Finding Steve

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

Submitted by
Marcelo, Jan Uriel A.
Pelagio, Trisha Gail P.
Ramirez, Bryce Anthony V.
Sison, Danielle Kirsten T.

Ms. Pau Rivera
Teacher

October 27, 2019

I. Introduction

The project serves as a representation of a game classified under the stealth genre. Hu (2014) describes this category typically considered as a subtype of other genres. In this way, stealth games revolve around the minimization visibility as the player or avatar conceals himself while simultaneously reaching the goal. Some popular titles of which are under the franchises of *Metal Gear Solid*, *Hitman*, and *Assassins Creed* series which have received significant approval from consumers. These are games that primarily revolve around utilizing stealth. Nonetheless, a number of which implements the stealth aspect without attributing it as the primary feature of a game. Examples of which are *Elder Scrolls V: Skyrim* and *World of Warcraft.*

A common denominator among stealth games focuses on the encounters on analogous obstacles which are aimed to be passed without alerting other elements or entities. Accomplishments are often associated towards navigating within and environment in order to reach a point or achieve specific actions. In this way, Borodovski (2016) elaborates that pathfinding is an essential aspect towards stealth games. These are both applied towards the player or avatar and game elements or objects that interact with the environment. These entities would often require the need to obtain the shortest or most optimal path to achieve certain goals.

The system implemented demonstrates this feature of stealth games with the application of artificial intelligence. There are 2 primary elements which are identified through the environment and the agent. Therefore, these likewise comprise the starting point and end point which represents the goal of the agent.

Agent

Finding Steve implements a single avatar which constitutes as the artificial intelligence in the project. With this, it is modeled as a zombie mob from the popular video game title *Minecraft*. As such, this likewise serves as the entity that perceives its environment throughout its implementation. Moreover, as these percepts are received, appropriate responses are generated in order to accomplish the goal of the program. The primary objective of the agent is to arrive at the goal using the most optimal path. However, there are aspects of which that are significant to consider through the implementation of the system. The agent only contains the capabilities to traverse through the environment spaces that are not identified as walls. Therefore, movement can be restricted depending on the specifications or geographical features of the environment.

Environment

The primary goal of the agent is to reach a defined endpoint in the environment generated. Through this, there research provides 2 components towards setting in which the agent traverses through. Empty spaces represent the areas which are available for movement for the zombie artificial intelligence. This is illustrated as the grass blocks from Minecraft. In addition, the environment can likewise contain walls, which portray spaces that cannot be traversed by the agent. These are visualized as wooden planks and are placed around the environment or map. Furthermore, the placements of walls are specifically defined by the user through the graphical user interface (GUI). The preparation of the map is an integral process of the system which precedes the traversal of the agent. With this, a preliminary procedure is to provide the dimensions of the environment. The project implements a map with a structure of a square, therefore the height and width are always identical. Walls are then placed based on the discretion of the user. After which, the start and end spaces of the system are to be defined in order to have reference points towards the accomplishment of the traversal.

Search Algorithms

The development of an agent that efficiently traverses the environment requires a search algorithm. These represent problems through simulated exploration of a defined state space by generating successors or proceeding states from the current conditions. Primarily, the project implements this feature through the utilization of nodes, which are the components that comprise the state. These states are the configuration of the map or environment which accurately describes the location of the agent. Nodes are generated, expanded, and explored through a tree structure in order to obtain an efficient solution. With this, the study uses a search algorithm under the heuristic category in order to obtain an optimal path for the agent. Specifically, the A* Search Algorithm is implemented as it provides an informed search that provides considerations on the estimated cost and actual cost of a given solution.

II. AI Features

System Flowchart

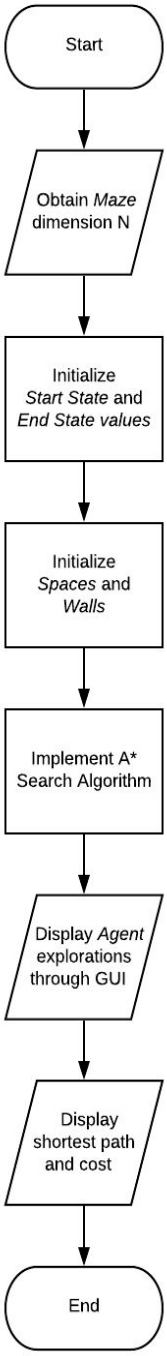

Figure 1: Flowchart of Program

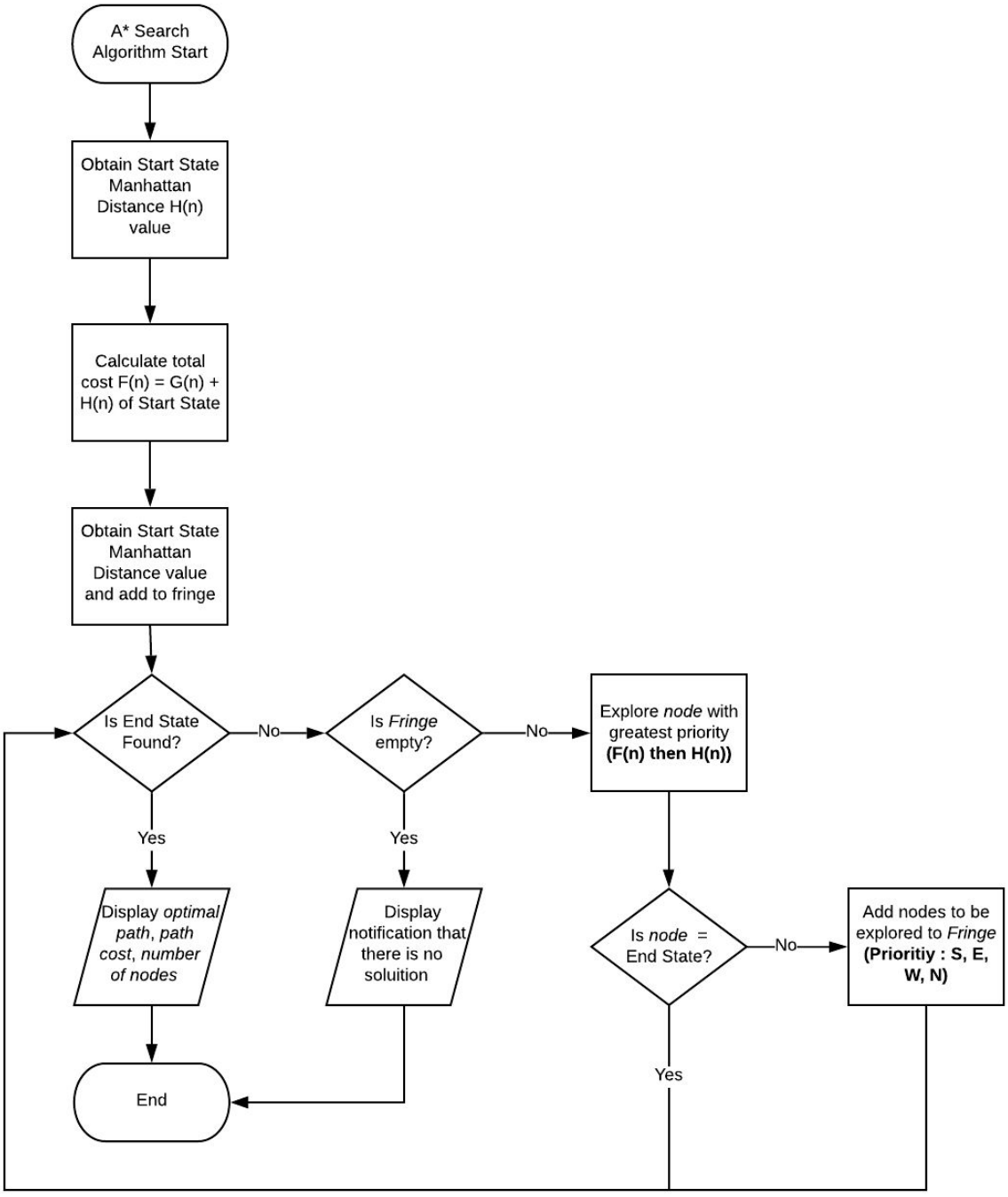

Figure 2: Flowchart of A* Algorithm

Data Structures Used

Abstract Collection

A foremost component of the construction of the program to be used, in highlight to the main solving algorithm, would be to test and try several algorithms that would output the best result in consideration with the time complexity which would concern the number of nodes created, space complexity that will take into account the maximum nodes to be generated and optimality which would give the most optimal solution to the problem. One way that the programmers have efficiently implemented this trial-and-error process was to find flexible ways in order to manipulate their data in order to suit the different needs of the algorithms. This was made effectual through the use of the Abstract Collection data structure.

The Abstract Collection data structure would allow polymorphism for several data structures such as hash maps, linked lists, stacks and trees. Through such, it would be effective in making a parent class, Solver that contains the Abstract Collection for the fringe, which contains the cells to be explored, and the explored list, which contains all explored cells. This Solver class will then be inherited by the different solving algorithms which could use their specific data structures that would fit the algorithm better. In the case of this program, three prospect algorithms were tested by the programmers namely, Depth First Search Algorithm, Breadth First Search Algorithm and lastly, the A* Algorithm.

The Depth First Search Algorithm utilized the data structure Stack because of how the traversal through each node would resemble that of a stack. The Breadth First Search however, utilized the Linked List Data structure in order to mimic the behavior of a queue that grows in a longitudinal manner. Lastly, the A* Algorithm, in order to make the simulation more efficient while considering the comparisons' time complexity during sorting, a priority queue was used. Through all this, code flexibility and versatility was considered in using the Abstract Collection data structure.

Array List

The program utilizes multiple Java ArrayLists a data structure that can store multiple objects. Primarily, these are used to temporarily store Nodes that are further processed for certain manipulations and operations. The primary advantage of which can be traced from the dynamic size ArrayLists provides. Sizes of the collection can be conveniently increased or decreased throughout the program. Similarly, access is likewise effective through the indexing rules of the dynamic array.

Specifically, the program uses an ArrayList object in order to obtain the possible nodes that can be explored in the traversal of the agent. These nodes are further processed based on several

considerations in order to be placed within the fringe of the search algorithm. The system primarily uses loops and the *get()* function implemented in ArrayLists in order to obtain and manipulated data throughout the movement of the artificial intelligence. These include calculating the actual and estimated costs that are used for the evaluation regarding which space is needed to be traversed.

Priority Queue

Search algorithms require collection data structures in order to store nodes that are explored and generated. With this, as the project considered applying the A* Search Algorithm, these data structures are implemented through *Priority Queues*. A primary reason for which is due to the nature of the search strategy that provides considerations on both the estimated and actual costs. The traversal of the agent is established through these values by obtaining the minimum sum among all the current nodes in the fringe. Thus, these are regarded with concern or priority in order to provide the succeeding actions of the agent.

The Priority Queue objects implement these considerations by associating every element by certain values. In this way, priority of access and retrieval is established based on the specified concerns. Through this, the priority of the program revolves around the evaluation function , where g(n) and h(n) are the actual cost and estimated cost respectively. With this, the retrieval of nodes are based on these values as implemented by the Priority Queue. Nonetheless, as the data structure is a queue, the first in first out policy is still inherited, if equal priority occurs.

Node

Nodes are an integral aspect of search algorithms as they comprise the various states of a problem. This data structure constitutes the search tree which is utilized to obtain efficient solutions. As such, the project implements such aspect through defining a separate class entitled *Node*. A property of which that a node comprises is value, which represents the state stored. These are of the *Maze* reference data type as it contains the configuration or specification of the map and the location of the agent. Therefore, this provides the information towards further considerations on the possible succeeding moves of the artificial intelligence. Moreover, the node likewise contains a *parent* property that represents the direct ancestor or parent of the current node. These properties are supplemented with functions towards their manipulation and access.

The Nodes serve as *states* in the program which would be considered in traversing the most optimal path to the goal. In order to store huge amount of information which would entail the entire state of the game, a copy of the whole maze state must be stored to the node. This would be applied to all of the searching algorithms implemented for the program test, thus the values and attributes are all focused on the Maze class of the program.

Point

The point data structure is a simplified version of the Space class wherein it only contains the x and y coordinate of the Space. This was made for optimality purposes since other attributes inside the Space class such as the maze, walls and heuristic costs are not being used in comparing whether the fringe or the explored list already contains a certain Space instance.

Another feature which allowed the Point class to optimize the performance of the program is its use of hashCode() function. This would reduce the time complexity in terms of determining if a certain list contains a point. Though not necessary, the use of this data structure would optimize the overall performance of the program since one of the major processes would involve determining if a certain list contains a point.

Artificial Intelligence Algorithms

Through testing all of the prospect algorithms, the programmers have decided to use A* with some modifications to its original algorithm. This was considered due to its optimality in all cases while maintaining an optimal growth rate in terms of time complexity and space complexity. Breadth First Search necessitates an exponential growth O(b^d) for both the time and space complexity while Depth First Search, though entailing a linear complexity in terms of space because of the node deletion, would have a chance of not finding the optimal path as it would also result to a possibility of ensuing to an infinite loop.

The A* Algorithm was chosen due to its accuracy and performance in terms of informed searches, since the end goal's position is known, a better simulation through the use of heuristics could be done. As stated by Pynes (2018), Later Hart, Nilsson and Raphael proofed A2, an improvement of the A1 which is a faster version of Dijkstra's algorithm, to be the most optimal in pathfinding and later on this algorithm was renamed as A*. DFS only considers neighboring paths and moves in the most immediate path to the goal while BFS finds a wider area of paths, but nothing specific. A* in contrast to this would calculate the distance of all neighboring paths while calculating the distance to the goal which would result to a depth guided search.

Program Run

As seen in the Figure 1 above, the program starts through the initialization of the map which was actuated through obtaining the Maze dimension as well as initializing the desired start and end states and lastly the spaces and walls. Through this, the AStar class will be initialized and called with the initialized maze as parameter and a Boolean value indicating which heuristic to use, this was mainly

used for the testing of the algorithm in order to determine the most optimal and admissible heuristic to be used for the program.

For the final program test, Manhattan Distance was chosen as the heuristic to be used since the bot is only restricted to move in four directions namely: Up, Down, Left and Right. Using the Euclidean distance might underestimate the total cost of the possible moves to get to a cell, especially considering that the maze has walls. Patel (2019) from Stanford has also stated that the most admissible heuristic for a simulation with 4 directions would be the Manhattan Distance. The AStar class's solve method will then be called to perform the simulation of path finding and output the most optimal path including explorations of the Agent and the cost for the whole simulation.

A* Algorithm

Initially, the start state would be initialized as the first node in the priority queue, thus its Manhattan distance will be obtained as well as its total cost F(n). This will then be added to the fringe to be the first node to be explored by the whole algorithm. The loop will start in which it would check if the End State is already found using a Boolean variable which is initialized to false at the start of the program.

A condition will be checked after passing the end state condition in order to check if the fringe is empty. In this condition, the program will know that all possible paths have been explored and no solution was found, thus a notification should be displayed that there is not solution to the problem. If the fringe is not yet empty, the bot will then explore the node with the greatest priority. At the start of the program, the node with the greatest priority is the start state. Throughout the whole run of the program, the greatest priority will be checked through the least total cost F(n) as the priority but if the F(n) of two nodes are equal then it would check which Node has the least heuristic value H(n). Another if condition will check if the node to be explored is already the end state, if not then it would add the possible nodes to be explored in to the Fringe.

At the point of adding the prospect nodes to be explored, several rules were followed. This is also where the significance of the modification in the native A-Star algorithm can be seen. The first rule is the priority of the directions to be traversed starting with South followed by East, West and lastly North. The programmers have decided that this would be the optimal traversal since the end goal would presumably be on the other end of the maze. Another rule to be considered in determining the spaces to be traversed would to consider whether the space is a wall or not. The second rule to be followed, that is the aforementioned modification to the original algorithm is that nodes that have already been explored or nodes that are already in the fringe will not be re-added to the fringe anymore since this would result in a redundant movement by the bot which would automatically be a

non-optimal path in finding the goal, since it is a maze and backtracking would already deduce that the path is non-optimal.

After the addition of nodes, the loop will then be iterated starting from checking if the end state is found followed by the condition if the fringe is empty and exploration of the nodes with greatest priority as checked by the Priority Queue data structure. This process will be repeated until the most optimal path is found or no solution was deduced. Through the end of the A* algorithm, the Graphical User Interface would then display the optimal path through a highlighted path as well as its total path cost together with the time complexity represented by the total number of nodes created for the whole run of the algorithm.

Rules Implemented

Direction Priority
IF Direction Priority = 'N' THEN store the space at the North of the current space
ELSE IF Direction Priority = 'S' THEN store the space at the South of the current space
ELSE IF Direction Priority = 'E' THEN store the space at the East of the current space
ELSE IF Direction Priority = 'W' THEN store the space at the West of the current space

This rule denotes the priority as to how the spaces will be traversed or added to the priority queue in order. This can be found in the Space class to be called by the A* Algorithm solver in obtaining the possible spaces to be traversed from the current position. The decision for this priority is critical to the assumption of where the end goal would be in order to create the least amount of nodes in traversing to find the most optimal path.

Node Exploration [f(n), h(n)]
IF a node contains a greater function value f(n) THEN that node obtains a greater priority
ELSE IF a node contains a lesser function value f(n) THEN that node obtains a greater priority
ELSE in the case of nodes with equal function values f(n) the node with greater heuristic value h(n) obtains a greater priority

The statements provided elaborates the evaluation considered in obtaining the succeeding node to be explored. Primarily, the priority is implemented based on the function value f(n) of a given node which are the sums of the estimated cost h(n) and actual cost g(n). As the nodes to be explored are stored within the fringe, a priority queue collection object, these are obtained depending on the node with the least function value.

Definitely, there are cases in which the nodes will have equal values during a traversal. In these cases, the priority of exploration is shifted towards the heuristic value h(n) of a state in the maze. Nonetheless, if these would likewise proved to be equal, exploration processing would then be associated towards the direction priority specified for the traversal. With all of these, rules regarding node exploration are implemented during the initialization of the AStar class. Specifically, the rules are performed by instantiating a comparator class for the fringe and explored Priority Queue objects.

Fringe Addition
IF the possible succeeding space contains a wall THEN the node containing such state is not added to the fringe
IF the node has been previously explored THEN the node is not added to the fringe
IF the node is already in the fringe THEN the node is not added to the fringe

The first if condition would ensure the correctness of all traversals of the AI bot. Aside from the start and end positions, walls are a key part of the system because it defines the structure of the maze and the uniqueness of each that would encompass major changes in terms of the possibility of the paths that can be taken by the AI bot. This condition can be seen in the getNexts() function of the space class which would return all possible spaces to be traversed from the bot's current position. This primarily checks if the neighbor spaces have walls that can be checked through the Boolean value found in each space class instance in order to return the correct potential spaces that can be traversed.

The second and third if conditions would can be found in the AStar algorithm program itself. It is part of the inner loop wherein in each iteration, the possible nodes from the explored node will be added to the fringe. This is one of the major modifications or improvements that the programmers have decided to add to the original A* algorithm in order to improve its complexity in terms of space and time. This condition would signify that nodes that have already been explored or are already in the fringe would not anymore be re-added to the fringe. For the second condition, the logic behind this is that re-exploring previously explored nodes would mean backtracking from the original position, thus it would definitely lead to a non-optimal path with the non-minimum cost. This solution will no longer be relevant to the overall solution; thus, it was omitted by the programmers.

Another condition that was formulated by the programmers in order to optimize the node traversals of the bot is the condition when a node is already in the fringe. This would be effective because a node already in the fringe would mean that it has already been explored by other another node which has higher priority. Thus, the state can be reached by another more optimal option so then it does not have to be traversed anymore.

Through all of this, while focusing on the construct and goal of the bot in reaching the end goal with minimum steps, the programmers have tried to make the bot more intelligent in devising conditions that would already omit and ignore irrelevant solutions to the overall problem.

Non-Existing Solutions
IF fringe is empty and solution is not found THEN display that there is no solution

The programmers have also considered cases wherein all possible paths have already been explored and there will be no way of reaching the end goal, thus in the main loop of the A star algorithm class, a condition was formulated such that when there are no more states to be explored and there is still no solution, a notification in the GUI will be displayed to inform the user that the bot cannot find a possible solution as there is no possible solution to the maze problem.

III. Results and Analysis

Capabilities

Optimal Path

The bot implemented in the project, Finding Steve, is able to effectively develop an optimal path given any descriptions or details of the map. These include configurations that either contain walls or not. A primary reason for which is due to the algorithm of choice implemented in the project. Specifically, this pertains towards the A* Search Algorithm. The solution provides considerations based on the estimated and actual values of the distances in order to effectively provide the optimal move at a specific state. In this way, the bot is able to think rationally through computing and comparing the values necessary to proceed to another space. Furthermore, this also implies that the bot is able to act rationally as it applies these computations towards its movement and traversal in the map. It produces the desired results depending on the decision produced within the values considered.

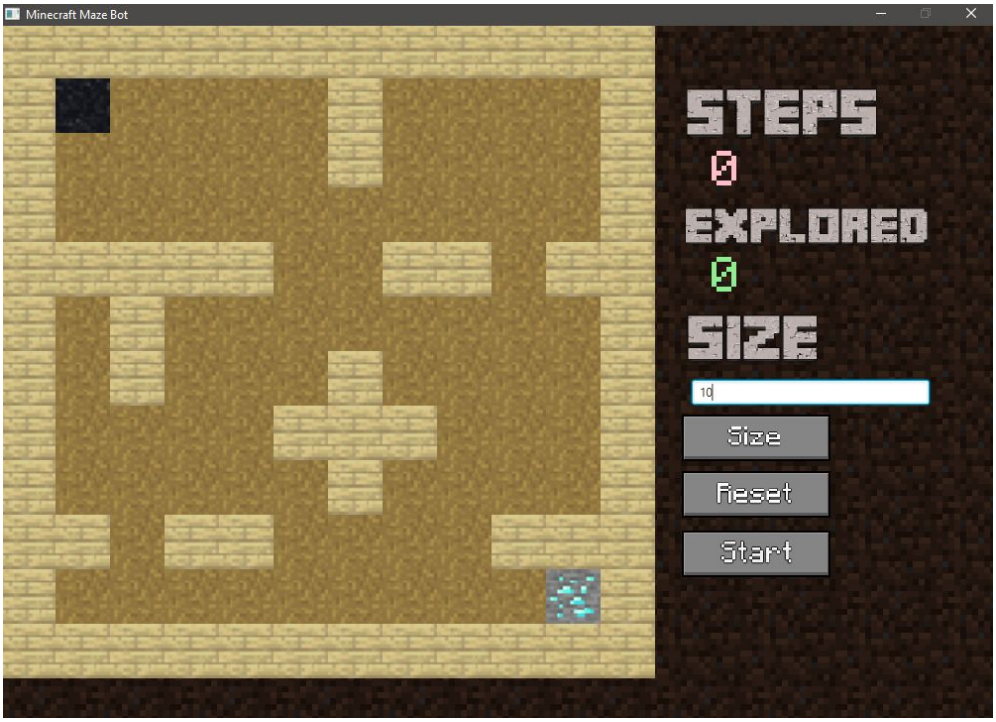

Figure 3. Sample Map Configuration

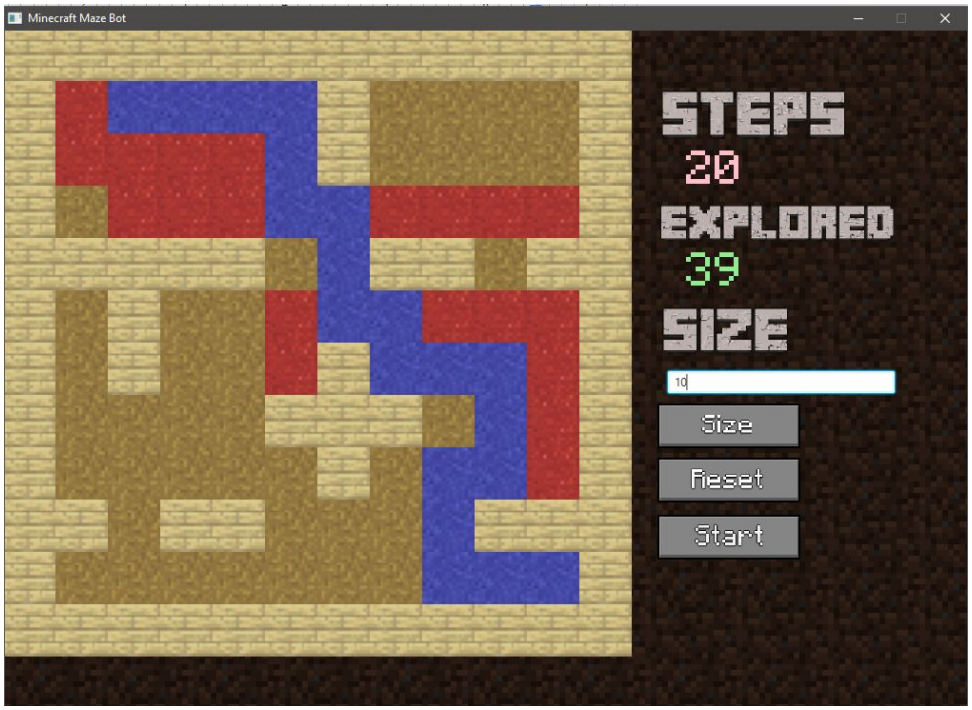

Figure 4. Optimal Path Result

Disregard possible paths that are irrelevant

Since the programmers have modified the A* Algorithm in order to fully maximize its optimality, the ability for backtracking a certain path was removed. In relation to this, a feature wherein the bot will be able to teleport in order to explore the most optimal path in the priority queue per move, if necessary, was added.

As the AI bot in the project traverses through the map, these would produce a number of actions that is in need of analysis. This is primarily because movement is based on the values comprised in a particular space or state. However, the project addresses this issue through disregarding the states and paths that are irrelevant. Primarily, this is observed within the fringe, as nodes which contains states that are previously explored are not added. Therefore, the bot does not backtrack towards the path it is currently taking to reach the end state. These would prove several advantages based on decreasing the possible decisions to be analyzed. Therefore, these certainly pertains towards the aspect of artificial intelligence regarding thinking rationally. In this way, the bot is instructed to only consider the states or moves that are necessary for its traversal.

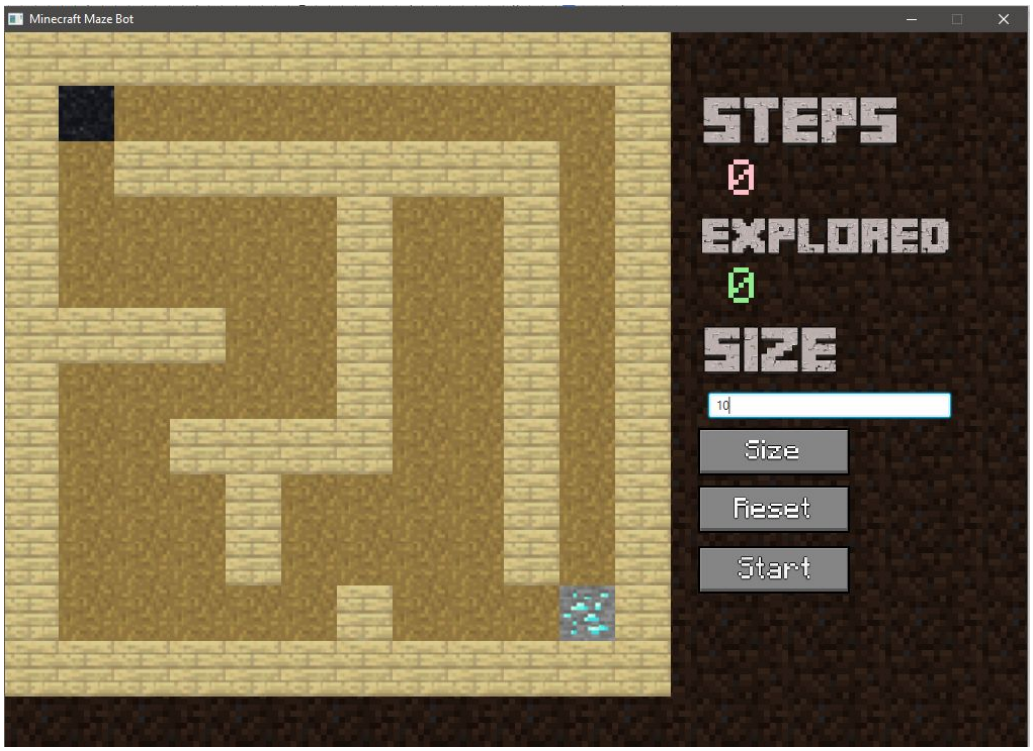

Figure 5. Map Configuration Containing Optimal and Non-Optimal Paths

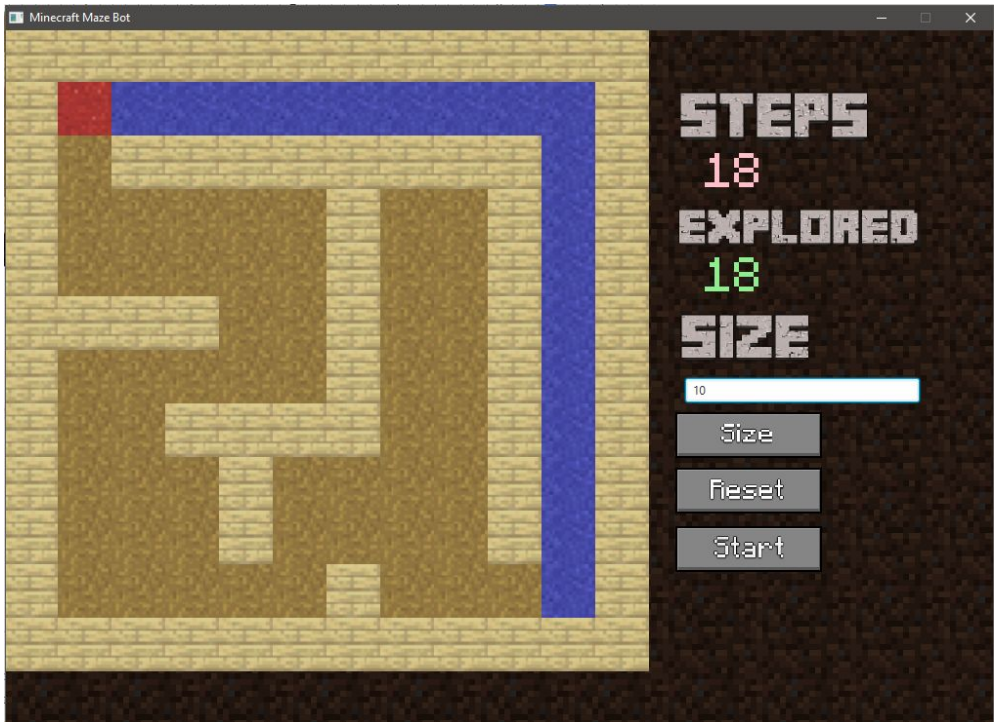

Figure 6. Maze Bot Disregarding the Non-Optimal Path

Efficient Time and Space Complexity

With the results generated in several trials, these certainly suggest significant improvements regarding its time and space complexities. This is in comparison towards other algorithms in terms of Uninformed or Blind Strategies and Informed or Heuristic Strategies. Specifically, the bot is able to perform more effectively and efficiently against the native A* Search Algorithm, which had served as the basis for the strategy implemented in terms of its traversal. The modifications implemented as explained in the second chapter are the primary causes for which.

The complexity of time in the context of artificial intelligence refers to the number of nodes generated by a program. The system had significantly considered this matter through decreasing the possibilities of states that are in need of evaluation. Specifically, nodes that are already explored or are currently in the fringe are automatically not assessed for further traversals. Similarly, these would likewise suggest improvements in terms of space complexity, which correspond towards the maximum number of nodes that are stored in memory. The bot does not consider paths that are not proven to be more optimal compared to the current path being taken. These are evaluated in terms of the heuristic and actual value of the distances given a specific state of the maze.

In these ways, the bot implemented in the system definitely corresponds to the feature of being intelligent. These considerations towards node creation and storage supplement the capabilities of the

bot towards its rational factor. It is able to reach or provide the needed results which ensures its completeness dimension. However, the project likewise considers optimality and efficiency both in terms of time and space.

Restrictions

Goal enclosed within walls

A problem was encountered in dealing with a situation wherein there will be no solution because all paths to the goal are blocked by walls. The A Star Algorithm only handles or explores based on the heuristic, it does not have prior knowledge on where the walls are, thus it is similarly related to a blind search in this manner. Due to this, the bot cannot conclude directly if the maze has no solution or if all paths to the goal are surrounded by walls unless all possible paths to the goal have already been explored. In cases such as this, the bot thinks humanly, as it is how a human would think in blind searches or in maze problems, exploring all possible paths, but it is not intelligent because it would still explore paths even if there is no actual solution to the problem.

In cases like this, it ended up in an infinite loop and the programmers have tried to devise a way to determine if there are no possible paths to the goal by determining if all paths have already been explored in order to be able to display an output that there is no solution to the problem.

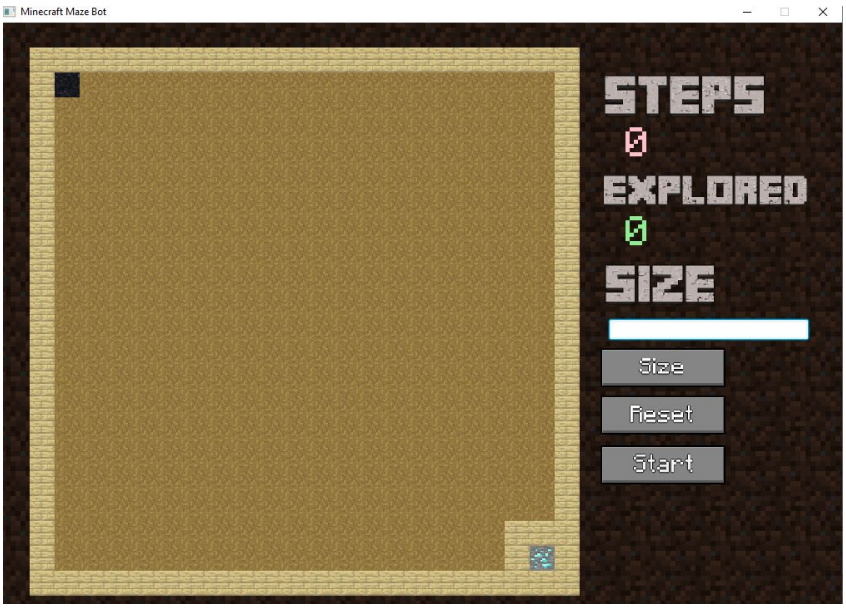

Figure 7. Map Configuration Comprising a Goal State Enclosed Within Walls

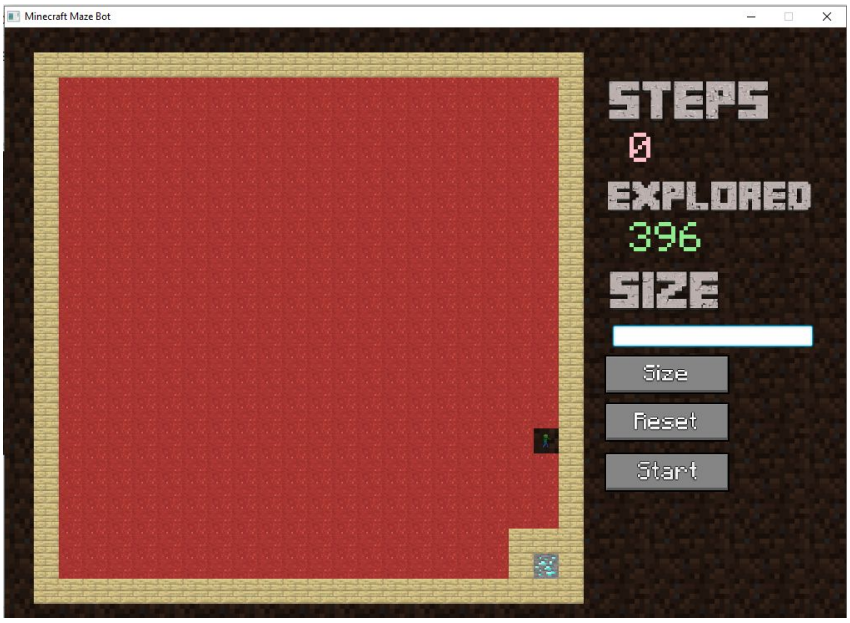

Figure 8. Handled Error Case on Goal State Enclosed Within Walls

Causality between walls and the generation of nodes in exploration

An observation was deduced by the programmers from several runs of the program. The AI Bot seems to be performing worse when more walls are added into the maze, thus the efficiency and intelligence of the bot in relation with the number of walls in the map is inversely proportional with each other. The most efficient path finding solution by the bot could be seen in cases wherein there are no walls added to the maze. In this case, no irrelevant path is explored because of the admissibility of the heuristic, only the steps which would lead to the most optimal path will be explored, thus it would produce the most optimal space complexity which would produce the least nodes possible in a solution.

Due to this, it can be inferred that the intelligence of the bot would depend mostly on the construction of the map, as well as how it would fit to the priority of the traversal of the bot starting from the north, east, west and south.

Figure 9. Map Configuration Containing No Walls

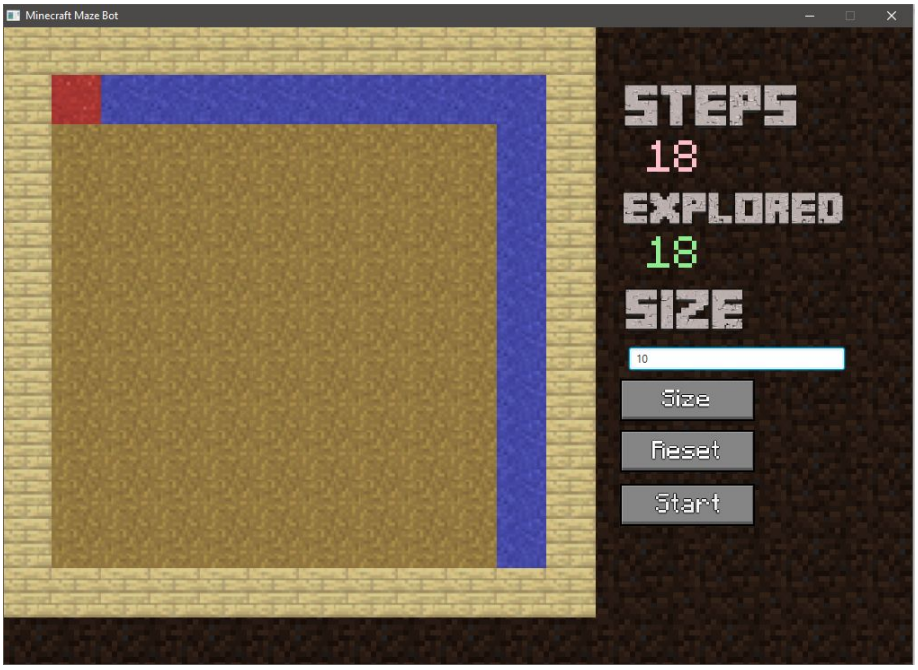

Figure 10. Traversal Result

IV. Recommendations

In creating a more optimal bot, consider the placement of walls in the program

One of the major problems in the algorithm that was devised for the Maze problem in Finding Steve is that the only consideration for the heuristic of each state was the manhattan distance of each state from the goal. One of the major considerations or restrictions for the bot, the placement of the walls, was not taken into account in this way.

Thus, in order to devise a more intelligent bot that would be able to find the path with the most optimal space and time complexity, the number of neighbor walls by the current state could also be considered. It could be added to the heuristic value of the manhattan wherein it would take into account the number of walls that are neighbor to the space. In this way, the bot would be able to define the more specific heuristic value of the state while not underestimating the walls surrounding it.

This change in the program could lessen the chances of exploring paths which would lead to a dead end, thus creating less nodes and having a better overall space complexity or maximum number of nodes required for the problem.

Searching Algorithm could simultaneously explore from the goal to the bot and vice versa (Divide and Conquer Algorithm)

Another major issue that would reduce the intelligence of the AI bot in the algorithm of Finding Steve is when there is no solution to the problem or all paths to the goal are blocked by walls. In a typical maze, walls would generate several paths, thus in order to make the exploration more efficient in terms of faster path finding, a divide and conquer algorithm could be used. In the case wherein the bot is already informed of the placement of the walls as well as the position of the goal, it possible possible that exploration could be done in two simultaneous ways, one from the AI bot and one from the goal state.

These explorations will only stop once there is an intersection already from spaces explored by the bot and the spaces explored from the goal state. In this case, the runtime will be faster and the detection of cases wherein there will be no solution to the problem can we detected in a faster way since when one of the explorations have already explored all paths possible, the bot can already conclude that there is no solution. This would make the bot more intelligent since it would give the bot a more informed approach that maximizes all knowledge and information in terms of wall placing ang goal position.

V. References

Borodovski, A. (2016, April). *Pathfinding in Dynamically Changing Stealth Games With Distractions*. Retrieved from Semantic Scholar: https://pdfs.semanticscholar.org/412b/abc908ea380418ab61ee105bad2f6396fdd6.pdf

Hu, M. (2014). *Game Design Patterns for Designing Stealth Computer Games*. Retrieved from Semantic Scholar: https://pdfs.semanticscholar.org/f872/c75468f6cca3a7e229f1eab8d47c9311f6fb.pdf

Patel, A. (2019, October 12). *A*'s Use of the Heuristic*. Retrieved from Stanford Computer Science Theory: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Pynes, D. (2018, June 18). *Graphs & Paths: A**. Retrieved from Towards Data Science: https://towardsdatascience.com/graphs-paths-a-getting-out-of-a-maze-a3d7c079a5c6

VI. Appendix A: Contribution of Members

Name	Contributions
Marcelo, Jan Uriel A.	Algorithm Design, Graphical User Interface
Pelagio, Trisha Gail P.	Algorithm Design, Report and Documentation
Ramirez, Bryce Anthony V.	Algorithm Design, Report and Documentation
Sison, Danielle Kirsten T.	Algorithm Design, Graphical User Interface