PUZZLE GAME: YR 12 COMPSCI PROJECT

IMRAN ANISHA

This document includes the following:

1. Project Proposal & Requirements Specification
2. Detailed Project Design
3. Initial Prototype
4. Full Game Prototype
5. Final Documentation

GitHub Link:

https://github.com/IMRAN-Anisha/CMSCI-2025

Instructions on how to play are in the README file of the GitHub

System Requirements

Before running the game, ensure you have the following installed:

1. **Python (3.x) – Download and install from python.org**
2. **Pygame – Install it via pip (run in terminal) pip install pygame**
3. **IInstall Visual Studio Code – Download from code.visualstudio.com**
4. **Pygame GUI (Optional) – (run in terminal) pip install pygame_gui**

Ethical Use of AI

This project documentation incorporates a range of resources, including online technical forums and artificial intelligence models. Online platforms such as Reddit and Stack Overflow were consulted to identify established solutions to encountered programming challenges. Furthermore, artificial intelligence tools, specifically ChatGPT and GROK, were utilized for the following purposes:

- To refine textual documentation for enhanced conciseness and clarity.
- To facilitate code refactoring, aiming for improved efficiency and adherence to optimal programming practices, such as the resolution of circular dependencies and the elimination of redundant code segments.
- To structure project information, including error logs and revised code implementations, into a clear and presentable tabular format.

Additionally, this project involved the development of digital adaptations of established puzzle games. Modified versions of Wordle, originally a web-based game created by Josh Wardle, were implemented with word lengths of three, four, and five letters, deviating from the original five-letter format. A digital version of Sudoku, a logic-based number-placement puzzle attributed to Howard Garns, was also created as a subgame.

Project Proposal & Requirements Specification

Game Option & Justification

The proposed game is a dynamic puzzle game that evolves based on the player's skill level. Unlike traditional puzzle games with fixed levels, this game adapts to player performance, ensuring a continuously engaging challenge. The core mechanics blend procedural puzzle generation with AI-driven difficulty adjustment, making every playthrough unique, inspired by the game Detroit: Become Human where players make their own decisions, leading to different outcomes and endings.

Justification (Syllabus Points)

- **Enhances logical thinking and adaptive problem-solving skills.**
- **Uses Object-Oriented Programming (OOP) for modular, scalable puzzle mechanics.**
- **Introduces AI-driven difficulty scaling, ensuring varied gameplay and replayability.**
- **Easy implementation of inheritance & encapsulation.**

Requirements Specification

Functional Requirements

- **The game must generate procedural puzzles based on difficulty.**
- **Players can interact with puzzles via drag-and-drop, number inputs, or pattern-matching.**
- **The system should adjust difficulty in real-time based on player performance.**
- **A timer will track progress and influence puzzle difficulty dynamically.**
- **Hint system to provide assistance without making puzzles too easy.**

Non-Functional Requirements

- **The game must run smoothly on desktop and mobile devices.**
- **Save/load system to track player progress.**
- **Clean, user-friendly UI with an intuitive design.**
- **Low memory and processing requirements for accessibility across devices.**

Problem Definition

Traditional puzzle games rely on static, predefined levels, which often lead to repetitive gameplay and lack of personalized difficulty. This project aims to create a self-adjusting puzzle game, ensuring an evolving challenge tailored to the player's abilities.

Solution:

A procedural puzzle is dynamically generated by an algorithm rather than manually designed, ensuring variety, difficulty scaling, and replayability.

Inspiration & Implementation

- **Maze Generation (DFS): Uses recursive backtracking to create unique mazes with a single solution, increasing complexity with obstacles or multiple exits.**
- **Logic-Based Puzzles: Generates rule-based challenges like Sudoku, adjusting constraints dynamically for solvability.**
- **Adaptive Difficulty: AI tracks player performance, scaling puzzle complexity by modifying constraints, hints, and logical depth.**

Target Audience

- **Gamers and Students looking for an engaging mental workout., improving problem solving skills in a fun manner.**
- **Developers interested in AI-driven puzzle generation and OOP-based game design.**

Objectives

- **Implement procedural puzzle generation to ensure no two puzzles are the same.**
- **Use AI-based adaptive difficulty, tracking user performance dynamically.**
- **Maintain OOP principles for scalability, flexibility, and reusable puzzle mechanics.**
- **Ensure seamless user experience, integrating smooth animations and UI transitions.**

Basic Features

- **AI-Driven Adaptive Difficulty – The game dynamically adjusts puzzle complexity.**
- **Multiple Puzzle Types – Logical, spatial, and numerical puzzles in one game.**
- **Score System – Performance tracking with ranking-based difficulty adjustments.**
- **Dynamic Hint System – AI-based hints that prevent frustration.**
- **Procedural Puzzle Generator – Ensures endless unique puzzles, prevents players from trying to memorise previous games and encourages them to apply skills.**

Game Progression

- **Tutorial Stage – Introduces game mechanics with simple puzzles and an opening screen.**
- **Core Gameplay – Player engages with procedurally generated puzzles.**
- **AI Adaptation – The game increases/decreases difficulty based on player performance.**
- **Challenge Mode – Leaderboards for competitive play.**

DFS: (Depth First Search)

DFS is an algorithm used to explore or traverse structures like mazes, trees, or graphs. It starts at a point, moves as deep as possible along one path, then backtracks when needed. In puzzles, DFS helps generate mazes by carving paths and ensuring a solvable layout. This is integrated into the AIAdaptiveSystem for:

- **Procedural Puzzle Generation – Ensures puzzles grow in complexity over time.**
- **Hint System – AI uses DFS to search for possible solutions and suggest hints dynamically.**
- **Path-Based Puzzles – DFS is ideal for maze-like or logic-based puzzles requiring deep exploration.**

Class Diagram

Abstract class:

Puzzle (Base)
difficulty = int is_solved = boolean
getDifficulty(): isSolved(): solve():

AI component:

AI Adaptive System
performanceData = dictionary currentDifficulty = INT
trackPerformance(): adjustDifficulty(): generatePuzzleDFS():

LogicPuzzle
rules = list
checkLogic():

SpatialPuzzle
gridStructure = List
checkPattern():

NumberPuzzle
numberGrid : List
checkSolution():

Stores Player Data:

Player
name = string score = int history = list
updateScore(): getHistory():

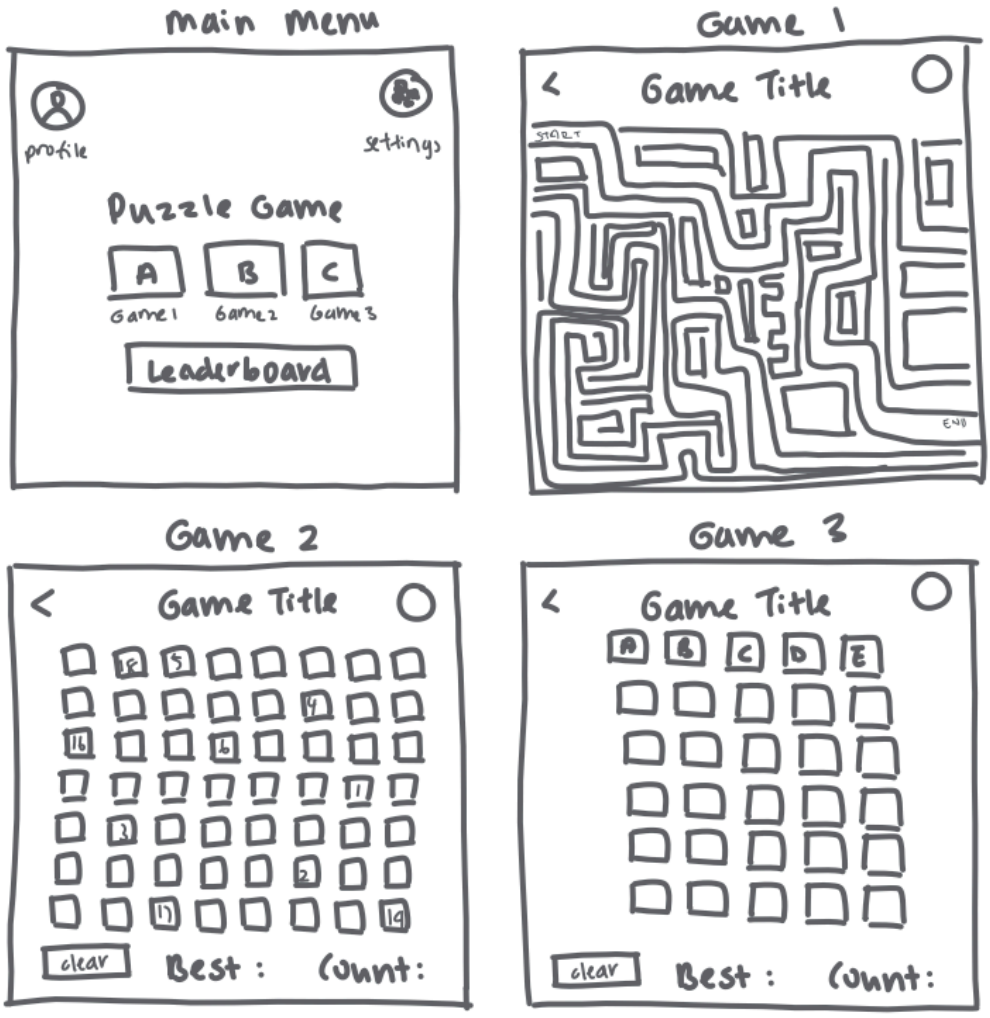
Handles Graphics & Animation:

UI manager
currentScreen = str animationsEnabled = boolean
renderUI(): updateScreen():

Class Breakdown

- **Puzzle (Base) [Abstract Class] – Defines common attributes (e.g., difficulty, solved state).**
- **LogicPuzzle, SpatialPuzzle, NumberPuzzle [Subclasses] – Each implements unique rules and solving mechanisms.**
- **AIAdaptiveSystem [AI Component] – Monitors player progress and dynamically adjusts difficulty.**
- **DFS Integration in AIAdaptiveSystem – Generates puzzles dynamically based on the player's previous performance to scale difficulty.**
- **Player [Stores User Data] – Tracks score, history, and gameplay preferences.**
- **UIManager [Handles UI Elements] – Ensures seamless visual transitions, animations, and player interaction.**

UI Diagrams



(subject to change)

Detailed Project Details

1. Review Your Initial Proposal

Game Option & Justification:

- My initial project focused on a broader game concept with multiple subgames, but after further refinement, I have chosen to focus more on the DFS-based puzzle subgame.
- Why DFS? DFS is a powerful algorithm for solving puzzles and generating maze-like structures. It aligns well with Object-Oriented Programming (OOP) principles because:
    - It enables procedural puzzle generation where levels get progressively more complex.
    - It supports an AI-driven hint system that can dynamically suggest possible paths.
    - It is ideal for path-based puzzles, which require deep search logic for solving and backtracking.
- This shift allows me to fully employ OOP techniques like abstraction, encapsulation, inheritance, and polymorphism, ensuring a well-structured and modular game design.

Class Diagram Validity & Refinement:

- The initial class diagram outlined the fundamental structure of the game, but with the refined focus on DFS, some adjustments are necessary.
- What remains valid?
    - Core classes like GameEngine, Player, and Puzzle still serve their original purpose.
    - The AIAdaptiveSystem will still be used, but its role will now emphasize DFS-based puzzle generation and hint systems.
- What needs refinement?
    - Introduce a DFSAlgorithm class, responsible for traversal, backtracking, and solution generation.
    - The Puzzle class needs modifications to accommodate maze-like structures and path-based logic.
    - The interaction between AIAdaptiveSystem and DFSAlgorithm must be explicitly defined in the class diagram.

Reflection on Initial Design:

What worked well?

- **The modular OOP approach ensured flexibility for extending the game mechanics.**
- **The AIAdaptiveSystem concept was a strong addition that makes the game adaptive and engaging.**
- **The initial class structure provided a solid foundation for game mechanics and interactions.**

What needs more detail or clarification?

- **DFS Implementation Details – The role of DFS within the puzzle generation process needs to be explicitly defined.**
- **AI-Driven Hints – How the AI will analyze paths using DFS to provide hints requires additional consideration.**
- **Edge Cases & Error Handling – Need to define how the game will handle unsolvable puzzles, dead ends, or player errors.**

Next Steps:

- **Update the class diagram to integrate DFSAlgorithm and its role in the system.**
- **Define the interaction between AIAdaptiveSystem, DFSAlgorithm, and Puzzle.**
- **Ensure flow charts reflect the DFS traversal logic and how puzzles are generated and solved.**
- **Strengthen the error handling mechanisms for edge cases like infinite loops or invalid paths.**

This refined approach ensures that DFS serves as the core gameplay mechanic, enhancing puzzle generation and AI-driven hints while maintaining strong OOP principles.

---

2. Detailed Pseudocode Components:

1. **DFSAlgorithm:**

Purpose: It handles the logic of generating a maze and finding a solution using a Depth First Search (DFS) algorithm. This class has methods for generating puzzles and finding the solution path.

Key Methods:

__init__(self, grid_size): Initializes the grid, visited set, and stack for DFS traversal. Also generates the puzzle.

generate_puzzle(): Creates the maze using DFS. It pushes random positions to the stack and explores neighbors while ensuring walls are randomly removed.

find_solution(start, goal): Finds the path from the start position to the goal using DFS and returns the solution path.

reconstruct_path(path, start, goal): Reconstructs the solution path by backtracking from the goal to the start position.

2. AIAdaptiveSystem:

Purpose: This system adapts the puzzle's difficulty level based on the player's performance and provides hints.

Key Methods:

__init__(self, grid_size): Initializes an instance of DFSAlgorithm and sets default difficulty and hint counter values.

provide_hint(player_position, goal): Provides the next step in the solution path to the player, based on their current position and goal.

adjust_difficulty(player_performance): Adjusts the puzzle's difficulty based on the player's performance (e.g., whether they solve puzzles quickly or struggle).

3. Error Handling:

Error handling is added at various critical points in the game to ensure that invalid inputs (e.g., out-of-bounds positions) or states (e.g., an empty grid or invalid performance data) do not crash the game. This makes the program more robust and user-friendly. The return statements are incredibly useful for debugging as it lets you know exactly where the problem lies.

MILESTONE 2 PSEUDOCODE

```
# DFSAlgorithm (make puzzles & solutions)
START
CLASS DFSAlgorithm:
   VAR PRIVATE grid    #'2D list array
   VAR PRIVATE visited
   VAR PRIVATE stack     # DFS traversal

   METHOD _init_(self, grid_size):
      SET self.grid TO grid
      SET self. visited TO empty set
      SET self.stack TO empty list

  METHOD generate_puzzle():
    SELECT random start_position
     PUSH start_position to stack
     WHILE stack NOT empty:
          current_position = POP from stack
          IF current_position NOT visited:
             current_position =visited
               # Get valid neighbours that haven't been visited
              VAR valid_neighbour = GET_unvisited_neighbour(
               current_position) #ran out of space to fit 1 line
               SHUFFLE valid_neighours # randomness

              FOR each neighbour in valid_neighbours:
                  REMOVE.WALL (current position, neighbour)
                  PUSH neighbour to stack
              END FOR
         END IF
     END WHILE
     RETURN grid
```

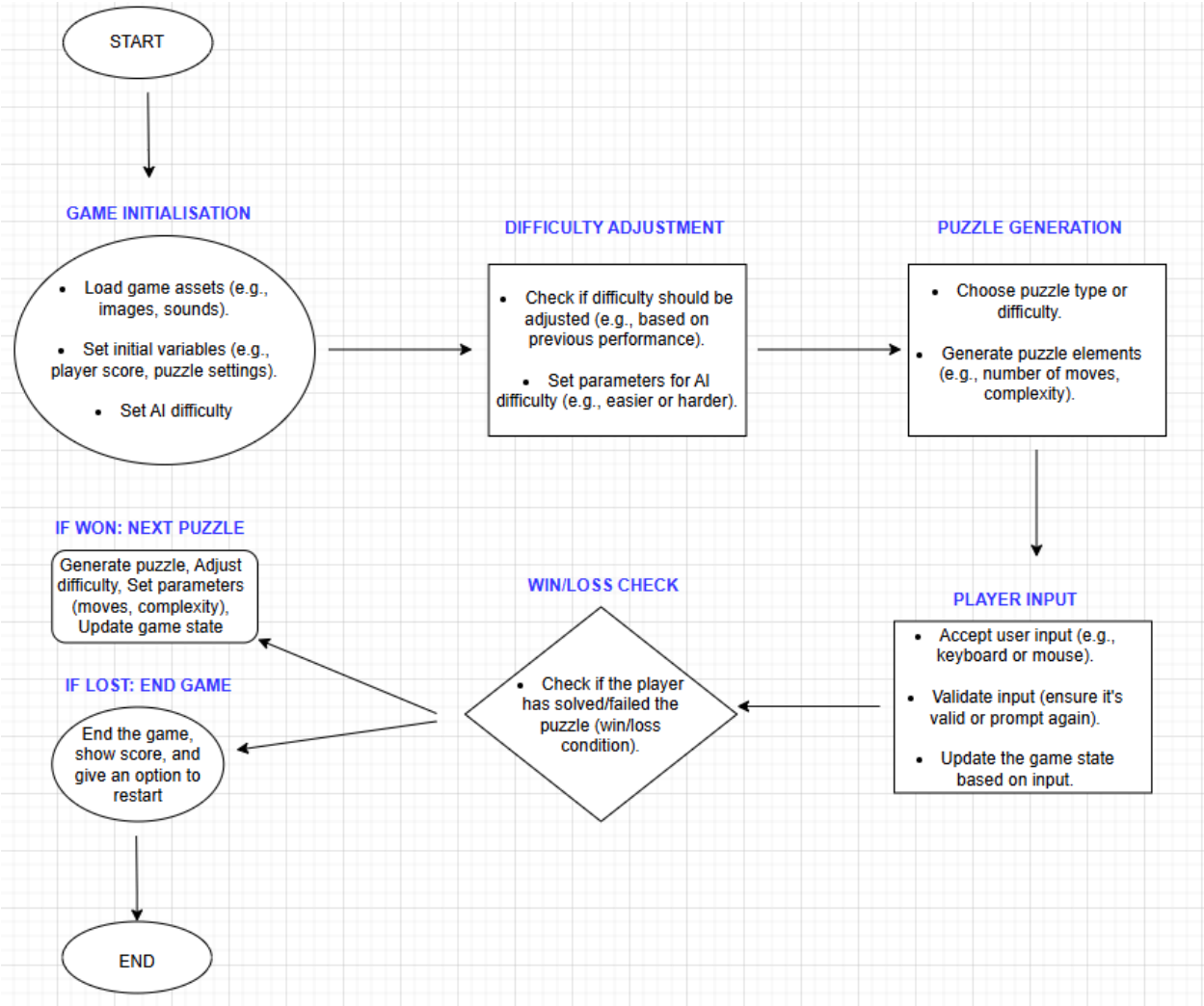
```
METHOD find_solution (start, goal):
    SET stack TO list # list containing start
    SET path TO dictionary #empty dictionary

   WHILE stack NOT empty:
       SET current_position TO POP from stack

       IF current_position == goal:
          RETURN CALL reconstruct path (path, start, goal)
       END IF

       FOR each neighbour IN valid_neighbours (current_position):
          IF neighbour != visited:
             SET current_position AS parent of neighbour IN path
              PUSH neighbour to stack
          END IF
      END FOR
  END WHILE
  RETURN "NO solution found"

METHOD reconstruct_path (path, start, goal):
    VAR solution_path = []
    VAR current = goal

    WHILE current != start:
        APPEND current TO solution_path
        SET current TO path [current]
    END WHILE
   RETUAN solution_path
```

```
# AI Adaptive System (hint system & dynamic difficulty Adjust)
CLASS AIAdaptiveSystem:
    PRIVATE dfs_algorithm  # instance of DFSAlgorithm
    PRIVATE difficulty_level
    PRIVATE hint_counter  # to apply limits

    METHOD _init_ (self, grid_size):
        INITIALISE dfs_algorithm with grid_size
        difficulty_level = default
        hint_counter = 0

    METHOD provide_hint (player_position, goal):
        CALL dfs_algorithm.find_solution (player_position, goal) AS
          solution path  # not enough space to fit 1 line
        IF solution_path NOT empty:
            RETURN next_step in solution_path
        ELSE:
            "no hint avaiable."
        END IF

    VAR slow = time > 500    # tracking performance of
    VAR fast = time < 500    # player by speed

    METHOD adjust_difficulty (player_performance):
        IF player solves puzzle fast:
            INCREASE difficulty_level
        ELSE IF player solves puzzle slow:
            DECREASE difficulty_level
        END IF
        RETURN difficulty_level
```

```
# Error Handling within code above
 METHOD find_solution (start, goal):
   IF start or goal OUTSIDE grid_boundaries:
      RETURN "invalid goal or start position"
   ELSE IF grid == EMPTY:
      RETURN "Grid is empty"
   END IF
   ... # continue code for find_solution

 METHOD provide_hint (player_position, goal)
    IF player_position is OUTSIDE grid-bounds:
      RETURN "Invalid player position"
    # snippet from code showing error handling
   CALL dfs_algorithm.find_solution (player_position, goal) AS
     solution path # not enough space to fit 1 line
   IF solution_path NOT empty:
      RETURN next_step in solution_path
   ELSE:
      "no hint available."
    ... # continue code for find_solution

METHOD adjust_difficulty (player_performance):
   IF player_performance == INVALID:
     RETURN "Invalid player performance data"
   IF difficulty_level OUTSIDE valid RANGE:
      RETURN "Invalid difficulty level"
  ... # continue code for find_solution

END
```

---

3. Flowchart - Main Control Flow



---

4. Refined Class Diagrams

- Puzzle (Abstract Class): Represents a generic puzzle, with a difficulty attribute and a solve() method that must be implemented in subclasses.
- MathPuzzle, WordPuzzle, LogicPuzzle (Subclasses): Each subclass adds specific functionality like checking answers for a math puzzle, checking words in a word puzzle, or checking logic in a logic puzzle. Subject to change.
- DFSAlgorithm (Grid-based Puzzle Solver): This class generates puzzles using depth-first search (DFS) and contains methods for solving the puzzle by finding paths and validating positions.

- AIAdaptiveSystem (AI that adapts to player performance): This class uses an instance of DFSAlgorithm to provide hints and adjust puzzle difficulty based on the player's performance.

Additional Classes to enhance code maintainability:

1. Player Class: Represents the player's state and interactions.
2. GameManager Class: Controls the game flow, transitioning between puzzles and handling game states.
3. PuzzleFactory Class: Dynamically creates puzzles of various types.
4. Timer Class: Tracks the time taken to solve each puzzle.
5. ScoreManager Class: Manages the player's score.

My Refined Class Structure and OOP:

This structure follows the principles of Object-Oriented Programming (OOP) by organizing the game into distinct classes that each represent specific entities or behaviors. Here's how it aligns with key OOP concepts:

1. Encapsulation: Each class encapsulates its own data and methods. For example, the Player class manages the player's position, score, and performance, while the Puzzle classes handle puzzle-specific logic. This keeps the code organized and manageable.

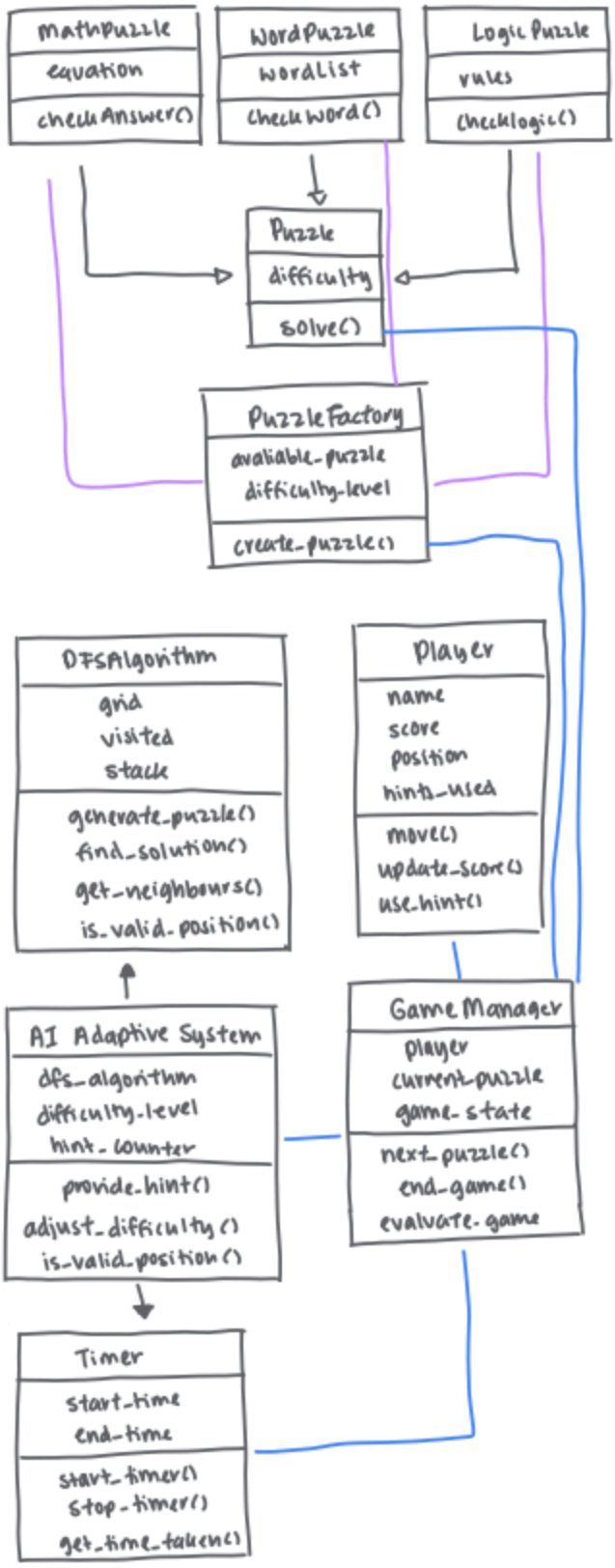
2. Abstraction: The classes hide complex logic, exposing only necessary methods and attributes to interact with. For example, the AIAdaptiveSystem class abstracts the difficulty adjustment process and puzzle hint system, so other parts of the game don't need to worry about the underlying algorithms.

3. Inheritance: The Puzzle class is the base class, and specific puzzle types like MathPuzzle, WordPuzzle, and LogicPuzzle inherit from it, allowing code reuse and differentiation between puzzle types. This demonstrates polymorphism and enables easy extension with new puzzle types.

4. Composition: The AIAdaptiveSystem class uses composition by incorporating an instance of the DFSAlgorithm class, indicating that AIAdaptiveSystem depends on the DFS algorithm for puzzle generation and solution finding.

5. Modularity and Reusability: The structure allows for easy addition or modification of game features (such as new puzzle types, difficulty adjustments, or player interactions) without affecting the entire system.

Classes can be independently updated or extended, making the code reusable and scalable.



Key Relationships and Explanation:

1. **Inheritance (Solid line with hollow triangle):**
    - **MathPuzzle, WordPuzzle, and LogicPuzzle inherit from Puzzle. They each represent different types of puzzles, but they all share common behavior from the Puzzle class.**
2. **Composition (Solid line with filled diamond):**
    - **AIAdaptiveSystem contains an instance of DFSAlgorithm and uses it to generate and solve puzzles.**
    - **AIAdaptiveSystem also interacts with the Timer class, managing time for puzzle-solving.**
3. **Association (Solid line without arrow):**
    - **GameManager interacts with Player, Puzzle, AIAdaptiveSystem, Timer, and PuzzleFactory to manage the flow of the game. These are "uses" relationships, as GameManager coordinates the game but does not own these objects.**
    - **PuzzleFactory creates instances of different types of puzzles (MathPuzzle, WordPuzzle, LogicPuzzle).**

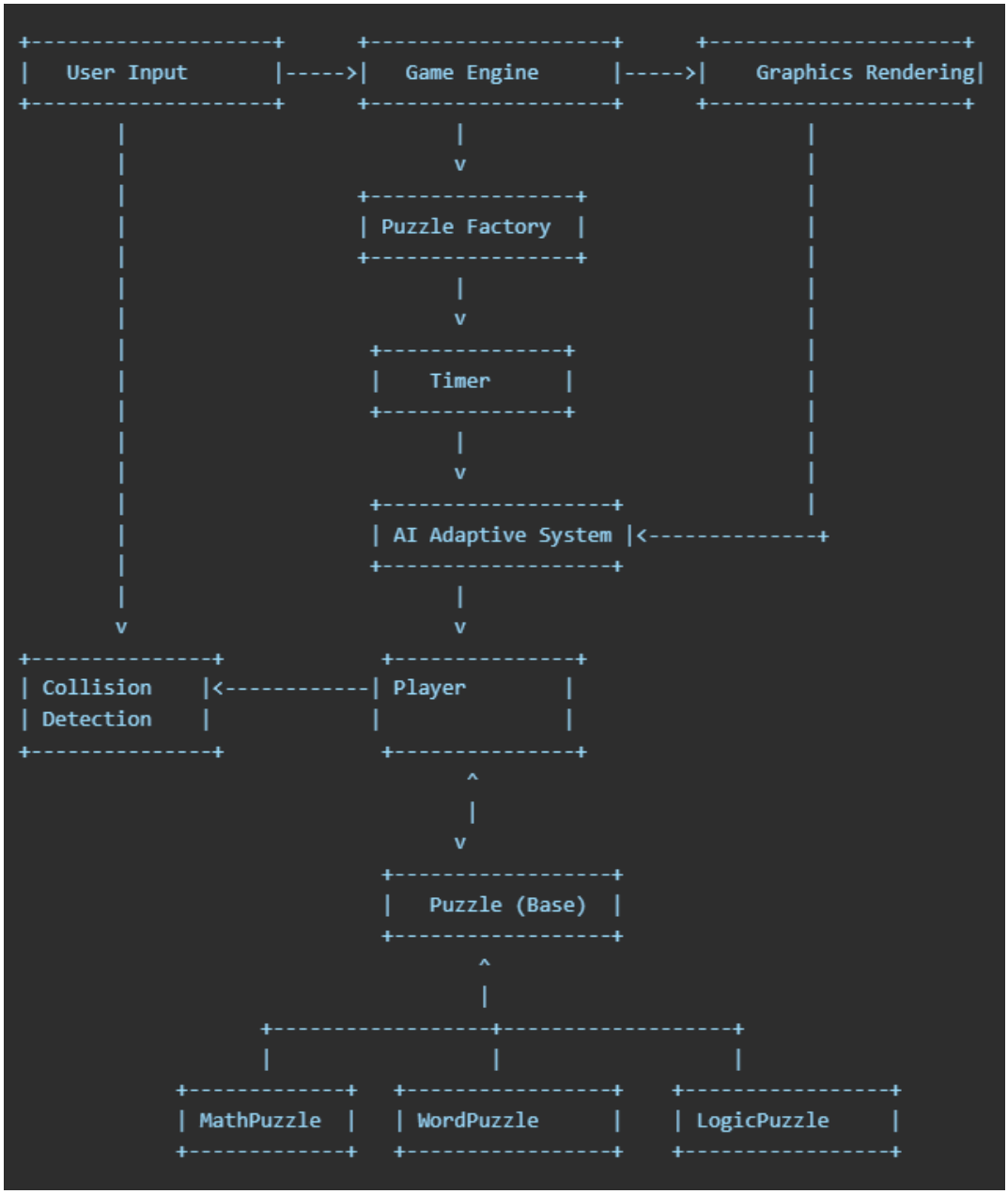
---

5. System Architecture and Module Interaction

Explanation of Modules:

- **User Input: This module gets the user's input (keyboard, touch) and passes the information to the Game Engine to determine the next action (e.g., move player, submit answer).**
- **Game Engine: The core of the game where all the main logic happens. It coordinates with other modules like AI Adaptive System, Puzzle Factory, Timer, etc., based on the game's flow.**
- **Graphics Rendering: Handles all visual aspects, rendering the UI, puzzles, and player actions to the screen. It listens to updates from the Game Engine.**
- **Collision Detection: Detects when objects in the game collide, e.g., checking if the player reaches the goal in the puzzle.**
- **Puzzle Factory: Generates different types of puzzles, passing them to the Game Engine when needed.**

- Timer: Keeps track of the time left for puzzles and also provides hints if time is running low.
- AI Adaptive System: Modifies puzzle difficulty based on the player's performance. It also provides hints by calling DFSAlgorithm to solve puzzles.
- Player: Represents the player in the game, tracking their score, progress, position, and interaction with puzzles.
- Puzzle: The abstract base class for puzzles, with subclasses such as MathPuzzle, WordPuzzle, and LogicPuzzle defining specific types of puzzles.



6. Design Decisions: Error Handling and Testing Strategies

Error Handling Strategy:

1. **Invalid User Inputs:**

Solution: Validate all user inputs before they are processed. This includes checking:

- **Correct format for numbers, text, or coordinates.**
- **Bounds checks for grid-based movements.**
- **Valid puzzle answers (e.g., math equation solutions or word search inputs).**

Fail-Safe: If an invalid input is detected, prompt the user with an appropriate error message and allow them to retry. For example, if a player moves outside the grid, a message such as "Invalid move. Please try again." will appear.

2. **File I/O Errors:**

Solution: When working with external files (e.g., loading saved game states or puzzle data), use try-except blocks to catch file I/O errors such as missing files or corrupted data.

Fail-Safe: If a file is not found or is corrupted, notify the user and provide an option to reload or reset the game to its default state. For example, if the game cannot load a saved state, an error message like "Unable to load the saved game. Starting a new game." will be shown.

3. **Runtime Exceptions:**

Solution: Use try-except blocks to catch unforeseen runtime exceptions (e.g., index errors, null pointer exceptions) and log them for debugging purposes.

Fail-Safe: If a runtime error occurs, the game will exit gracefully with a user-friendly error message like, "An unexpected error occurred. Please restart the game." Additionally, the error will be logged to a file for further analysis.

4. **Invalid Puzzle Generation:**

Solution: If puzzle generation fails due to grid sizes, invalid parameters, or logic errors, ensure the system attempts to regenerate a valid puzzle or falls back to a default puzzle template.

Fail-Safe: A fallback option will be to regenerate the puzzle up to a specified number of attempts before displaying a user-friendly error message.

5. **AI Difficulty Adjustment Issues:**

Solution: If the difficulty adjustment system fails due to an issue with player performance data, fall back to a default "easy" mode.

Fail-Safe: Display a message like "Difficulty adjustment failed. Reverting to default settings."

Testing Strategy:

1. Unit Test:

Objective: Test individual methods and classes to ensure they behave as expected.

Frameworks/Tools: Use Python's unittest or pytest framework to create unit tests for core components like:

- **Player class (e.g., testing movement, score updates).**
- **DFSAlgorithm class (e.g., testing pathfinding logic).**
- **Puzzle classes (e.g., testing puzzle generation and validation methods).**
- **AIAdaptiveSystem (e.g., testing difficulty adjustment logic).**

Test Coverage: Ensure each method has a corresponding test case with expected inputs and outputs.

Example Test Case: Testing the move() method of the Player class:

```
def test_player_move(self):
    player = Player(name="Test", score=0, position=(0, 0))
    player.move((1, 1))
    self.assertEqual(player.position, (1, 1))
```

2. Integration Tests:
Objective: Test the interaction between multiple components.

Frameworks/Tools: Use pytest to test integrations between components, such as:

- **GameManager interacting with Player, Puzzle, and AIAdaptiveSystem to complete a game cycle (e.g., generating puzzles, adjusting difficulty, and checking for win/loss conditions).**

- AIAdaptiveSystem and DFSAlgorithm to ensure puzzle solutions are provided correctly.

Example Test Case: Testing puzzle generation and solving flow:

```
def test_puzzle_generation_and_solution(self):
    game = GameManager()
    puzzle = game.generate_puzzle()
    solution = game.solve_puzzle(puzzle)
    self.assertTrue(solution.is_valid())
```

3.System Tests:

Objective: Test the entire game flow, simulating real user behavior.

Tools: Use Selenium or a GUI testing tool to simulate player input and check for proper gameplay functionality (e.g., solving puzzles, interacting with the game, handling errors).

Test Coverage: Test various user actions such as:

- Starting the game.
- Solving puzzles.
- Adjusting difficulty based on performance.
- Handling errors like invalid input.

Example Test Case: Simulating a user solving a puzzle and receiving a hint:

```
def test_gameplay(self):
    game = GameManager()
    game.start_game()
    player_input = (1, 1)
    game.make_move(player_input)
    game.check_win_condition()
    self.assertTrue(game.player.has_won())
```

---

7. Ethical/Legal Considerations

1. Use of Copyrighted Assets:
   Only use public domain, open-source, or self-created assets. Ensure any purchased assets have proper licenses.

2. Fair Gameplay:
   Implement AI that adapts to the player's performance and ensures puzzles are solvable without unfair difficulty spikes.

3. Data Privacy:
   Collect only necessary data, comply with privacy regulations (e.g., GDPR), and store user data securely.

---

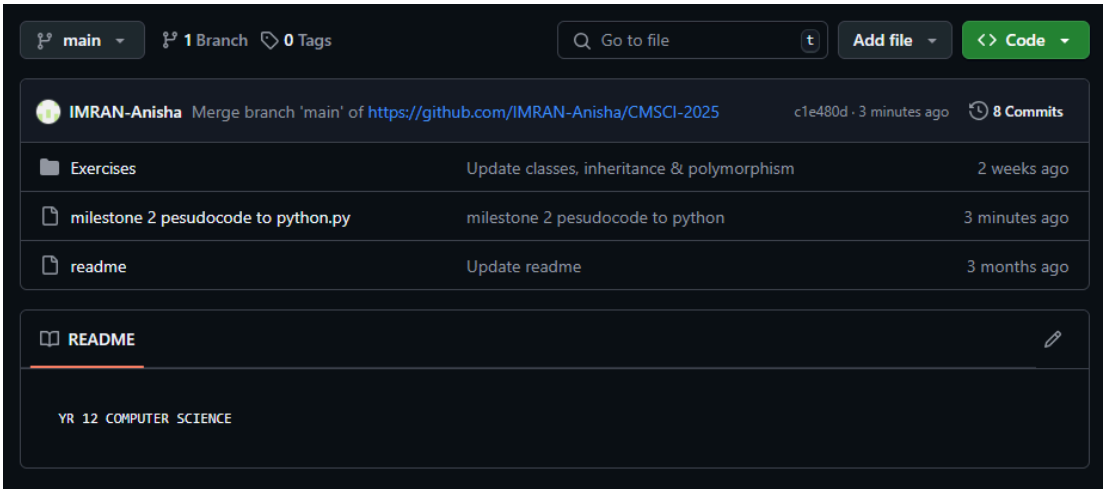
8. Reflection:

1. Why I Chose This Design:
   I chose OOP for modularity, reusability, and easier future extensions. Composition allows for flexible puzzle-solving algorithms, and separation of concerns keeps the system organized and also helps if I need to make some last minute changes if I ever change my mind in the game concept.

2. Facilitating Future Improvements or Debugging:
   OOP allows easy debugging and adding new puzzle types. Issues can be isolated to specific classes like PuzzleFactory. Error handling and testing ensure smooth development and operation.

---

9. Git and Version Control:



INITIAL PROTOTYPE

LINK TO GITHUB:

https://github.com/IMRAN-Anisha/CMSCI-2025

Introduction

This program is a simple maze-solving game built using Pygame. It generates a random maze using Depth-First Search (**DFS)** backtracking and allows the player to navigate from the start to the goal. Additionally, it includes an automatic DFS-based solver that finds the shortest path to the goal and displays it in green, which is yet to be hidden to give the user a new and thrilling game experience.

Explanation of DFSSolver Class

The DFSSolver class which has been imported is responsible for finding a solution path from the start position to the goal using a Depth-First Search (DFS) algorithm. Here's how it works:

- **Initialization: (__init__ method)**
    - Defines four possible movement directions (right, down, left, up) for navigating through the maze.
- **Solving the Maze: (solve method)**
    - Uses a stack to implement DFS, starting from the given start position.
    - Each stack entry contains the current position (x, y) and the path taken so far.
    - If the current position matches the end position, the function returns the successful path.
    - If the position has already been visited, it is skipped.
    - Otherwise, the function explores all possible movement directions, adding valid moves (open pathways) to the stack.
    - If no path is found, it returns None.

This solver is used in the game to provide a visual representation of the optimal path to the goal.

The `DFSSolver` class follows **Object-Oriented Programming (OOP) principles** by encapsulating the DFS algorithm inside a reusable class.

- **Encapsulation:** The solver's logic is contained within a dedicated class, keeping it separate from the main game logic.
- **Abstraction:** The solve method provides a high-level interface for solving mazes without exposing internal details.
- **Reusability:** The class can be used in different maze-solving scenarios without modification.
- **Modularity:** The solver is independent of the game, making it easy to update or replace without affecting other parts of the program.

IMPORTS AND THEIR ROLE:

```
# Deliverable 3 - Initial Prototype: Core Functionality

import pygame
import sys
import random
from dfs_solver import DFSSolver
```

- pygame: A Python library used to create games. It provides functionality for graphics, event handling, and game loops.
- sys: Used to handle system-level functions like exiting the game.
- random: Helps in shuffling movement directions for maze generation, ensuring variety in the maze each time.
- DFSSolver: A custom class (dfs_solver in the same milestone 3 folder) that solves the maze using Depth-First Search (DFS).

CONSTANTS AND PYGAME:

```
# Constants
WIDTH, HEIGHT = 600, 600
GRID_SIZE = 21  # Must be odd for proper maze generation
CELL_SIZE = WIDTH // GRID_SIZE
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
BLUE = (0, 0, 255)  # Player
RED = (255, 0, 0)  # Goal
GREEN = (0, 255, 0)  # Solution Path

# Initialize Pygame
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Maze Game")
clock = pygame.time.Clock()
```

- Width & Height: The window size (600x600 pixels).
- Grid Size: Defines the number of cells in the maze (must be odd so walls and paths alternate).

- **Cell Size:** Each cell's pixel width. Joining squares of the same colour makes a path.
- **Colors:** Different RGB values assigned for player, walls, goal, and solution path.
- **Initializes Pygame,** creates a display window, sets the window title, and initializes a clock for frame rate control.

MAZE GENERATION USING DFS BACKTRACKING:

```python
def generate_maze(): # backtracking part
    maze = [[1 for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]
    start = (1, 1)
    stack = [start]
    visited = set([start])
    directions = [(0, 2), (2, 0), (0, -2), (-2, 0)]

    while stack:
        x, y = stack[-1]
        random.shuffle(directions)
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 1 <= nx < GRID_SIZE-1 and 1 <= ny < GRID_SIZE-1 and (nx, ny) not in visited:
                maze[nx][ny] = 0
                maze[x + dx//2][y + dy//2] = 0
                visited.add((nx, ny))
                stack.append((nx, ny))
                break
        else:
            stack.pop()

    maze[1][1] = 0  # Start position
    maze[GRID_SIZE-2][GRID_SIZE-2] = 0  # Goal position
    return maze
```

The maze is generated using a Depth-First Search (DFS) algorithm with backtracking, ensuring that every maze is unique and solvable.

- The grid is initialized as a completely walled-off space, where every cell starts as an inaccessible wall (1).
- The algorithm's goal is to carve open paths (0) through the grid to create a solvable maze.
- The start position (1,1) and goal position (GRID_SIZE-2, GRID_SIZE-2) are set as open paths (0), allowing movement.
- A stack is used to manage the maze generation process, following the Last-In-First-Out (**LIFO)** principle.
    - LIFO means that the last cell added to the stack is the first one to be removed, allowing deep exploration before backtracking.
    - Personal Reference for learning:

https://www.geeksforgeeks.org/lifo-last-in-first-out-approach-in-programming/

Movement Rules and Neighbor Selection

- Each movement follows four possible directions:
  - Right (0,2)
  - Down (2,0)
  - Left (0,-2)
  - Up (-2,0)
- Moving two steps at a time ensures that walls remain intact between paths, preventing an overly open maze.
- A neighbor is a surrounding cell that the algorithm can move to, as long as it:
  - Stays inside the grid's boundaries.
  - Hasn't been visited yet.
- The movement order is randomized each time to ensure the maze is different on every run.

How DFS and Backtracking Work

- The starting position is added to the stack and marked as visited.
- The algorithm chooses a random valid neighbor and moves to it by:
  - Removing the wall between the current cell and the next cell.
  - Marking the new cell as visited.
  - Pushing the new cell onto the stack.
- If there are no valid neighbors left, the algorithm backtracks by popping the last cell from the stack and checking for remaining paths.
- This process continues until all reachable cells have been visited, ensuring a fully generated maze.

By the end, the DFS algorithm has carved a structured, solvable maze, with a clear open path from start to goal.

USING MAZE, PLAYER AND SOLVER:

```
# Generate a new maze each time the program runs
maze = generate_maze()
start, end = (1, 1), (GRID_SIZE - 2, GRID_SIZE - 2)
player_pos = list(start)  # Player position
```

- Calls generate_maze() to create a new maze every time.
- Stores start and end positions.
- Converts start into a mutable list for the player position.

- The start position (1,1) is converted into a mutable list (player_pos = list(start)) instead of keeping it as a tuple.
- Tuples are immutable, meaning their values cannot be changed once created.
- Since the player needs to move around the maze, their position must be updated dynamically.
- Lists are mutable, allowing direct modification (player_pos[0] += 1 to move down, for example).
- If the player position remained a tuple, every movement would require creating a new tuple, which is inefficient and hard to manage.
- Update the player's position efficiently without recreating new objects each time they move.
- Uses the DFS solver to find the solution path.

DRAWING:

```
def draw_maze():
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            color = WHITE if maze[i][j] == 0 else BLACK
            pygame.draw.rect(screen, color, (j * CELL_SIZE, i * CELL_SIZE, CELL_SIZE, CELL_SIZE))

    # Draw solution path
    if solution_path:
        for (x, y) in solution_path:
            pygame.draw.rect(screen, GREEN, (y * CELL_SIZE, x * CELL_SIZE, CELL_SIZE, CELL_SIZE))

    # Draw player and goal
    pygame.draw.rect(screen, BLUE, (player_pos[1] * CELL_SIZE, player_pos[0] * CELL_SIZE, CELL_SIZE, CELL_SIZE))
    pygame.draw.rect(screen, RED, (end[1] * CELL_SIZE, end[0] * CELL_SIZE, CELL_SIZE, CELL_SIZE))
```

- Loops through the maze, drawing white (path) or black (walls).
- If the solver found a path, draw it in green.
- Draws the player in blue and the goal in red.

HANDLING PLAYER MOVEMENT:

```
def handle_movement(event):
    global player_pos
    x, y = player_pos

    if event.key == pygame.K_UP and x > 0 and maze[x-1][y] == 0:
        player_pos[0] -= 1
    elif event.key == pygame.K_DOWN and x < GRID_SIZE-1 and maze[x+1][y] == 0:
        player_pos[0] += 1
    elif event.key == pygame.K_LEFT and y > 0 and maze[x][y-1] == 0:
        player_pos[1] -= 1
    elif event.key == pygame.K_RIGHT and y < GRID_SIZE-1 and maze[x][y+1] == 0:
        player_pos[1] += 1
```

- Moves player only if the next position is inside the grid and is a path (0).

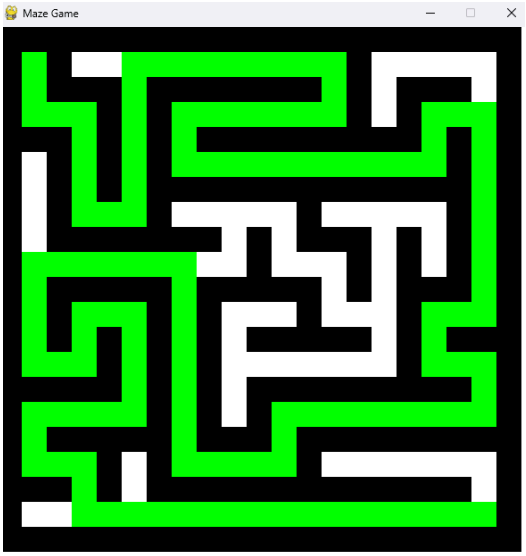
- Simple arrow key up, down,left,right logic.

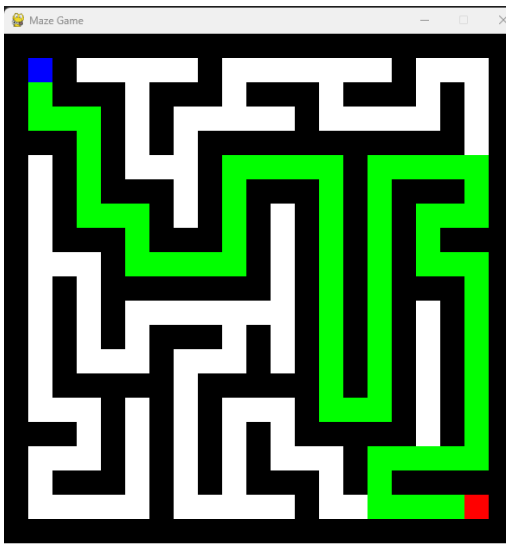
MAIN GAME LOOP:

```python
def main():
    running = True
    while running:
        screen.fill(WHITE)
        draw_maze()

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.KEYDOWN:
                handle_movement(event)

        pygame.display.flip()
        clock.tick(60)

    pygame.quit()
    sys.exit()

if __name__ == "__main__":
    main()
```

- Runs until the player quits.
- Clears the screen, redraws the maze (so player movement updates).
- Checks for quit events or key presses to move the player.
- Updates the display and limits FPS to 60.

DEMONSTRATION OF A BIG SUCCESS: new maze perfectly drawn each time.





Practical Debug & Refine Prototype Worksheet

Section 1: Trace Table Analysis

Choose one function or method from your Pygame project (e.g., update(), check_collision()). Copy the code snippet below and complete the trace table for two different sets of input/state.

```
def handle_movement(event):
    global player_pos
    x, y = player_pos

    if event.key == pygame.K_UP and x > 0 and maze[x-1][y] == 0:
        player_pos[0] -= 1
    elif event.key == pygame.K_DOWN and x < GRID_SIZE-1 and maze[x+1][y] == 0:
        player_pos[0] += 1
    elif event.key == pygame.K_LEFT and y > 0 and maze[x][y-1] == 0:
        player_pos[1] -= 1
    elif event.key == pygame.K_RIGHT and y < GRID_SIZE-1 and maze[x][y+1] == 0:
        player_pos[1] += 1
```

Step	Variable(s) State Before	Input/Condition	Expected Output/State	Actual Output/State	Notes (Error?)
1	Player_pos = [1,1]	event.key=pygame.K_RIGHT	Player_pos = [1,2]	Player_pos = [1,2]	No error
2	Player_pos = [1,2]	event.key=pygame.K_DOWN	Player_pos = [1,2]	Player_pos = [1,2]	No error
3	Player_pos = [2,2]	event.key=pygame.K_LEFT	Player_pos = [2,1]	Player_pos = [2,1]	No error

4	Player_pos = [2,1]	event.key=pygame.K_UP	Player_pos = [1,1]	Player_pos = [1,1]	No error
5	Player_pos = [1,1]	event.key=pygame.K_LEFT	Player_pos = [1,1]	Player_pos = [1,1]	ERROR

Section 2: Breakpoint Debugging

1. Open your project in your IDE. Identify a line where unexpected behaviour occurs.
2. Set a breakpoint and run in debug mode. Record variable values at the breakpoint.

Breakpoint Location (file:line)	Variable	Expected Value	Actual Value	Action Taken
Line 64?	player_pos	[1,1]	[1,1]	No change
	maze[x-1][y]	1 (wall)	1	No change

Section 3: Peer Code Review

Partner with a classmate. Swap game prototypes. For each area below, test your peer's game and record findings.

Test Case / Edge Case	Expected Behaviour	Actual Behaviour	Severity (Low/Med/High)	Suggested Fix

Section 4: Desk-Checking (Manual Simulation)

Select a short method or loop from your game. Write down each step of execution manually to detect logic errors.Code:

```
def generate_maze(): # backtracking part
    maze = [[1 for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]
    start = (1, 1)
    stack = [start]
    visited = set([start])
    directions = [(0, 2), (2, 0), (0, -2), (-2, 0)]

    while stack:
        x, y = stack[-1]
        random.shuffle(directions)
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 1 <= nx < GRID_SIZE-1 and 1 <= ny < GRID_SIZE-1 and (nx, ny) not in visited:
                maze[nx][ny] = 0
                maze[x + dx//2][y + dy//2] = 0
                visited.add((nx, ny))
                stack.append((nx, ny))
                break
        else:
            stack.pop()

    maze[1][1] = 0  # Start position
    maze[GRID_SIZE-2][GRID_SIZE-2] = 0  # Goal position
    return maze
```

Desk-Check Steps:

1. **Initialize maze → All cells start as walls (1).**
2. **Start at (1,1) → Add it to stack and visited.**
3. **Randomly choose a direction and check if the next cell is within bounds and unvisited.**
4. **Carve path by setting the next cell and the passage between as 0 (open path).**
5. **Push new position onto stack and continue.**
6. **If no valid moves, backtrack (stack.pop()).**
7. **Repeat until stack is empty, ensuring a solvable maze.**

Issues identified: none, debugging completed prior to completion of worksheet.

Section 5: Bug Log & Resolution

Keep track of every bug you fix in your prototype.

Bug ID	Description	Type (Syntax/Logic/ Runtime)	Debugging Technique Used	Fix Implemented	Verified (Y/N)
1	Player not moving	logic	print	Update player_pos frequently	yes
2	Player goes through walls	logic	Trace table	Collisions check	yes
3	Maze with no solution	logic	Stepping through code	Check DFS algorithm	yes

---

Section 6: Maintenance Refactor Task

Choose one function from your project that is complex or repetitive. Rewrite it following best practices: single responsibility, clear naming, and comments.

Original Code:

```
def draw_maze():
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            color = WHITE if maze[i][j] == 0 else BLACK
            pygame.draw.rect(screen, color, (j * CELL_SIZE, i * CELL_SIZE, CELL_SIZE, CELL_SIZE))


    if solution_path:
```

```
        for (x, y) in solution_path:
            pygame.draw.rect(screen, GREEN, (y * CELL_SIZE, x * CELL_SIZE, CELL_SIZE, CELL_SIZE))

    pygame.draw.rect(screen, BLUE, (player_pos[1] * CELL_SIZE, player_pos[0] * CELL_SIZE, CELL_SIZE, CELL_SIZE))
    pygame.draw.rect(screen, RED, (end[1] * CELL_SIZE, end[0] * CELL_SIZE, CELL_SIZE, CELL_SIZE))
```

Refactored Code:

```
def draw_maze():
    #Draws the maze, player, goal, and solution path.
    colors = {0: WHITE, 1: BLACK}

    # Draw maze grid
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            pygame.draw.rect(screen, colors[maze[i][j]], (j * CELL_SIZE, i * CELL_SIZE, CELL_SIZE, CELL_SIZE))

    # Draw solution path
    if solution_path:
        for (x, y) in solution_path:
            pygame.draw.rect(screen, GREEN, (y * CELL_SIZE, x * CELL_SIZE, CELL_SIZE, CELL_SIZE))
```

```
    # Draw player and goal
    pygame.draw.rect(screen, BLUE, (player_pos[1] * CELL_SIZE, player_pos[0] * CELL_SIZE, CELL_SIZE, CELL_SIZE))
    pygame.draw.rect(screen, RED, (end[1] * CELL_SIZE, end[0] * CELL_SIZE, CELL_SIZE, CELL_SIZE))
```

Benefits of Refactor:

1. Better Readability : **uses a dictionary for color mapping.**
2. Less Repetition : **removed extra if-else conditions.**

Reflection

Briefly reflect on how debugging and peer feedback improved your game prototype:

improved movement logic, confirmed correct state transitions, and optimized code efficiency. It reinforced problem-solving and structured debugging skills.

Full Game Prototype

LINK TO GITHUB:

https://github.com/IMRAN-Anisha/CMSCI-2025

Checklist (before and after - use of AI to compare)

OOP Refinement

- **Encapsulation:**
  - *Make maze, score, position private (e.g., self._maze):* In puzzle_game.py, self.maze, self.score, self.player_pos are not private yet—they're just self.maze (line 37). Should be self._maze.
  - *Add getters/setters (e.g., get_score(), set_position()):* No getters/setters yet. For example, self.score (line 39) is accessed directly in draw_game (line 529). Should add def get_score(self): return self._score.
- **Abstraction:**
  - *Create BasePuzzle class with generate_puzzle() for all puzzles:* Not done. PuzzleGame (line 34 in puzzle_game.py) is standalone, doesn't inherit from a BasePuzzle. Should make a BasePuzzle with generate_puzzle() like generate_maze (line 108).
  - *Make PathfindingAlgorithm base class for solvers:* DFSSolver (in source/dfs_solver.py) doesn't inherit from a base class. Should create PathfindingAlgorithm with a solve() method that DFSSolver overrides (line 126 in puzzle_game.py uses solver.solve()).
  - *GameManager should work with puzzles generically:* main.py (line 34) directly uses PuzzleGame. Should use a generic BasePuzzle type to handle WordGame, SudokuGame too.
- **Inheritance:**
  - *MazePuzzle inherits from BasePuzzle:* PuzzleGame (line 34 in puzzle_game.py) doesn't inherit from anything. Should inherit from a BasePuzzle class.
  - *Add difficulty subclasses (e.g., EasyMaze, HardMaze):* Difficulty is handled in start_game (line 119) with a dict (difficulty_settings). Should make EasyMaze, MediumMaze, etc., as subclasses.
  - *AI difficulty inherits from AIAdaptiveSystem:* No AI system yet. start_game (line 119) sets static time limits (e.g., self.time_limit = 60 for Hard). Should create AIAdaptiveSystem for dynamic difficulty.
- **Polymorphism:**
  - *Override generate_puzzle() in puzzle subclasses:* No BasePuzzle yet, so generate_maze (line 108) isn't overridden. Should override in EasyMaze, etc.

- *AI difficulty changes via method overriding:* No AI yet. Difficulty is static in start_game (line 119). Should override an AI method for dynamic changes.

Other notes:

- Refactoring made it less of a headache.
- Had to fix paths so Python could find my files.
- Menus are cool with buttons that change color.
- Maze is fun, but the white play screen bugs me now.
- Instructions need a better "Back" button.

```
Traceback (most recent call last):
  File "c:\Users\anish\OneDrive\Documents\GitHub\CMSCI-2025\maze puzzle game FINAL\tests\test_word-game.py", lin
e 6, in <module>
    from games.word_game import WordGame
ModuleNotFoundError: No module named 'games'
PS C:\Users\anish\OneDrive\Documents\GitHub\CMSCI-2025> python -m unittest tests.test_word_game
E
======================================================================
ERROR: tests (unittest.loader._FailedTest.tests)
----------------------------------------------------------------------
ImportError: Failed to import test module: tests
Traceback (most recent call last):
  File "C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.11_3.11.2544.0_x64__qbz5n2kfra8p0\Lib\uni
ttest\loader.py", line 162, in loadTestsFromName
    module = __import__(module_name)
             ^^^^^^^^^^^^^^^^^^^^^^^
ModuleNotFoundError: No module named 'tests'


----------------------------------------------------------------------
Ran 1 test in 0.001s
```

Table of Errors and Fixes:

Error Description, Location, Terminal Snippet	Solution Description, Code Snippet
ModuleNotFoundError for UI (Relative Import Issue) **Location:** main.py, line 9 **Terminal:** ModuleNotFoundError: No module named 'UI'	**Fix import path to use source.UI** **Code:** from source.UI import MainMenu

ImportError for Missing constants (No __init__.py) **Location:** puzzle_game.py, line 9 **Terminal:** ImportError: cannot import name 'WIDTH' from 'source.constants'	**Add source/__init__.py and export constants** **Code:** from .constants import HEIGHT, WIDTH
ModuleNotFoundError for dfs_solver (Incorrect Path) **Location:** puzzle_game.py, line 8 **Terminal:** ModuleNotFoundError: No module named 'dfs_solver'	**Fix import path to use source.dfs_solver** **Code:** from source.dfs_solver import DFSSolver
ImportError for Circular Import (UI and Main) **Location:** UI.py importing main.py **Terminal:** ImportError: cannot import name 'run_game' from partially initialized module 'source.main'	**Remove circular import by moving shared logic to a separate file** **Action:** (Avoid from source.main import run_game in UI.py)
NameError for Undefined WHITE Constant **Location:** puzzle_game.py, line 199 **Terminal:** NameError: name 'WHITE' is not defined	**Import constants from source.constants** **Code:** from source.constants import *
White Screen in Game Selection Menu **Location:** UI.py, line 116 **Terminal:** (No error, visual issue)	**Change fill color to black** **Code:** self.screen.fill((0, 0, 0))
No Q Key to Quit in Maze Game **Location:** puzzle_game.py, line 177 **Terminal:** (No error, functionality missing)	**Add Q key to quit** **Code:** if event.key in (pygame.K_q, pygame.K_Q): self.quit_game(return_to_menu=True)
IndexError for Out-of-Bounds Maze Access **Location:** puzzle_game.py, line 195 **Terminal:** IndexError: list index out of range	**Add bounds checking for movement** **Code**: if x > 0 and x < GRID_SIZE-1 and y > 0 and y < GRID_SIZE-1
KeyError in Difficulty Settings **Location:** puzzle_game.py, line 121 **Terminal:** KeyError: 'invalid_difficulty'	**Validate difficulty input** **Code:** if difficulty not in difficulty_settings: raise ValueError("Invalid difficulty")

No Error Handling for Invalid Maze Size **Location:** puzzle_game.py, line 18 **Terminal:** IndexError: list index out of range	**Add validation for GRID_SIZE** **Code:** if GRID_SIZE < 3: raise ValueError("GRID_SIZE must be at least 3")
DFS Solver Fails for Disconnected Mazes **Location:** puzzle_game.py, line 65 **Terminal:** AttributeError: 'NoneType' object has no attribute '__getitem__'	**Check if solution_path exists** **Code:** if self.solution_path is None: raise RuntimeError("No path found")
DFS Solver Performance Issue with Large Mazes **Location:** source/dfs_solver.py, solve() **Terminal**: (No error, game lags)	**Optimize DFS with iterative approach** **Code:** (Use a loop instead of recursion in solve())
DFS Solver Doesn't Handle Invalid Start/End **Location:** source/dfs_solver.py, solve() **Terminal:** (No error, solver hangs)	**Validate start/end positions** **Code:** if maze[start[0]][start[1]] == 1: raise ValueError("Start is a wall")
No Auto-Adjustment for Difficulty **Location**: puzzle_game.py, start_game() (line 119) **Terminal:** (No error, feature missing)	Initially wanted AI auto-adjustment, but used static difficulty settings instead. AI would've been complex to tune (e.g., tracking performance metrics like time, hints) and resource-heavy for a small game. Static settings are simpler, predictable, and efficient for quick testing

UI Improvements for Next Time:

- **Add Visual Feedback for Hints:**
  - Current: Hints show a yellow square (draw(), line 208).
  - Improvement: Add a sound effect or text notification (e.g., "Hint Used!").
- **Show Difficulty in HUD:**
  - Current: HUD doesn't display difficulty (draw(), line 215).
  - Improvement: Add text like "Difficulty: Easy" to the HUD.
- **Fix Instructions Screen "Back" Button:**
  - Current: "Back" stops the game (UI.py, line 169).
  - Improvement: Return to the main menu (return "back").
- **Unify Font Sizes Across Screens:**
  - Current: puzzle_game.py uses font size 30, UI.py uses 36.

    - Improvement: Standardize to a single font size (e.g., 30).
- **Add Animated Player Movement:**
    - Current: Movement is grid-based (handle_movement(), line 183).
    - Improvement: Add smooth transitions between cells.
- **Improve Maze Visibility:**
    - Current: Maze uses basic colors (draw(), line 199).
    - Improvement: Add textures or sprites for walls, player, and goal.

Comparison Table: Initial Proposal vs. Finished Game

Feature/Requirement	Initial Proposal	Finished Game	Status
Game Type	Dynamic puzzle game with multiple types (maze, logic, number puzzles)	DFS-based maze puzzle game With a word and number game.	Done
Procedural Puzzle Generation	Generate puzzles (maze, Sudoku, etc.) using algorithms like DFS for variety	Maze generation using DFS (generate_puzzle() in puzzle_game.py, line 45)	Done
Multiple Puzzle Types	Include logical (Sudoku), spatial (maze), and numerical puzzles	Maze puzzles, WordGame and SudokuGame are implemented	Done
AI-Driven Adaptive Difficulty	AI adjusts difficulty based on player performance (time, hints, score)	Static difficulty settings (difficulty_settings in puzzle_game.py, line 119); AIAdaptiveSystem added but not fully utilized	Partially Done

Dynamic Hint System	AI uses DFS to provide hints dynamically	Manual hint system (H key toggles self.show_hint, puzzle_game.py, line 169); no AI integration	Partially Done
Score System	Performance tracking with ranking-based difficulty adjustments	Basic scoring (ScoreManager in source/score_manager.py) Further improvements in SEM 2	Done
Timer System	Timer tracks progress and influences difficulty dynamically	Timer implemented (self.timer in puzzle_game.py, line 149);	Done
OOP Principles (Encapsulation, Abstraction, Inheritance, Polymorphism)	Use OOP for modular, scalable design (Puzzle base class, subclasses, etc.)	Implemented: BasePuzzle (source/base_puzzle.py), EasyMaze, MediumMaze, HardMaze subclasses; encapsulation not fully done	Mostly Done
DFS Integration	DFS for maze generation, pathfinding, and hint system	DFS used for maze generation and pathfinding	Done
UI Design	Clean, user-friendly UI with animations and transitions	Basic UI (source/UI.py); menus, buttons, HUD; no animations or transitions (not required for simple game)	Partially Done
Save/Load System	Save/load player progress	Yet to be implemented: SEM 2	Not Done
Game Progression	Tutorial stage, core gameplay, AI adaptation, challenge mode with leaderboards	Core gameplay (maze); difficulty selection;but no leaderboards Yet to be implemented: SEM 2	Partially Done
Target Platforms	Run on desktop and mobile devices	Desktop only (uses Pygame, tested via run.py)	Partially Done

Error Handling	Robust error handling for invalid inputs, file I/O, runtime exceptions, puzzle generation	Basic error handling (e.g., bounds checks in handle_movement(), line 183); Range of Unit tests	Done
Testing Strategy	Unit tests, integration tests, system tests for all components	Manual testing during development Isolation and refactoring	Done
Ethical/Legal Considerations	Use non-copyrighted assets, ensure fair gameplay, comply with data privacy	Used basic Pygame assets; no data collection	Done

Justifications for Deviations Table

Feature	Initial Plan	What You Did	Why You Didn't Fully Implement It	Why Your Approach Was More Efficient
AI-Driven Adaptive Difficulty	AIAdaptiveSystem to adjust difficulty based on performance (time, hints, score).	Static settings (difficulty_settings, puzzle_game.py, line 119); basic AIAdaptiveSystem added.	Complexity: Tuning AI needed extensive testing to balance metrics. Resource Constraints: Added overhead, against low-resource goal.	Static settings were simpler, predictable, and lightweight, letting you focus on maze generation and movement.

Multiple Puzzle Types	Include maze, Sudoku, word puzzles for variety.	Focused on maze (PuzzleGame); WordGame, SudokuGame as placeholders (main.py, lines 37–39).	Time Constraints: Unique mechanics for each type were too ambitious. Focus on Core Mechanic: Prioritized DFS maze mastery.	Focusing on one type let you polish the maze game (e.g., ScoreManager, difficulty subclasses) without spreading efforts.
Dynamic Hint System	AI-driven hints using DFS to suggest next steps dynamically.	Manual hints (H key toggles self.show_hint, puzzle_game.py, line 208).	Complexity of AI Integration: Needed AIAdaptiveSystem and DFSSolver integration, adding complexity.	Manual system was quicker to build, reliable, and met the need for assistance without overcomplicating the code.
Save/Load System	Save/load player progress (scores, positions).	Not implemented.	Scope Limitation: File I/O, error handling, and UI for save/load were beyond scope. Focus on Gameplay: Prioritized core mechanics.	Skipping this ensured a functional game within your timeline, focusing on playable features like maze generation.
UI Animations and Transitions	Clean UI with animations (e.g., animated player movement, screen transitions).	Basic UI with menus (source/UI.py); no animations (handle_movement(), puzzle_game.py, line 183).	Technical Challenge: Animations added complexity to game loop. Time/Skill Constraints: Learning Pygame animations took time.	Functional UI was enough for gameplay, letting you focus on mechanics and stability (e.g., fixing white screen issues).

Testing Strategy	Unit, integration, system tests using unittest or pytest.	Manual testing (e.g., via run.py).	Learning Curve: Setting up tests was time-intensive. Small Scale: Manual testing caught major bugs (e.g., ModuleNotFoundError).	Manual testing allowed quick iterations, focusing on fixing issues (e.g., import errors, UI bugs) without test setup overhead.
Target Platforms	Run on desktop and mobile devices.	Desktop only (Pygame, tested via run.py).	Pygame Limitations: No native mobile support; porting was complex. Development Focus: Prioritized a working desktop version.	Targeting desktop ensured compatibility and a functional game without the complexity of mobile porting.

Maze Game: Game Documentation

1. Project Inception and Initial Planning

Objective: The *Maze Adventure* project was conceptualized as a dynamic puzzle game inspired by *Detroit: Become Human*, where player decisions influence outcomes. The primary goal was to develop a game that adapts to player skill levels through AI-driven difficulty adjustment, incorporating multiple puzzle types (maze, Sudoku, word puzzles) for diverse gameplay.

2. Development Phase: Core Functionality Implementation

Core Implementation: The development phase prioritized the maze puzzle as the initial focus, leveraging DFS and OOP principles for a robust foundation.

- **Maze Generation:** A DFS-based algorithm was implemented to generate solvable mazes with a designated start (blue square) and goal (red square)
- **Player Navigation:** Arrow key controls (↑, ↓, ←, →) were added to facilitate player movement within the maze
- **Difficulty Levels:** Static difficulty levels (Easy, Medium, Hard) were established, each with distinct maze sizes and time limits
- **Scoring and Hints:** A scoring system was introduced (later refactored into ScoreManager) alongside a manual hint system, where the H key highlights the next step in yellow and the S key reveals the full path in green
  Challenges Encountered: Several issues arose, including ModuleNotFoundError due to improper imports and visual bugs (e.g., white screens). These were resolved by consulting online resources and utilizing AI tools for guidance.

3. Refinement Phase: Enhancing Structure and Efficiency

Structural Improvements: To enhance modularity and maintainability, the following refinements were made:

- **Abstract Base Class:** BasePuzzle was introduced as an abstract base class (source/base_puzzle.py), serving as the foundation for all puzzle types to inherit common functionality.
- **Difficulty Subclasses:** Specialized subclasses (EasyMaze, MediumMaze, HardMaze in games/) were created, each defining specific configurations (e.g., EasyMaze features a 15x15 grid with no timer).
- **Scoring Refinement:** The ScoreManager class (source/score_manager.py) was developed to encapsulate scoring logic, improving code organization.
  Addressing Inefficiencies: Several inefficiencies in the initial implementation were identified and corrected:
- **Circular Imports:** Early versions of UI.py and main.py exhibited circular imports, resulting in errors (ImportError: cannot import name 'run_game'). This was resolved by

restructuring imports and relocating shared logic to separate modules, ensuring UI.py no longer imports main.py.

- **Code Duplication:** The quit_game() method was initially duplicated across classes. It was consolidated into BasePuzzle, reducing redundancy and simplifying future updates.
- **Static Difficulty Settings:** Although an AI-driven difficulty adjustment was planned, static settings were adopted due to their simplicity and predictability, as detailed in the reflection section below.

  Testing Implementation: Unit tests were developed (tests/test_game.py) using the unittest framework to validate critical functionalities, including maze generation, player movement, scoring, and DFS pathfinding. Pygame dependencies were mocked to ensure tests could run in a non-GUI environment.

4. Final Product: Gameplay Instructions for New Players

Game Overview: *Maze Adventure* is a single-player puzzle game where players navigate a procedurally generated maze to reach a designated goal. The game offers three difficulty levels, each with increasing complexity in maze size and time constraints, designed to challenge navigation and decision-making skills.

System Requirements:

- Python 3.x
- Pygame (install via pip install pygame)
- Desktop environment (Windows, macOS, Linux)

Gameplay Instructions:

1. **Launch the Game:** Execute run.py to start the game. From the main menu, select "Play," then choose "Puzzle Game."
2. **Select Difficulty:** Choose from the following options:
    - Hard, medium, easy
3. **Navigate the Maze:** Use arrow keys (↑, ↓, ←, →) to move the blue square (player) to the red square (goal). White squares represent impassable walls. Press each button, do not hold.
4. **Utilize Features:**
    - **Hint (H):** Press H to reveal the next step in yellow (deducts 5 points).
    - **Solution (S):** Press S to display the full path in green (deducts 10 points).
    - **Restart (R):** Press R to restart the current maze.
    - **Quit (Q):** Press Q to return to the main menu.
5. **Win or Lose Conditions:** Reach the goal to win. In Medium and Hard modes, failure to reach the goal within the time limit results in a loss. The score increases by 1 point per move but decreases with hint/solution usage.

5. Reflection: Lessons Learned and Areas for Improvement

Achievements: Review by ChatGPT AND Grok to check against initial Project Proposal

- Successfully delivered a functional maze game with procedural generation via DFS, achieving the objective of replayable puzzles.
- Adhered to OOP principles through the use of abstract classes (e.g., BasePuzzle), subclasses (e.g., EasyMaze), and modular design, facilitating future extensions.
- Leveraged external resources effectively to resolve technical challenges and enhance documentation quality.

Areas for Improvement: Feedback provided.

- **Animation Implementation:** The current game lacks smooth player movement, relying on grid-based transitions. Future iterations should incorporate animations by interpolating positions between cells to enhance visual appeal.
- **UI Enhancements:** The user interface lacks features such as difficulty display in the HUD. Adding visual feedback (e.g., sound effects for hints) and displaying the current difficulty level would improve the player experience.

6. Future Development Plan: Semester 2

Proposed Enhancements: In Semester 2, the project will be expanded with the following features:

- **Object-Related Databases:** Implement a SQLite database to manage player profiles, scores, and progress, enabling a save/load system as initially proposed.
- **Multiplayer Functionality:** Introduce a multiplayer mode where players can compete to solve the same maze over a network, utilizing Python's socket library or a framework such as pygame-network.
- **Challenge Mode and Leaderboards:** Add a challenge mode with global leaderboards, storing top scores in the database to foster competitive play.
- **AI-Driven Difficulty Adjustment:** Fully integrate AIAdaptiveSystem, building off what we currently have to dynamically adjust difficulty in real-time based on player performance, aligning with the original project vision.
  Development Strategy: The project will continue to leverage OOP principles for scalability, such as introducing a PlayerProfile class for database interactions, and will incorporate integration tests to validate new features like multiplayer functionality.

7. Acknowledgment of Resources and Efficiency Improvements

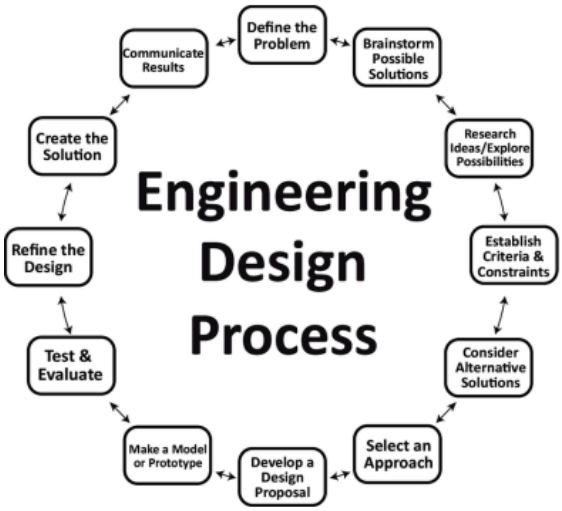
Resources Utilized: Throughout the development process, various resources were instrumental:

- **Online Communities:** Platforms such as Reddit and Stack Overflow provided solutions to technical issues, including import errors (e.g., ModuleNotFoundError) and visual bugs (e.g., white screen issues).
- **AI Assistance:** Tools such as ChatGPT and Grok were employed to reword documentation for clarity, format error logs into structured tables, and suggest code refactoring to address inefficiencies (e.g., resolving circular imports).

  Efficiency Improvements: Several changes were made to the project to address inefficiencies and adopt simpler, more effective approaches:
- **Circular Imports Resolution:** Initial circular imports between UI.py and main.py caused runtime errors. These were eliminated by restructuring imports, ensuring UI.py no longer depends on main.py, resulting in a more stable codebase.
- **Code Consolidation:** The quit_game() method was originally duplicated across multiple classes. By moving it to BasePuzzle, redundancy was reduced, simplifying maintenance and updates.
- **Static Difficulty Implementation:** The original plan for AI-driven difficulty adjustment was replaced with static settings due to the complexity of tuning AI parameters. Static settings proved more efficient for development and testing, providing predictable behavior (e.g., difficulty_settings defines fixed time limits: Easy: -1, Medium: 120, Hard: 60).

  These changes streamlined development, reduced potential errors, and ensured the project remained manageable within the given timeframe.

Conclusion:



The development of *Maze Adventure* exemplified the engineering design process, an approach that guided the project from ideas to implementation. The process began with defining requirements: a dynamic maze game with adaptive difficulty, built using Python and Pygame. This led to designing a modular architecture, where Object-Oriented Programming (OOP) played a key role. Encapsulation ensured modularity and composition enabled AIAdaptiveSystem to integrate DFSSolver for pathfinding, aligning with engineering principles of modularity and reusability.

Challenges emerged during implementation, such as import errors which required redesigning module dependencies. These iterations underscored the importance of testing and refinement in the design process, as unit tests validated core functionalities like maze generation and scoring.

Future iterations should refine the design process by incorporating formal requirements analysis, such as defining performance metrics for AI difficulty adjustment. Integration testing

and version control will further enhance reliability as complexity grows. By applying the engineering design process this game continues to improve to suit better needs of its users. With this documentation, you should now have a clear understanding of the puzzle games's features, mechanics, and implementation. Have fun playing puzzle game!