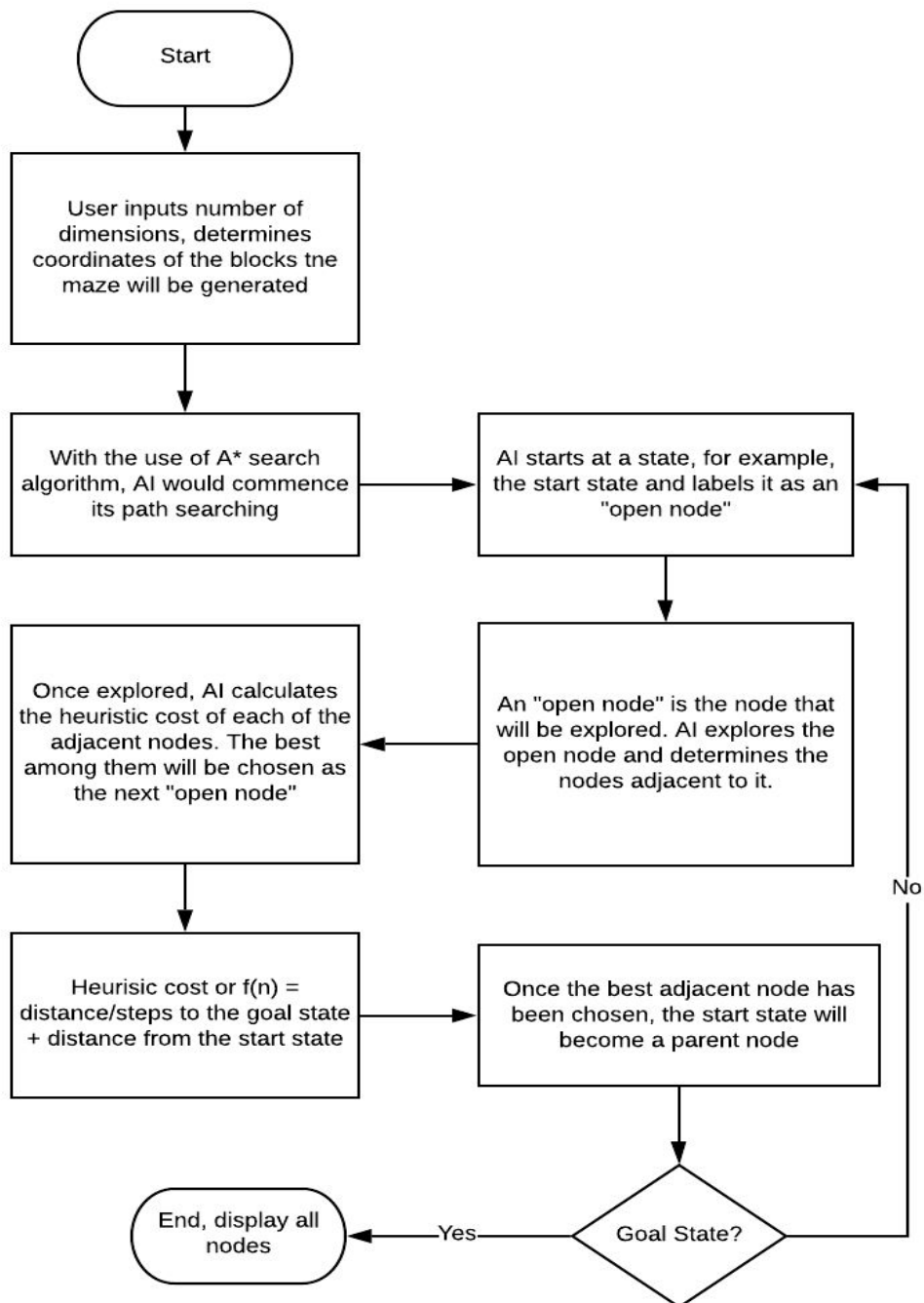
A generic stealth game is a type of game that emphasizes stealthy movements as their fundamental workings (Alkaisy, 2011). According to the Cambridge Dictionary (n.d.), stealth is defined as “movement that is quiet and careful in order to avoid notice, or secret or indirect action. “ This means that the player’s behavior is based upon hiding and avoiding any confrontation with possible danger or enemies in the game. And as such, stealth games require the ability to explore the environment and have several different solutions and ways of achieving objectives using different pathways, gadgets or approaches. It is to understand the possible paths players take through a given scenario and how they are impacted by different mechanics provided through the application of artificial intelligence (AI) (Trembly, Torres & Verbrugge, 2014).

In this execution, a maze environment was made and the AI (Maze Solver Bot) used for this game uses the A* search algorithm. According to Russell & Norvig (1995), A* search algorithm is an informed search algorithm, or a best-first search which its basic concept is based on using heuristic methods to achieve optimality and completeness. This means it is formulated in terms of weighted graphs and is guaranteed to find the shortest path between two points (start node and goal node) while attempting to minimize the cost that is defined for each specific problem it is applied to. In short, it avoids expanding paths that are already expensive, but expand the most promising first. A deeper explanation of the algorithm of the AI will be further discussed in later parts.

Overall, stealth games differ in the interpretation and reaction of the AI's player actions which gives game designers the ability to evaluate the relative existence of a stealthy path solution.

II. AI Features

Flowchart of Algorithm



Explanation of Algorithm

The Maze Solver Bot uses the A* Search Algorithm to traverse through the maze. It uses a heuristic function $[h(n)]$ to determine the best possible choices to make to arrive to a goal state. In the program, our heuristic function is the euclidean distance of the current point to the end point. In this section of the paper, we will describe, in detail, the actions being made by the program to solve the maze.

First, the user will be greeted by the prompt screen requesting for an integer input (n) to be used in the generation of the n x n maze board. This is all done in the `MainMenu` class. The input is then passed on through the constructor of the `MakeBoard` class. In this class, upon instantiation, the important variables are first populated such as the size of the board. After which, the class calls an initialize method from the JavaFX library right after. In this method, the board is generated as an n x n `GridPane` (JavaFX). The `GridPane`'s cells are then populated by `Rectangle` (JavaFX) objects that are initially filled with the color black. At the same time, a 2D String Matrix is being created and populated with the character 'X'. This 2D matrix will be later passed to the Search algorithm to start its traversal on the maze. In this case, black cells are equivalent to 'X' on the matrix which for the maze, is a wall, while white cells are equivalent to spaces on the String matrix which in the maze, is an open path. The cells are populated with a method that when the user clicks on the cell, the `Rectangle` will change color, and change its String register on the String matrix. After a maze is made, the player can now click on the 'Run AI' button and the Search algorithm will start its traversal.

When the 'Run AI' button is pressed, the active `MakeBoard` object holding the GUI Thread is passed on to the `A_StarAlgorithm` class. In this class, the inputted String matrix is read through the `InputHandler` class. Inputs look like this:

T XXX	'X'	Wall
X XXX	' '	Space
X	'T'	Bot (Start point)
X XX	'G'	Goal (End point)
X XXG		

The `InputHandler` class returns an `SquareGraph` object with the data of the map in it through the `readMap` method. This `SquareGraph` object is called using the `executeAStar` method to start the traversal. This method returns an `ArrayList` of custom-made Nodes in which these nodes have properties:

<code>costFromStart</code>	Used to determine the shortest path after the complete traversal of Search algorithm
<code>costToTarget</code>	Used by the heuristic function to determine which path left to take to get to the target

<code>totalCost</code>	The sum of <code>costFromStart</code> and <code>costToTarget</code>
<code>parent</code>	The source node precedent to the current node
<code>state</code>	Whether it is open (unexplored), or closed (explored)
<code>type</code>	Whether it is a wall or a space
<code>x</code>	Its X coordinate in the board / <code>GridPane</code>
<code>y</code>	Its Y coordinate in the board / <code>GridPane</code>

These properties serve much importance for the search algorithm to compute for the shortest path from start to end point. The `executeAStar` method starts the traversal by getting the Node format of the start point and the end point. This will be constantly be referred to by the execute method for the calculation of the heuristic function. The algorithm then gets the 3 cost properties of the start node and opens the nodes adjacent to it.

Opening nodes means that it recognizes the neighbouring space-type nodes next to the current node and make then available for traversal. At this stage, the algorithm then uses the heuristic function to determine which path to take. Though, using the algorithm implemented, the bot is split off into all possible areas. It will only traverse to its neighbor node if it is open or it checks if the `tentativeCost` (`totalCost` of the current node) is less than the `costFromStart` of the neighbor node. Traversal is done by changing either the X property or the Y property of the node. The traversal continues until either it reaches the end point or it hits a dead end -- in this case, all traverse nodes from the split point until the last point of the dead end path are changed to the closed state where it will be closed for exploration.

During program runtime, there will be many locations that will be considered current due the splitting of the bot. Though, at each stage, only one will be accessed and further traversed. This one current node will be determined by the heuristic function. The algorithm will keep looping through this until it reaches the end. As soon as it reaches the end, the algorithm will then trigger the `reconstructPath` method that reverse traverses the maze using the parent property as a reference to its source node. It also takes into consideration the `costFromStart` property so that the result will be the shortest path possible. The result of shortest path is then return as an `ArrayList` of nodes to the GUI for display to the user.

The GUI will display closed nodes as Grey cells while the path will be displayed in the green color. There will be cases that the inputted maze of the user is impossible to solve. This will be recognized by the program as such if all the space-type nodes are of closed-state. Upon recognition, the program will cease to execute and will return 'There is no path to target' text. If the traversal was successful, the Maze Solver Bot will display the maze and its corresponding colors, as well as the number of steps needed to reach the goal and the coordinates of the nodes of the path.

The Following are rules implemented in the AI:

IF

all sides are possible adjacent nodes or all sides are not blocks

THEN

AI will calculate for each heuristic cost and choose the best/has the least value among the choices. The chosen node will become the next open node and the current state where the AI is, will become a closed node. The AI will never choose the state where it came from because it will always have a higher heuristic cost compared to other adjacent nodes unless it is a complex maze.

This rule was used in line of using the heuristic function in the A* algorithm, where the node with least heuristic value gets expanded and searched. With this rule implemented in the AI, it will always go with the node that has the less heuristic value; meaning that the node is closer to the goal.

IF

no sides are possible adjacent nodes or all sides are blocks

THEN

The AI will stop and end the program saying it has reached a dead end or no possible paths could lead to the goal.

It is important to have this rule to prevent the AI from doing an infinite loop by passing through the same path. Also, this rule was implemented to check if there is a possibility of finding a path towards the goal. With this rule implemented, it always checks the possibility before commencing with searching for the path towards the goal.

IF

the coordinates of the adjacent node is inside the map and the coordinates of the adjacent node is not blocked

THEN

The AI takes into account the node and stores it in a list of adjacent nodes that are open and possible nodes that lead to the goal state.

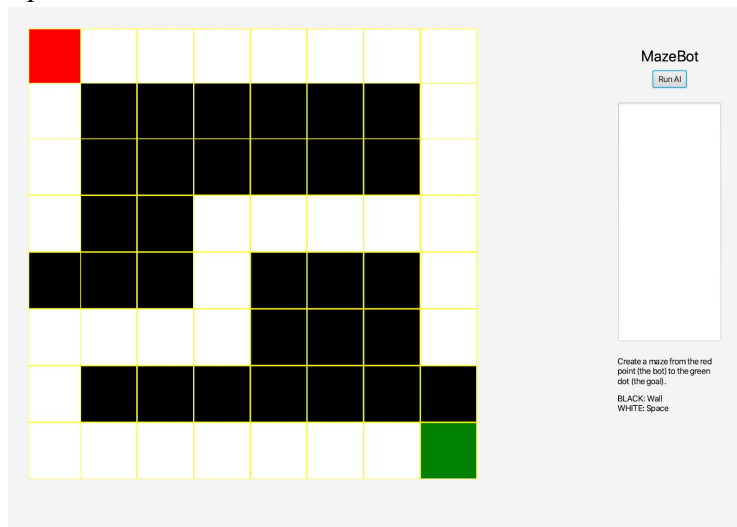
This rule was implemented since the code for finding adjacent nodes also includes the possibility of the AI checking outside the board. Thus, the function `IsInsideMap()` was created to prevent the AI from taking the coordinates of a node outside the board. Furthermore, The AI checks if the path inside the map is an open path. If it isn't, the AI does not add the node in the list of nodes possible to lead to the goal.

III. Results and Analysis

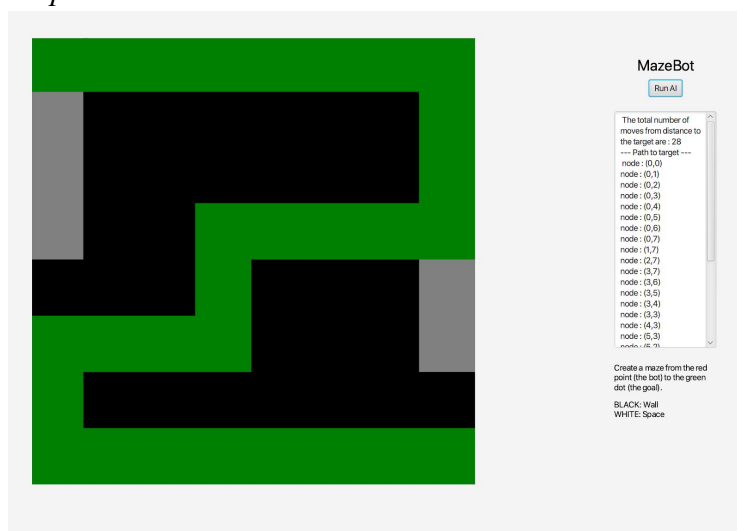
The AI bot can handle from 8 x 8 matrix up to 64 x 64 matrix, as mentioned in the specifications needed. The answers are generated instantly but due to simulation purposes, a delayed animation is place to given the user a glimpse of how the computer is traversing through the maze one by one. The more increased dimensions and the more complicated the maze, the longer it takes to generate an answer. Since the A* search algorithm will always check for its adjacent nodes and will always calculate for the heuristic cost, it will always take a longer time with complicated mazes. A simple maze like this can be easily solved by the algorithm used:

Lower-Limit: 8 x 8

Input

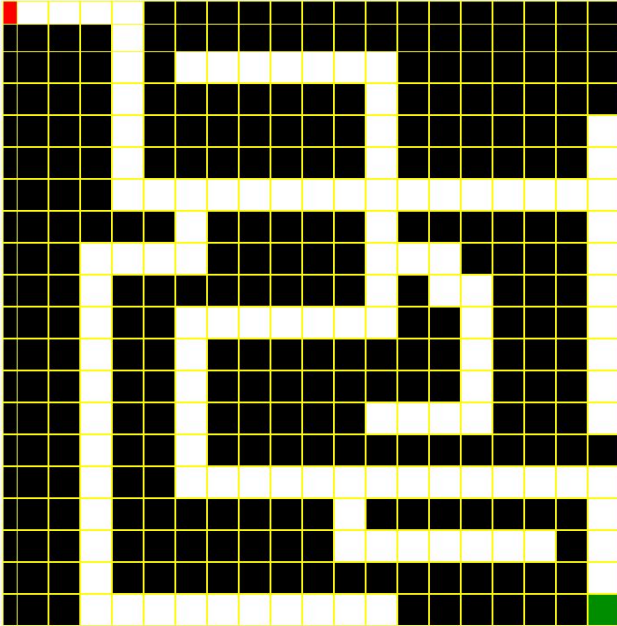


Output



20 x 20

Input



MazeBot

Run AI

Create a maze from the red point (the bot) to the green dot (the goal).

BLACK: Wall
WHITE: Space

Output



MazeBot

Run AI

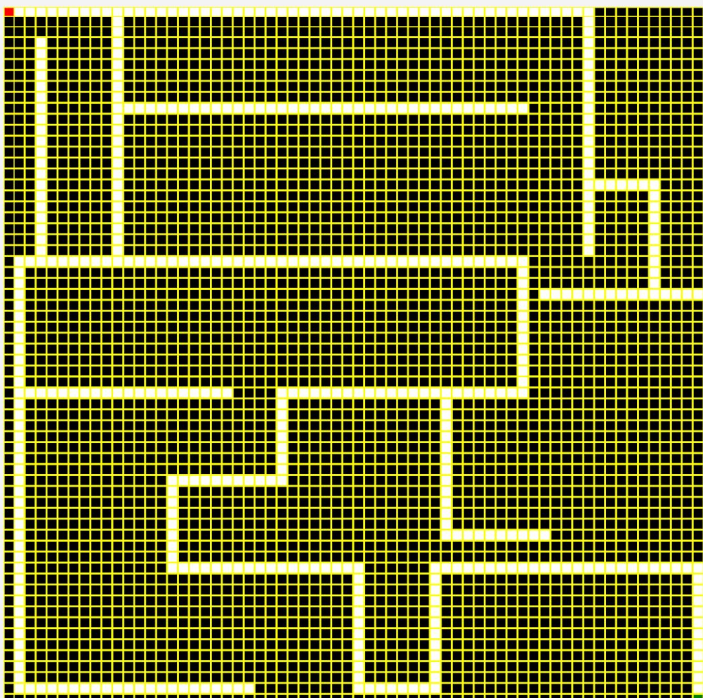
The total number of moves from distance to the target are : 50
--- Path to target ---
node : (0,0)
node : (0,1)
node : (0,2)
node : (0,3)
node : (0,4)
node : (1,4)
node : (2,4)
node : (3,4)
node : (4,4)
node : (5,4)
node : (6,4)
node : (6,5)
node : (6,6)
node : (6,7)
node : (6,8)
node : (6,9)
node : (6,10)
node : (6,11)

Create a maze from the red point (the bot) to the green dot (the goal).

BLACK: Wall
WHITE: Space

Upper-limit: 64 x 64

Input



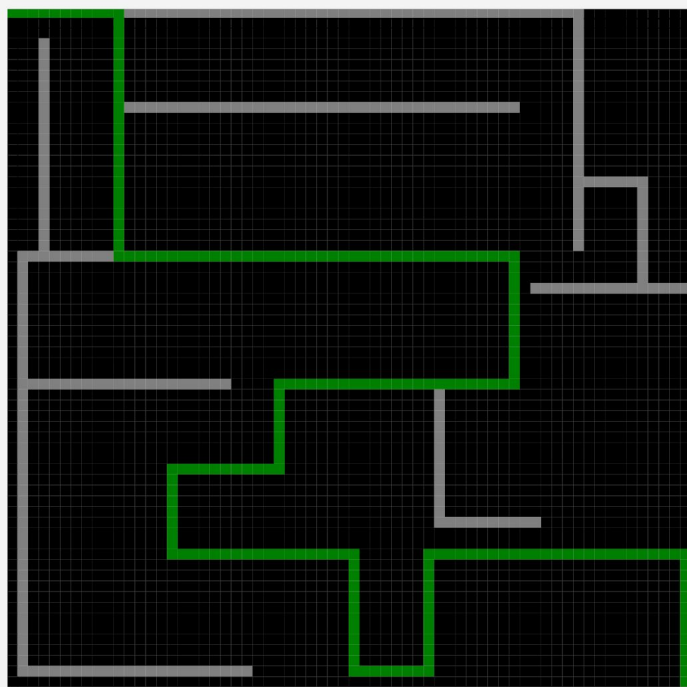
MazeBot

Run AI

Create a maze from the red point (the bot) to the green dot (the goal).

BLACK: Wall
WHITE: Space

Output



MazeBot

Run AI

The total number of moves from distance to the target are : 212
--- Path to target ---
node : (0,0)
node : (0,1)
node : (0,2)
node : (0,3)
node : (0,4)
node : (0,5)
node : (0,6)
node : (0,7)
node : (0,8)
node : (0,9)
node : (0,10)
node : (1,10)
node : (2,10)
node : (3,10)
node : (4,10)
node : (5,10)
node : (6,10)
node : (7,10)

Create a maze from the red point (the bot) to the green dot (the goal).

BLACK: Wall
WHITE: Space

Once the user follows all the specifications, the AI can handle any type of maze. The AI considered all the cases in its implementation, and was able to handle such cases. Furthermore, the theory behind the capability of the algorithm depends on the dimensions of the board. Simply put, the AI follows this theory:

The A* algorithm finds the shortest path possible towards the goal if and only if the dimensions of the board is less than $2^{32} \times 2^{32}$ dimension.

The generation of the board for the pathfinder made use of integers, which is usually 4 Bytes, and can take up to 2^{32} distinct states. With this in mind, the maximum integer value it can handle is up until 2^{32} . anything greater than this will result for the program to crash.

IV. Recommendations

- (1) As mentioned above, one of the weaknesses of the AI is that it also checks adjacent nodes outside of the board. Hence, the use of the method `isInsideMap`. In line with this, we recommend the next programmers that will code for pathfinding AIs to limit the bounds within the board itself so that unnecessary checking won't be done, hence removing `isInsideMap`.
- (2) Another weakness of the AI is that it traverses every available space within the board. Although it would mean that the AI won't miss the shortest path towards the goal, the big downside of this is that it won't be as fast in finding the best path towards the goal. Thus, we recommend to create a faster and optimal code that it only traverses the shortest path towards the goal. One suggestion might be a code similar to Deep Blue, a Chess AI that was able to look 12 steps ahead for every possible move the opponent will make. Thus, it must be possible to create code similar to this so that it will reduce the possibility of the AI traversing other open paths that may or may not lead towards the goal.
- (3) Another weakness of the AI is that before it returns if there is a path towards the goal or not, it needs to traverse first. Thus, it becomes more inefficient and time consuming as it was not able to identify instantly that there is no path towards the goal without traversing the board first. Thus, one recommendation would be to implement the algorithm with the capability of being able to look ahead the possible paths available towards the goal before traversing.
- (4) One weakness of the AI is that when the dimensions of the board is large, for example, a 64 x 64 board, the AI takes a longer time in finding the shortest path towards the goal. Thus, another recommendation would be to create a code that can find the shortest path possible faster and generate the nodes or steps to take towards the goal faster. Perhaps, for example, the use of divide and conquer or greedy algorithms might help in making the

code more efficient.

V. References

- Alkaisy, C. (2011). The history and meaning behind the 'Stealth genre' [Blog post], Retrieved October 25, 2019, from https://www.gamasutra.com/blogs/MuhammadAlkaisy/20110610/7764/The_history_and_meaning_behind_the_Stealth_genre.php
- Raptis, Dimos (2014) Artificial Intelligence - A* algorithm for pathfinding source code (Version 3.0) [Source code]. <https://github.com/dimosr/Pathfinding>
- Russell, S. J., & Norvig, P. (1995). *Artificial intelligence: a modern approach*. Noida, India: Pearson India Education Services Pvt. Ltd.
- Stealth [Def. 1]. (n.d.). In *Cambridge Dictionary*, Retrieved October 25, 2019, from <https://dictionary.cambridge.org/us/dictionary/english/stealth>.
- Tremblay, J., Torres, P. A., & Verbrugge, C. (2014, April). Measuring risk in stealth games. In FDG. Retrieved October 25, 2019, from http://www.fdg2014.org/papers/fdg2014_paper_33.pdf

VI. Appendix A: Contributions of Members

Name	Contributions
Cansana, Jose Lorenzo M.	Source Code, Algorithm, GUI
Jamalul, Gabriel Rasheed H.	Documentation, Algorithm
Loyola, Leanne Marie C.	Documentation, Algorithm
Marasigan, Giann Jericho Mari F.	Source Code, Algorithm