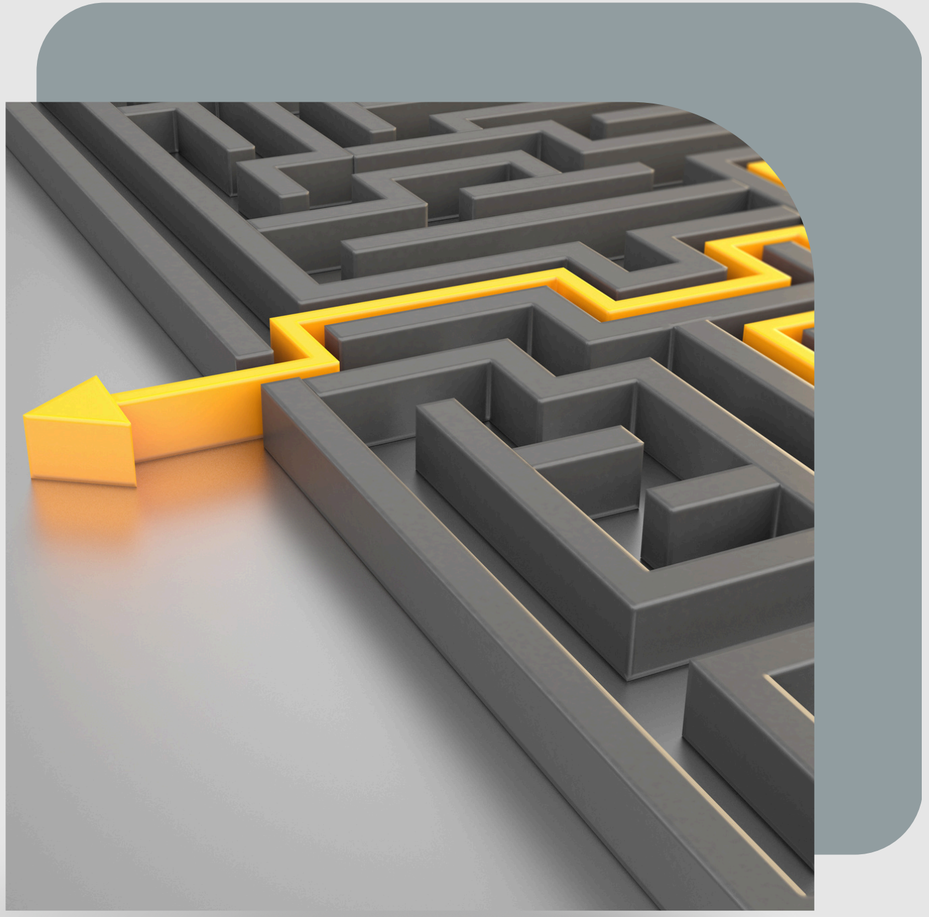
# AI-Powered Maze Solver




**PROJECT BY**
**AI NINJAS TEAM**

**SUPERVISED BY**
**DR. TAMER EMARA**

# Meet Our Team

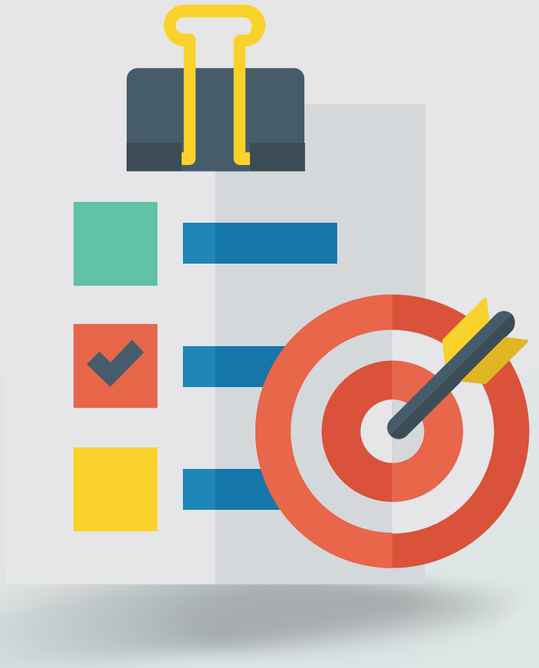
محمود السيد عبد الحميد شريف

محمود محمد الحسيني عبد الغني

توفيق احمد توفيق المزين

# Table of Content

# Introduction



# Project Overview

THIS PROJECT FOCUSES ON THE DESIGN AND IMPLEMENTATION OF AN EFFICIENT AI-POWERED MAZE-SOLVING ALGORITHM. IT DEMONSTRATES THE APPLICATION OF ENGINEERING PRINCIPLES IN DEVELOPING INTELLIGENT SYSTEMS CAPABLE OF NAVIGATING COMPLEX ENVIRONMENTS.

# Relevance

PATHFINDING ALGORITHMS PLAY A CRITICAL ROLE IN FIELDS LIKE ROBOTICS, AUTONOMOUS SYSTEMS, GAMING, AND LOGISTICS. THIS PROJECT NOT ONLY ENHANCES PROBLEM-SOLVING SKILLS BUT ALSO CONTRIBUTES TO PRACTICAL REAL-WORLD APPLICATIONS.

# Project Goals

## Primary Objective:

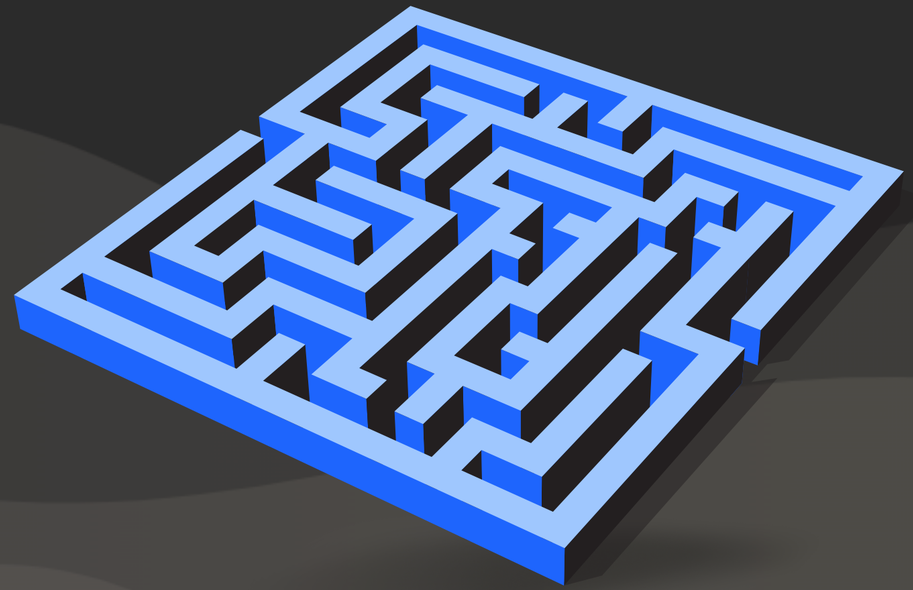
DESIGN AND IMPLEMENT AN EFFICIENT ALGORITHM CAPABLE OF NAVIGATING AND SOLVING A COMPLEX MAZE.

# Project Goals

## Secondary Objectives:

- DEMONSTRATE THE INTEGRATION OF ENGINEERING PRINCIPLES IN AI PATHFINDING.
- OPTIMIZE THE ALGORITHM FOR EFFICIENCY AND RELIABILITY IN VARYING MAZE LAYOUTS.
- PRESENT A CLEAR VISUAL REPRESENTATION OF THE MAZE-SOLVING PROCESS.

# Maze Design



# Maze Representation

```
XXXXXXXXXXXXXGX
XEEEEEEEEEEEEEX
XXXEXXXXXXXXXXX
XXXEXXXXXXXXXXX
XXXEXEEEEEEEEEX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXEEEEEEEXXXXX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXSXXXXXXXXXXX
```

- **INPUT FORMAT:**

**THE MAZE IS DEFINED IN A TEXT FILE (MAP.TXT) USING A GRID-LIKE STRUCTURE. EACH CHARACTER REPRESENTS A SPECIFIC ELEMENT:**

- **S: START POINT.**
- **G: GOAL.**
- **E: EMPTY SPACES FOR MOVEMENT.**
- **X: WALLS OR OBSTACLES.**

```python
# Loading the maze from map.txt
with open('map.txt', 'r') as file:
    maze = [line.strip() for line in file]

# Example output (raw maze format)
for row in maze:
    print(row)

```

# Maze Conversion

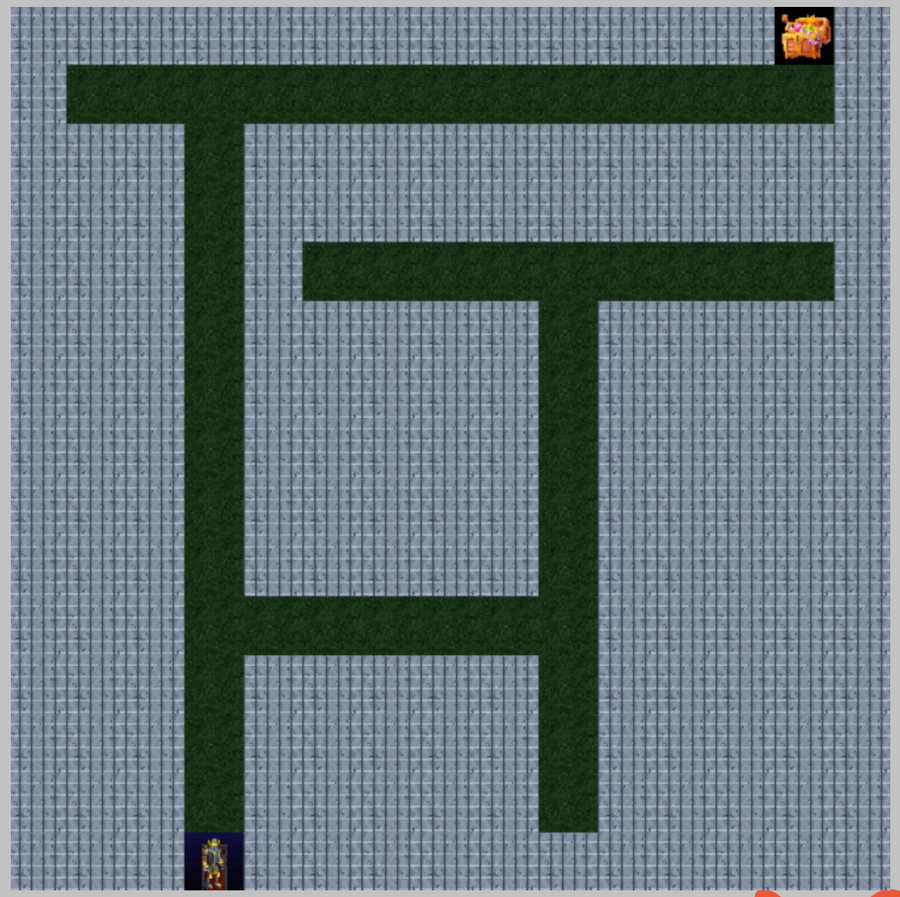
- **PROCESS:**

**THE RAW TEXT-BASED MAZE IS CONVERTED INTO A GRAPHICAL MAP FOR BETTER VISUALIZATION. EACH SYMBOL IS MAPPED TO ITS CORRESPONDING TEXTURE USING PYTHON AND PYGAME.**

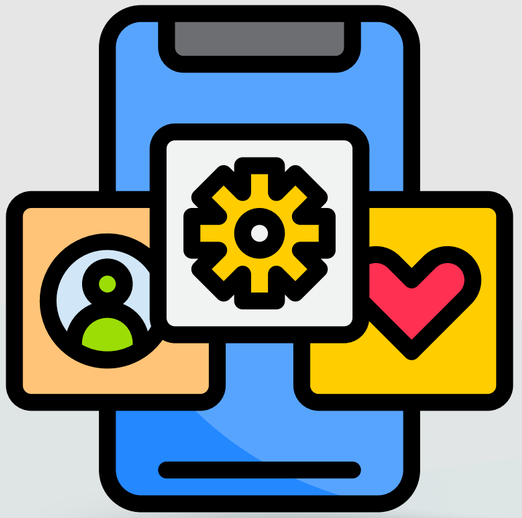
```python
# Loading textures for the graphical representation
def load_textures():
    textures = {
        'X': pygame.image.load("assets/wall.png"),
        'E': pygame.image.load("assets/floor.png"),
        'S': pygame.image.load("assets/start.png"),
        'G': pygame.image.load("assets/goal.png")
    }
    for key in textures:
        textures[key] = pygame.transform.scale(textures[key], (TILE_SIZE, TILE_SIZE))
    return textures
```

# Maze Conversion

```
XXXXXXXXXXXXXGX
XEEEEEEEEEEEEEX
XXXEXXXXXXXXXXX
XXXEXXXXXXXXXXX
XXXEXEEEEEEEEEX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXEEEEEEEXXXXX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXEXXXXXEXXXXX
XXXSXXXXXXXXXXX
```



# Game Features



# Manual Play

```python
if event.type == pygame.KEYDOWN and not solving:
    if event.key == pygame.K_UP and maze[player_pos[0] - 1][player_pos[1]] in ('E', 'G', 'S'):
        player_pos[0] -= 1
    elif event.key == pygame.K_DOWN and maze[player_pos[0] + 1][player_pos[1]] in ('E', 'G', 'S'):
        player_pos[0] += 1
    elif event.key == pygame.K_LEFT and maze[player_pos[0]][player_pos[1] - 1] in ('E', 'G', 'S'):
        player_pos[1] -= 1
    elif event.key == pygame.K_RIGHT and maze[player_pos[0]][player_pos[1] + 1] in ('E', 'G', 'S'):
        player_pos[1] += 1
```

**PLAYERS CAN MANUALLY NAVIGATE THE MAZE USING ARROW KEYS TO REACH THE GOAL.**

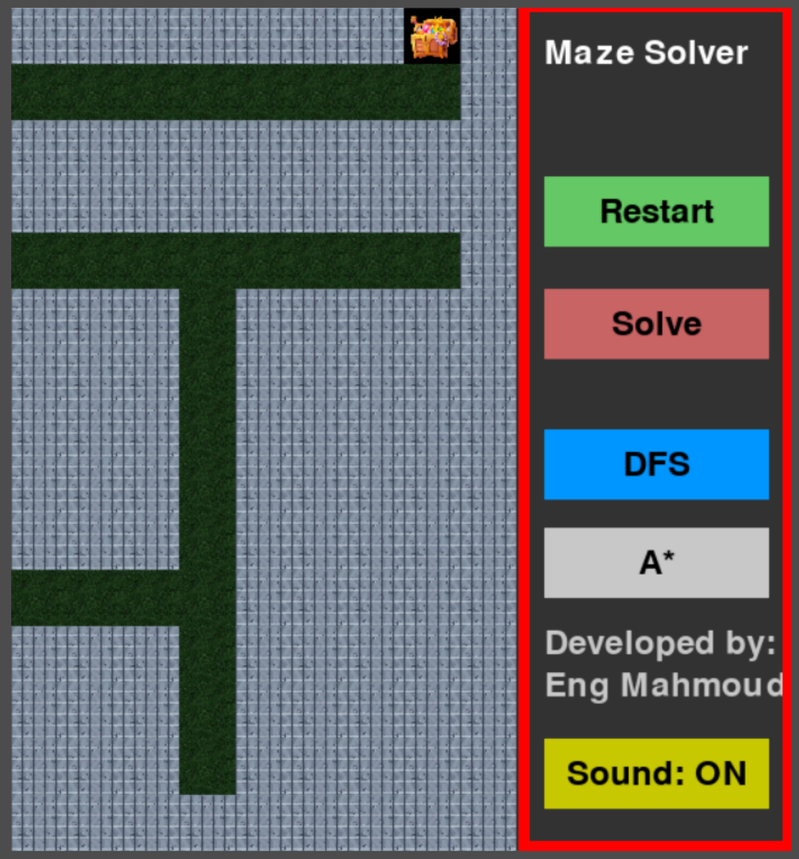
# Sound Effects and Background Music:

```
1  pygame.mixer.init()
2  pygame.mixer.music.load("assets/background_music.mp3")
3  pygame.mixer.music.play(-1)  # Looping background music
4
5  click_sound = pygame.mixer.Sound("assets/click.wav")
6  victory_sound = pygame.mixer.Sound("assets/victory.wav")
```

**IMMERSIVE SOUND EFFECTS AND BACKGROUND MUSIC ENHANCE THE GAMEPLAY EXPERIENCE**

# Sidebar Functionality:

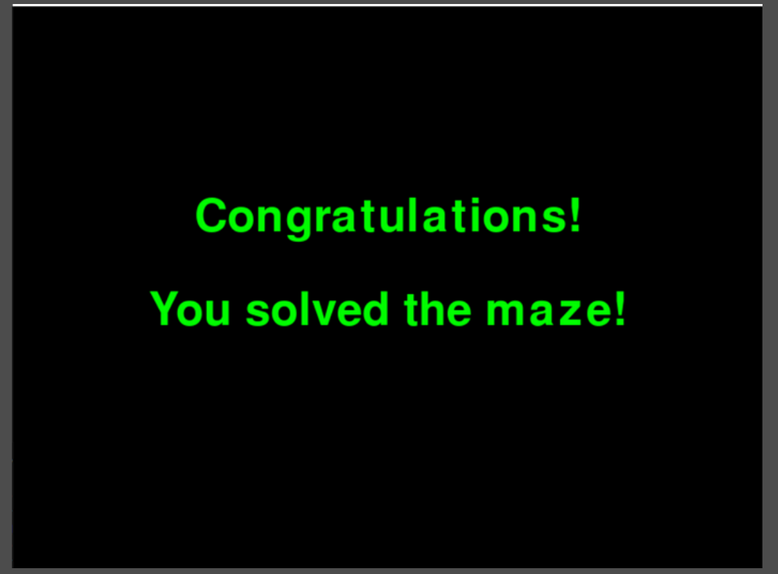
```python
def draw_sidebar(screen, width, height, selected_algorithm, sound_on):
    restart_rect = pygame.Rect(width - SIDEBAR_WIDTH + 20, 120, 160, 50)
    solve_rect = pygame.Rect(width - SIDEBAR_WIDTH + 20, 200, 160, 50)
    dfs_rect = pygame.Rect(width - SIDEBAR_WIDTH + 20, 300, 160, 50)
    a_star_rect = pygame.Rect(width - SIDEBAR_WIDTH + 20, 370, 160, 50)

    # Drawing buttons and toggling states
    pygame.draw.rect(screen, (100, 200, 100), restart_rect)
    pygame.draw.rect(screen, (200, 100, 100), solve_rect)
```




**INTERACTIVE SIDEBAR FOR RESTARTING THE GAME, TOGGLING SOUND, AND SELECTING ALGORITHMS.**

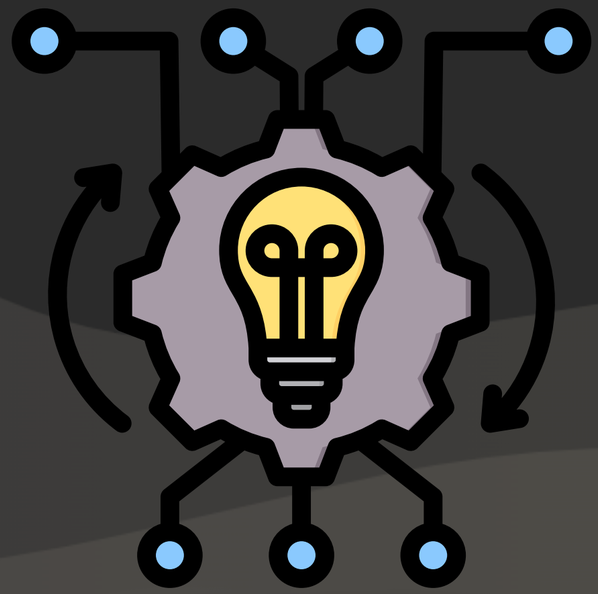
# Victory Message:

```python
def victory_message(screen, screen_width, screen_height):
    font = pygame.font.Font(None, 72)
    message = font.render("Congratulations!", True, (0, 255, 0))
    screen.fill((0, 0, 0))
    screen.blit(message, (screen_width // 2 - message.get_width() // 2, screen_height // 3))
    pygame.display.flip()
```



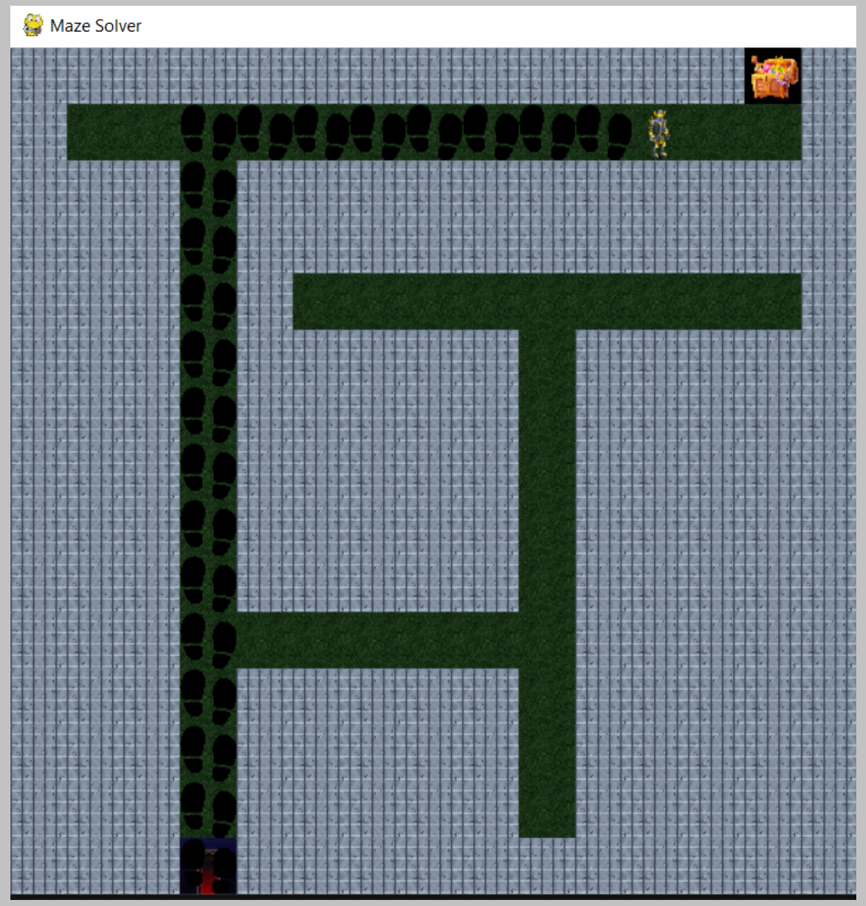

**CONGRATULATORY MESSAGE DISPLAYED WHEN THE GOAL IS REACHED**

# Algorithm Implementation



# Depth-First Search (DFS)

- **OVERVIEW:**
  - DFS IS AN ALGORITHM THAT EXPLORES ALL POSSIBLE PATHS EXHAUSTIVELY, BACKTRACKING WHEN A DEAD END IS REACHED, UNTIL THE GOAL IS FOUND.
- **ADVANTAGES:**
  - SIMPLE AND EASY TO IMPLEMENT.
  - EFFECTIVE FOR SMALLER OR LESS COMPLEX MAZES.
- **DISADVANTAGES:**
  - MAY EXPLORE REDUNDANT PATHS, MAKING IT LESS EFFICIENT FOR LARGE OR COMPLEX MAZES.



# Depth-First Search (DFS)

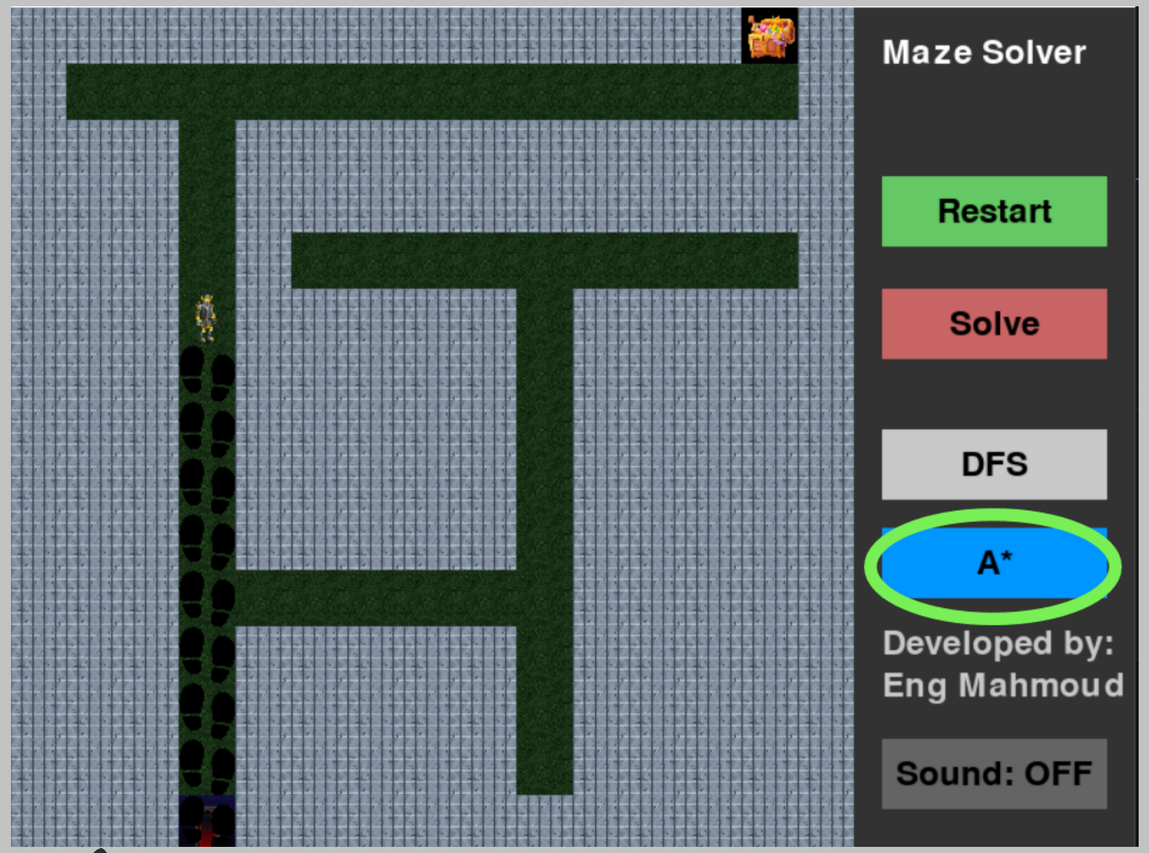
```python
def solve_with_dfs(screen, maze, start, goal, textures, player_texture, footprint_texture, screen_width, screen_height, sound_on):
    stack = [(start, [])]
    visited = set()
    while stack:
        (x, y), path = stack.pop()
        if (x, y) in visited:
            continue
        visited.add((x, y))
        if (x, y) == goal:
            return path + [(x, y)], len(visited), time.time() - start_time
        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and maze[nx][ny] in ('E', 'G') and (nx, ny) not in visited:
                stack.append(((nx, ny), path + [(x, y)]))
```

# Code Explanation:

- INITIALIZATION:
  - STACK: KEEPS TRACK OF THE CURRENT POSITION AND THE PATH TAKEN TO GET THERE. INITIALIZED WITH THE START POSITION AND AN EMPTY PATH.
  - VISITED: A SET TO KEEP TRACK OF VISITED CELLS, PREVENTING REDUNDANT EXPLORATION.
- WHILE LOOP:
  - THE ALGORITHM ITERATES UNTIL THE STACK IS EMPTY, INDICATING ALL PATHS HAVE BEEN EXPLORED OR THE GOAL IS FOUND.
- POP OPERATION:
  - RETRIEVES THE LAST NODE (X, Y) AND THE PATH LEADING TO IT FROM THE STACK.
- VISITED CHECK:
  - SKIPS ANY NODE THAT HAS ALREADY BEEN VISITED.
- GOAL CHECK:
  - IF THE CURRENT NODE MATCHES THE GOAL POSITION, THE FUNCTION RETURNS:
    - THE COMPLETE PATH (PATH + [(X, Y)]).
    - THE NUMBER OF NODES VISITED (LEN(VISITED)).
    - THE TOTAL TIME TAKEN TO SOLVE THE MAZE (TIME.TIME() - START_TIME).
- NEIGHBOR EXPLORATION:
  - ITERATES OVER FOUR POSSIBLE DIRECTIONS: RIGHT, DOWN, LEFT, UP.
  - FOR EACH MOVE, CALCULATES THE NEW POSITION (NX, NY) AND VALIDATES IT:
    - IT MUST BE WITHIN THE MAZE BOUNDS.
    - IT MUST BE AN EMPTY CELL ('E') OR THE GOAL ('G').
    - IT MUST NOT HAVE BEEN VISITED.
- STACK UPDATE:
  - VALID MOVES ARE ADDED TO THE STACK ALONG WITH THE UPDATED PATH.

# A* Algorithm

- **OVERVIEW:**
  - A IS AN OPTIMIZATION OF PATHFINDING THAT COMBINES THE ACTUAL COST OF THE PATH SO FAR AND A HEURISTIC ESTIMATE OF THE REMAINING DISTANCE TO THE GOAL ('G').
- **ADVANTAGES:**
  - EFFICIENTLY FINDS THE SHORTEST PATH.
  - IDEAL FOR LARGER AND MORE COMPLEX MAZES.
- **DISADVANTAGES:**
  - MORE COMPUTATIONALLY INTENSIVE DUE TO HEURISTIC CALCULATIONS.



# A* Algorithm

```python
def solve_with_a_star(screen, maze, start, goal, textures, player_texture, footprint_texture, screen_width, screen_height, sound_on):
    open_set = []
    heapq.heappush(open_set, (0, start, [], 0))  # (priority, current_position, path, cost)
    visited = set()
    while open_set:
        _, (x, y), path, cost = heapq.heappop(open_set)  # Get the node with the lowest priority
        if (x, y) in visited:
            continue
        visited.add((x, y))  # Mark the node as visited
        if (x, y) == goal:  # Goal Check
            return path + [(x, y)], len(visited), time.time() - start_time
        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:  # Explore neighbors
            nx, ny = x + dx, y + dy
            if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and maze[nx][ny] in ('E', 'G') and (nx, ny) not in visited:
                priority = cost + 1 + abs(nx - goal[0]) + abs(ny - goal[1])  # Cost + heuristic
                heapq.heappush(open_set, (priority, (nx, ny), path + [(x, y)], cost + 1))
```

# Code Explanation:

- INITIALIZATION:
  - OPEN_SET: A PRIORITY QUEUE (MIN-HEAP) THAT STORES NODES ALONG WITH THEIR PRIORITY (ESTIMATED TOTAL COST).
    - INITIALIZED WITH THE START POSITION, AN EMPTY PATH, AND A COST OF 0.
  - VISITED: A SET TO KEEP TRACK OF VISITED NODES.
- HEAP OPERATIONS:
  - HEAPQ.HEAPPOP: RETRIEVES THE NODE WITH THE LOWEST PRIORITY FROM THE OPEN_SET.
    - THIS ENSURES THE ALGORITHM ALWAYS PROCESSES THE MOST PROMISING PATH FIRST.
- VISITED CHECK:
  - SKIPS NODES THAT HAVE ALREADY BEEN EXPLORED.
- GOAL CHECK:
  - IF THE CURRENT NODE MATCHES THE GOAL, THE FUNCTION RETURNS:
    - THE COMPLETE PATH (PATH + [(X, Y)]).
    - THE NUMBER OF VISITED NODES (LEN(VISITED)).
    - THE SOLVING TIME (TIME.TIME() - START_TIME).
- NEIGHBOR EXPLORATION:
  - ITERATES OVER FOUR POSSIBLE DIRECTIONS: RIGHT, DOWN, LEFT, UP.
  - FOR EACH NEIGHBOR, CALCULATES THE NEW POSITION (NX, NY) AND VALIDATES IT:
    - IT MUST BE WITHIN THE MAZE BOUNDS.
    - IT MUST BE AN EMPTY CELL ('E') OR THE GOAL ('G').
    - IT MUST NOT HAVE BEEN VISITED.
- PRIORITY CALCULATION:
  - THE PRIORITY IS COMPUTED AS:
    - COST: THE ACTUAL PATH COST SO FAR.
    - + HEURISTIC: THE MANHATTAN DISTANCE FROM THE NEW POSITION TO THE GOAL (ABS(NX - GOAL[0]) + ABS(NY - GOAL[1])).
- HEAP UPDATE:
  - VALID NEIGHBORS ARE PUSHED ONTO THE OPEN_SET WITH THEIR CALCULATED PRIORITY.

# Algorithm Selection Process

- PURPOSE:

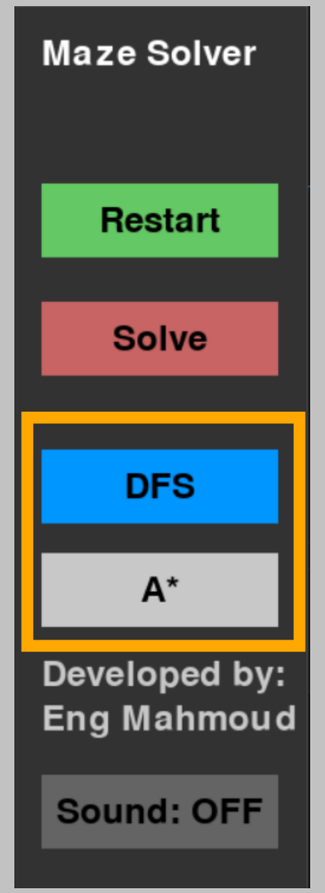
ALLOWING THE USER TO CHOOSE BETWEEN DFS AND A ENSURES FLEXIBILITY FOR SOLVING DIFFERENT TYPES OF MAZES EFFICIENTLY.

- **Comparison of DFS and A*:**

| Aspect | DFS | A* |
|---|---|---|
| **Efficiency** | May explore redundant paths | Optimized with heuristics |
| **Suitability** | Smaller, simpler mazes | Larger, more complex mazes |
| **Complexity** | Easier to implement and debug | More computationally intensive |

# Algorithm Selection Process

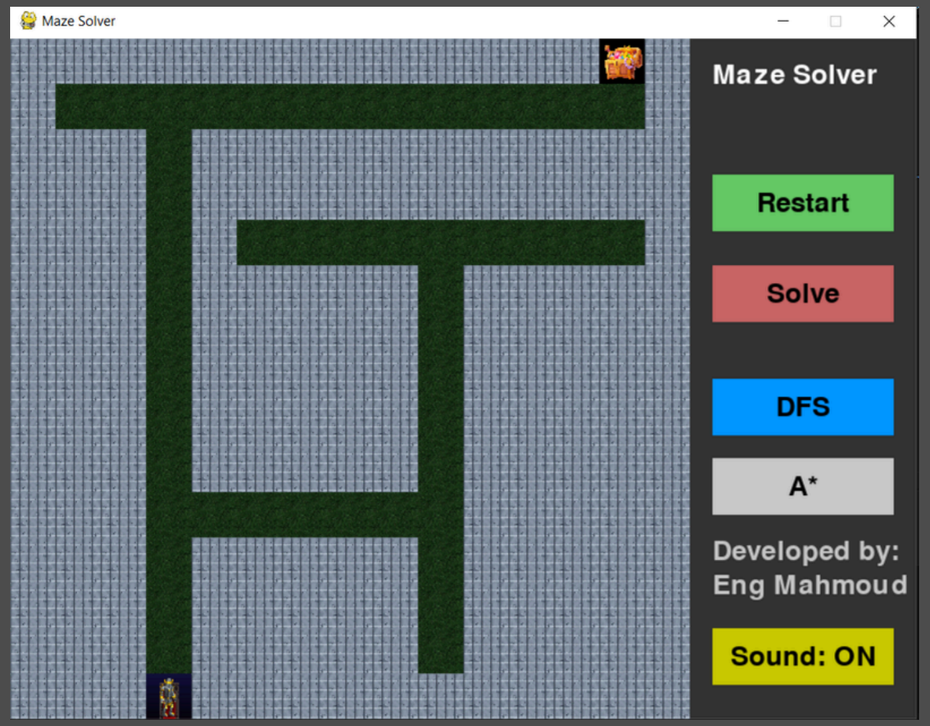
```
1  elif dfs_rect.collidepoint(event.pos):
2      selected_algorithm = "DFS"
3  elif a_star_rect.collidepoint(event.pos):
4      selected_algorithm = "A*"
```

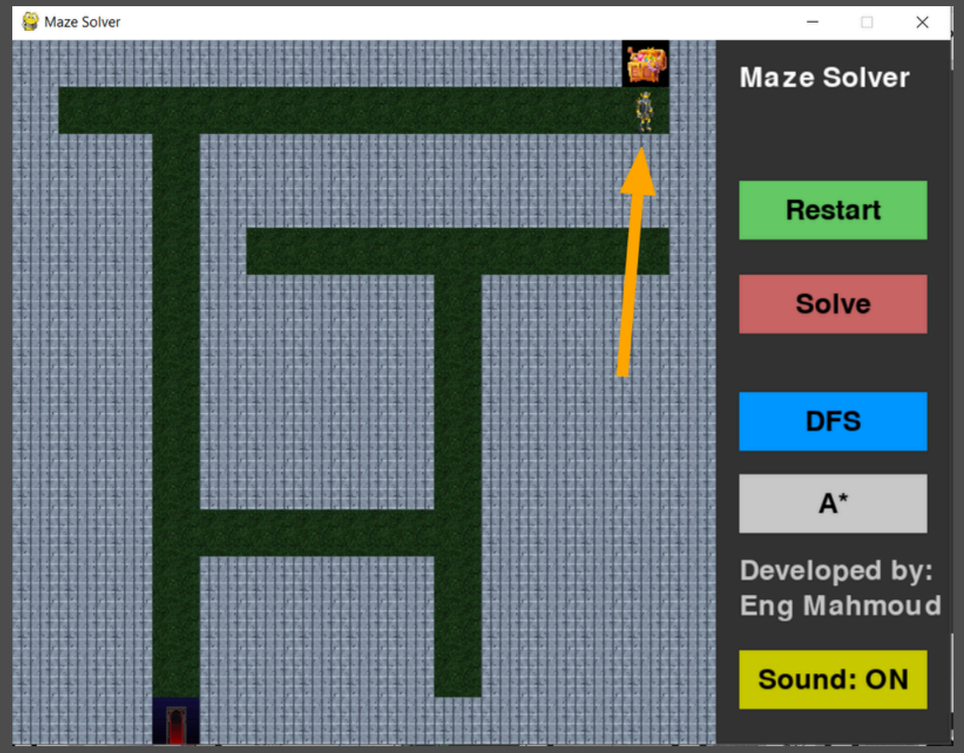


# Demo

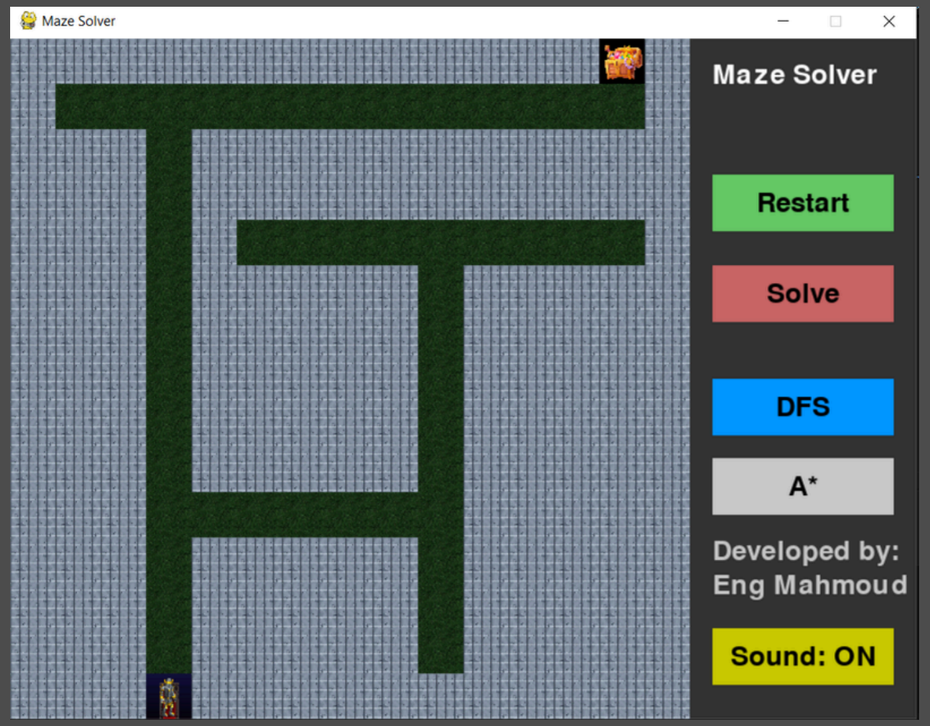
# Gameplay

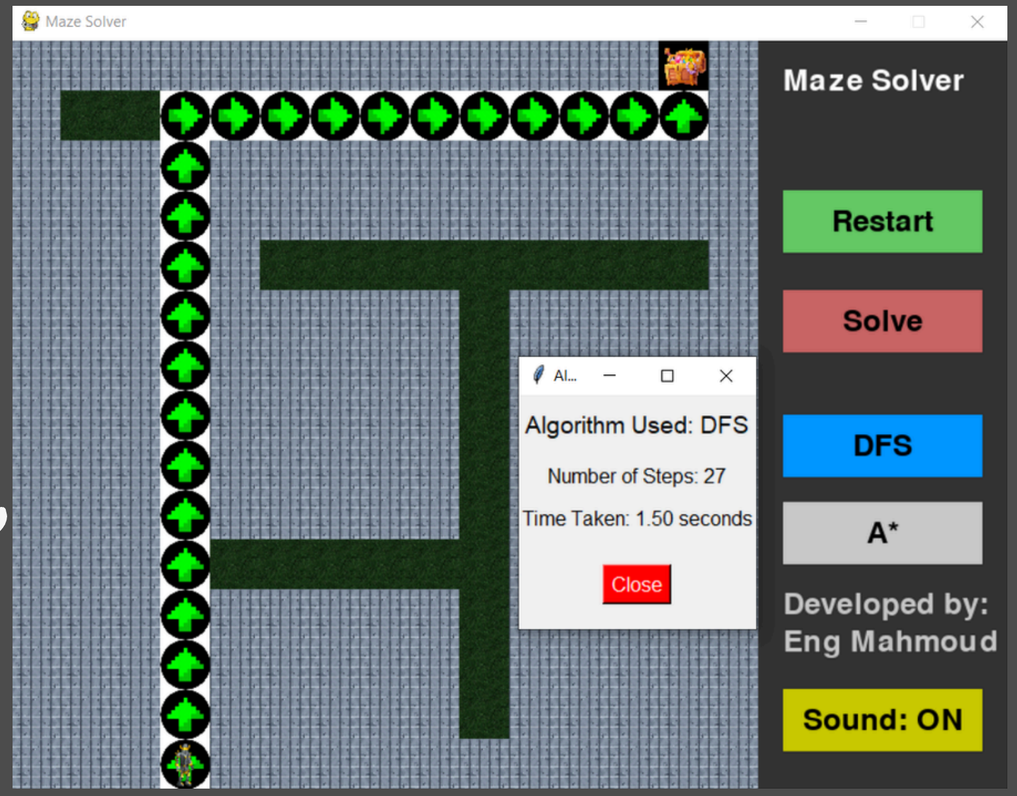
# Manual Play


Maze Solver
Maze Solver
Restart
Solve
DFS
A*
Developed by:
Eng Mahmoud
Sound: ON


Maze Solver
Maze Solver
Restart
Solve
DFS
A*
Developed by:
Eng Mahmoud
Sound: ON

# Automatic Play



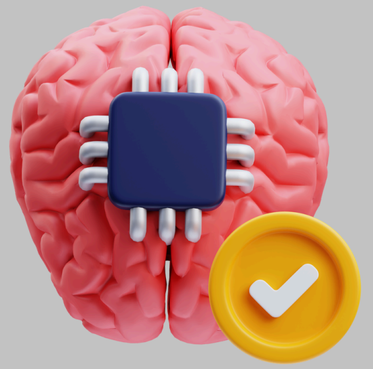



# Challenges and Solutions

- CHALLENGE 1: OPTIMIZING ALGORITHM PERFORMANCE.
  - SOLUTION: IMPLEMENTED A* FOR FASTER AND MORE EFFICIENT PATHFINDING.
- CHALLENGE 2: DEBUGGING MAZE LOGIC AND PLAYER MOVEMENT.
  - SOLUTION: STEP-BY-STEP TESTING OF MOVEMENT MECHANICS AND COLLISION HANDLING.
- CHALLENGE 3: INTEGRATING SOUND AND VISUALS SEAMLESSLY.
  - SOLUTION: USED PYGAME FOR GRAPHICS AND AUDIO SYNCHRONIZATION, ENSURING SMOOTH TRANSITIONS.

# Future Enhancements

- DYNAMIC MAZE ELEMENTS:
  - "INTRODUCE MOVING WALLS OR TIMED OBSTACLES TO INCREASE GAMEPLAY COMPLEXITY."
- IMPROVED VISUALS:
  - ENHANCE GRAPHICS WITH ANIMATIONS, BETTER TEXTURES, AND LIGHTING EFFECTS.
- AI OPPONENTS:
  - IMPLEMENT AI PLAYERS COMPETING TO SOLVE THE MAZE FOR A COMPETITIVE EXPERIENCE.
- MULTIPLAYER SUPPORT:
  - "ALLOW MULTIPLE PLAYERS TO NAVIGATE THE MAZE SIMULTANEOUSLY."
- ADDITIONAL FEATURES:
  - CUSTOMIZABLE MAZE SIZES.
  - SCORING SYSTEM BASED ON EFFICIENCY AND SPEED.



# Conclusion

THIS PROJECT SUCCESSFULLY INTEGRATES AI ALGORITHMS, INTERACTIVE GAMEPLAY, AND EFFICIENT DESIGN PRINCIPLES TO SOLVE MAZES. IT HIGHLIGHTS THE APPLICATION OF ENGINEERING AND PROBLEM-SOLVING TECHNIQUES IN PRACTICAL SCENARIOS.