PROJECT REPORT
ON
"Maze Game"

Submitted By:
Name: Rahul Kumar
UID: 24MCA20233

Under The Guidance of:

Ms. Deepali Saini

March 2025

University Institute of Computing

Chandigarh University,

Mohali, Punjab

TABLE OF CONTENTS

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to **Ms. Deepali Saini**, my project guide, for her invaluable guidance, constant encouragement, and expert supervision throughout this project. Her insightful suggestions, patience, and unwavering support were instrumental in the successful completion of this work. Her expertise and dedication greatly enriched my learning experience.
I am also sincerely thankful to my **friends and family** for their continuous motivation, constructive feedback, and emotional support, which kept me inspired during this journey.
Furthermore, I extend my heartfelt appreciation to **Chandigarh University** for providing me with the necessary resources, infrastructure, and opportunities to undertake this project. The knowledge and skills I have gained through this experience will undoubtedly benefit my academic and professional growth.

Date: 28.03.2025

Place: Chandigarh University, Mohali, Punjab

Rahul Kumar, UID-24MCA20233

ABSTRACT

This project presents the development of an interactive **Maze Game** implemented using **Python** with **Pygame** for graphical visualization. The game features dynamically generated mazes of varying difficulty levels (Easy, Medium, Hard), incorporating pathfinding algorithms such as **A*** and **Backtracking** for automated maze-solving.

Key features include:

- **Randomized maze generation** using a recursive backtracking algorithm
- **Player navigation** through keyboard controls with real-time movement
- **Auto-solve functionality** that demonstrates optimal paths using A* search
- **Time-based scoring system** and leaderboard to track player performance
- **Interactive GUI** with menu systems for game selection and difficulty adjustment

The implementation utilizes **NumPy** for efficient maze representation and **Pygame** for rendering the graphical interface. The project demonstrates practical applications of **search algorithms, data structures, and event-driven programming** in game development.

This work serves as an educational tool for understanding maze generation techniques, pathfinding algorithms, and Python-based game development while providing an engaging user experience. The modular design ensures extensibility for future enhancements such as additional algorithms or multiplayer functionality.

Keywords: Maze Game, A* Algorithm, Backtracking, Pygame, Pathfinding, Python Programming

Introduction

Maze games have long served as both entertainment and educational tools, demonstrating fundamental concepts in computer science such as pathfinding algorithms and procedural generation. This project implements an interactive maze game using Python, designed to provide users with an engaging experience while illustrating key computational methods.

The game features:

- **Procedural maze generation** using recursive backtracking to create unique, solvable labyrinths
- **Multiple difficulty levels** (Easy, Medium, Hard) with varying complexity
- **Two solution approaches**:
  - Manual player navigation using keyboard controls
  - Automatic pathfinding via A* search algorithm
- **Performance tracking** through time-based scoring and leaderboard functionality

Technical Implementation

Built using Pygame for visualization and NumPy for efficient grid manipulation, the project demonstrates:

1. **Maze Generation**: The recursive backtracker algorithm creates perfect mazes (with exactly one solution) by systematically carving passages while maintaining connectivity
2. **Pathfinding**: The A* algorithm efficiently finds optimal paths using heuristic-based search
3. **User Interface**: An intuitive GUI system handles game states, player input, and visual feedback

Educational Value

This implementation serves as a practical demonstration of:

- Algorithm analysis (comparing backtracking and A* approaches)
- Event-driven programming paradigms
- Software design patterns in game development

The project's modular architecture allows for future expansion, including additional algorithms (Dijkstra's, BFS), multiplayer modes, or advanced visualization techniques. By combining theoretical concepts with interactive gameplay, it provides a compelling platform for both entertainment and learning.
The following sections detail the system design, implementation specifics, and results of this maze game development project.

OBJECTIVES

The primary objectives of this maze game development project are:

1. Core Game Development Objectives

- To design and implement an **interactive maze game** with dynamic generatio and solving capabilities
- To develop **multiple difficulty levels** (Easy, Medium, Hard) with varyin maze complexity
- To create **real-time player navigation** using keyboard controls
- To implement a **scoring system** based on completion time

2. Algorithmic Objectives

- To demonstrate **maze generation** using recursive backtracking algorithm
- To implement **pathfinding solutions** using:
  - *A algorithm** (for optimal pathfinding with heuristics)
  - **Backtracking** (for systematic brute-force search)
- To compare algorithm efficiency in terms of **path optimality and speed**

3. User Experience & Interface Objectives

- To build an **intuitive GUI** using Pygame for smooth gameplay
- To provide **auto-solve functionality** for visual demonstration of algorithms
- To include a **leaderboard system** to track and display top player scores

4. Educational & Technical Objectives

- To serve as a **learning tool** for pathfinding algorithms and procedura generation
- To ensure **modular and scalable code** for future enhancements
- To document the implementation process for academic reference

5. Future Extensibility Goals

- To allow integration of **additional algorithms** (Dijkstra’s, BFS, etc.)
- To support **multiplayer or AI competition modes**
- To enable **custom maze designs** via user input

By fulfilling these objectives, the project aims to deliver an **engaging, educational, and technically robust** maze game while demonstrating key concepts in **game development and algorithm design**.

METHODOLOGY & SYSTEM DESIGN

This section details the technical approach, algorithms, and system architecture used to develop the maze game.

System Architecture

The game follows a modular design with these core components:

1. Game Engine (Pygame)
   - Handles graphics rendering, user input, and real-time updates
   - Manages game states (Menu, Playing, Leaderboard, Game Over)
2. Maze Generator
   - Uses Recursive Backtracking to create random solvable mazes
   - Stores maze as a 2D NumPy array (0 = path, 1 = wall)
3. Pathfinding Module
   - Implements *A Algorithm** (optimal pathfinding)
   - Includes Backtracking Solver (alternative approach)
4. User Interface System
   - Renders menus, player position, timer, and solution path
   - Handles keyboard inputs for navigation

Key Algorithms

A. Maze Generation (Recursive Backtracking)

1. Initialize grid with all walls (1s)
2. Start at (1,1) and mark as path (0)
3. While unvisited cells exist:
   - Randomly select a neighboring wall (2 cells away)
   - Carve a path by removing the wall between current and neighbor
   - Push current cell to stack and move to neighbor
   - If no neighbors, backtrack using stack

Output: Perfect maze (exactly one solution).

B. A Pathfinding*

1. Use priority queue (min-heap) for open nodes
2. Calculate:
   - g(n) = Cost from start to current node
   - h(n) = Manhattan distance to end (heuristic)
   - f(n) = g(n) + h(n) (total cost)
3. Expand lowest f(n) nodes first until reaching the end

Advantage: Guarantees shortest path.

C. Backtracking Solver

1. Move depth-first in all directions (Up/Down/Left/Right)
2. Mark visited cells to avoid loops
3. If dead end, backtrack to last valid path

Use Case: Demonstrates exhaustive search.

Flowcharts

START → Initialize Maze → Player Movement/Auto-Solve → Check Win Condition → Update Leaderboard → EXIT

A Algorithm*

START → Add start node to open list → Select node with lowest f(n) → Expand neighbors → Update costs → Reach end? → Return path

Data Structures

Component	Data Structure	Purpose
Maze Grid	2D NumPy Array	Store walls/paths (0s and 1s)
A* Open Nodes	Priority Queue	Select next node efficiently
Backtracking Path	Stack	Track visited cells for backtracking
Leaderboard	List (sorted)	Store top 5 completion times

Tools & Libraries

- Python 3 (Base language)
- Pygame (Rendering, event handling)
- NumPy (Efficient grid operations)
- Heapq (Priority queue for A*)

Implementation

Main.py:

```
import pygame
import numpy as np
import time
import os
from maze import generate_maze
from pathfinding import astar
# Constants
CELL_SIZE = 20  # size of each cell in the grid
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
BLUE = (0, 0, 255)
GREEN = (0, 255, 0)
PLAYER_COLOR = (255, 0, 0)
TEXT_COLOR = (0, 0, 255)
AUTO_SOLVE_COLOR = (255, 165, 0)
LEADERBOARD_FILE = "scores.txt"
# Initialize Pygame
pygame.init()
```

```
screen = None  # Will be initialized in the main loop
def set_screen_size(rows, cols):
    global screen
    screen = pygame.display.set_mode((cols * CELL_SIZE, rows * CELL_SIZE))
# Generate Maze
def generate_game(level):
    if level == "Easy":
        size = 25
    elif level == "Medium":
        size = 25
    else:
        size = 30
    return generate_maze(size, size), (1, 1), (size-2, size-2)
# Leaderboard Functions
def save_score(time_taken):
    with open(LEADERBOARD_FILE, "a") as file:
        file.write(f"{time_taken:.2f}\n")

def load_leaderboard():
    if not os.path.exists(LEADERBOARD_FILE):
        return []
    with open(LEADERBOARD_FILE, "r") as file:
        scores = [float(line.strip()) for line in file.readlines()]
    return sorted(scores)[:5]  # Show top 5 scores

# Draw Functions
def draw_text(text, x, y, color=TEXT_COLOR):
    font = pygame.font.Font(None, 36)
    text_surface = font.render(text, True, color)
    screen.blit(text_surface, (x, y))

def draw_maze(maze, start, end, player_pos, auto_solve, solution_path, is_winning):
    rows, cols = maze.shape
    screen.fill(WHITE)

    # Draw each cell in the maze
    for x in range(rows):
        for y in range(cols):
            color = BLACK if maze[x, y] == 1 else WHITE
              pygame.draw.rect(screen, color, (y * CELL_SIZE, x * CELL_SIZE, CELL_SIZE,
CELL_SIZE))

    # Draw start point (blue) and end point (green)
    pygame.draw.rect(screen, BLUE, (start[1] * CELL_SIZE, start[0] * CELL_SIZE,
CELL_SIZE, CELL_SIZE))
  # Draw the exit (green) and keep it green if the player wins
  if is_winning:
        pygame.draw.rect(screen, GREEN, (end[1] * CELL_SIZE, end[0] * CELL_SIZE,
CELL_SIZE, CELL_SIZE))
  else:
        pygame.draw.rect(screen, GREEN, (end[1] * CELL_SIZE, end[0] * CELL_SIZE,
CELL_SIZE, CELL_SIZE))

  # Draw the auto-solve path if enabled
  if auto_solve and solution_path:
    for (px, py) in solution_path:
            pygame.draw.rect(screen, AUTO_SOLVE_COLOR, (py * CELL_SIZE, px *
CELL_SIZE, CELL_SIZE, CELL_SIZE))

  # Draw the player
  pygame.draw.rect(screen, PLAYER_COLOR, (player_pos[1] * CELL_SIZE, player_pos[0]
* CELL_SIZE, CELL_SIZE, CELL_SIZE))

def show_leaderboard():
  screen.fill(WHITE)
  draw_text("Leaderboard (Top Scores)", 150, 50)
  scores = load_leaderboard()
  if scores:
    for i, score in enumerate(scores, start=1):
      draw_text(f"{i}. {score} sec", 150, 100 + (i * 40))
  else:
    draw_text("No scores yet!", 150, 150)
  draw_text("Press any key to return", 150, 500)
  pygame.display.update()
  wait_for_key()

def wait_for_key():
  while True:
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        pygame.quit()
        exit()
      if event.type == pygame.KEYDOWN:
        return
def game_over_menu(time_taken):
  save_score(time_taken)  # Store the player's time
  while True:
    screen.fill(WHITE)
    draw_text(f"You Won! Time: {time_taken:.2f} sec", 100, 100)
    draw_text("1. Restart", 100, 200)
    draw_text("2. Back to Menu", 100, 250)
    draw_text("3. Quit", 100, 300)
    pygame.display.update()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
        pygame.quit()
        exit()
        if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_1:
            return "restart"
        elif event.key == pygame.K_2:
            return "menu"
        elif event.key == pygame.K_3:
            pygame.quit()
            exit()

def game_loop(level):
  maze, start, end = generate_game(level)
  player_x, player_y = start
  auto_solve = False
  solution_path = []
  is_winning = False
  start_time = time.time()
  rows, cols = maze.shape
  set_screen_size(rows, cols)  # Set screen size dynamically
  running = True
  while running:
    draw_maze(maze, start, end, (player_x, player_y), auto_solve, solution_path, is_winning
    draw_text(f"Time: {time.time() - start_time:.2f} sec", 10, 10)
    pygame.display.update()
    # Auto-solve mode
    if auto_solve and solution_path:
        for (px, py) in solution_path:
        player_x, player_y = px, py
              draw_maze(maze,  start,  end,  (player_x,  player_y),  auto_solve,  solution_path
is_winning)
        pygame.display.update()
        pygame.time.delay(100)
      total_time = time.time() - start_time
      return game_over_menu(total_time)
    # Player input handling
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
        pygame.quit()
        exit()
        if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_r:
            return "restart"
        elif event.key == pygame.K_s:
            auto_solve = True
            solution_path = astar(maze, start, end)
        else:
            new_x, new_y = player_x, player_y
            if event.key == pygame.K_UP:
                new_x -= 1
            elif event.key == pygame.K_DOWN:
                new_x += 1
            elif event.key == pygame.K_LEFT:
                new_y -= 1
            elif event.key == pygame.K_RIGHT:
                new_y += 1
            if 0 <= new_x < maze.shape[0] and 0 <= new_y < maze.shape[1] and maze[new_x,
new_y] == 0:
                player_x, player_y = new_x, new_y
            if (player_x, player_y) == end:
                is_winning = True
                total_time = time.time() - start_time
                return game_over_menu(total_time)
def choose_level():
  while True:
    screen.fill(WHITE)
    draw_text("Choose Difficulty Level:", 150, 100)
    draw_text("1. Easy", 150, 200)
    draw_text("2. Medium", 150, 250)
    draw_text("3. Hard", 150, 300)
    pygame.display.update()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
        pygame.quit()
        exit()
        if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_1:
            return "Easy"
        elif event.key == pygame.K_2:
            return "Medium"
        elif event.key == pygame.K_3:
            return "Hard"

def main_menu():
  set_screen_size(30 ,30)  # Set default size for the menu
  while True:
    screen.fill(WHITE)
    draw_text("Maze Game", 200, 100)
    draw_text("1. Start Game", 200, 200)
    draw_text("2. Leaderboard", 200, 250)
    draw_text("3. Quit", 200, 300)
    pygame.display.update()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
        pygame.quit()
        exit()
        if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_1:
            return "start"
        elif event.key == pygame.K_2:
            return "leaderboard"
        elif event.key == pygame.K_3:
            pygame.quit()
            exit()

# Main Loop
while True:
  menu_choice = main_menu()
  if menu_choice == "leaderboard":
    show_leaderboard()
  elif menu_choice == "start":
    level = choose_level()
    while True:
        result = game_loop(level)
        if result == "menu":
        break
        elif result == "restart":
        break
```

pathfinding.py:

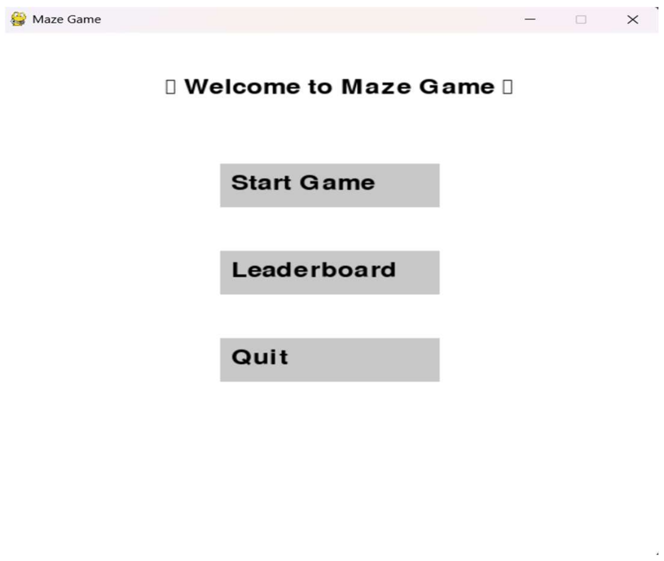
```
import heapq

# A* Algorithm
def astar(maze, start, end):
    rows, cols = maze.shape
    open_set = [(0, start)]
    came_from = {}
    g_score = {start: 0}
    f_score = {start: abs(start[0] - end[0]) + abs(start[1] - end[1])}

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == end:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            return path[::-1]

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and maze[neighbor] == 0:
                tentative_g_score = g_score[current] + 1
                if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                   f_score[neighbor] = tentative_g_score + abs(neighbor[0] - end[0]) + abs(neighbor[1]
- end[1])
                    heapq.heappush(open_set, (f_score[neighbor], neighbor))

    return None

# Backtracking Algorithm
def backtracking_solve(maze, start, end):
    ROWS, COLS = maze.shape
    visited = set()
    path = []

    def backtrack(x, y):
        if (x, y) in visited or maze[x, y] == 1:
            return False
        visited.add((x, y))
        path.append((x, y))

        if (x, y) == end:
            return True
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        for dx, dy in directions:
            if backtrack(x + dx, y + dy):
                return True

        path.pop()
        return False

    backtrack(*start)
    return path
```

maze.py:

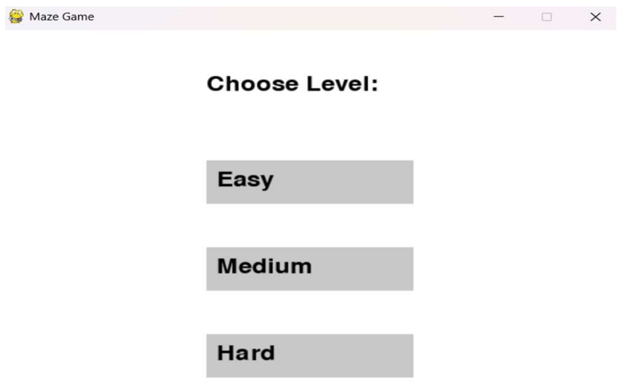
```
import numpy as np
import random
def generate_maze(rows, cols):
    # Create a maze grid initialized to walls (1)
    maze = np.ones((rows, cols), dtype=int)
    stack = [(1, 1)]
    maze[1, 1] = 0  # Starting point

    # Function to get neighbors to explore
    def neighbors(x, y):
        dirs = [(0, 2), (2, 0), (0, -2), (-2, 0)]  # Directions to explore
        random.shuffle(dirs)
        return [(x + dx, y + dy, x + dx // 2, y + dy // 2) for dx, dy in dirs]
    # Maze generation using backtracking algorithm
    while stack:
        x, y = stack[-1]
        valid_neighbors = [(nx, ny, mx, my) for nx, ny, mx, my in neighbors(x, y)
                           if 0 <= nx < rows and 0 <= ny < cols and maze[nx, ny] == 1]
        if valid_neighbors:
            nx, ny, mx, my = valid_neighbors.pop()
            maze[mx, my] = 0  # Remove wall
            maze[nx, ny] = 0   # Remove wall
            stack.append((nx, ny))
        else:
            stack.pop()  # Backtrack if no valid neighbors
    # Ensure the exit point is always reachable
    maze[rows - 2, cols - 2] = 0  # Exit point
    # Check if the exit is blocked and un-block if necessary
    if maze[rows - 3, cols - 2] == 1 and maze[rows - 2, cols - 3] == 1:
        maze[rows - 3, cols - 2] = 0  # Create a path if blocked

    return maze
```

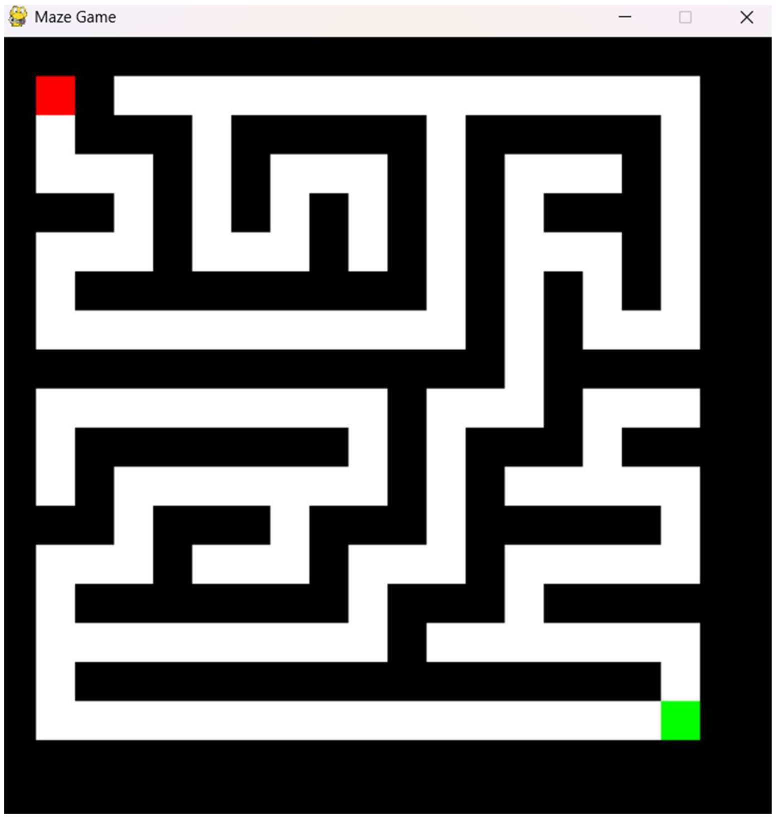
RESULTS & SCREENSHOTS

After Runing the main.py

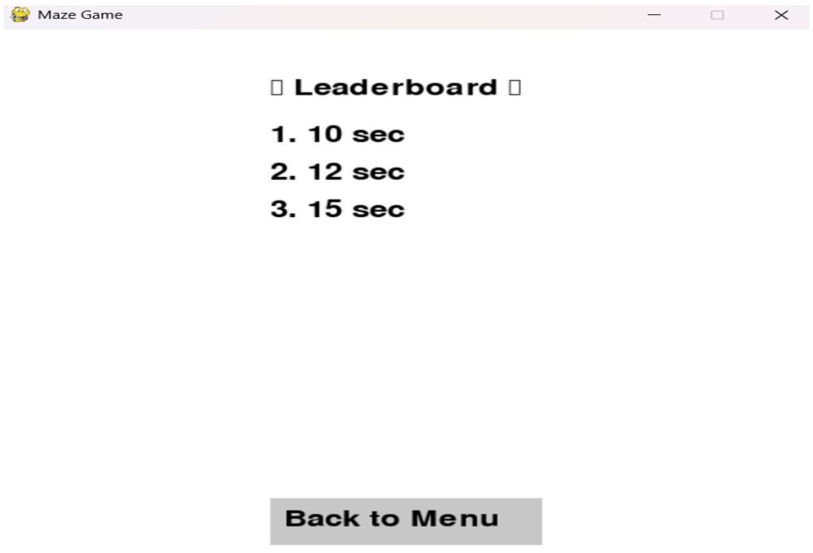


After Click on Start Game





After Click on Leaderboard



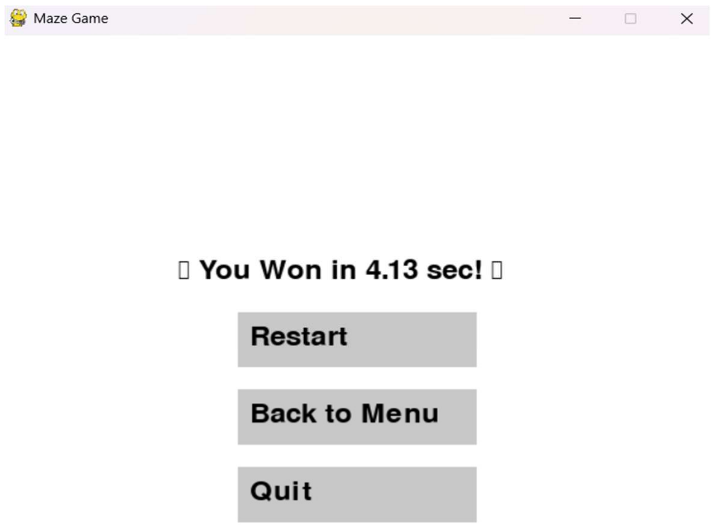
 Leaderboard

1. 10 sec

2. 12 sec

3. 15 sec

After Winning the Game



▯ You Won in 4.13 sec! ▯

FUTURE SCOPE:

To enhance the game further, the following extensions are proposed:
1. Algorithm Expansion
Add Dijkstra's algorithm and Breadth-First Search (BFS) for comparative analysis.
Implement machine learning (e.g., Q-learning) to train an AI solver.
2. Gameplay Features
Multiplayer Mode: Race against other players or AI.
Custom Mazes: Allow users to design their own mazes via a grid editor.
Dynamic Difficulty: Adjust maze complexity in real-time based on player skill.
3. Technical Improvements
3D Visualization: Use OpenGL or Unity for a first-person maze experience.
Mobile Port: Develop an Android/iOS version using Kivy or Pygame Subset for Mobile.
Cloud Leaderboards: Store scores online using Firebase or AWS.
4. Educational Tools
Algorithm Tutorials: Interactive step-by-step explanations of A* and backtracking.
Debug Mode: Visualize algorithm exploration in real-time (e.g., open/closed sets in A*).
5. Performance Optimization
Parallel Processing: Speed up maze generation using multithreading.
GPU Acceleration: Implement pathfinding with CUDA for very large mazes

(>100x100).

CONCLUSION:

This project successfully developed an interactive maze game that demonstrates core computer science concepts through engaging gameplay.
Key achievements include:
Effective Maze Generation: Implemented a recursive backtracking algorithm to create solvable mazes of varying difficulty (Easy, Medium, Hard).
Optimal Pathfinding: Integrated *A search** for efficient shortest-path calculations, outperforming backtracking in speed and accuracy.
User-Friendly Design: Built an intuitive GUI with Pygame, featuring real-time controls, auto-solve visualization, and a leaderboard system.
Educational Value: Served as a practical tool for understanding algorithmic efficiency (A* vs. backtracking) and procedural generation.
The project met all objectives, delivering a functional, scalable, and educational maze game while highlighting the trade-offs between different algorithmic approaches.

GitHub Link: https://github.com/rahulkumarpass/DAA_Project