**HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION**

**FACULTY OF INFORMATION TECHNOLOGY**

Logo

Description automatically generated

**FINAL TERM PROJECT**

**Course name: ARTIFICIAL INTELLIGENCE**

**Design and implement Maze game using AI algorithms**

**Lecturer name**: Assoc. Prof. Hoang Van Dung

**List of members**

|  |  |  |
| --- | --- | --- |
| **Student ID** | **Student Name** | **Contribution(%)** |
| 22110062 | Đào Nguyên Phúc | 100% |
| 22119054 | Lưu Trọng Dũng | 100% |

*Ho Chi Minh City, 2024*

*CONTENT*

*1. Project descriptions* ……………………………………………………………………………………2

1.1.Introduction……………………………………………………………………………………..2

1.2.Objectives………………………………………………………………………………………2

1.3.Scope and Target Audience………………………………………………………………..2

*2. Background knowledges* ……………………………………………………………………………..3

2.1.Programming Tools and Environment…………………………………………………..3

2.2. Methods and Techniques……………………………………………………………….….3

*3. Design system………………………………………………………………………………………..5*

*3.1.DFS……………………………………………………………………………………………5*

*3.2.BFS……………………………………………………………………………………………6*

*3.3.checkPosition function……………………………………………………………..……8*

*3.4.Herwinning function………………………………………………………………..……10*

*3.5.Heuristic function………………………………………………………………………..10*

*3.6.Hill climbing………………………………………………………………………………..11*

*3.7.A\*………………………………………………………………………………………………12*

*3.8.Greedy……………………………………………………………………………………..…14*

*3.9.UCS……………………………………………………………………………………………16*

*3.10.Beam Search……………………………………………………………………………18*

*4. Implement, Test case, results, and discussions…………………………………………..21*

*4.1. User Interface…………………………………………………………………………… 21*

*4.2.**Test case, results…………………………………………………………………………23*

*5 Conclusions…………………………………………………………………………………………31*

References

**1.Project description**

**1.1. Introduction**

This program was developed to simulate various pathfinding algorithms commonly used in games, enabling players or AI systems to find the optimal route from a start point to an endpoint on a map. The algorithms implemented include **DFS (Depth-First Search)**, **BFS (Breadth-First Search)**,**Hill Climbing ,A\*, Greedy, UCS ,Beam search**. The program also allows users to create maps, set obstacles, and check for win/loss conditions within the game environment.

**1.2. Objectives**

**-Applying AI Pathfinding Algorithms**: The program simulates basic pathfinding algorithms within a game environment, aiming to help users understand and compare different pathfinding approaches.

**-Creating Map Environments**: It allows players to generate custom maps with obstacles, designate start and end positions, and test the functionality of each pathfinding algorithm.

**-Providing an Intuitive Graphical Interface**: The environment uses the Pygame library to display the map and game status, including the movements of the player or AI algorithms.

**1.3. Scope and Target Audience**

The program is designed for students, educators, and AI enthusiasts interested in learning about **search algorithms** and **pathfinding** techniques. It is intended for educational environments, as well as for self-study, where users can directly observe how different algorithms navigate paths with varying complexity.

**2.Background Knowledges**

**2.1.Programming Tools and Environment**The project is implemented in **Python** due to its flexibility and the availability of visualization libraries. Key tools and libraries include:

* **Pygame** for creating an interactive graphical interface.
* **NumPy** for efficient data manipulation.
* **Algorithmic Techniques** such as **graph traversal** for the pathfinding process.
* **Random:** used for introducing randomness, such as generating mazes or placing obstacles, start, and goal positions dynamically.
* **Copy:** used to duplicate objects, especially when implementing algorithms that require exploring multiple states without affecting the original maze structure.
* **Queue and PriorityQueue**: implementing search algorithms like BFS, DFS, or A\*, … which are commonly used in maze-solving.

**2.2.Methods and Techniques**The main algorithms implemented are:

* **DFS**: Traverses paths deeply before backtracking, ideal for observing exhaustive searches.
* **BFS**: Uses a queue to explore all neighboring nodes, which is useful for identifying shortest paths in an unweighted grid.
* **Hill Climbing**: Moves towards the goal based on a heuristic, illustrating how heuristic-based methods can be efficient but may get trapped in local optima.
* **UCS**  :helps find the optimal path to the goal by considering path costs. It systematically explores the maze using a priority queue, expanding the least-cost paths first.
* **A\*** :improves on UCS by using a heuristic to guide the search, balancing cost and distance to the goal.
* **Beam Search:** a heuristic search strategy that is particularly useful in scenarios where finding the best solution quickly is more important than exhaustively exploring every possibility.
* **Greedy:** a problem-solving approach that makes decisions step-by-step by choosing the locally optimal solution at each step with the hope that these local choices will lead to a globally optimal solution.

**3.Design System**

**3.1.DFS**:

Iteratively or recursively explores each branch, backtracking when reaching dead ends, until the goal is found.

Start at posStart:

* Add the start position to self.info["dfs"][0] and depth.
* Mark it as visited.

Explore Possible Moves:

* Shuffle moves for randomness.
* Check each direction (up, left, down, right):
  + If valid (within bounds, not visited, walkable):
    - Move to the new position.
    - Update the path and stack.

Backtrack if Stuck:

* If no valid moves, backtrack to the previous position in depth.
* If no positions remain in depth, terminate.

Terminate on Success or Exhaustion:

* If the goal is reached, the solution is stored in self.info["dfs"][0].
* Otherwise, no path is found.

A screen shot of a computer program

Description automatically generated

**3.2.BFS**:

Utilizes a queue to explore all nodes at each level, efficiently finding the shortest route in the grid.

**Initialization**:

* **Queue**: A deque stores nodes to be explored, initialized with the start position.
* **Visited Set**: Tracks visited nodes to avoid revisiting cells.
* **Parent Dictionary**: Keeps track of each cell's parent, enabling reconstruction of the path from start to end.
* **Moves**: Defines possible movement directions (up, left, down, right).

**Breadth-First Traversal**:

* Dequeue the front node of the queue (queue.popleft()).
* Check if it's the goal position (self.posEnd). If yes, exit the loop.
* Shuffle the moves to introduce randomness in path exploration.
* For each valid neighbor:
  + If the neighbor is within bounds, not a wall (self.map[nx][ny] != 1), and not visited:
    - Mark it as visited.
    - Add it to the queue for future exploration.
    - Record its parent.

**Path Reconstruction**:

* Start from the goal position and backtrack using the parent dictionary to construct the path.
* Append each cell in the path to self.info["bfs"][0].
* Reverse the path so it starts from self.posStart.

**Return Value**:

* If a valid path is found, return True. If not (e.g., the maze is unsolvable), return False.

A screen shot of a computer program

Description automatically generated

**3.3.checkPosition function**

This is a helper function used to validate positions in the game map. It takes two parameters:

**+ current**: A tuple containing (x,y) coordinates to check

+ **visited**: An optional set of previously visited positions (defaults to None)

The function performs 4 key checks:

- 0 <= current[0] < self.sizeMap[0]: Checks if x-coordinate is within map bounds

- 0 <= current[1] < self.sizeMap[1]: Checks if y-coordinate is within map bounds

- self.map[current[0]][current[1]] == 0: Verifies the position is not a wall (walls are marked as 1)

- current not in visited: Ensures the position hasn't been visited before (if a visited set is provided)

The function returns True only if ALL these conditions are met, meaning:

- The position is within the map boundaries

- The position is not a wall

- The position hasn't been visited before (if tracking visited positions)

This function is particularly useful in pathfinding algorithms (like DFS, BFS, A*etc.) to:*

- Prevent going out of bounds

- Avoid walking through walls

- Prevent revisiting the same positions (when needed)

- Validate potential moves before making them

It's a crucial safety check that helps maintain valid movement throughout the game.

A computer screen shot of a program code

Description automatically generated

**3.4 .Herwinning function**

This is a helper function used to reconstruct the path from the start position to the end position in pathfinding algorithms.

- Takes two parameters:

+ **current**: The current position (usually the end/goal position)

+**parents**: A dictionary that maps each position to its parent position

- Used by algorithms like A*, BFS, and others to reconstruct the final path after finding the goal*

The function:

+Starts from the end position

+ Follows the parent links backwards until reaching the start position (which has no parent, i.e., None)

+Reverses the path so it goes from start → end instead of end → start

**A computer screen shot of text

Description automatically generated**

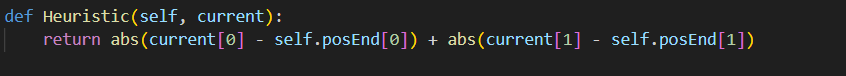
**3.5.Heuristic Function**

- This is a Manhattan distance heuristic that estimates how far a position is from the goal

+ It calculates the sum of horizontal and vertical distances between the current position and end position

+Lower values mean closer to the goal

+ Used to guide the hill climbing search towards promising directions



**3.6.Hill Climbing**

Evaluates possible moves based on proximity to the goal, using heuristics but lacking the ability to backtrack effectively.

- **Initialization**:

+ Starts at the **posStart** position

+ Creates an empty **path** list with the starting position

+ Sets up possible moves in four directions (up, down, left, right)

- **Main Loop**:

+ Continues until either reaching the end position or getting stuck

+ For each iteration:

+ Finds all valid neighboring positions

+ Checks if any neighbor has a better (lower) heuristic value than current position

+ Moves to the neighbor with better heuristic if found

+ Records the move in both **path** and **allPath**

- **Breaking Conditions**:

+ Stops if there are no valid neighbors

+ Stops if it reaches the end position

+ Stops if it revisits a position (detects a loop)

- **Result Handling**:

+ If successfully reaches the end: saves the complete path

+ If fails to reach the end: saves only the starting position

- **Heuristic Function**:

+ Uses Manhattan distance to estimate distance to goal

+ Lower value means closer to goalA screen shot of a computer program

Description automatically generated

**3.7.A\***The algorithm guarantees finding the shortest path by always exploring the most promising nodes first, based on both actual cost and estimated remaining distance.

- **Initialization**:

+ Creates a priority queue for open nodes

+Sets up tracking dictionaries for:

**+ gnList:** Cost from start to current node

**+ fnList**: Total estimated cost (g(n) + h(n))

**+ parents**: Keep track of path

+ Defines possible moves (up, down, left, right)

- **Main Loop**:

+ While there are nodes to explore:

+ Gets the node with lowest f(n) from queue

+ Marks it as visited

+ Records the path taken

+ Checks if reached destination

- **Node Expansion**:

+ For each possible direction:

+ Calculates neighbor position

+ If position is valid (not wall, within bounds):

+ Calculates new cost to reach neighbor

+ If new path is better than existing:

~ Updates costs

~ Sets parent

~ Adds to queue with priority f(n)

- **Cost Calculation**:

+ g(n): Actual cost from start (tracked in gnList)

+ h(n): Heuristic estimate to goal (Manhattan distance)

*+ f(n) = g(n) + h(n)*

- **Path Recovery**:

+ Once goal found, uses parent links to reconstruct path

+ Stores final path in info dictionary

A screen shot of a computer program

Description automatically generated

**3.8.Greedy**

This is a "greedy" approach because it always chooses the node that appears closest to the goal, regardless of the path taken to reach that node.

**Initialization**:

+ Creates an empty priority queue (**openList**) to store nodes to explore

+ Creates an empty set (**visited**) to track visited nodes

+ Adds the starting position to both the queue and visited set

+ Creates a dictionary (**parents**) to track the path

+ Sets up possible moves (up, down, left, right)

- **Main Loop**:

+ While the priority queue is not empty:

+ Gets the next node with lowest heuristic value from queue

+ Adds current position and count to path history

+ Checks if reached end position (goal)

+ If not at goal, explores neighbors:

+ Calculates neighbor positions in all 4 directions

+ For valid unvisited neighbors:

* Marks as visited
* Sets parent reference
* Adds to queue with heuristic priority

- **Heuristic**:

+ Uses Manhattan distance (sum of x and y differences) to estimate distance to goal

+ Lower heuristic values are explored first

- **Path Recording**:

+ Tracks the path using parent references

+ When goal is reached, reconstructs path using **Herwinning** function

+ Records exploration order in **allPath**["greedy"]

- **Key Differences from Other Algorithms**:

+ Only considers heuristic value (estimated distance to goal)

+ Doesn't consider path cost like A

+ Makes locally optimal choice at each step

+ May not find optimal path but usually finds goal quickly

A screen shot of a computer program

Description automatically generated

**3.9.UCS**

- **Initialization**:

+ Creates a priority queue called **openList** to store nodes to explore

+ Puts the start position in the queue with initial cost 0

+ Creates two dictionaries:

+ **gnList**: tracks the cost to reach each node

+ **parents**: keeps track of each node's parent for path reconstruction

- **Main Loop**:

+ While the queue is not empty:

+ Gets the node with lowest cost from the queue

+ Records the path taken in **allPath**["ucs"][0]

+ Checks if current node is the end position

+ If yes: reconstructs path using **Herwinning** and returns

- **Neighbor Exploration**:

+ For each possible move (up, down, left, right):

* + Calculates the neighbor's position
  + If the position is valid (using **checkPosition**):
  + Calculates new cost to reach neighbor

+ If neighbor hasn't been visited OR new path is cheaper:

* Updates cost in **gnList**
* Updates parent in parents
* Adds neighbor to queue with new cost

A computer screen shot of a program code

Description automatically generated

**3.10.Beam Search**

This implementation of Beam Search aims to find a path to the goal efficiently by focusing on the most promising directions, but it may not always find the optimal path due to its limited search width.

**-Initialization:**

+ A priority queue openList is created, starting with the initial position.

+Sets for visited nodes and a dictionary for parent nodes are initialized.

+A counter count is set up to track the order of visited nodes.

**-Main Loop:**

The algorithm continues while there are nodes in the **openList**:

a.Beam Selection:

+Up to 3 nodes are selected from the openList to form the current beam.

+Each selected node is added to allPath["beam"][0] with its count.

+If the goal is reached, the path is reconstructed and returned.

b. Neighbor Generation:

+For each node in the current beam, all valid neighbors are generated.

+Valid neighbors are those that are within bounds, not walls, and not visited.

+Each neighbor is added to the neighbors list with its heuristic value.

c. Beam Pruning and Extension:

+The neighbors list is sorted based on heuristic values.

+ Then the neighbors are added to openList.

+Termination:

* If the goal is reached, the path is returned.
* If openList becomes empty without reaching the goal, only the start position is returned.

+Key Points:

* The beam width is 3, meaning it considers the 3 most promising nodes at each step.
* It uses a heuristic function to estimate the promise of each node.
* It's a compromise between breadth-first search (exploring all possibilities) and greedy best-first search (following only the best option).

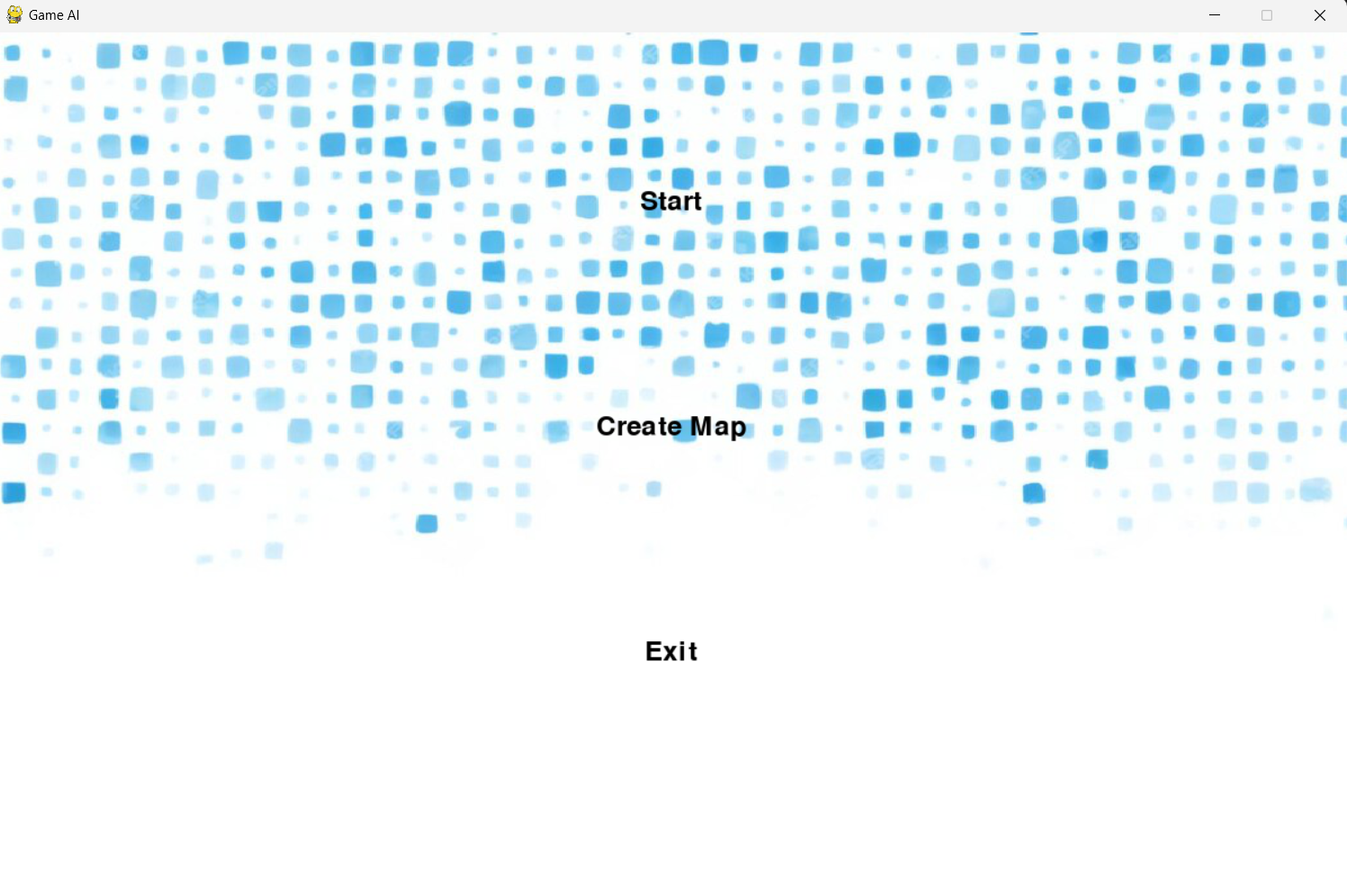
A screen shot of a computer program

Description automatically generated

**4.Implement, Test case, results, and discussions**

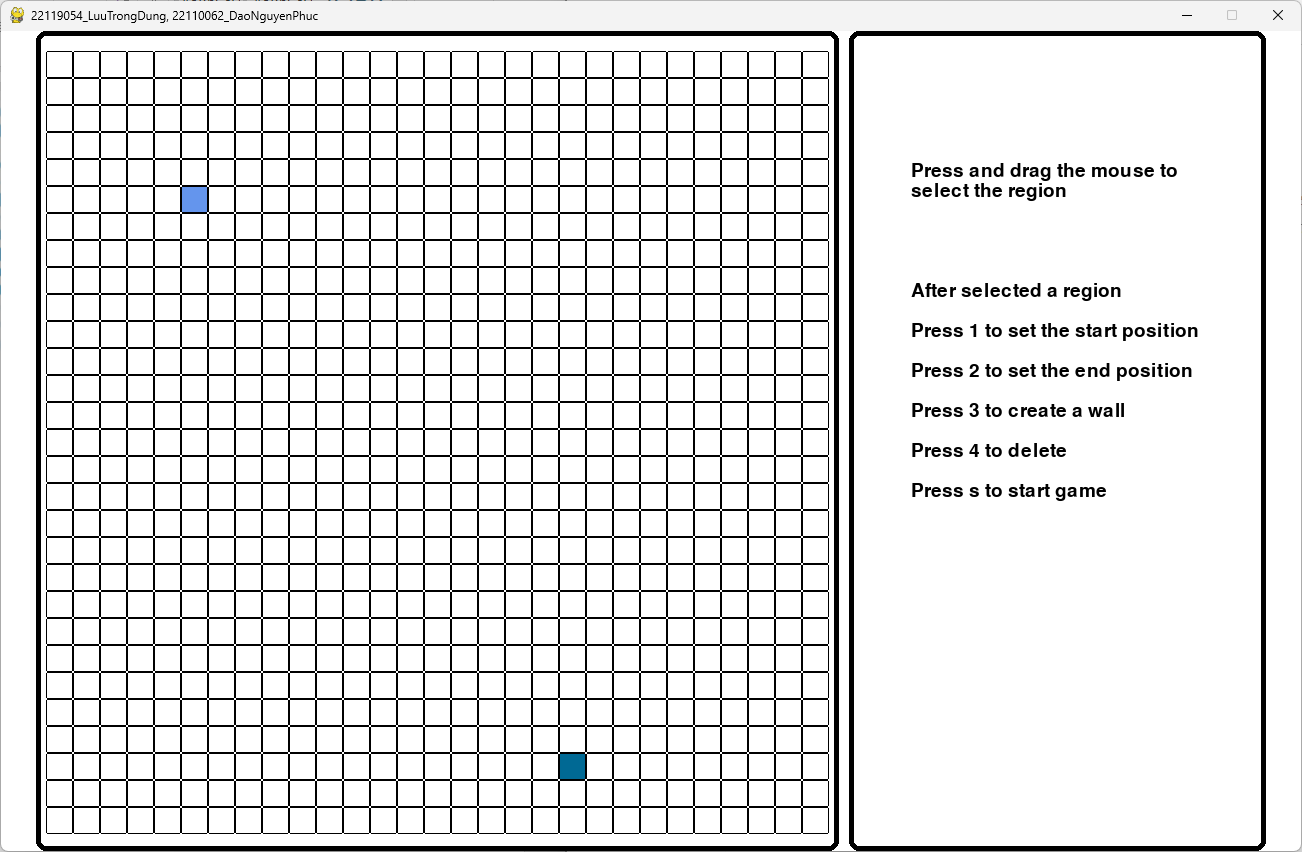
**4.1.User Interface**

**4.1.1.Main**



-Start button to play game

-Can create your own map



- Provides the ability to create maps by pressing and dragging the mouse to select tiles.

**4.1.2.Gameplay Interface**

**A screenshot of a video game

Description automatically generated**

-There are 7 balls similar to 7 methods and 1 ball for player

-Time is the time executing program

-The program also calculates each step of method to reach the goal destination.

**4.1.3.Play game**

To control the starting node, we use the following key codes: (W, S, A, D)

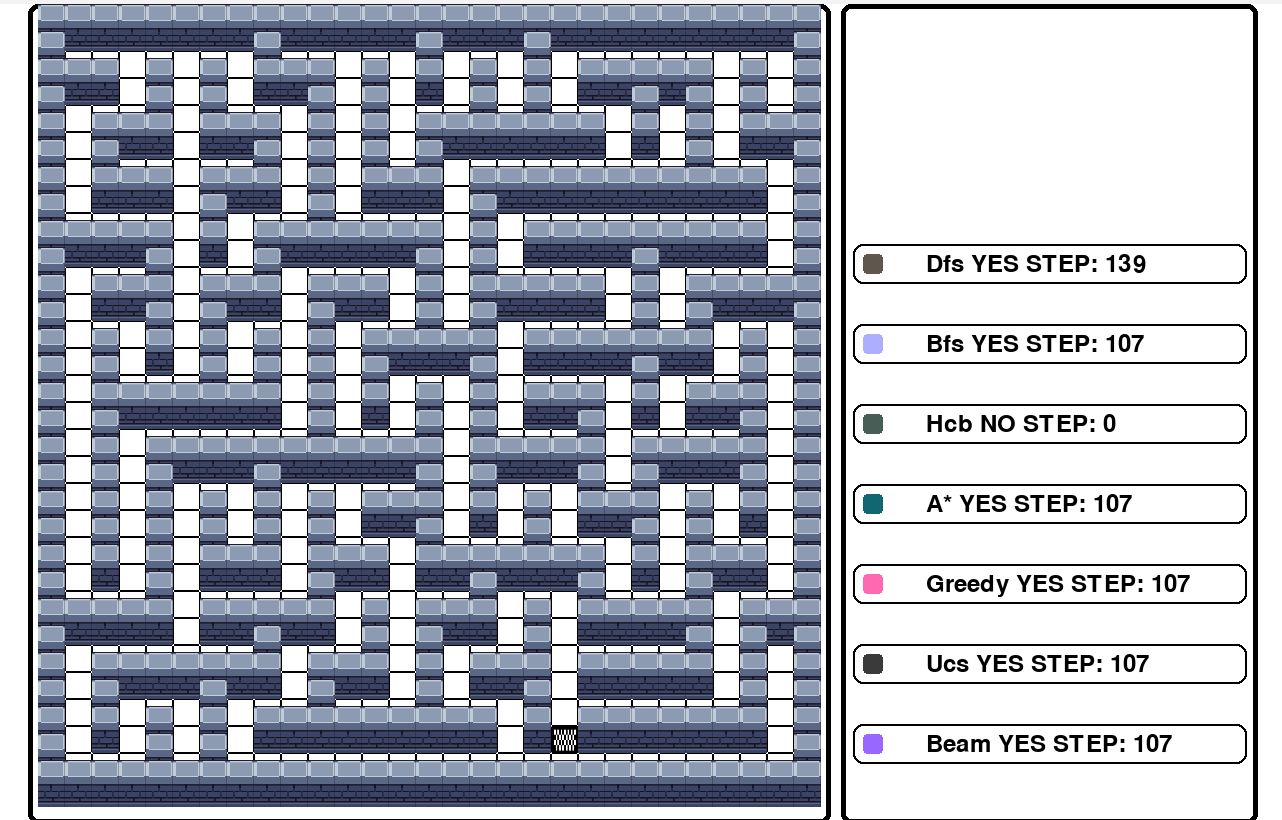
* **W**: to move the character upwards
* **S**: to move the character downwards
* **A**: to move the character to the left
* **D**: to move the character to the right

**4.2. Test case, results**

**4.2.1. Automatic map generation mode**



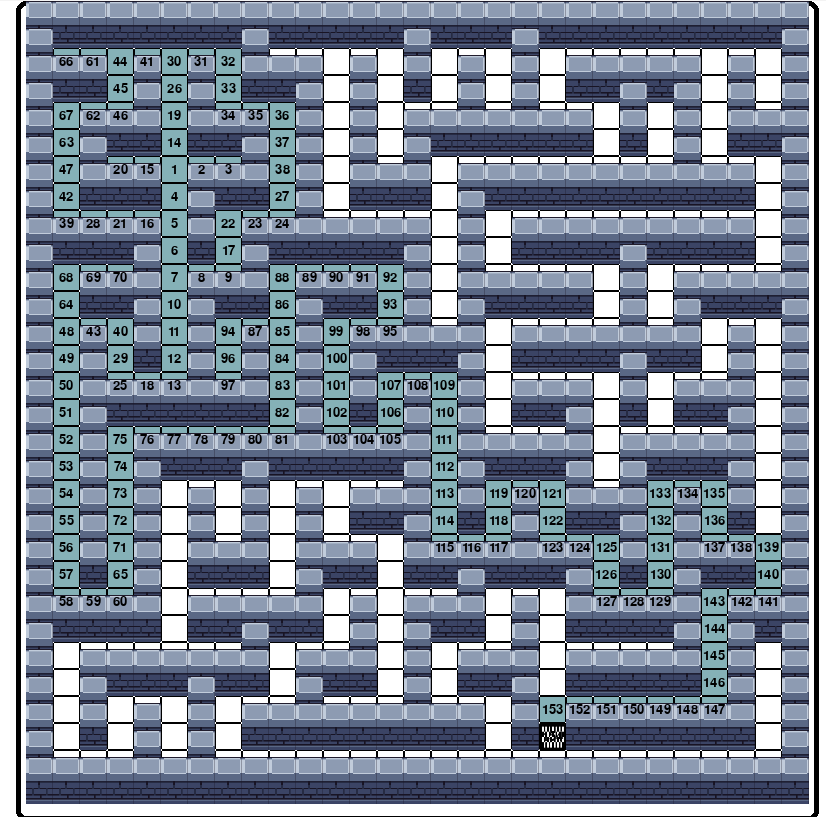
Hit skip to take the player to the finish line without walking.

Press X1 to speed up the Bots' journey (up to X5)****

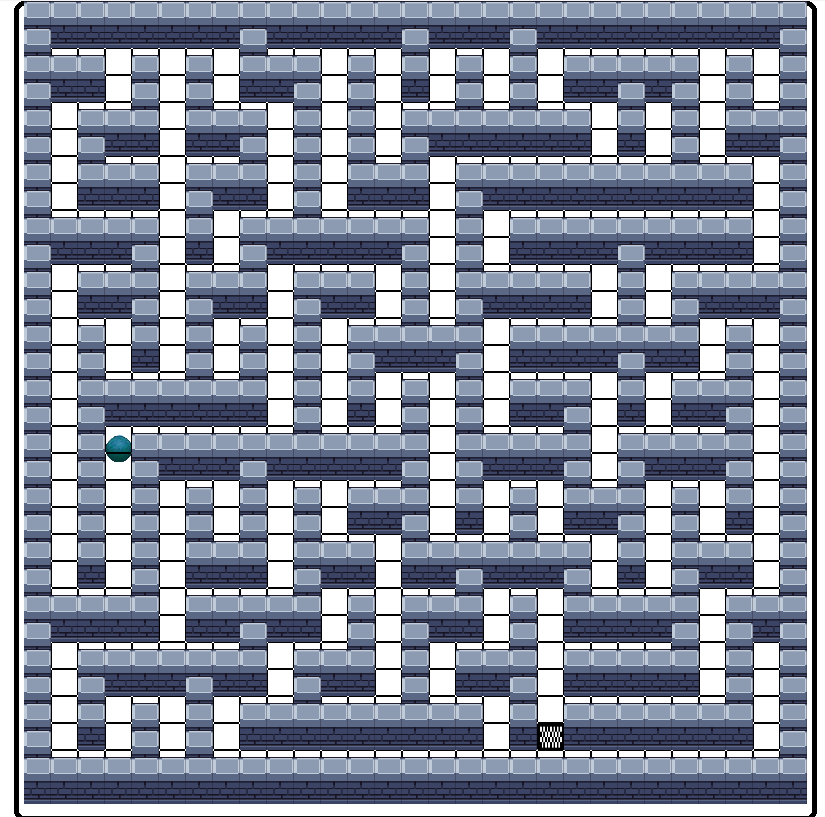
After all players and bots reach the finish line (end of migration)

Tap on the algorithm you need to review its pathfinding process

It will replot the algorithm's pathfinding process in the maze (with 1 being the location that the algorithm considers first, then the next time until the destination is found).

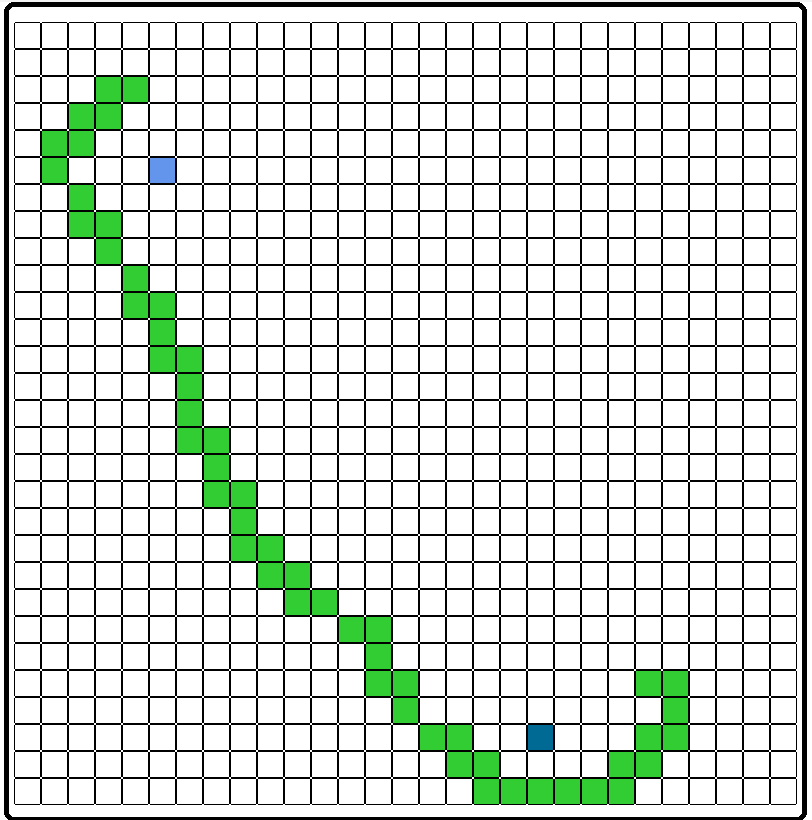


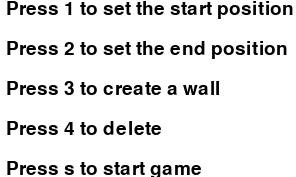
To review the last move of the algorithm, click the square in front of each algorithm (The final path of the algorithm will be redrawn).



As the red figure is the final path of the algorithm being redrawn.

**4.2.2. Map Creation Mode**

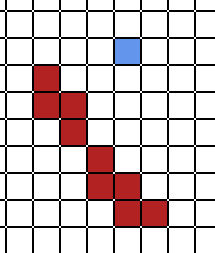
Players press and drag the mouse to select tiles and zones.



After selecting the area, you can press the keys:

+ Press 1 to set the starting position (provided that only 1 cell is filled in green).

+ Press 2 to set the end position (provided that only 1 box is filled in green).

+ Press key 3 to create an obstacle (wall). 

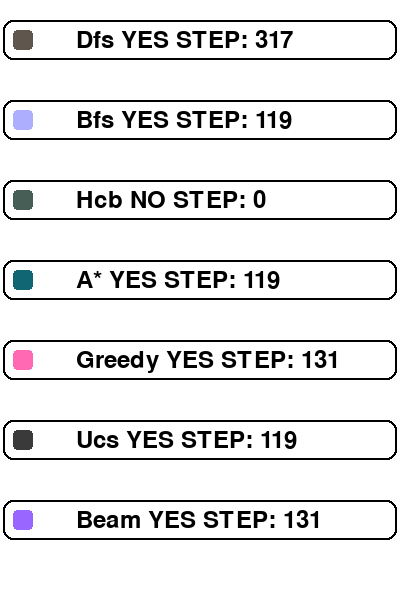
+ Press key 4 to clear obstructions

+ After successfully creating 1 map, press the S key to start the level

Note: you have to create a map so that there is always a way, otherwise you will not be able to start the game.

The gameplay is similar to the automatic map creation.

**4.2.3. Results**



+ DFS finds the path but does not ensure the optimal (shortest). The large number of steps caused by browsing deep into the first branch, can be a waste of resources.

+ Bfs algorithms (Bfs algorithms are the most time-consuming and memory consuming because they have to browse all the cells), A\* it always finds the most optimal path (shortest).

+ Hillclimbing algorithm because it always chooses the best value for the next step without a rollback mechanism, so if it encounters a dead end or a next value that is not better than the previous position, it will stop the algorithm.

+ Beam is better than Bfs in that it saves the next n positions best (with n called beams), so it can have 3 options in the next value, saving 3 positions reduces the memory required and the time it takes to consider the positions more than Bfs.

+ Greedy in some cases it can also find the most interrupted path such as Bfs or A\* thanks to the best saving mechanism for the next position, it does not get stuck like hillclimbing because it has a mechanism to turn back to find other tiles when stuck.

+ UCS is similar to BFS but applies to weighted graphs, ensuring the shortest path

**5.Conclusion**

This pathfinding visualization program is a sophisticated educational tool and interactive game implemented using Pygame. At its core, the program showcases seven different pathfinding algorithms: Depth-First Search (DFS), Breadth-First Search (BFS), Hill Climbing, A Search, Greedy Best-First Search, Uniform Cost Search (UCS), and Beam Search. The program features a user-friendly interface with a menu system that allows users to start predefined mazes, create custom maps, or exit the game.

The educational value of this program cannot be understated. It serves as an excellent tool for understanding different pathfinding algorithms, visualizing search patterns, comparing algorithm efficiency, and learning about maze generation and solving. The program successfully combines these educational aspects with interactive gameplay elements, making it an effective tool for both learning and entertainment purposes. Its potential applications extend to teaching computer science concepts, algorithm visualization, pathfinding algorithm comparison, interactive learning tools, and game development prototyping.

In conclusion, this program represents a well-balanced combination of technical sophistication, educational utility, and user engagement, making it a valuable resource for both learning and practical applications in pathfinding visualization.