# The Puzzle Pathfinder: The Legend's Guidefinder!

MCO1 (Gamebot) Report
for the course
Introduction to Intelligent Systems (CSINTSY)

Submitted by
Adrada, Jasper John N.
Natividad, John Austin Mikhail T.
Ramirez, Benmar Sim G.
Razon, Luis Miguel Antonio B.
Romblon, Kathleen Mae V.

Dr. Raymund C. Sison
Professor

March 4, 2023

## I. Introduction

The Gamebot is a program that searches for the optimal path from the start to the goal state. It traverses through a maze, and this maze can be set up by the user by defining its size which ranges from an 8 by 8 to a 64 by 64 grid. The user can also choose which tiles in the grid are walls, which are displayed to be yellow, while the walkable tiles are displayed as white. The start of the maze will always be at the tile at row 1, column 1, and the goal will always be at row *n*, column *n*, where *n* is the size of the maze. Once the user is finished setting up the grid with walls, they can pick from three search algorithms and then choose to start their run of the game. The maze exists so that when the Gamebot searches for the optimal path from the beginning to the end state using each of the three algorithms, it can be determined which among them returns the most cost-optimal path.

The user can choose from three search algorithms to apply to their run of the Gamebot, which are the Uniform-Cost search, Greedy Best-First search, and A* search. These three algorithms can be implemented into the Best-first search algorithm, their only difference being the evaluation function. The evaluation functions that are applied to the Best First Search algorithm are used by the Gamebot to find the path from the beginning tile to the goal, and it may use only the actual cost from the beginning to the current tile, only the heuristic estimate, which is the path cost from the current tile to the goal state, or both actual cost from beginning to current tile and the heuristic estimate. The Uniform-Cost search considers the path with the lowest cost, the Greedy Best-First search considers the path with the best heuristic estimate to the goal, and the A* search considers both the path with the lowest cost as well as the best heuristic estimate to the goal.

The point of the program is to compare each of the three search algorithms with each other by how they find the most optimal path in terms of cost. It is expected that we find the most optimal path using the A* search algorithm since it makes use of both actual path cost from the beginning to the current node as well as the heuristic estimate of each node based on how far they are from the goal state. Applying both actual path cost and heuristic estimate gives the estimated cost of the best path that continues from each node to the goal state. The heuristic estimate used for this algorithm is the Manhattan distance formula, which is an admissible heuristic as it does not overestimate the path cost, and is best used when movement is restricted to four directions only, namely up, down, left, and right.

## II. AI Features

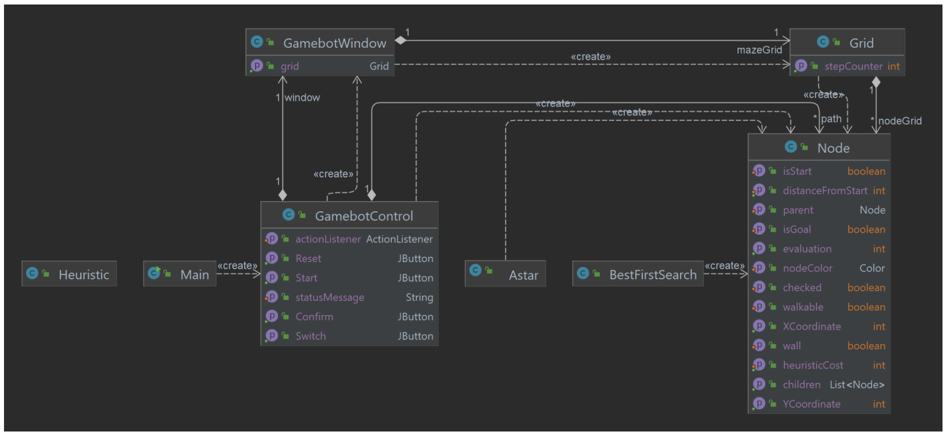


*Image 1. UML Properties of each Class*

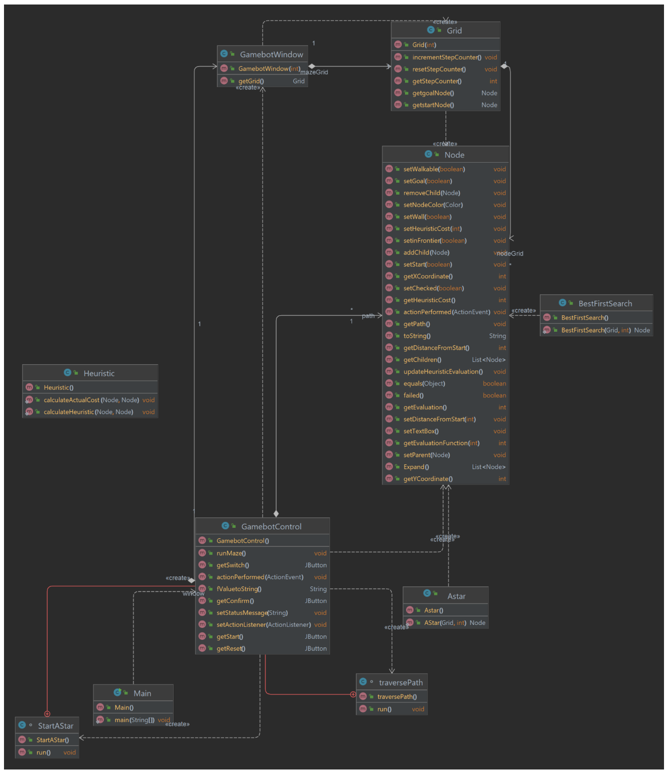


*Image 2. UML of Methods and Interaction between Classes*

Before tackling the artificial intelligence (AI) of the bot, it is essential to first describe how its environment works and how the bot will be interacting with it. As seen in *Images 1 and 2,* the program starts off from the *Gamebot Control* class which is responsible for initializing the other objects that are necessary to generate the environment or the maze which includes the bot and the start and end goal states. The class also acts as a way for the bot to act to the environment, apart from the "*Confirm"* button which generates the environment (Refer to *Image 3*), there are also the "*Start*" and "*Switch*" buttons which cause the bot to begin the search and changes the search algorithm that the bot is using respectively.

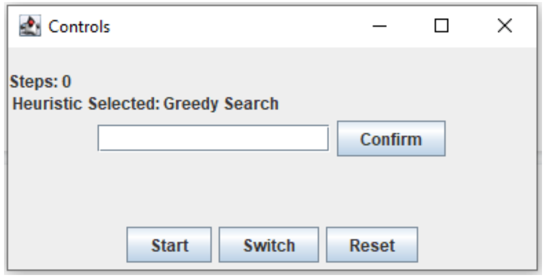
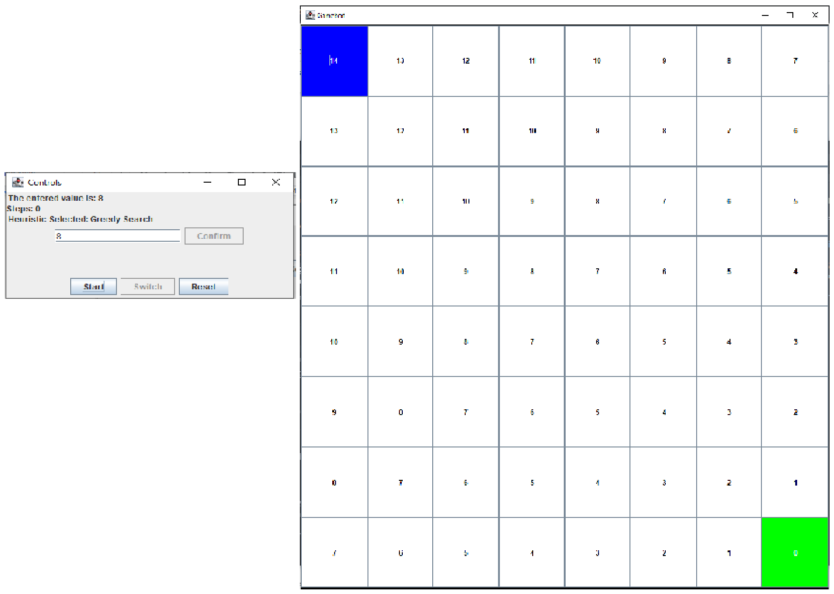

Steps: 0
Heuristic Selected: Greedy Search

*Image 3. Gamebot Control class window*

The textbox and "*Confirm*" button in *Image* 3 are responsible for generating the grid where the maze and the bot are initialized (Refer to *Image 4*), When generating the grid, the *Gamebot control* class calls and initializes the *Gamebot window* class which loads a window containing objects from the *Grid* class which is responsible for loading *n* by *n* amount of objects from the *Node* class. For ease of reference, the object from the *Grid class* will be called grid and the objects from the *Node* class will be called nodes.

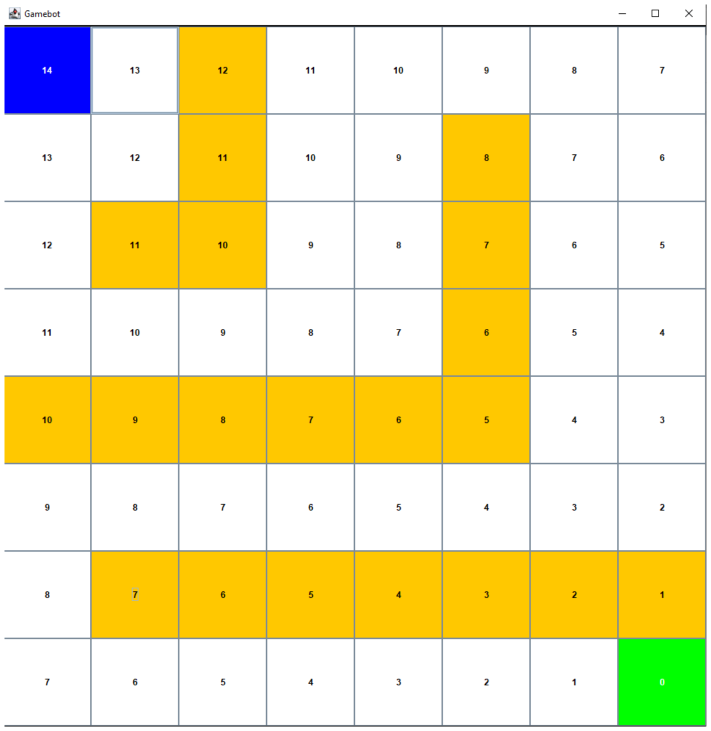
The grid contains two special types of nodes: the *start* and *end* nodes which will be considered as the initial and goal states of the AI when it traverses into the maze. As for the normal nodes, each of them has its own states which are set to none by default. These states include: *none, wall, frontier, checked, and path*. Each state has its own representative color to help the user distinguish each node: blue represents the *start* node or the starting position of the bot, green for the *end* and *path* nodes, dark gray for the *frontier* node, light gray for the *checked* node, and white for the *none* or default node. Before the search algorithm starts, the user has a choice of which nodes they will set to *none* and *wall* which will dictate the maze that the bot will traverse (Refer to *Image 5)*. With the nodes and the grid set up, the bot can officially traverse through the maze by clicking the "*Start*" button. When the grid is initialized, it loads n x n amount of nodes where they each have three essential attributes that are initialized: the children, parent, and heuristic evaluation attributes. The children attribute is a list of nodes containing nodes that are to the north, south, east, and west of the node if applicable. The parent node is initialized as null and will be used as a reference for the path during the search algorithm. Finally, the heuristic evaluation is calculated through the Manhattan heuristic and is represented by the number displayed in the middle of the node. The heuristic evaluation will be discussed further in later portions of the paper.



The entered value is: 8
Steps: 0
Heuristic Selected: Greedy Search

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 |
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 |
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 |
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 |
| 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

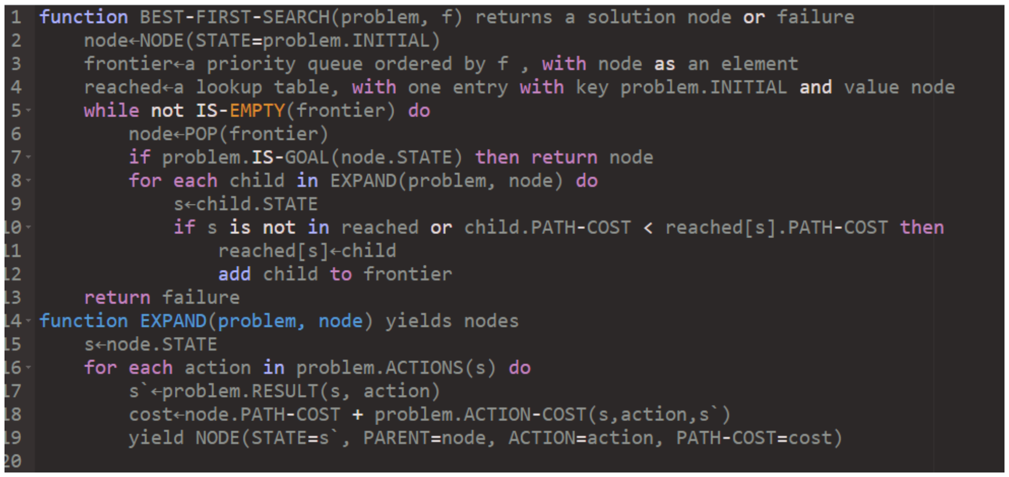
*Image 4: Gamebot window showing an 8 by 8 grid as specified by the Gamebot control window*



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 |
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 |
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 |
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 |
| 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

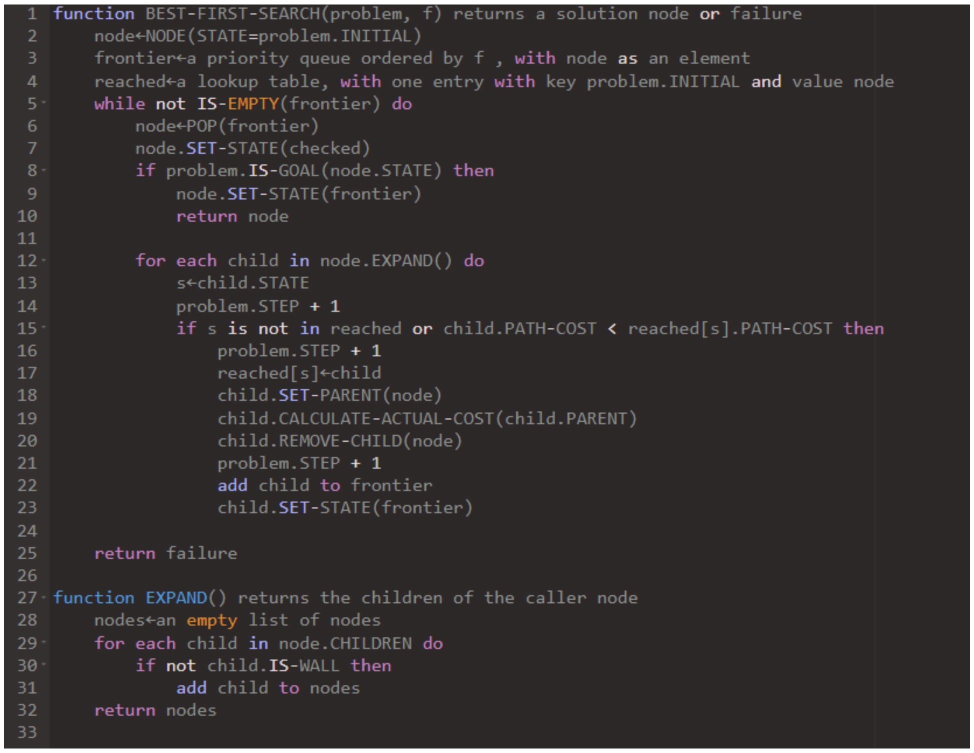
*Image 5: A grid where the user specified the walls by clicking the nodes*

For the AI of the bot, it was implemented using the best-first search algorithm. The search algorithm was modified to best suit the needs of the AI to cooperate with its environment while maintaining the same rules as the original algorithm. This search algorithm was specifically chosen because of its ease of implementation and it provides us with the option to use and test other search algorithms by only changing the *f* of the function.



```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
    node←NODE(STATE=problem.INITIAL)
    frontier←a priority queue ordered by f , with node as an element
    reached←a lookup table, with one entry with key problem.INITIAL and value node
    while not IS-EMPTY(frontier) do
        node←POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            s←child.STATE
            if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
                reached[s]←child
                add child to frontier
    return failure
function EXPAND(problem, node) yields nodes
    s←node.STATE
    for each action in problem.ACTIONS(s) do
        s`←problem.RESULT(s, action)
        cost←node.PATH-COST + problem.ACTION-COST(s,action,s`)
        yield NODE(STATE=s`, PARENT=node, ACTION=action, PATH-COST=cost)
```

*Image 6: Best-first search algorithm (Russell, 2022)*



```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
    node←NODE(STATE=problem.INITIAL)
    frontier←a priority queue ordered by f , with node as an element
    reached←a lookup table, with one entry with key problem.INITIAL and value node
    while not IS-EMPTY(frontier) do
        node←POP(frontier)
        node.SET-STATE(checked)
        if problem.IS-GOAL(node.STATE) then
            node.SET-STATE(frontier)
            return node

        for each child in node.EXPAND() do
            s←child.STATE
            problem.STEP + 1
            if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
                problem.STEP + 1
                reached[s]←child
                child.SET-PARENT(node)
                child.CALCULATE-ACTUAL-COST(child.PARENT)
                child.REMOVE-CHILD(node)
                problem.STEP + 1
                add child to frontier
                child.SET-STATE(frontier)

    return failure

function EXPAND() returns the children of the caller node
    nodes←an empty list of nodes
    for each child in node.CHILDREN do
        if not child.IS-WALL then
            add child to nodes
    return nodes
```

*Image 7: Modified best-first search algorithm*

When the method for the algorithm is first called, it initializes three variables, namely *node*, *frontier*, and *reached*.

*Node* is initialized with the initial node which is the node located at (1,1) and is distinguished by the color blue. *Frontier* is a priority queue that will contain nodes which are sorted by the *f* of each node in ascending order which can differ, depending on the type of search that will be done – the types of searches available for the program will be tackled in depth in the latter part of this section. Lastly, *reached* is a table or a hashmap containing key-value pairs where the keys are the node and the values are the actual costs calculated from the initial node to the current node. After all of these have been initialized, it is now time to proceed to the logic behind the AI.

First, the bot checks if the *frontier* is empty because if the *frontier* is empty, then it would mean that there are no more nodes that could be expanded and added to the *frontier* because all the nodes that could be expanded have already been explored and the goal state could not be found in any of them. This happens if the goal state cannot be reached by the algorithm due to the goal state or the initial state being enclosed with no possible path leading to each other. If the *frontier* is not empty, then that would mean that there are still unexplored nodes that might possibly lead to the goal state and the program will continue where the next node that would be explored is the node at the front of the *frontier*, the next node is assigned to *node* and its state is set to *checked* since it is eliminated from the frontier.

After the node is assigned the object at the top of the *frontier*, it then checks if the node is the goal node. This serves as the terminating clause for the loop since if the method does not return anything before the loop ends, it will indicate that there is no goal node and failure will be returned instead. If the current node is indeed the goal node, it then returns the current node, therefore, terminating the flow since it has no need to continue, else the program will keep on going where it loops through the children of the node that can be obtained through the expand method that will be discussed after the best-first search algorithm.

As it loops through the children of the node, it checks if the child is in the *reached* table or if the path cost of the child is less than the path cost of the same node in the *reached* table. It does this to make sure that there are no duplicate nodes in *frontier* and *reached* since it might affect runtime and present a worse solution instead of the good one. If the child passes these conditions, then it gets added or updated to *reached*, the node becomes the child's parent, calculates the child's path cost from the path cost of the parent node, the parent node gets removed from the children of the child node, and lastly, the child gets added to *frontier* and the state of the child gets set to the *frontier* status. If the child does not apply to any of the conditions, it gets skipped.

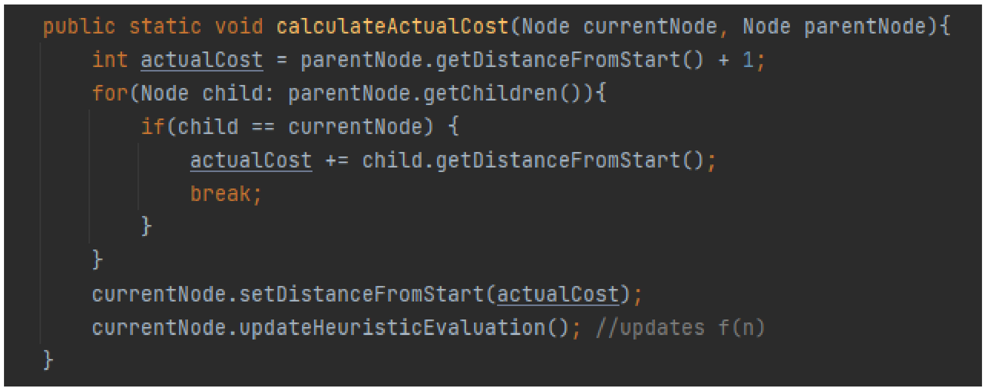
After it loops through all the children, it goes back to the start of the loop where it checks if *frontier* is empty.

As for the expand method, it was decided that it would be implemented in the *Node* class so that it will just return the array of children that it had when the grid was initialized. This made getting the children easier since it was already part of the nodes which meant that it only needs to return the *children* array of the nodes. This method was safely implemented because of the statement in the modified Best-First Search algorithm where the parent node is removed from the children, therefore preventing reached nodes from getting into *frontier* again, and followed up by the condition in the method where the nodes are returned if and only if they do not have the wall status.

The Gamebot is capable of switching between three intelligent search algorithms that utilize the Best-First Search algorithm, with different evaluation functions.

**Uniform-Cost search algorithm**

The Uniform-Cost search algorithm is an intelligent search algorithm that systematically chooses the lowest path cost. As it chooses the lowest cumulative path, the search algorithm will always return the optimal solution. To achieve this, the algorithm uses the *frontier*, a priority queue implemented in the Best First Search algorithm. For this search algorithm, the evaluation function of each node is expressed as $f(n) = g(n)$ where $g(n)$ is the total path cost from the node. The algorithm was modified in such a way that the *frontier* will sort the total path of each node enqueued in the *frontier* in ascending order. This will achieve the goal of a Uniform-Cost search of systematically choosing successor nodes with a smaller path cost. In the code, the total path cost of a node is determined by incrementing the node's parent's path cost by 1 and storing the new path cost in the node. The calculation of the path cost is implemented in the method named *calculateActualCost*.

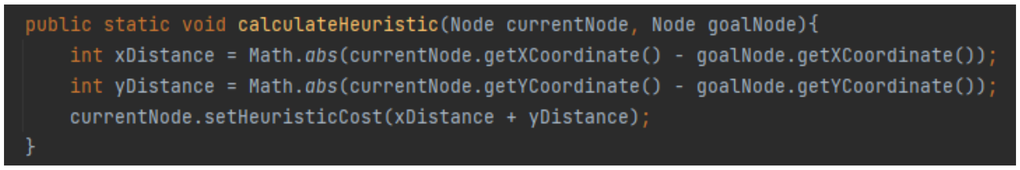


```java
public static void calculateActualCost(Node currentNode, Node parentNode){
    int actualCost = parentNode.getDistanceFromStart() + 1;
    for(Node child: parentNode.getChildren()){
        if(child == currentNode) {
            actualCost += child.getDistanceFromStart();
            break;
        }
    }
    currentNode.setDistanceFromStart(actualCost);
    currentNode.updateHeuristicEvaluation(); //updates f(n)
}
```

*Image 8: Utility method to calculate the value of g(n).*

**Greedy Best-First search algorithm**

The Greedy Best-First search algorithm is another intelligent search algorithm that is implemented in the Gamebot. This search algorithm, compared to the Uniform-Cost search algorithm, recognizes a heuristic value $h(n)$ rather than the path cost, which results to an evaluation function of $f(n) = h(n)$. The heuristic is the estimated cost of the cheapest path from the current state to the goal state. In this program, the group decided to implement the Manhattan distance heuristic as it was deemed the most efficient out of the other admissible distance heuristics. In a uniform grid, such as the maze in this program, the Manhattan distance heuristic is an efficient heuristic estimate as it calculates the distance by calculating the sum of the absolute difference between the vectors of a state and the goal state, in this case, a node's *x* and *y* coordinates. In the program, this is implemented through the calculateHeuristic method which performs the aforementioned calculations.



```java
public static void calculateHeuristic(Node currentNode, Node goalNode){
    int xDistance = Math.abs(currentNode.getXCoordinate() - goalNode.getXCoordinate());
    int yDistance = Math.abs(currentNode.getYCoordinate() - goalNode.getYCoordinate());
    currentNode.setHeuristicCost(xDistance + yDistance);
}
```

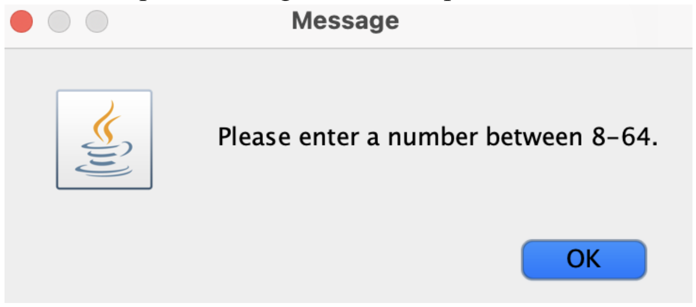
*Image 9: Utility method to calculate the value of h(n)*

**A* search algorithm**

The A* search algorithm is a search algorithm that determines the shortest path between two nodes in a graph. Compared to the aforementioned search algorithms, this search algorithm could be observed as a combination of both the Uniform-Cost algorithm and the Greedy Best-First search algorithm as the evaluation function of the A* search algorithm uses both the actual cost $g(n)$ and the heuristic cost $h(n)$ to form the evaluation function $f(n) = g(n) + h(n)$ of a node that will be enqueued to the *frontier*. This function allows the algorithm to take into account the current path cost of the node while having an idea of how far the goal state is which allows the AI to follow more informed and rational decisions as it is able to select the cheapest path toward the goal state. The heuristic function of this algorithm is also the same as the function used in the Greedy Best-First search, due to the same reasons. The algorithm is designed to cover both the limitations of Uniform-Cost search and Greedy Best-First search by being able to be more efficient by expanding fewer nodes while also being able to return the shortest path through the use of the *frontier*.

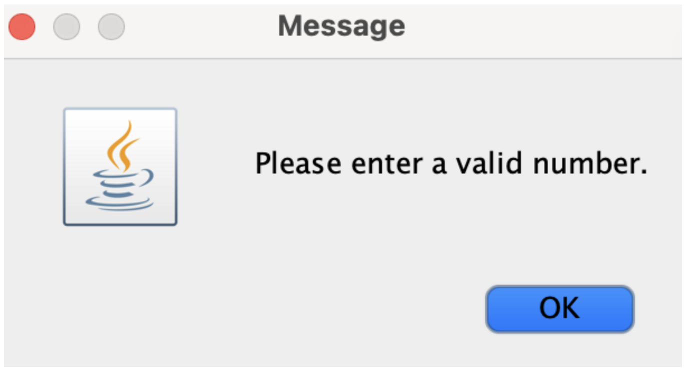
## III. Results and Analysis

To illustrate what the Gamebot is capable of, it is given three general objectives throughout the whole program. The first is to receive and comprehend numerical input, in this case, the size of *n* within a range of 8 to 64. The Gamebot has a text field and is able to identify if the input is not within the specified range, or if the input isn't a number.



Please enter a number between 8-64.

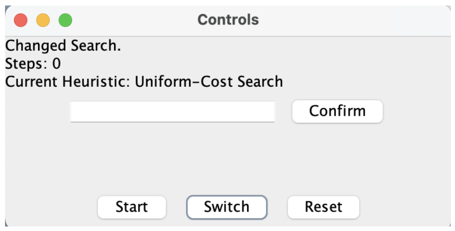
*Image 10: Message Dialog for input range*



Please enter a valid number.

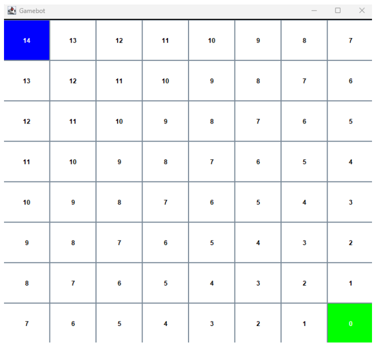
*Image 11: Message Dialog for input validity*

In terms of its capabilities, the Gamebot's second objective is to display the windows that allow the user to interact with the bot. The bot's programming generates a control window upon running that houses the text field and other buttons used to generate, reset and switch the heuristic of the created grid. This means it is able to provide us with visual data and other features to interact with the grid. Additionally, to help with the user's experience, tooltips are present for each button with its respective function description.



Changed Search.
Steps: 0
Current Heuristic: Uniform-Cost Search

*Image 12: Control window of Gamebot*


Gamebot

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 |
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 |
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 |
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 |
| 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

*Image 7: Grid window of Gamebot*

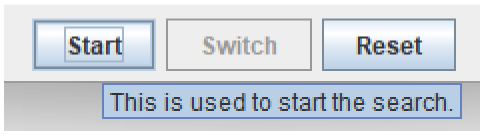
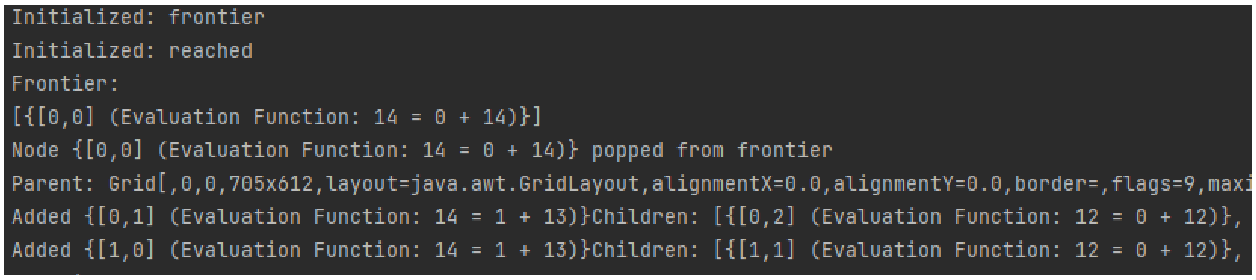


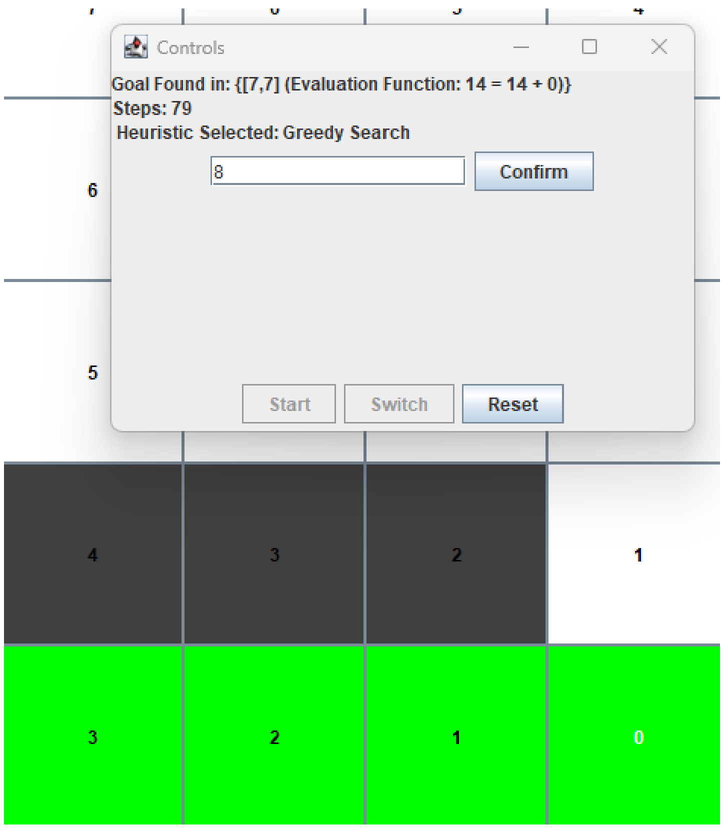
*Image 13: Tooltip for buttons*

Lastly, the bot’s third objective is to successfully and intelligently use the set of provided AI algorithms which are A* search, Greedy Best-First search, and Uniform-Cost Search (UCS) to achieve its goal of reaching the start and end nodes while navigating through the maze and its walls.



```
Initialized: frontier
Initialized: reached
Frontier:
[{[0,0] (Evaluation Function: 14 = 0 + 14)}]
Node {[0,0] (Evaluation Function: 14 = 0 + 14)} popped from frontier
Parent: Grid[,0,0,705x612,layout=java.awt.GridLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=9,maxi
Added {[0,1] (Evaluation Function: 14 = 1 + 13)}Children: [{[0,2] (Evaluation Function: 12 = 0 + 12)},
Added {[1,0] (Evaluation Function: 14 = 1 + 13)}Children: [{[1,1] (Evaluation Function: 12 = 0 + 12)},
```

*Image 8: Terminal when Greedy Best-First Search is running*

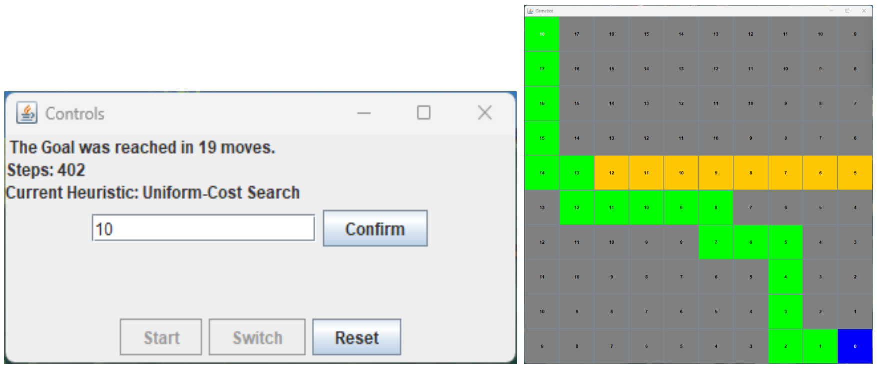

Controls

Goal Found in: {[7,7] (Evaluation Function: 14 = 14 + 0)}
Steps: 79
Heuristic Selected: Greedy Search

*Image 14: Status and Grid display when goal is reached*

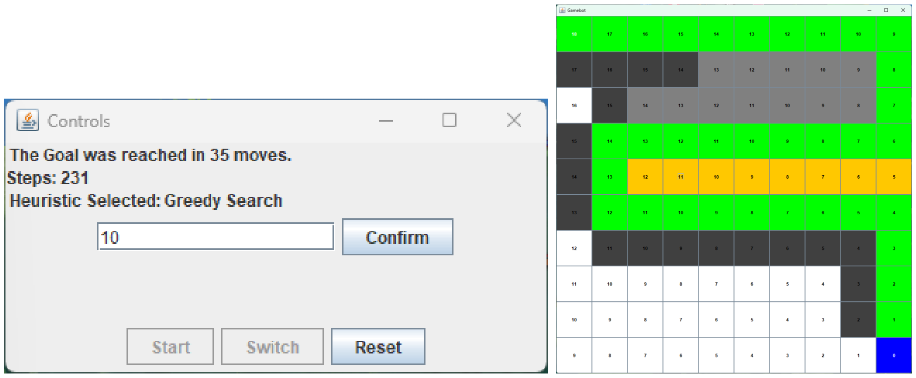
The bot's intelligence in efficiently solving the maze can be attributed to its combination of AI algorithms and rule base. A* algorithm helped the bot make informed decisions about which nodes to explore next, reducing search space and optimizing the search process. UCS ensured that the bot explored the maze with the lowest cost, while the Greedy Best-First search, despite not being the most accurate, provided heuristics that guided the bot in the right direction. Additionally, the rule base played a crucial role in enabling the bot to use these search algorithms and loop until it reached its goal. This demonstrates the bot's intelligence within its limitations and specific purpose.

Additionally, below are the results of each algorithm and their step counts. It is also apparent in the images the different approaches each AI algorithm takes.


Controls

The Goal was reached in 19 moves.
Steps: 402
Current Heuristic: Uniform-Cost Search

*Figure 1: Status (left) and Grid display (right) when using Uniform-Cost search*

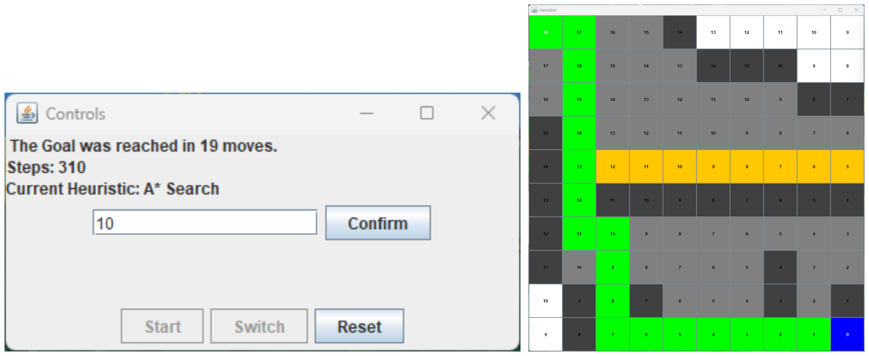

Controls

The Goal was reached in 35 moves.
Steps: 231
Heuristic Selected: Greedy Search

10 Confirm

Start Switch Reset

Figure 2: Status (left) and Grid display (right) when using Greedy Best-First search


Controls

The Goal was reached in 19 moves.
Steps: 310
Current Heuristic: A* Search

10 Confirm

Start Switch Reset

Figure 3: Status (left) and Grid display (right) when using A* search

To illustrate the differences between each algorithm's performance, efficiency, and results, a test was made wherein the algorithms were compared in a 10 by 10 maze with a horizontal wall on the middle of the grid, but leaving 2 spaces on the left.

The Uniform-Cost search algorithm was able to locate the path of 19 moves toward the goal state; however, this algorithm was the least efficient as it had to perform 402 steps of checking and expanding. The algorithm always ends up systematically checking all of the nodes it can traverse to, as its evaluation function results in the frontier arranging the expanded nodes depending on each node's path cost.

The Greedy Best-first search algorithm was the worst one of all in terms of locating a path toward the goal state with 35 moves. However, this was the most efficient out of the three where it was able to locate the goal state in only 231 steps since it only sorts the frontier by the heuristic function, which forces the algorithm to search closer and closer toward the goal state but as seen with the results, it trades off finding the optimal solution for finding a solution quickly.

The A* search algorithm was able to return the path of 19 moves as well, similar to the Uniform-Cost search. However, unlike the Uniform-Cost search which ends up with relatively higher steps compared to the Greedy Best-First search algorithm, it was able to locate the goal state much faster with a step count of 310. Although the Greedy Best-first search algorithm was able to locate the goal state much quicker, the A* search algorithm was able to return the optimal path, due to the design of its evaluation function. In a sense, A* was the middle ground of the two search algorithms, providing the optimal path in a slightly efficient way.

The Uniform-Cost search algorithm had the highest number of steps, while the Greedy Best-first search algorithm had the least number of steps. However, the Greedy Best-first search algorithm was not able to find the optimal solution. On the other hand, the A* search algorithm was able to find the optimal solution with a relatively lower step count than the Uniform-Cost search algorithm. The choice of the Gamebot's search algorithm will depend on the user's selection; although, the trade-offs between finding the optimal solution and finding a solution quickly should be taken into account.

The power of AI algorithms in solving complex problems is demonstrated by the bot's ability to find the best answer for the shortest path through a maze, even in complex maze configurations. It is a highly effective tool for solving maze problems due to the use of the A* search, greedy search, and uniform cost search algorithms, which allow it to take various factors into account and determine the most effective path through the maze. The program's one and only restriction, the variety of grid size options, might be problematic for some users. The range of specifications, however, is probably enough for the majority of maze problems. The Gamebot, in its entirety, is a sophisticated and successful tool for resolving maze issues. Its algorithmic capabilities, adaptability, and capacity to spot errors in user input make it an invaluable tool for anyone trying to solve a maze problem correctly and quickly.

## IV. Recommendations

While it is capable of executing multiple search algorithms with different evaluation functions, this Gamebot does not apply the search functions for Breadth-first search and Depth-first search, whose evaluation functions are $f(n) = d$ and $f(n) = 1/d$ respectively, where $d$ is the number of actions it takes to reach a node from the root and the mapping is set to be a tree. In other words, $d$ is the depth of a node in a tree. These two algorithms may not return the optimal path to the goal as they are uninformed search algorithms, but applying them to the program where the beginning and goal states are the same for each search can help in understanding how uninformed search algorithms compare with informed ones. It also aids in identifying whether the bot is intelligent or not.

The Gamebot also only applies one computation of the heuristic estimate, which is the Manhattan distance formula, as it can only move around in four directions: up, down, left, and right. For the A* search, the heuristic estimate decides whether the path it takes will be cost-optimal, and the condition for this search to return a cost-optimal path is for the heuristic estimate to be admissible. If the maze specifications allow for the Gamebot to move in any eight directions which are adjacent to it, the formula for the Diagonal distance can be used as an admissible heuristic estimate.

The A* search is an algorithm that requires a lot of time and space to execute, especially when the size of the search is large. This form of A* search is what was implemented for the Gamebot, but there are other implementations that use less time and space. One alternative is to change the computation for the heuristic estimate to be an inadmissible one - meaning that it will overestimate the cost from the current node to the goal state. Another could be to put more weight on the heuristic, therefore making the evaluation function become $f(n) = g(n) + W \times h(n)$ for some $W > 1$. The drawback is that it would not always return the most cost-optimal path, but it will still return a good enough solution for what the problem asks for as well as take less time to search for the path.

## V. References

GeeksforGeeks. (2022, May 30). *A Star Search Algorithm*. https://www.geeksforgeeks.org/a-search-algorithm/

GeeksforGeeks. (2023, February 15). *Greedy Algorithms*. https://www.geeksforgeeks.org/greedy-algorithms/

GeeksforGeeks. (2023, February 14). *Uniform Cost Search  Dijkstra for large Graphs*. https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

GeeksforGeeks. (2023, January 11). *Search Algorithms in AI*. https://www.geeksforgeeks.org/search-algorithms-in-ai/

Russell, S., & Norvig, P. (2022). *Artificial Intelligence: A Modern Approach,* 4th Ed. New Jersey: Pearson.

## VI. Contribution of Members

| Name | Contribution |
|---|---|
| Adrada, Jasper John N. | Code Editing and Creation, Documentation, Brainstorming, Error Checking, III |
| Natividad, John Austin Mikhail T. | Code Editing and Creation, Documentation, Brainstorming, Error Checking, II and III |
| Ramirez, Benmar Sim G. | Code Editing and Creation, Documentation, Brainstorming, Error Checking, III and II |
| Razon, Luis Miguel Antonio B. | Code Editing and Creation, Documentation, Brainstorming, Error Checking, II and III |
| Romblon, Kathleen Mae V. | Code Editing and Creation, Documentation, Brainstorming, Error Checking, I and IV |