# Artificial Intelligence Project

## Fire Escape Route Agent

Under supervision:

Dr/Ahmed Selim

Eng/Omar Khaled

Eng/Ahemd Sobhy

**Group members:**

1- Maram Hazem Fouad Ismail Ahmed (CS, C4)
2- Menna Ahmed Ibrahim Agamy (CS, C5)
3- Wessam Mohammed El-Sayed El-Hanafy (CS, C5)
4- Heba Ahmed Ibrahim Agamy (CS, C5)
5- Mohamed Elsayed Mohamed Ahmed Aboelsoud (CS, C4)
6- Mohamed Khaled El-Daheesh Ahmed (CS, C4)
7- Mohamed Refat Mostafa Abd-Elmajid Naser (CS, C4)

## Problem Description

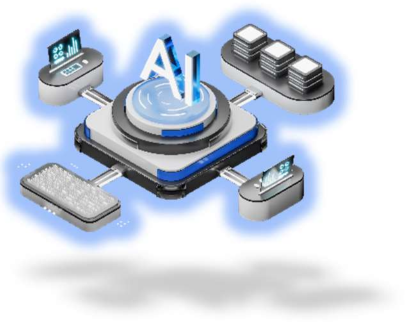
“Fire Escape Route Agent” is a pathfinding problem, which studies and aims to the find the fastest and most safe path to take in a mall in case of a fire. The environment of the study is a 3D maze of size 25x30 and the actor is an agent that routes the maze through different AI algorithms in order to find the fastest and most safe path to the goal and select the AI algorithm that led it to this path in the shortest time as the best algorithm. Depending on the results of the study, a Surviving Trolley that transports people will take the optimal path towards the exit of the mall safely.

## Used Algorithms

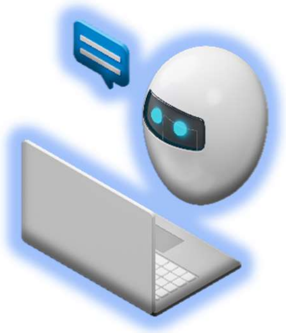
The chosen algorithms for this study are:

1- Breadth-First Search (BFS)
2- Depth-First Search (DFS)
3- Uniform-Cost Search (UCS)
4- A* Search
5- Iterative-Deepening Search (IDS)
6- Hill Climbing Search

Each algorithm implementation and results will be discussed in the algorithms implementation section.



## Implementation

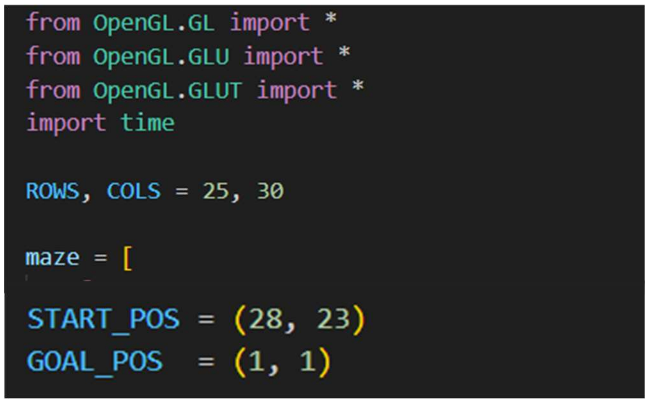


The code used was divided into 3 modules:

### 1- The Maze Module(maze.py)

This module defines the maze environment and provides 3D visualization capabilities. It generates and displays the final maze and agent through the following steps:

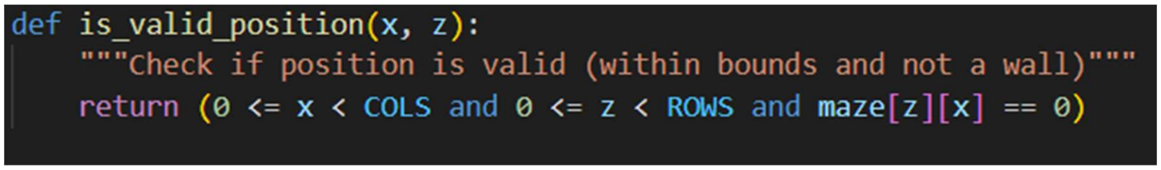
1) **Maze Definition:** Contains a 25×30 grid representing the fire escape environment, where 1s are walls (obstacles) and 0s are walkable paths. Defines start position (28, 23) as the fire location and goal position (1, 1) as the safe exit.



```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import time

ROWS, COLS = 25, 30

maze = [

START_POS = (28, 23)
GOAL_POS  = (1, 1)
```

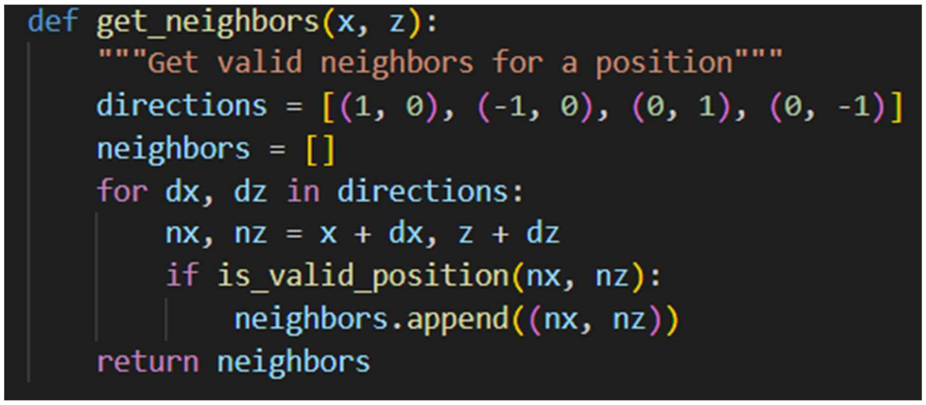
2) **Navigation Utilities:** Using the following functions to validate moves and retrieve adjacent walkable cells, enabling 4-directional movement (no diagonals):

***is_valid_position(x, z)*:**Checks if a position is valid (within maze bounds and not a wall), and returns True if the position at coordinates (x, z) is walkable (value 0 in maze array).



```
def is_valid_position(x, z):
    """Check if position is valid (within bounds and not a wall)"""
    return (0 <= x < COLS and 0 <= z < ROWS and maze[z][x] == 0)
```

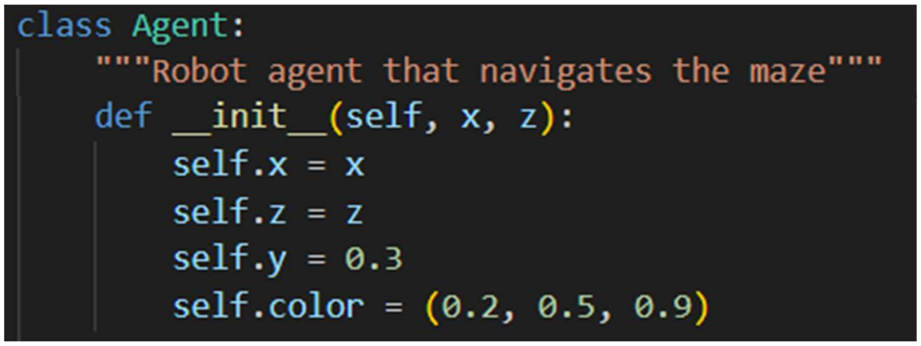
***get_neighbors(x, z)*:** Returns all valid adjacent positions (up, down, left, right) for a given position, uses 4-directional movement (no diagonals), and filters out walls and out-of-bounds positions.



```
def get_neighbors(x, z):
    """Get valid neighbors for a position"""
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    neighbors = []
    for dx, dz in directions:
        nx, nz = x + dx, z + dz
        if is_valid_position(nx, nz):
            neighbors.append((nx, nz))
    return neighbors
```

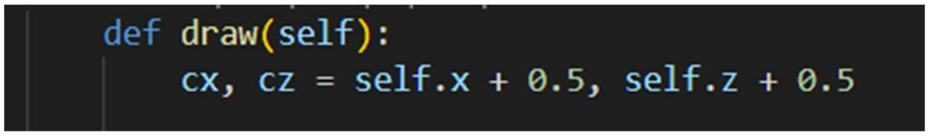
**3)Agent Class:** Implements a 3D robot model with detailed geometry, and handles agent positioning and rendering using OpenGL primitives. It contains the following functions:

***__init__(x, z)*:** Initializes agent at position (x, z) with blue color.



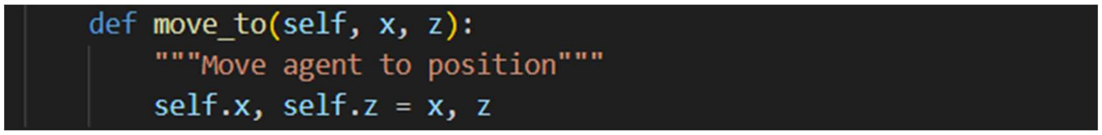
```
class Agent:
    """Robot agent that navigates the maze"""
    def __init__(self, x, z):
        self.x = x
        self.z = z
        self.y = 0.3
        self.color = (0.2, 0.5, 0.9)
```

***draw()*:** Renders the 3D robot model with body, head, eyes, antenna, arms, legs, and chest light using OpenGL.



```
    def draw(self):
        cx, cz = self.x + 0.5, self.z + 0.5
```

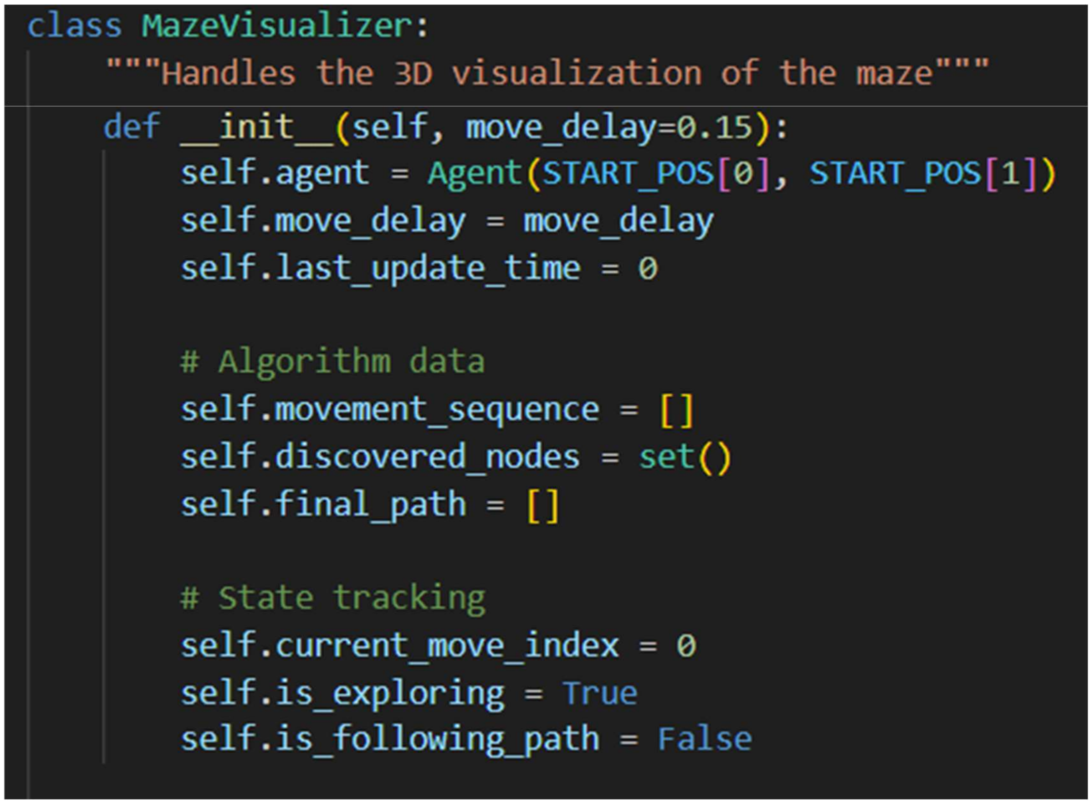
***move_to(x, z)*****:** Updates agent's position to new coordinates.



```
    def move_to(self, x, z):
        """Move agent to position"""
        self.x, self.z = x, z
```

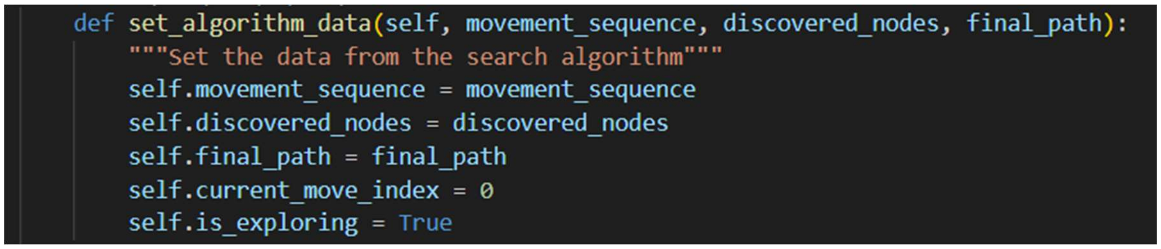
**4) MazeVisualizer Class:** Handles the 3D visualization and animation through the following functions:

***__init__(move_delay)*****:** Initializes visualizer with movement delay between steps.



```
class MazeVisualizer:
    """Handles the 3D visualization of the maze"""
    def __init__(self, move_delay=0.15):
        self.agent = Agent(START_POS[0], START_POS[1])
        self.move_delay = move_delay
        self.last_update_time = 0

        # Algorithm data
        self.movement_sequence = []
        self.discovered_nodes = set()
        self.final_path = []

        # State tracking
        self.current_move_index = 0
        self.is_exploring = True
        self.is_following_path = False
```

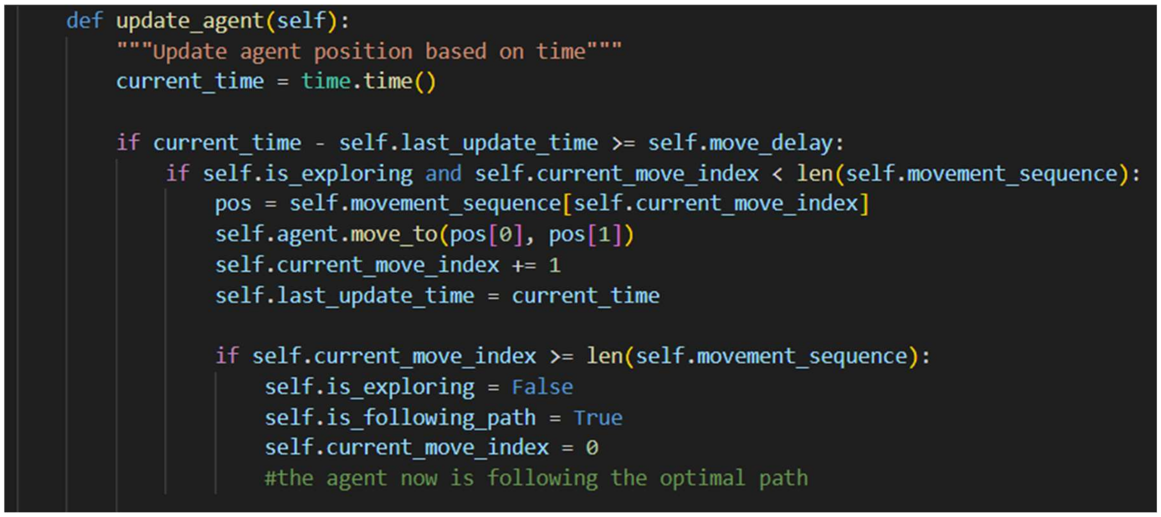
***set_algorithm_data(movement_sequence, discovered_nodes, final_path)*:** Loads data from search algorithm for visualization.



```
    def set_algorithm_data(self, movement_sequence, discovered_nodes, final_path):
        """Set the data from the search algorithm"""
        self.movement_sequence = movement_sequence
        self.discovered_nodes = discovered_nodes
        self.final_path = final_path
        self.current_move_index = 0
        self.is_exploring = True
```

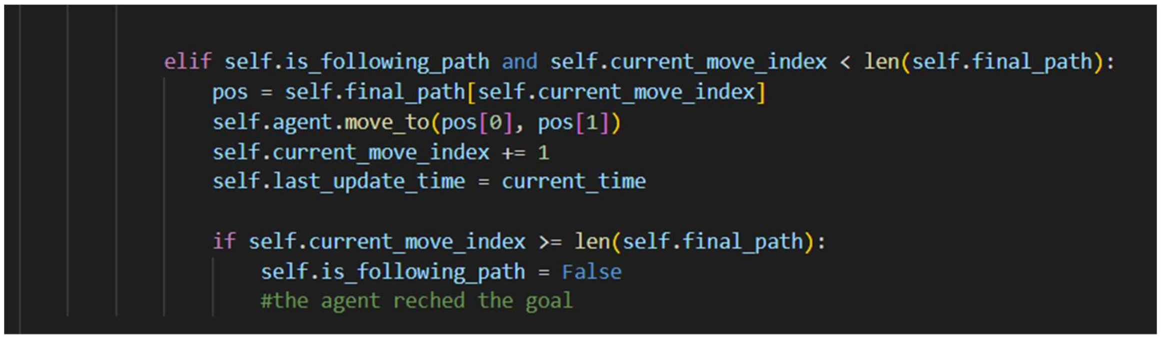
***update_agent()*:** Updates agent position based on elapsed time, handles two phases:

(Phase 1) agent explores maze (shows search process)



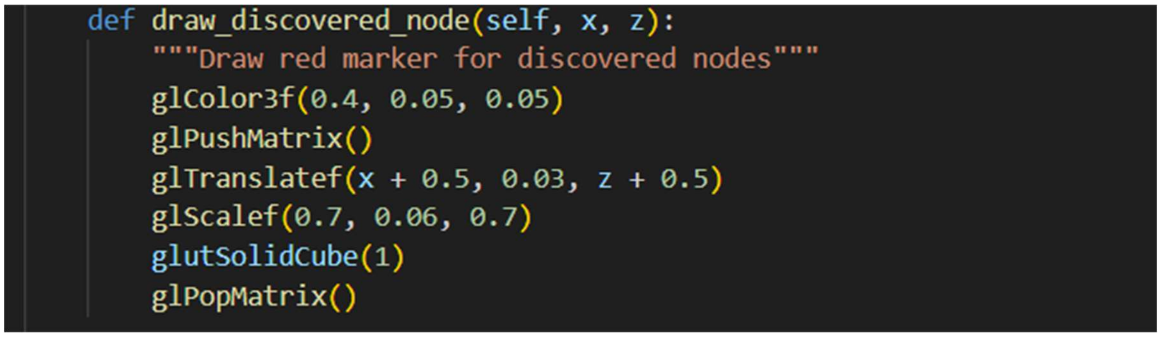
```
    def update_agent(self):
        """Update agent position based on time"""
        current_time = time.time()

        if current_time - self.last_update_time >= self.move_delay:
            if self.is_exploring and self.current_move_index < len(self.movement_sequence):
                pos = self.movement_sequence[self.current_move_index]
                self.agent.move_to(pos[0], pos[1])
                self.current_move_index += 1
                self.last_update_time = current_time

                if self.current_move_index >= len(self.movement_sequence):
                    self.is_exploring = False
                    self.is_following_path = True
                    self.current_move_index = 0
                    #the agent now is following the optimal path
```

(Phase 2) agent follows optimal path to goal



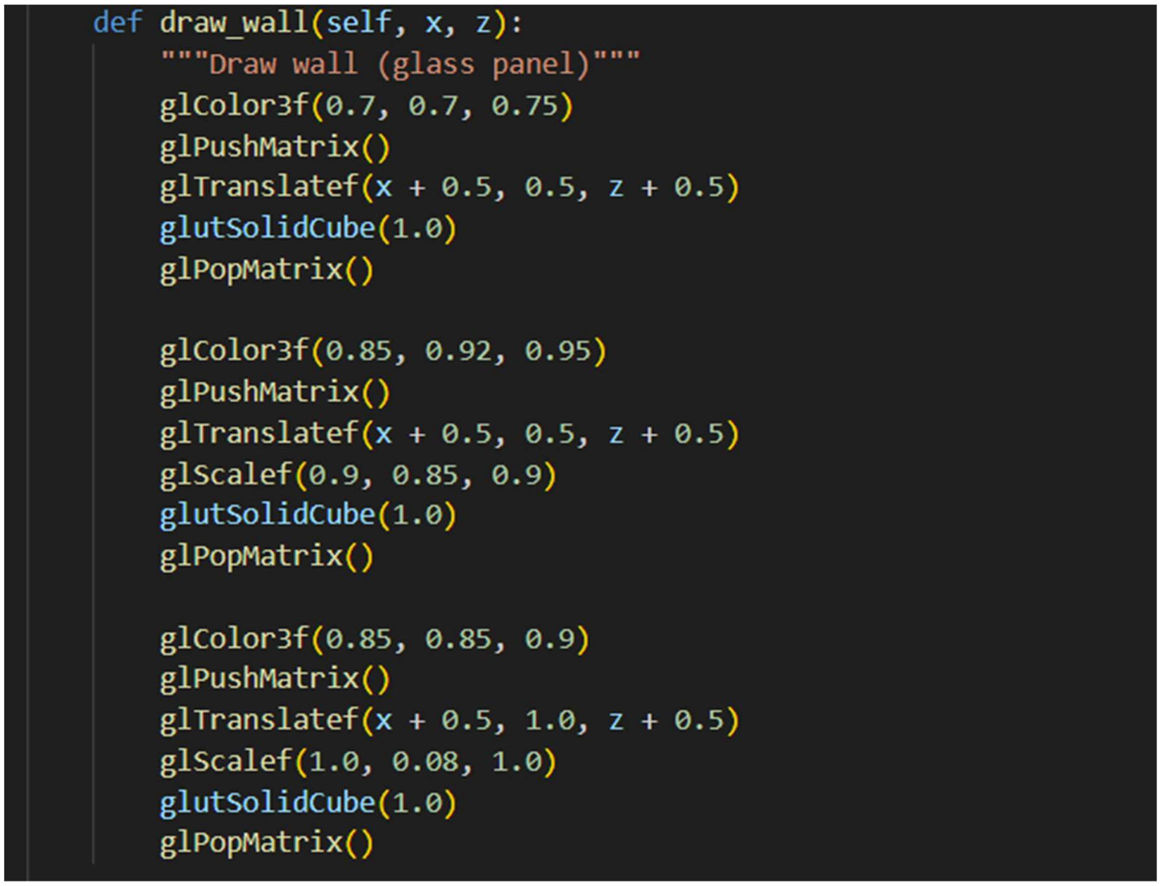
```

            elif self.is_following_path and self.current_move_index < len(self.final_path):
                pos = self.final_path[self.current_move_index]
                self.agent.move_to(pos[0], pos[1])
                self.current_move_index += 1
                self.last_update_time = current_time

                if self.current_move_index >= len(self.final_path):
                    self.is_following_path = False
                    #the agent reched the goal
```

***draw_discovered_node(x, z)*:** Draws red markers for nodes explored during search.



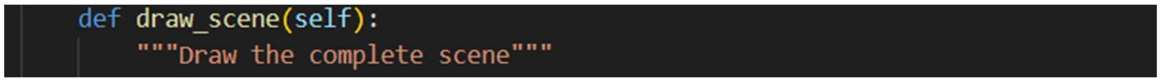
```
def draw_discovered_node(self, x, z):
    """Draw red marker for discovered nodes"""
    glColor3f(0.4, 0.05, 0.05)
    glPushMatrix()
    glTranslatef(x + 0.5, 0.03, z + 0.5)
    glScalef(0.7, 0.06, 0.7)
    glutSolidCube(1)
    glPopMatrix()
```

***draw_wall(x, z)*:** Renders 3D glass-panel walls.



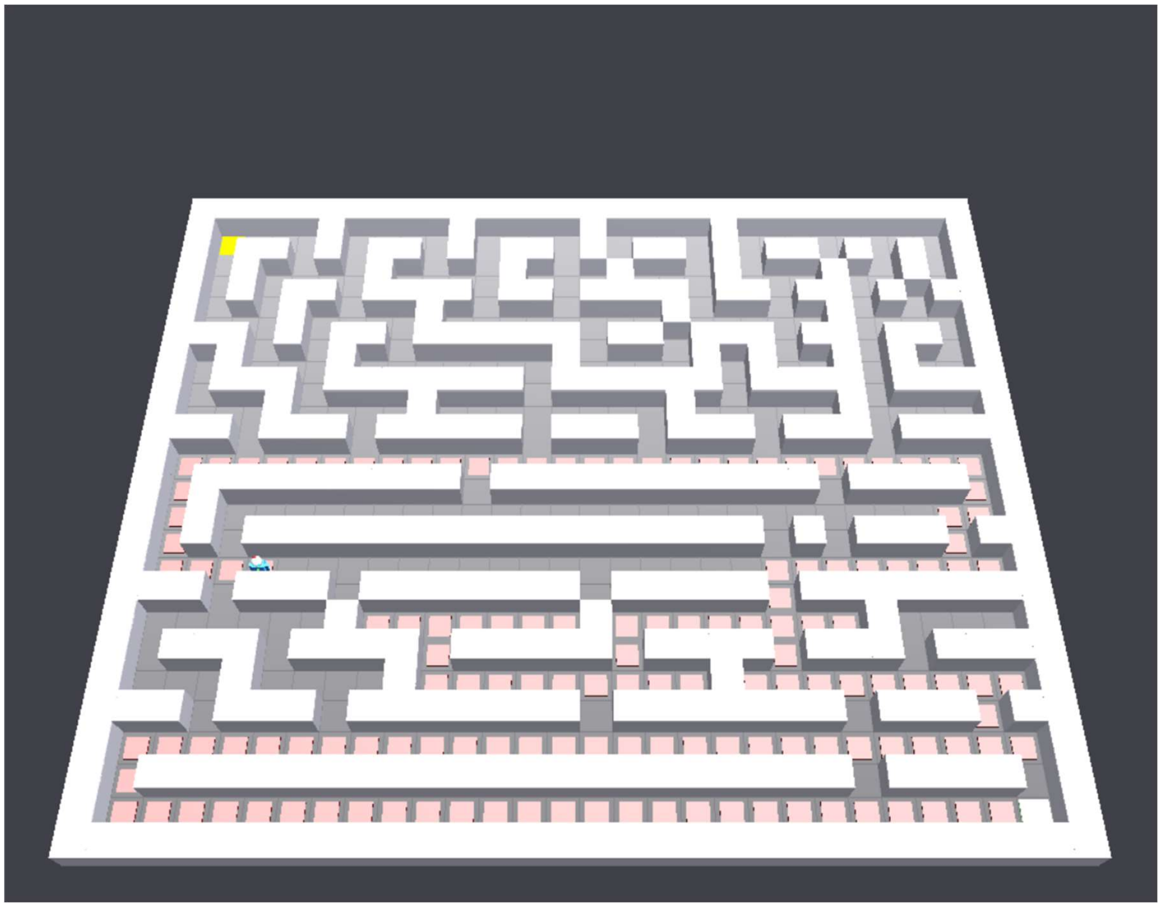
```
def draw_wall(self, x, z):
    """Draw wall (glass panel)"""
    glColor3f(0.7, 0.7, 0.75)
    glPushMatrix()
    glTranslatef(x + 0.5, 0.5, z + 0.5)
    glutSolidCube(1.0)
    glPopMatrix()

    glColor3f(0.85, 0.92, 0.95)
    glPushMatrix()
    glTranslatef(x + 0.5, 0.5, z + 0.5)
    glScalef(0.9, 0.85, 0.9)
    glutSolidCube(1.0)
    glPopMatrix()

    glColor3f(0.85, 0.85, 0.9)
    glPushMatrix()
    glTranslatef(x + 0.5, 1.0, z + 0.5)
    glScalef(1.0, 0.08, 1.0)
    glutSolidCube(1.0)
    glPopMatrix()
```

***draw_scene()*:** Renders complete scene including floor, grid lines, discovered nodes, walls, start (green), goal (gold), and agent.



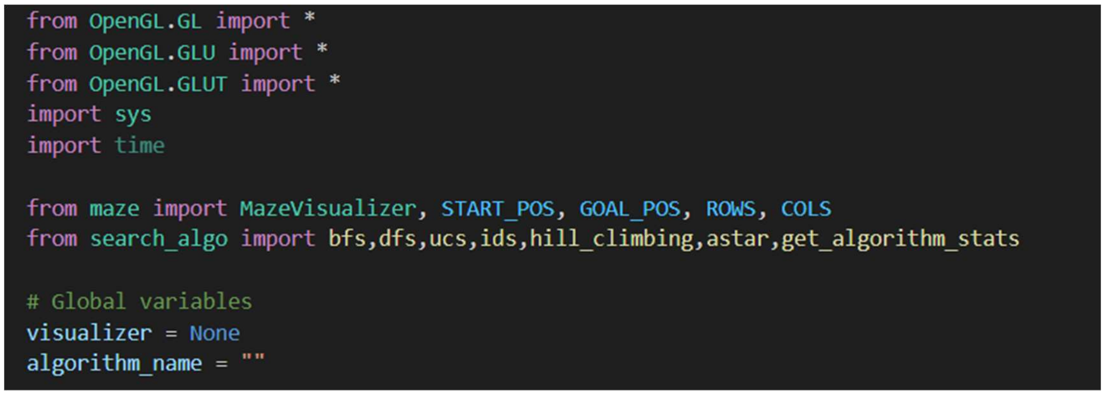
```
    def draw_scene(self):
        """Draw the complete scene"""
```

## The Generated Maze & Agent:



## 2- The Main Module(main.py)

This module acts as the program's entry point and orchestration layer.



```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import sys
import time

from maze import MazeVisualizer, START_POS, GOAL_POS, ROWS, COLS
from search_algo import bfs,dfs,ucs,ids,hill_climbing,astar,get_algorithm_stats

# Global variables
visualizer = None
algorithm_name = ""
```

**Included Functions:**

***display()*****:** OpenGL callback that renders each frame, sets up camera view and draws the scene.

***reshape(w, h)*****:** Handles window resizing and adjusts viewport and projection matrix.

***init_opengl()*****:** Initializes OpenGL settings (depth testing, lighting, colors).

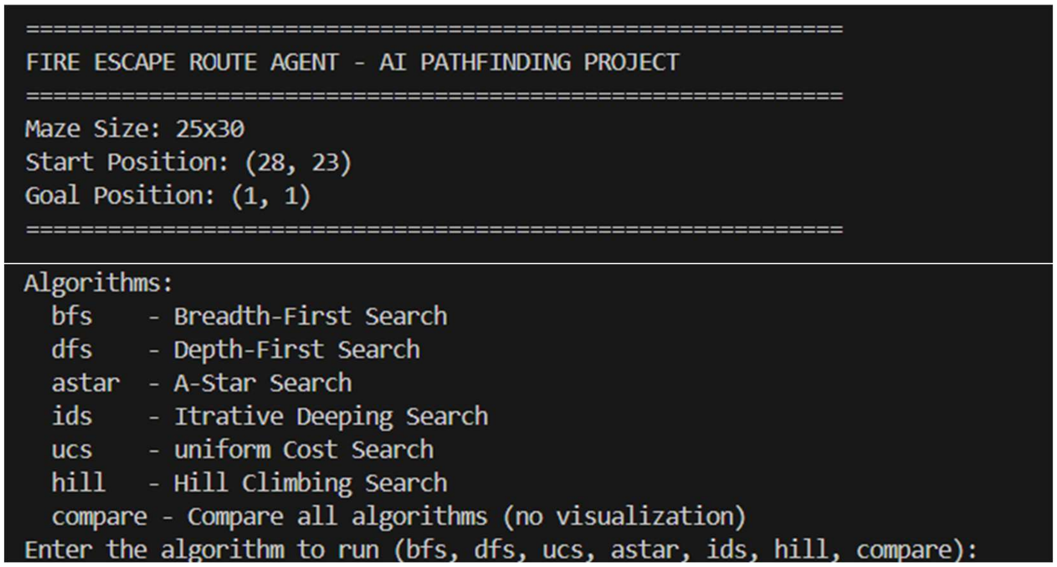
***run_algorithm(algorithm_choice, move_delay)*****:** Executes the selected search algorithm, creates visualizer with results, and prints statistics(nodes explored, movements, path length, visualization time).

***compare_algorithms()*****:** Runs all 6 algorithms without visualization and displays comparison table of performance metrics.

***main()*****:** Entry point of the program, handles user input for algorithm section, and initializes OpenGL window and starts visualization.

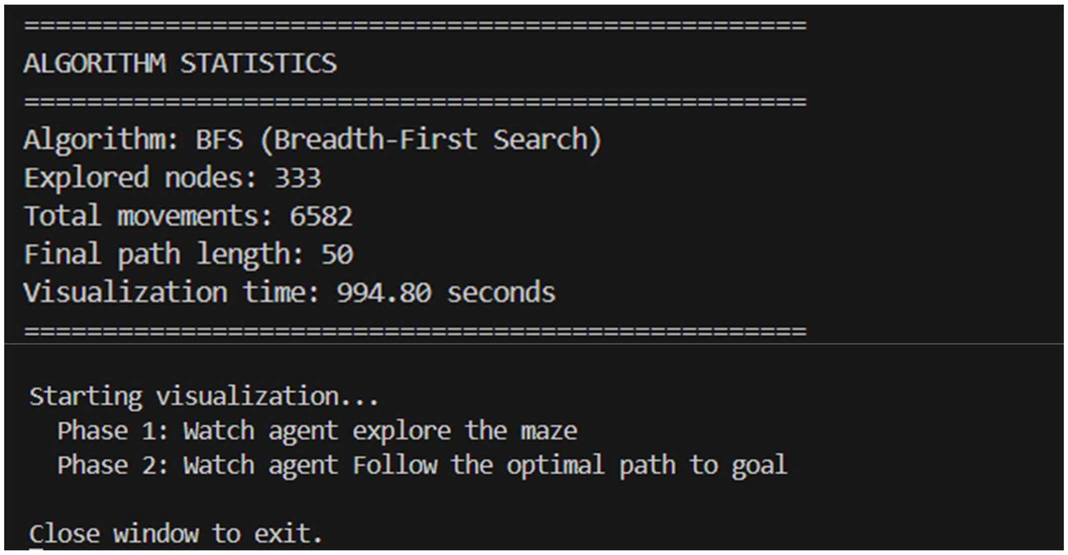
## Output:

When running the main module, the following program appears to the user:



```
============================================================
FIRE ESCAPE ROUTE AGENT - AI PATHFINDING PROJECT
============================================================
Maze Size: 25x30
Start Position: (28, 23)
Goal Position: (1, 1)
============================================================
Algorithms:
  bfs     - Breadth-First Search
  dfs     - Depth-First Search
  astar   - A-Star Search
  ids     - Itrative Deeping Search
  ucs     - uniform Cost Search
  hill    - Hill Climbing Search
  compare - Compare all algorithms (no visualization)
Enter the algorithm to run (bfs, dfs, ucs, astar, ids, hill, compare):
```

The user then enters one of the 6 algorithms to run( bfs, for example). After entering the algorithm, the program starts executing the algorithm on the maze and agent and display the algorithm statistics, then it visualizes the maze and how the agent explores the maze through the algorithm. Finally, it visualizes the agent following the optimal path after discovering it.



```
==================================================
ALGORITHM STATISTICS
==================================================
Algorithm: BFS (Breadth-First Search)
Explored nodes: 333
Total movements: 6582
Final path length: 50
Visualization time: 994.80 seconds
==================================================

Starting visualization...
  Phase 1: Watch agent explore the maze
  Phase 2: Watch agent Follow the optimal path to goal

Close window to exit.
```

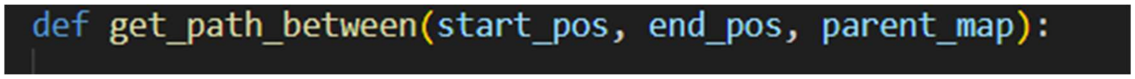
## 3- The Search Algorithms Module(search_algo.py)

This module implements the 6 AI pathfinding algorithms for fire escape route planning.

**Common Functions:**

**1)*get_path_between()*:** This function is called by all algorithms to calculate the movement path between any two nodes in the search tree. It does the following:

- **Traces** paths from both nodes upward to find their lowest common ancestor.
- **Constructs** backtrack path (current position → ancestor) and forward path (ancestor → target), and combines paths to create realistic agent movement sequence.
- **Enables** visualization of backtracking behavior in



```
def get_path_between(start_pos, end_pos, parent_map):
```

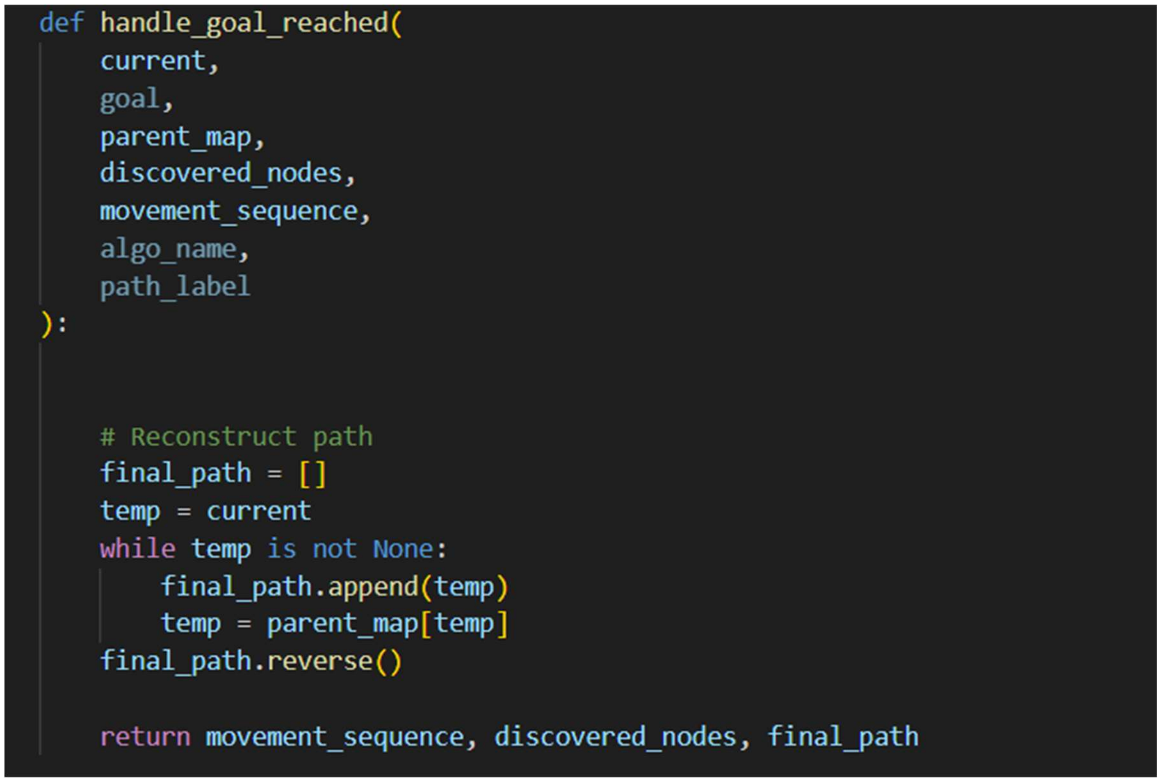
algorithms.

By doing the previous steps, it provides realistic animation by making the agent physically traverse the maze rather than teleporting, accurately representing the search algorithm's exploration pattern.

**2) *handle_goal_reached()*:** This function is called by all algorithms when the goal is reached to finalize results. It does the following:

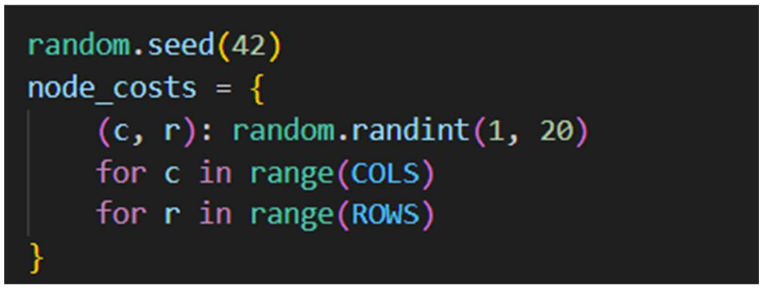
- **Reconstructs the final path** from start to goal by backtracking through the parent_map.
- **Reverses the path** to get correct order (start → goal).
- **Returns** the following:

- **movement_sequence:** Tracks the agent's movement step-by-step for visualization.
- **discovered_nodes:** Stores all nodes discovered during the search (shown as red markers).
- **current_position:** Represents the agent's current location in the maze.
- **final_path:** The optimal path from start to goal



```python
def handle_goal_reached(
    current,
    goal,
    parent_map,
    discovered_nodes,
    movement_sequence,
    algo_name,
    path_label
):


    # Reconstruct path
    final_path = []
    temp = current
    while temp is not None:
        final_path.append(temp)
        temp = parent_map[temp]
    final_path.reverse()

    return movement_sequence, discovered_nodes, final_path
```

**3)** ***random.seed(42)*****:** This function is called by UCS, A*, and Hill Climbing algorithms. It sets the random number generator seed to 42 to ensue reproducibility - the same "random" costs are generated every time the program runs. This makes algorithms comparisons fair and debugging consistent.



```python
random.seed(42)
node_costs = {
    (c, r): random.randint(1, 20)
    for c in range(COLS)
    for r in range(ROWS)
}
```

4) ***return movement_sequence, discovered_nodes, []*:** If the goal cannot be reached after exploring all reachable nodes, the function returns the explored data with an empty path []. This indicates no solution exists in the connected component.

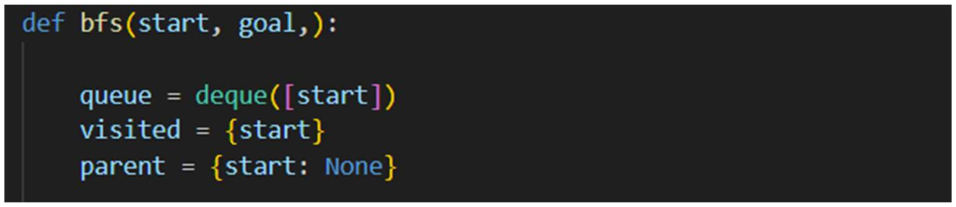
## Algorithms Implementation

- **Breadth-First Search(BFS)**

**1) Overview**

Breadth-First Search(BFS) is an uninformed search that explores nodes level by level from left to right, according to FIFO(First In First Out) queue structure.

**2) Main Function & Data Structures Initialization**



```
def bfs(start, goal,):

    queue = deque([start])
    visited = {start}
    parent = {start: None}
```

**Main Function: bfs(start, goal)**

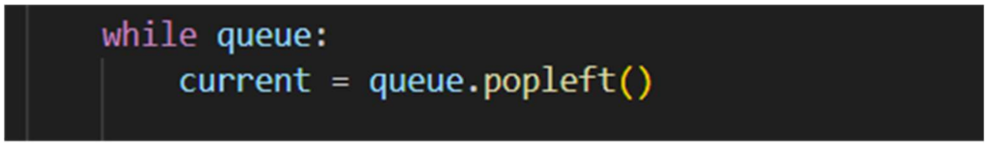
**Purpose:** This function searches for the shortest path from the start node to the goal node using a FIFO queue.

**Data Structures Initialization**

**Explanation:**

- **queue:** A FIFO (First-In-First-Out) data structure that stores nodes to be explored in order.
- **visited**: Keeps track of nodes that have been discovered to avoid revisiting them.
- **parent**: Stores the predecessor of each node to reconstruct the final path.

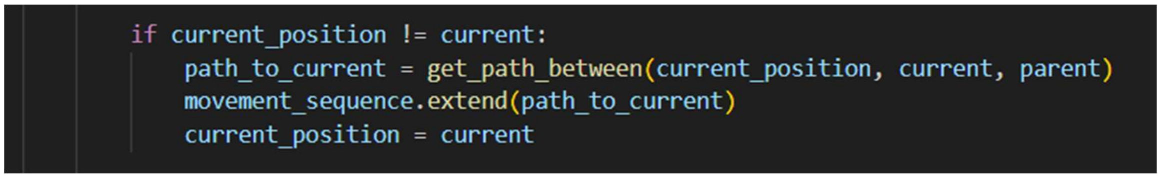
### 3) Main Search Loop



```python
while queue:
    current = queue.popleft()
```

**Purpose:** The algorithm continues searching as long as there are nodes in the queue, popleft() removes and returns the first node from the queue (FIFO behavior). This is the core principle of BFS; nodes are explored in the order they are discovered.

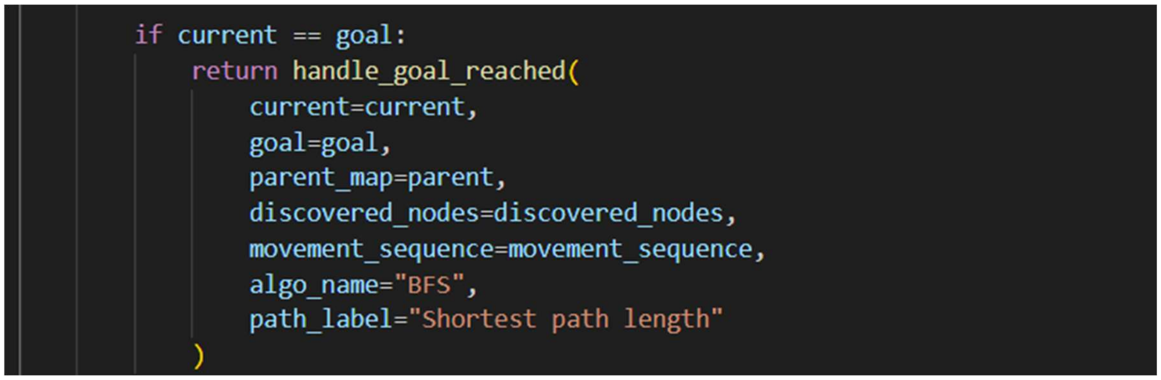
### 4) Agent Movement Update



```python
if current_position != current:
    path_to_current = get_path_between(current_position, current, parent)
    movement_sequence.extend(path_to_current)
    current_position = current
```

- Updates the movement path whenever the agent needs to move to a new node.
- Uses get_path_between() to calculate the physical path the agent must take.

### 5) Goal Test

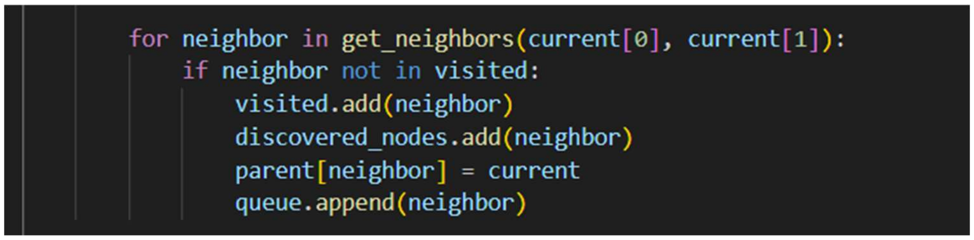


```python
if current == goal:
    return handle_goal_reached(
        current=current,
        goal=goal,
        parent_map=parent,
        discovered_nodes=discovered_nodes,
        movement_sequence=movement_sequence,
        algo_name="BFS",
        path_label="Shortest path length"
    )
```

**Purpose:**

- Once the goal node is reached, the algorithm terminates.
- BFS guarantees that this path has the minimum number of steps (shortest path in unweighted graphs).
- Calls the common handle_goal_reached() function to reconstruct and return the path.

## 6) Neighbor Expansion



```
for neighbor in get_neighbors(current[0], current[1]):
    if neighbor not in visited:
        visited.add(neighbor)
        discovered_nodes.add(neighbor)
        parent[neighbor] = current
        queue.append(neighbor)
```

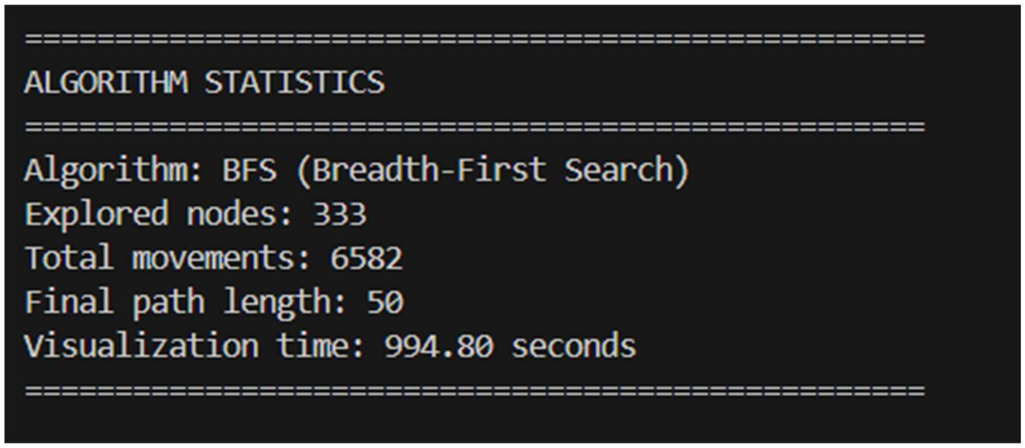
**Explanation:**

- Each neighboring node (up, down, left, right) is explored.

If a neighbor hasn't been visited before:

- Mark it as visited
- Add it to discovered nodes
- Record its parent for path reconstruction
- Add it to the end of the queue (FIFO order)

### 7) BFS Results



```
=====================================================
ALGORITHM STATISTICS
=====================================================
Algorithm: BFS (Breadth-First Search)
Explored nodes: 333
Total movements: 6582
Final path length: 50
Visualization time: 994.80 seconds
=====================================================
```

**Explored nodes:** 333
**Total movements:** 6582
**Final path length:** 50
**Visualization time:** 994.80 seconds

- **Depth-First Search(DFS)**

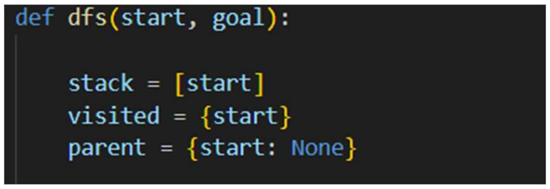
### 1) Overview

DFS is an uninformed search algorithm that explores nodes branch by branch from left to right, according to LIFO(Last In First Out) stack structure.

### 2) Main Function & Data Structures Initialization

**Main Function:** dfs(start, goal)

**Purpose:** This function searches for a path from the start node to the goal node using a LIFO stack, exploring deeply before backtracking.



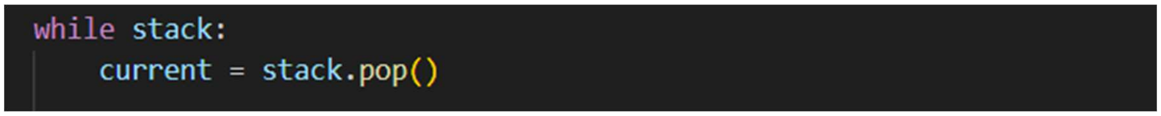
```python
def dfs(start, goal):

    stack = [start]
    visited = {start}
    parent = {start: None}
```

**Data Structures Initialization**

**Explanation:**

- **stack:** A LIFO (Last-In-First-Out) data structure that stores nodes to be explored.
- **visited**: Keeps track of nodes that have been discovered to avoid cycles and revisiting.
- **parent**: Stores the predecessor of each node to reconstruct the final path.

## 3) Main Search Loop

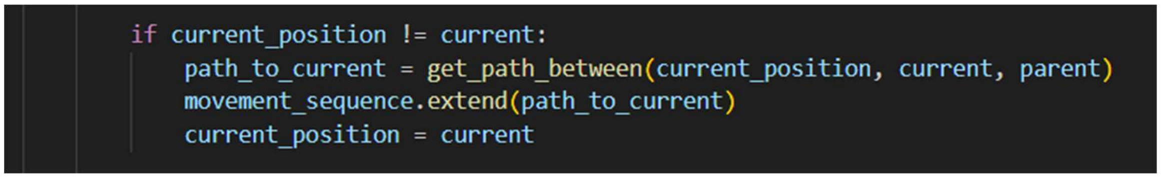


```python
while stack:
    current = stack.pop()
```

**Purpose:**

The algorithm continues searching as long as there are nodes in the stack, pop() removes and returns the **last** node from the stack (LIFO behavior). This ensures depth-first exploration(the most recently discovered node is explored first).

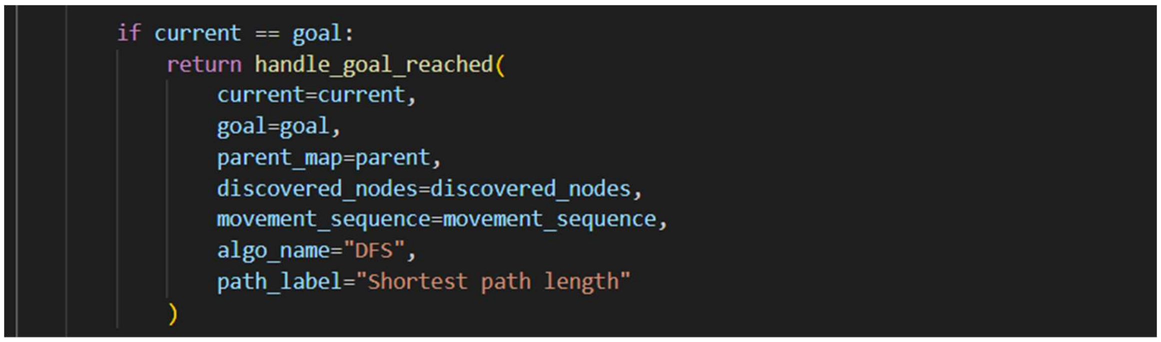
## 4) Agent Movement Update



```python
        if current_position != current:
            path_to_current = get_path_between(current_position, current, parent)
            movement_sequence.extend(path_to_current)
            current_position = current
```

- Updates the movement path whenever the agent needs to move to a new node.
- Uses get_path_between() to calculate the physical path the agent must take.
- **DFS backtracking visualization:** When a dead-end is reached, the agent physically moves back up the search tree to explore other branches.

## 5) Goal Test

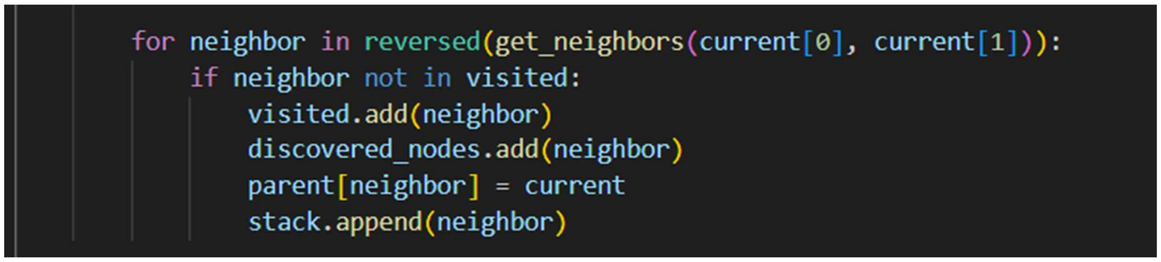


```python
if current == goal:
    return handle_goal_reached(
        current=current,
        goal=goal,
        parent_map=parent,
        discovered_nodes=discovered_nodes,
        movement_sequence=movement_sequence,
        algo_name="DFS",
        path_label="Shortest path length"
    )
```

**Purpose:**

- Once the goal node is reached, the algorithm terminates.
- **DFS does NOT guarantee the shortest path**—it returns the first path found, which may be longer than optimal.
- Calls the common handle_goal_reached() function to reconstruct and return the path.

## 6) Neighbor Expansion



```python
for neighbor in reversed(get_neighbors(current[0], current[1])):
    if neighbor not in visited:
        visited.add(neighbor)
        discovered_nodes.add(neighbor)
        parent[neighbor] = current
        stack.append(neighbor)
```

**Explanation:**

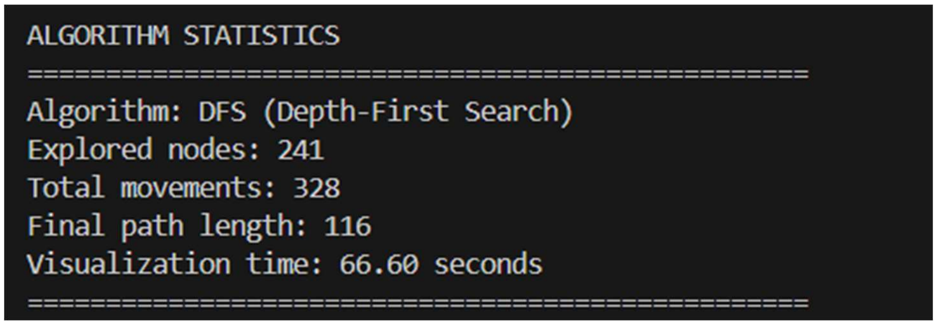
- Each neighboring node (up, down, left, right) is explored in reversed order.
- **reversed()**: Ensures consistent exploration order (implementation detail for predictable behavior).

If a neighbor hasn't been visited before:

- Mark it as visited immediately (prevents cycles)
- Add it to discovered nodes
- Record its parent for path reconstruction

- Push it onto the top of the stack (LIFO order)

**7) DFS Results**



```
ALGORITHM STATISTICS
==================================================
Algorithm: DFS (Depth-First Search)
Explored nodes: 241
Total movements: 328
Final path length: 116
Visualization time: 66.60 seconds
==================================================
```

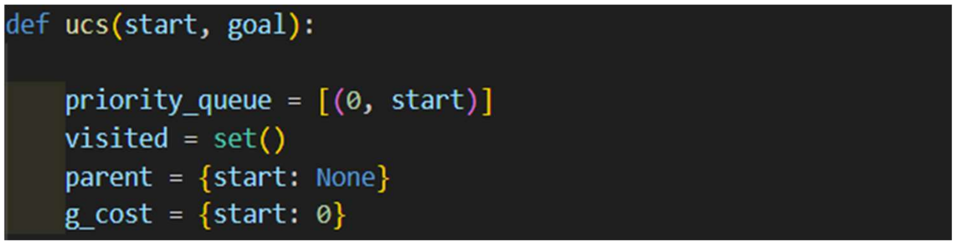
**Explored nodes:** 241
**Total movements:** 328
**Final path length:** 116
**Visualization time:** 66.60 seconds

- **Uniform-Cost Search(UCS)**

  **1) Overview**

  Uniform Cost Search (UCS) is an uninformed search algorithm used to find the **lowest-cost path** from a start node to a goal node. Unlike Breadth-First Search, UCS considers the **actual cost of moving between nodes**, ensuring that the solution found has the **minimum total path cost**, assuming all costs are positive.

  **2) Main Function & Data Structures Initialization**



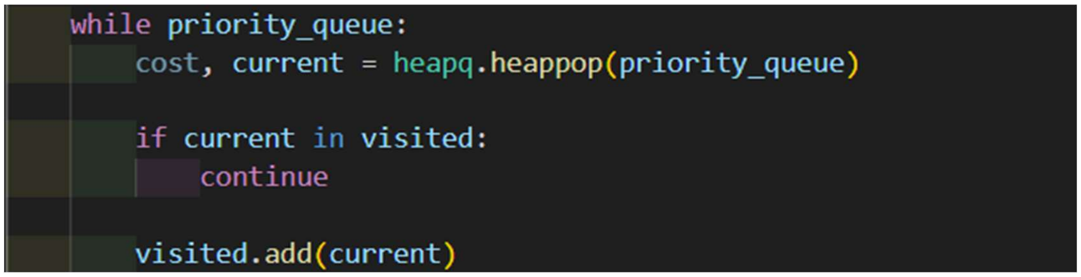
```python
def ucs(start, goal):

    priority_queue = [(0, start)]
    visited = set()
    parent = {start: None}
    g_cost = {start: 0}
```

**Main Function: ucs(start, goal)**
**Purpose:** This function searches for the least-cost path from the start node to the goal node using a priority queue.

**Data Structures Initialization**
**Explanation:**

- **priority_queue:** Stores nodes ordered by their cumulative cost from the start.
- **visited**: Keeps track of nodes that have been discovered to avoid cycles and revisiting.
- **parent**: Stores the predecessor of each node to reconstruct the final path.
- **g_cost**: Records the minimum known cost to reach each node.

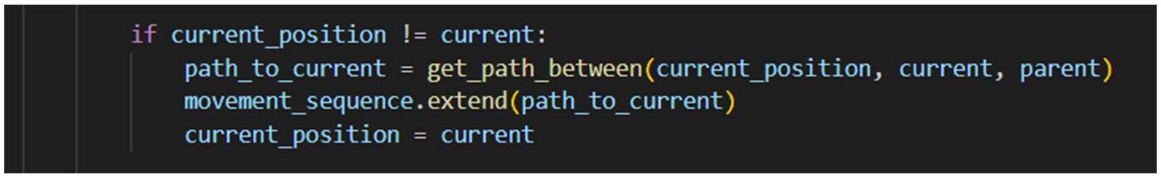
**3) Main Search Loop & Node Expansion**



```
while priority_queue:
    cost, current = heapq.heappop(priority_queue)

    if current in visited:
        continue

    visited.add(current)
```

**Main Search Loop:** The algorithm continues searching as long as there are nodes to explore.
**Node Expansion:** The node with the **lowest cumulative cost** is selected for expansion. Already visited nodes are skipped to prevent redundant processing.

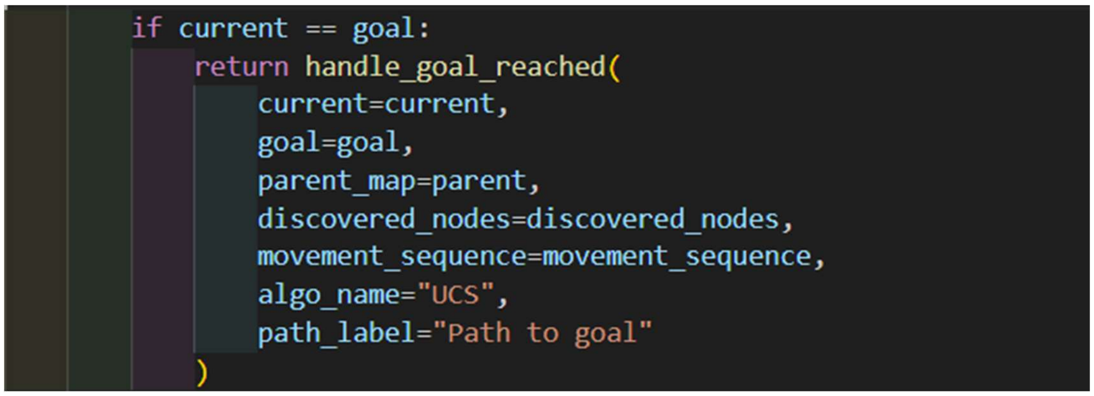
## 4) Agent Movement Update



```
if current_position != current:
    path_to_current = get_path_between(current_position, current, parent)
    movement_sequence.extend(path_to_current)
    current_position = current
```

- Updates the movement path whenever the agent needs to move to a new node.
- Uses get_path_between() to calculate the physical path the agent must take.

## 5) Goal Test

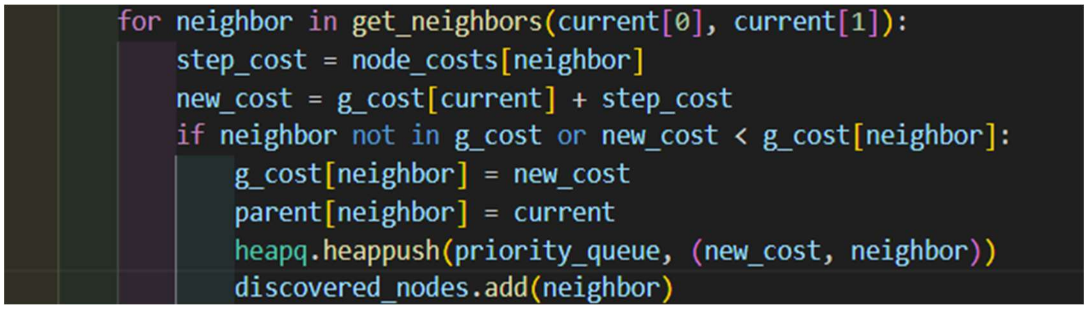


```
if current == goal:
    return handle_goal_reached(
        current=current,
        goal=goal,
        parent_map=parent,
        discovered_nodes=discovered_nodes,
        movement_sequence=movement_sequence,
        algo_name="UCS",
        path_label="Path to goal"
    )
```

**Purpose:**

- Once the goal node is reached, the algorithm terminates.
- UCS guarantees that this path has the **minimum possible cost**.

## 6) Neighbor Expansion & Cost Update



```
for neighbor in get_neighbors(current[0], current[1]):
    step_cost = node_costs[neighbor]
    new_cost = g_cost[current] + step_cost
    if neighbor not in g_cost or new_cost < g_cost[neighbor]:
        g_cost[neighbor] = new_cost
        parent[neighbor] = current
        heapq.heappush(priority_queue, (new_cost, neighbor))
        discovered_nodes.add(neighbor)
```

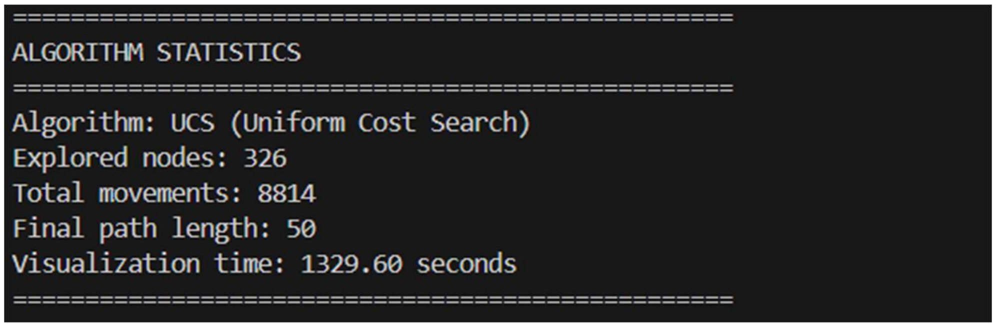
**Explanation:**

- Each neighboring node is explored.
- The new cumulative cost is calculated using the node's movement cost.

If a cheaper path to the neighbor is found:

- The cost is updated.
- The parent node is recorded.
- The neighbor is added to the priority queue.

## 7) UCS Statistics



```
==================================================
ALGORITHM STATISTICS
==================================================
Algorithm: UCS (Uniform Cost Search)
Explored nodes: 326
Total movements: 8814
Final path length: 50
Visualization time: 1329.60 seconds
==================================================
```

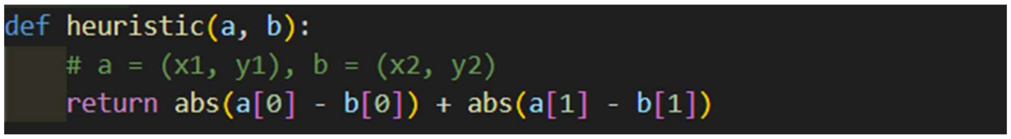
**Explored nodes:** 326
**Total movements:** 8814
**Final path length:** 50
**Visualization time:** 1329.60 seconds

- **A* Search**

**1) Overview**

A* search is an **informed search algorithm** that finds the optimal path from a start node to a goal node by combining the **actual cost from the start** and a **heuristic estimate** of the remaining distance. By using a heuristic function, A* significantly reduces the number of explored nodes compared to uninformed algorithms such as Uniform Cost Search.

**2) Heuristic Function**



```
def heuristic(a, b):
    # a = (x1, y1), b = (x2, y2)
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

**Explanation:**

- This is the **Manhattan Distance heuristic**.
- It is included for the **A*** algorithm.
- In A*, it estimates the remaining cost from a node to the goal.

**3) Cost Function Used in A***

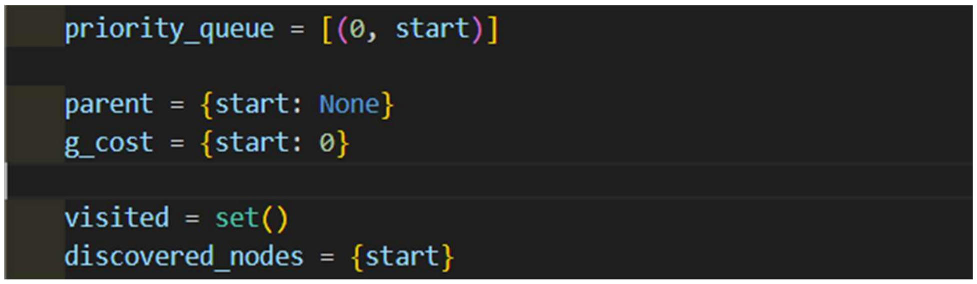
A* evaluates each node using the following equation:

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

**Where:**

- g(n) is the actual path cost from the start node to node n
- h(n) is the heuristic estimate from node n to the goal
- f(n) is the estimated total cost of the solution through node n

## 4) Main Function & Data Structures Initialization



```python
priority_queue = [(0, start)]

parent = {start: None}
g_cost = {start: 0}

visited = set()
discovered_nodes = {start}
```

**Main Function: astar(start, goal)**

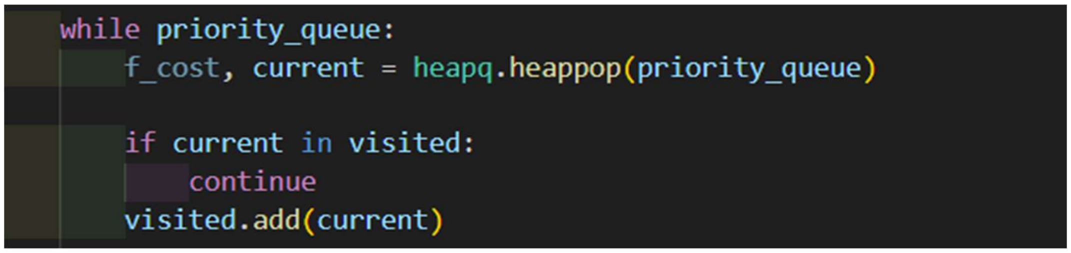
**Purpose:** This function searches for the lowest-cost path from the start node to the goal node while prioritizing nodes that appear closer to the goal using a heuristic.

**Data Structures Initialization**

**Explanation:**

- priority_queue: Stores nodes ordered by their **f-cost**.
- parent: Keeps track of each node's predecessor for path reconstruction.
- g_cost: Stores the minimum cost from the start to each node.
- visited: Prevents re-expansion of nodes.
- discovered_nodes: Stores all nodes discovered during the search.

## 5) Main Search Loop & Node Selection and Expansion



```python
while priority_queue:
    f_cost, current = heapq.heappop(priority_queue)

    if current in visited:
        continue
    visited.add(current)
```

**Main Search Loop**

**Purpose:**

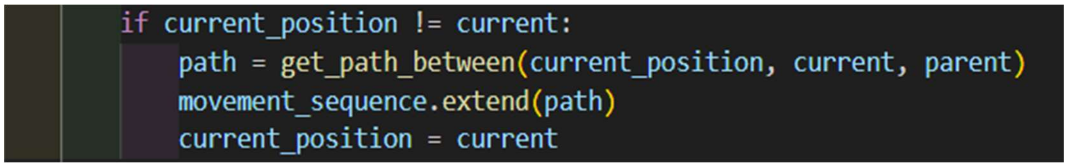
- The loop continues as long as there are nodes available for exploration.

**Node Selection and Expansion**

**Purpose:**

- The node with the lowest estimated total cost (f-cost) is selected.
- Already visited nodes are skipped to avoid redundant processing.

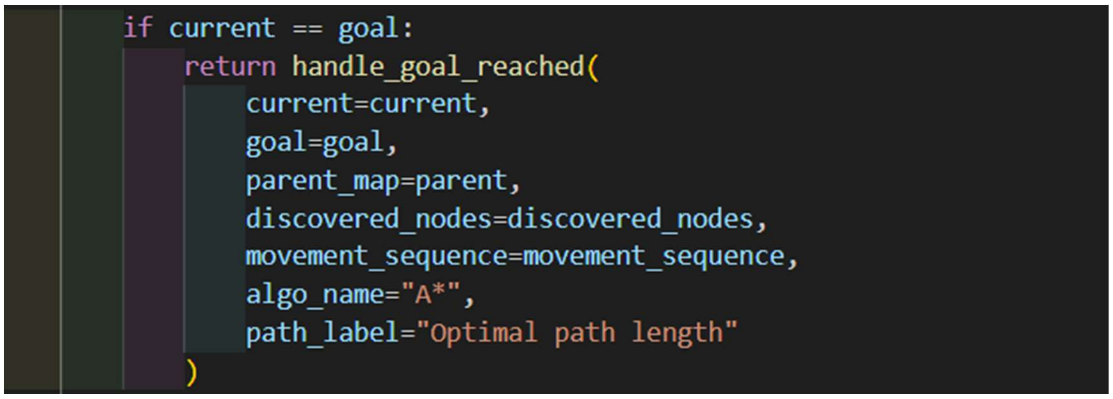
## 6) Agent Movement Update



```
if current_position != current:
    path = get_path_between(current_position, current, parent)
    movement_sequence.extend(path)
    current_position = current
```

**Purpose:** Updates the movement path whenever the agent moves to a new node.

## 7) Goal Test

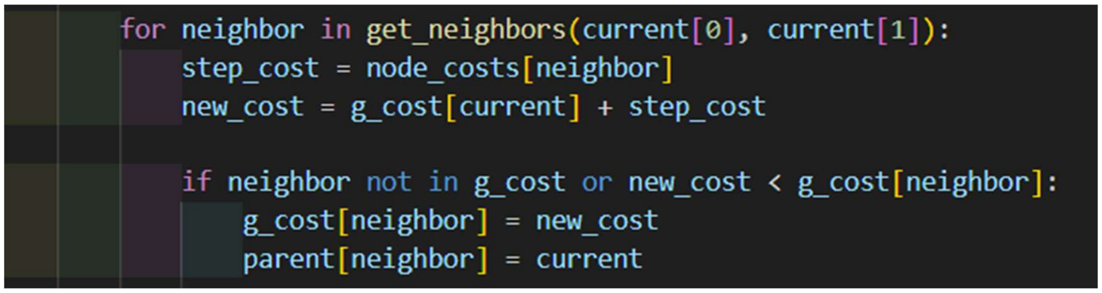


```
if current == goal:
    return handle_goal_reached(
        current=current,
        goal=goal,
        parent_map=parent,
        discovered_nodes=discovered_nodes,
        movement_sequence=movement_sequence,
        algo_name="A*",
        path_label="Optimal path length"
    )
```

**Explanation:**

- The algorithm stops immediately once the goal node is reached.
- With an admissible heuristic, A* guarantees that this path is optimal.

## 8) Neighbor Expansion & Cost Update

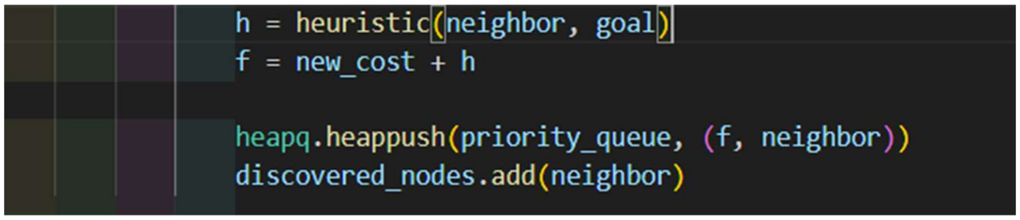


```python
for neighbor in get_neighbors(current[0], current[1]):
    step_cost = node_costs[neighbor]
    new_cost = g_cost[current] + step_cost

    if neighbor not in g_cost or new_cost < g_cost[neighbor]:
        g_cost[neighbor] = new_cost
        parent[neighbor] = current
```

**Explanation:**

- Calculates the actual cost of reaching each neighboring node.
- Updates the best known path to the neighbor.

## 9) Heuristic Calculation & Priority Queue Update



```python
h = heuristic(neighbor, goal)
f = new_cost + h

heapq.heappush(priority_queue, (f, neighbor))
discovered_nodes.add(neighbor)
```

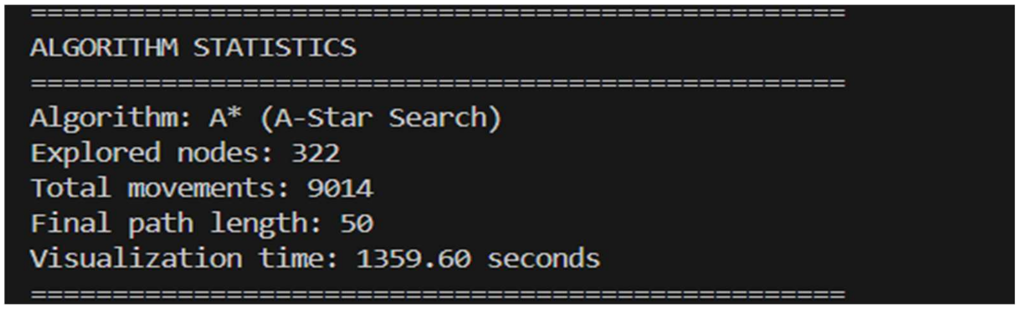
**Heuristic Calculation**
**Explanation:**

- Estimates the remaining distance to the goal.
- Guides the search toward promising nodes.

**Priority Queue Update**
**Explanation:**

- Combines actual cost and heuristic to compute the f-cost.
- Nodes with lower estimated total cost are expanded first.

### 10) A* Statistic



```
======================================================
ALGORITHM STATISTICS
======================================================
Algorithm: A* (A-Star Search)
Explored nodes: 322
Total movements: 9014
Final path length: 50
Visualization time: 1359.60 seconds
======================================================
```

**Explored nodes:** 322
**Total movements:** 9014
**Final path length:** 50
**Visualization time:** 1359.60 seconds

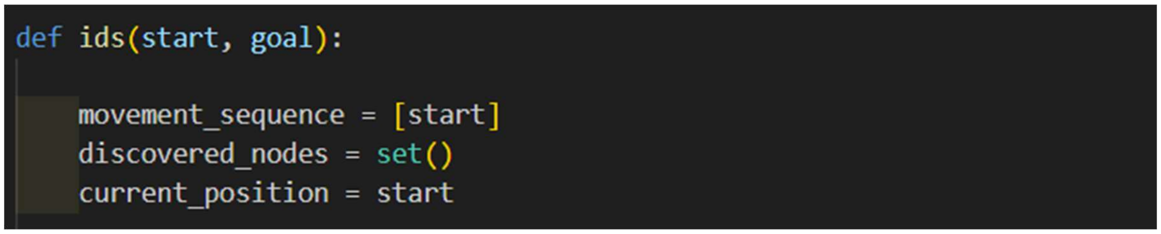
- **Iterative-Deepening Search(IDS)**

### 1)Overview

Iterative Deepening Search (IDS) is an uninformed search algorithm that combines the space efficiency of Depth-First Search with the optimality guarantee of Breadth-First Search. IDS repeatedly performs depth-limited searches with increasing depth limits (0, 1, 2, 3, ...) until the goal is found. This strategy explores the search space level by level while using minimal memory.

### 2) Main Function & Data Structures Initialization

**Main Function:** ids(start, goal)



```python
def ids(start, goal):

    movement_sequence = [start]
    discovered_nodes = set()
    current_position = start
```

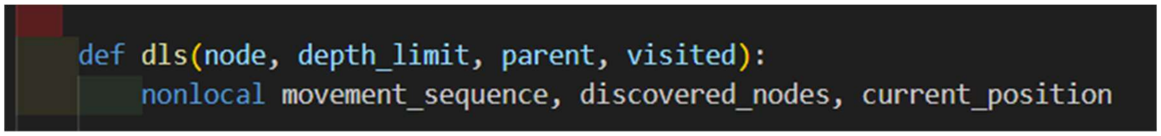
**Purpose:** This function searches for the shortest path by performing multiple depth-limited searches with incrementally increasing depth limits.

**Data Structures Initialization**

**Explanation:**

- **movement_sequence:** Tracks the agent's movement step-by-step for visualization.
- **discovered_nodes:** Stores all nodes discovered during the search (shown as red markers).
- **current_position:** Represents the agent's current location in the maze.

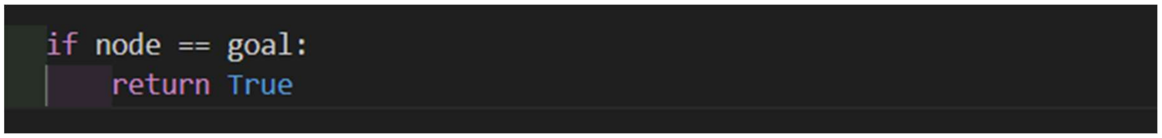
## 3) Depth-Limited Search (DLS) Function



```
def dls(node, depth_limit, parent, visited):
    nonlocal movement_sequence, discovered_nodes, current_position
```

**Purpose:** A recursive helper function that performs depth-limited DFS. Uses nonlocal to access and modify outer function variables across iterations.

## 4) Goal test

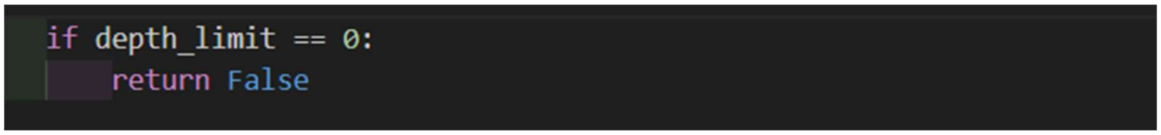


```
if node == goal:
    return True
```

**Purpose:**

- If the goal is found at the current depth, immediately return True.
- This signals success and stops all further searching.

**5) Depth-Limit Check**

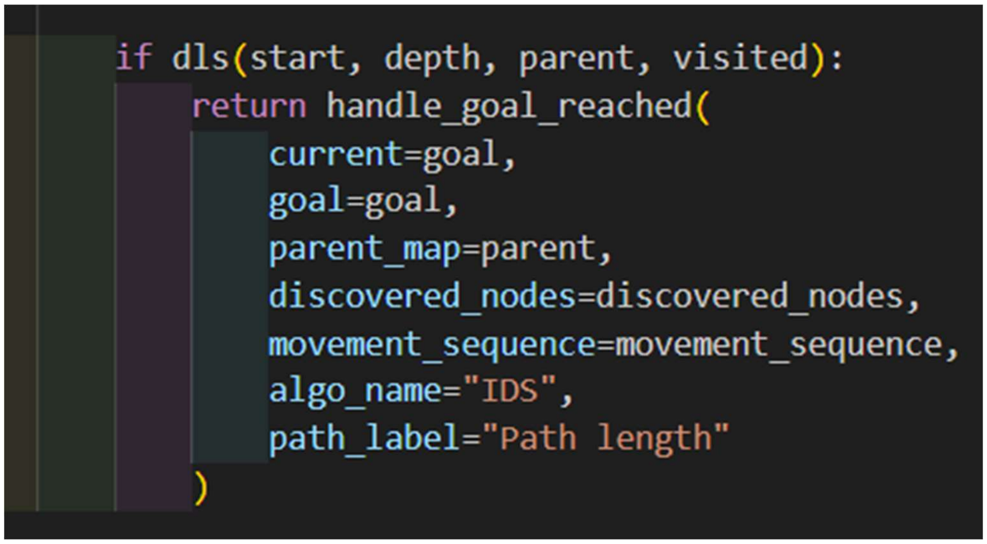


```python
if depth_limit == 0:
    return False
```

**Explanation:**

- If the current depth limit is reached and the goal hasn't been found, stop exploring this branch.
- **Cutoff condition**: Prevents going deeper than the current iteration's limit.
- The node may be re-explored in future iterations with higher depth limits.

**6) Success Case**

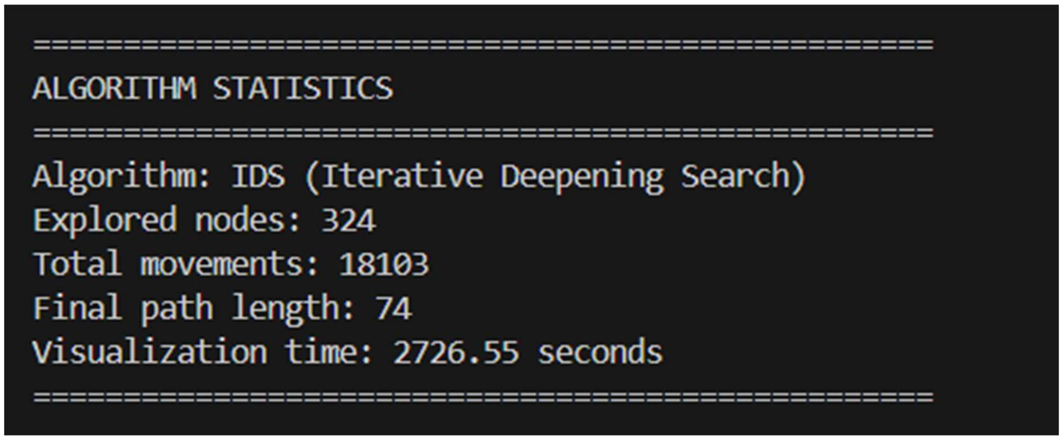


```python
        if dls(start, depth, parent, visited):
            return handle_goal_reached(
                current=goal,
                goal=goal,
                parent_map=parent,
                discovered_nodes=discovered_nodes,
                movement_sequence=movement_sequence,
                algo_name="IDS",
                path_label="Path length"
            )
```

**Purpose:**

- When dls() returns True, the goal has been found at the current depth.
- Calls handle_goal_reached() to reconstruct the optimal path.
- **IDS guarantees optimality**: The first depth at which the goal is found yields the shortest path.

### 7) IDS Statistics



```
=====================================================
ALGORITHM STATISTICS
=====================================================
Algorithm: IDS (Iterative Deepening Search)
Explored nodes: 324
Total movements: 18103
Final path length: 74
Visualization time: 2726.55 seconds
=====================================================
```

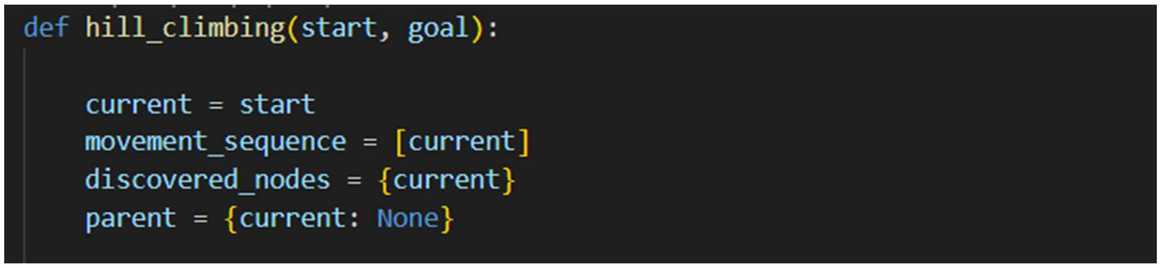
**Explored nodes:** 324
**Total movements:** 18103
**Final path length:** 74
**Visualization time:** 2726.55 seconds

- **Hill Climbing Search**

### 1) Overview

Hill Climbing is a heuristic search algorithm that moves to the neighbor that looks closest to the goal, It does not explore all paths and can get stuck in local optimal.

### 2) Main Function & Data Structures Initialization



```
def hill_climbing(start, goal):

    current = start
    movement_sequence = [current]
    discovered_nodes = {current}
    parent = {current: None}
```

**Main Function:** def hill_climbing(start, goal)
**Puropse:** The function initializes the search, repeatedly selects the best neighboring  node using a heuristic, updates

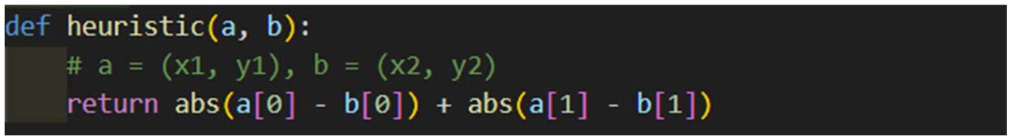
the current position, and stops when the goal is reached or no improvement is possible.

**Data Structures Initialization**

**Explanation:**

- Current: Represents the present position.
- **movement_sequence:** Tracks the agent's movement step-by-step for visualization.
- **discovered_nodes:** Stores all nodes discovered during the search (shown as red markers).
- Parent: Maps each node to its parent (predecessor) node and reconstruct the path from start to goal (final path).

## 3) Heuristic Function

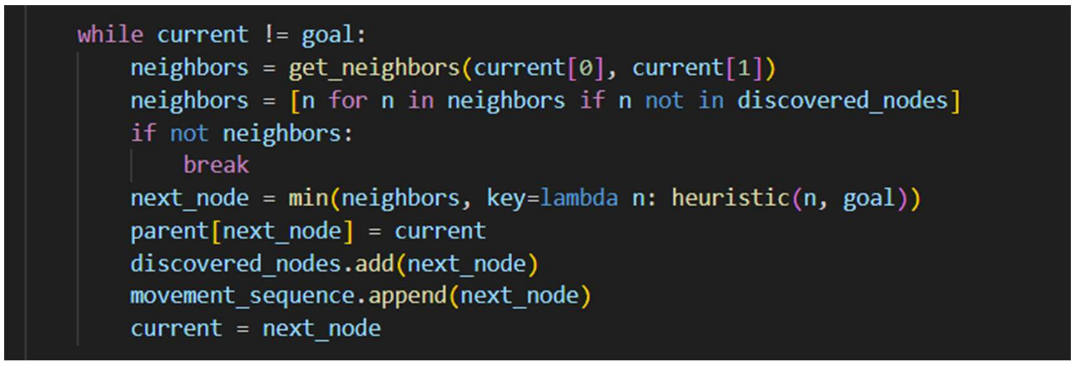


```python
def heuristic(a, b):
    # a = (x1, y1), b = (x2, y2)
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

**Explanation:**

Calculates Manhattan distance and estimates how close a node is to the goal.

## 4) Neighbors Expansion



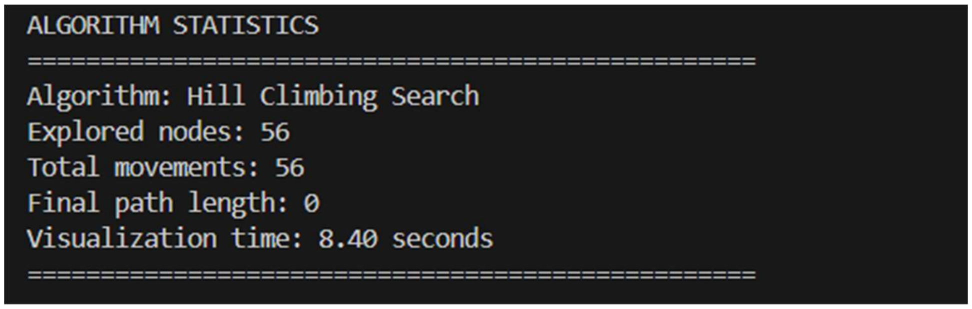
```
while current != goal:
    neighbors = get_neighbors(current[0], current[1])
    neighbors = [n for n in neighbors if n not in discovered_nodes]
    if not neighbors:
        break
    next_node = min(neighbors, key=lambda n: heuristic(n, goal))
    parent[next_node] = current
    discovered_nodes.add(next_node)
    movement_sequence.append(next_node)
    current = next_node
```

**Explanation:**

- This loop runs as long as the goal has not been reached.
- Retrieves all valid neighboring nodes of the current position.
- Prevent the algorithm from visiting neighbors that were already visited.
- Checks if there are no available neighbors to move to, which means that the algorithm is stuck in a local optimal; therefore the loop stops even if the goal was not reached.
- next_node: evaluates each neighbor using the heuristic function and Selects the neighbor with the smallest heuristic value which is most close to the goal.
- parent[next_node]: Stores the current node as the parent of the selected neighbor.
- discovered_nodes.add(next_node): Marks the selected neighbor as visited.

- movement_sequence.append(next_node): Records the movement to the next node.
- Current: Moves the algorithm to the selected neighbor.

## 5) Hill Climbing Statistics



```
ALGORITHM STATISTICS
==================================================
Algorithm: Hill Climbing Search
Explored nodes: 56
Total movements: 56
Final path length: 0
Visualization time: 8.40 seconds
==================================================
```

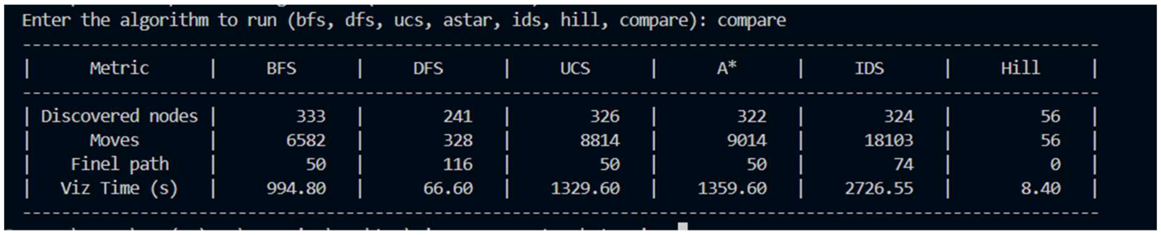
**Explored nodes:** 56
**Total movements:** 56
**Final path length:** 0
**Visualization time:** 8.40 seconds

## Algorithms Comparison

after executing all six search algorithms on the same maze environment (25×30) with identical start and goal positions, comparison was conducted based on the following:

* Number of explored (discovered) nodes
* Total agent movements
* Final path length
* Visualization execution time

---



```
Enter the algorithm to run (bfs, dfs, ucs, astar, ids, hill, compare): compare
-----------------------------------------------------------------------------------------------
|      Metric      |      BFS      |      DFS      |      UCS      |      A*      |      IDS      |      Hill      |
-----------------------------------------------------------------------------------------------
| Discovered nodes |           333 |           241 |           326 |          322 |           324 |             56 |
|      Moves       |          6582 |           328 |          8814 |         9014 |         18103 |             56 |
|    Finel path    |            50 |           116 |            50 |           50 |            74 |              0 |
|   Viz Time (s)   |        994.80 |         66.60 |       1329.60 |      1359.60 |       2726.55 |           8.40 |
-----------------------------------------------------------------------------------------------
```

---

### Breadth-First Search (BFS)

- BFS successfully reached the goal and produced the shortest path (50 steps). However, it explored a large number of nodes (333) and required many movements, leading to a long execution time.
- Reliable and optimal
- slow and inefficient for emergency evacuation.

---

**Depth-First Search (DFS)**

- DFS reached the goal quickly with fewer explored nodes and movements, but generated a non-optimal long path (116 steps).
- Fast and memory-efficient, but unsafe due to lack of optimality.

---

**Uniform-Cost Search (UCS)**

- UCS reached the goal and found an optimal path (50 steps), but required high computational effort and long execution time due to cost-based exploration.
- Optimal but impractical for real-time fire escape scenarios.

---

**A* Search**

- A* successfully reached the goal with an optimal path (50 steps) and explored slightly fewer nodes than BFS and UCS. However, visualization time remained high.
- Best balance between optimality and guided search
- most suitable algorithm for this problem.

---

**Iterative Deepening Search (IDS)**

- IDS reached the goal but with a longer path (74 steps) and the highest execution time due to repeated depth-limited searches.
- Correct but highly inefficient
- time-consuming.

**Hill Climbing Search**

- Hill Climbing failed to reach the goal and stopped at a local optimum, despite being very fast and exploring few nodes.
- Fast but unreliable
- unsafe for evacuation planning.

---

## Final Comparison Conclusion

Overall Comparison and Best Algorithm Selection
Based on the experimental results:

- Optimal and successful algorithms: BFS, UCS, A*, IDS
- Fast but non-optimal: DFS
- Fast but failed: Hill Climbing

Among all algorithms, A* Search provides the best balance between optimality and intelligent guidance, making it the most suitable choice for a fire escape route agent when heuristics are available.
However, if heuristic information is weak or unavailable, BFS and UCS remain a reliable alternative.
For real-world emergency evacuation scenarios, algorithms must balance speed, safety, and reliability, and A* Search best fulfills these requirements in this study.