Final Year Project Report

Full Unit - Final report

Playing Games and Solving Puzzles Using AI

Ong, Man Hei

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Dr Wahlstrom, Magnus

Department of Computer Science
Royal Holloway, University of London

April 5, 2025

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count on Context only: *[5878]* Word Count including diary: *[7020]*

Student Name: Ong, Man Hei

Date of Submission: 12/12/2024

Signature: Ong, Man Hei

Table of Contents

Chapter 1: **Introduction**

1.1 Background Information

This project investigates which algorithms can be effectively applied to the popular puzzle game Unblock Me, also known as Rush Hour. This classic sliding block puzzle game requires the player to maneuver a red block to the designated exit by moving the other blocks around it using the fewest moves possible, making it an ideal candidate for testing different kinds of search algorithms. The main objective of this project is to develop a functional Unblock Me game and implement an AI that can automatically solve the levels created by the student in real time. A few search algorithms have been chosen, for instance, Breadth-First Search (BFS), Depth-First Search (DFS), Greedy Search, Dijkstra, and the A* algorithm, to implement into the game as a function and solve puzzles of varying difficulty game levels. These kinds of algorithms are frequently used in AI applications to find an optimal solution or path. This project explores different algorithms' strengths, weaknesses, and efficiency by testing these algorithms in the puzzle context.

1.2 Research Motivation

The motivation behind this research is my passion for different kinds of puzzle games, as I was always fascinated by different types of challenging puzzles when I was young, including Unblock Me/Rush Hour. Beyond puzzle games, my interest extends to video games in general and I always inquire about how the game's NPCs follow their pathfinding algorithms. One of my goals is to create a successful video game using my imagination and creativity and I am also curious about how a machine could solve those problems so quickly and efficiently, seemingly in the blink of an eye. This curiosity sparked my interest in exploring creating a functional modern video game and understanding the logic behind the algorithms behind "artificial intelligence ."

Chapter 2: **Background theory**

2.1 Python

Python is a high-level programming language widely used in game development, AI, and data science due to its simplicity and readability. Its libraries and frameworks, such as Pygame, make it ideal for developing simple 2D games like Unblock Me. Python also allows developers to create graphs for data science. In this case, it will be used to plot the time graph for the different time graphs to compare the performance of various algorithms, providing a clear visualization of their efficiency. In other words, it could create a benchmark for the algorithms.

2.2 Research Background of Unblock Me

Rush Hour, also known as Unblock Me, is a sliding block puzzle game invented by Nobuyuki Yoshigahara, a mechanical and mathematical puzzle inventor, during the 1970s [10]. The original game involved maneuvering cars within a 6 x 6 traffic-like grid and the main objective was to clear a path for a specific vehicle to exit the traffic. Later, in 2009, Unblock Me was initially released. It adapted Yoshigahara's original concept by replacing cars with blocks. The game requires the player to maneuver yellow blocks, blocking a designated red block through a 6 x 6 grid. Depending on their orientation, the player can move the yellow blocks vertically or horizontally, while the red block can only move horizontally. The game's main objective is to guide the red block to the exit using the fewest possible moves. The hardest possible configuration requires a minimum of 93 steps to solve[6], which is one reason why mathematicians and computer scientists are interested in this puzzle.

2.3 Levels in my game

I enhanced my project by using select levels from Michael Fogleman's Rush Hour puzzle database[8] instead of making my own. Michael Fogleman's levels provided a well-structured foundation, enabling me to focus on implementing and refining my game mechanics instead of spending so much time designing each level. Their inclusion allowed for testing various gameplay scenarios, ensuring the game's functionality and challenge progression were both engaging and effective.

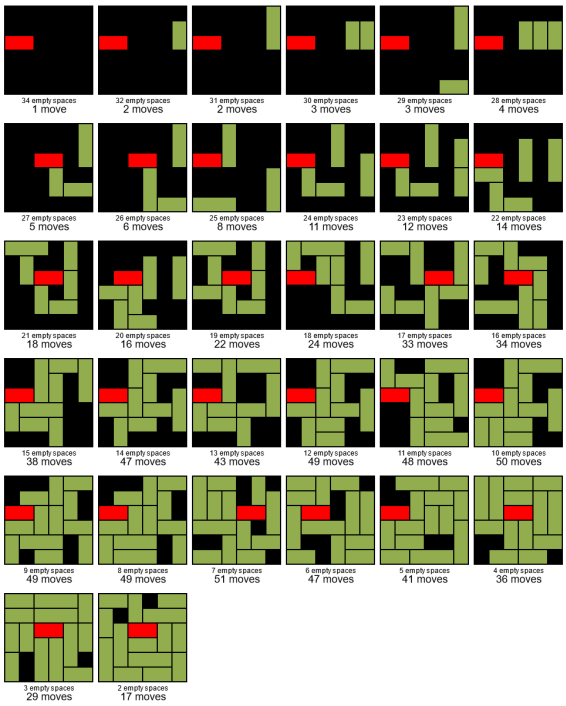


Figure 2.1: Full credit for these levels goes to Michael Fogleman

2.4 Application of Algorithms in Unblock Me

Different search algorithms used in solving puzzles in Unblock Me could result in different times. Currently, five different Algorithms have been chosen for the project.

Breadth-First Search (BFS)

Breadth-first search is a data structure that uses a "queue" system to manage the collection of states. It operates based on the principle of "First In, First Out (FIFO)," which means the first element added to the queue is the first one to be popped out of the list. Theoretically, in BFS each state represents a node in a graph or a position in a problem's state space. By systematically exploring all neighbors of a node before moving to the next layer, starting from the root, the search is expanded from the root's child node to its neighbors and then to their neighbors. BFS guarantees to find the shortest path to a solution if one exists [7]. However, the drawback is that BFS is more exhaustive and memory-intensive since all states are systematically evaluated layer by layer, ensuring the shortest path is found. (See Figures 2.2)

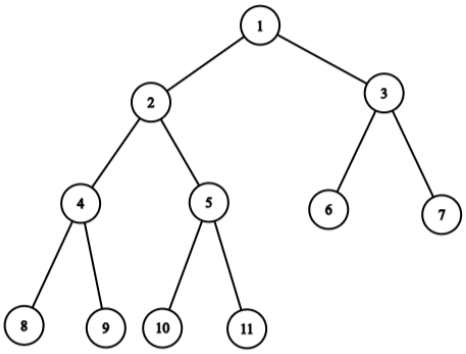


Figure 2.2: BFS node tree

Depth-First Search (DFS)

Depth-first search is a data structure that uses a "stack" system to manage the collection of states. It operates similarly to BFS but follows another principle, "Last In, First Out (LIFO)," in which the last element added to the stack is the first to pop out. This allows DFS to dive deep into a search tree branch before considering other branches. It also efficiently explores deep paths[15], but it requires backtracking when a dead-end of a branch is reached. When a dead-end is reached, or no further progress can be made along the current path, DFS performs backtracking, allowing the steps to be retraced to the last unexplored state, and the search continues. Regarding the backtracking method, it ensures that the DFS method explores all possible paths without getting stuck in a loop, from the parent node to the child node, and eventually it will find the solution [14]. Additionally, DFS's advantage is that it utilises less memory, as it only needs to store the current path in the stack rather than all states at a given level. (See Figures 2.3)

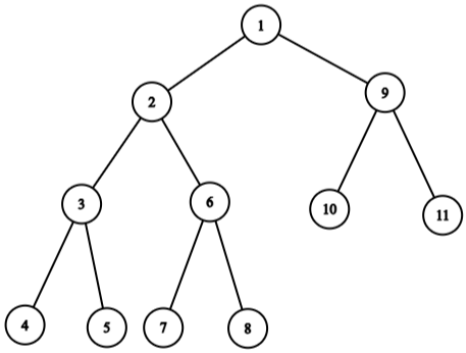


Figure 2.3: DFS node tree

Greedy Search

Greedy Search is a heuristic-based search algorithm that makes decisions solely based on the heuristic function given by the developer. To be effective, Greedy Search requires an appropriate heuristic function.A typical formula such as "Manhattan distance"[12] could calculate the distance between two points to estimate the estimated minimal cost from the current state to the target state. Mathematically, assuming if the red block is at the coordinate (x1,y1) and the exit is at the coordinate *(x2,y2)*, then the Manhattan Distance[2] formula is represented as:

$$MH(u, v) = |x_1 - x_2| + |y_1 - y_2|$$

or formula generalized as:

$$MH(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} |x_i - y_i|$$

Dijkstra's Algorithm

Dijkstra's Algorithm ensures that the solution found is optimal by automatically exploring each state in increasing order of path cost. Developers can use functions like priority queues to efficiently manage all the states that are appended to the list, allowing the algorithm to explore the least costly path first.

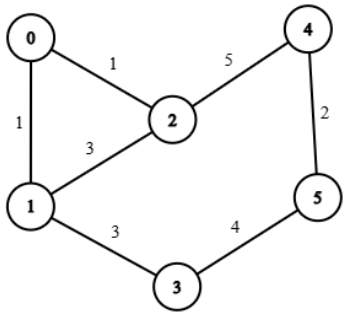


Figure 2.4: Dijkstra's Algorithm node graph

A* Search Algorithm

A* Search Algorithm is an advanced pathfinding algorithm that is widely used in many AI search applications in this decade, what makes it special is that it uses both the actual path cost (known as g(n)) represents the cost from the start of the state and an estimate heuristic cost (also known as h(n)) to the goal. By combining both costs[11], which is the total cost function is

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

, A* effectively computes and finds the most promising paths. In the context of Unblock Me, the state with the lowest f(n)value is processed first using Heapq. Every block move is checked to see if they are valid before the cost function is updated accordingly. When a new state is discovered, the algorithm calculates the new g(n)and adds the heuristic estimate h(n), resulting in the output in the total of f(n).

2.5 State space

A state space 4.4 refers to the entire set of possible configurations that a problem can take during the search for a suitable path. Take this project as an example, the state space consists of all the potential moveable blocks on the grid as they move towards solving the puzzle.

2.6 Research Purpose

- **Engine Selection:** Python's Pygame engine is preferred over others powerful engine like Unity and Unreal Engine for this project because it is more aligned with the fundamental objectives of developing and testing search algorithms for Unblock Me. The simplicity of Pygame allows me to focus on the remainder of the algorithmic side of the project rather than getting caught up in the complexities of a fully featured game engine. Unlike Unity or Unreal, which offer many useful tools and libraries for 3D rendering, physics simulation, and advanced graphical effects, Pygame provides a minimalistic framework ideal for a 2D puzzle game. This makes the development of the Unblock Me puzzle game easier and allows me to focus on the main goal which is refining the search algorithms without unnecessary distractions.
- **Algorithm Selection:** Theoretically, BFS and A* searching methods are considered some of the most effective algorithms because they guarantee finding the optimal path in a search space. On the other hand, DFS is much faster and more memory-efficient[1], but it is less consistent in producing an optimal path. In this study, each algorithm will have its own performance benchmark to keep track of its execution time and evaluate whether it is able to find the optimal path.

Chapter 3: **Library Overview**

3.1 Pygame

Pygame is an open-source Python library[13] designed for creating 2D games or multimedia applications and is based on the Simple Direct Media Layer (SDL) library [9]. It provides library functions such as handling graphics render, sound files and input devices (Mouse and Keyboard), making it an excellent choice for developing a simple 2D puzzle game like Unblock Me. Those backend hander libraries allow the developer to focus on the logic and gameplay mechanics of the game, streamlining the development process and enabling quicker prototyping and testing.'

3.2 Collections (Deque)

The official Python module "Collections"[3] offers the function "Deque," used in this project. "Deque" is a data structure specially designed for effective operations at both ends of the queue, or, to put it another way, it provides quick and effective insertions and deletions from both ends of a queue. The function of "Deque" allows for O(1) time complexity for these operations, making it more efficient than the standard list, which requires O(n) time when popping elements from the front due to the need to shift all remaining elements. Therefore, Deque is ideal for handling search algorithms like BFS, where the algorithm requires frequent and efficient popping of the first element in the queue.

3.3 copy (deepcopy)

The official Python module "copy"[4] provides the function ".deepcopy" which is used in this project. A "shallow copy" only copies reference to those compound objects, so it will also affect the original object since both the original and copied objects share the same references to those compound objects. However, a "deep copy" creates an entirely new object, and all objects nested within it; this further ensures that any modifications made to the deep copied object will not be affected by the original game, making it ideal for saving out states without altering the initial state.

3.4 heapq

The official Python module "heapq"[5] provides an efficient way to implement heaps, also known as priority queues. Heapq focuses on managing a collection of elements, maintaining the order based on their priority, and allowing the smallest or largest element to pop out quickly. Transferring the logic into this project, the heapq module manages collections of states based on their priority, allowing the algorithm to continuously process the state with the lowest path cost.

What makes the function heapq effective is its *O(log n)* time complexity for insertion and

deletion operations, compared to the $O(n)$ time complexity of basic list-based sorting. This means the algorithm remains efficient and quick to make decisions, making it critical in algorithms like Dijkstra's, Greedy's, and A* search, where determining the next best state to explore based on cost is essential.

Chapter 4: **Project Progress Overview**

4.1 Game Rules

Unblock Me is a simple yet challenging puzzle game where the player needs to pave a path in a 6x6 grid for the red block to exit the board; the board consists of various blocks with different lengths, and the sizes are usually 2x1 or 3x1 long and can be moved either horizontally or vertically depends on their orientation. The player must move the yellow blocks out of the way while limited by the small space within the grid: horizontal blocks can only move left or right, and vertical blocks can only move up or down. The game challenges the player to find the most efficient sequence of moves with the fewest possible moves, as the game becomes progressively more problematic as the number of blocks and complexity of their arrangement increases.

4.2 Gameplay Mechanics(blocks)

- **Starting the Drag:**
  - When the player clicks on a block, it is detected as a Pygame. Rect acts as a hitbox, and the start_drag() function checks if the mouse is hovering over the block. This is determined by comparing the mouse position, then the grid_x and grid_y are calculated based on the tile size and the block's position and size. If the mouse is over the block, the dragging flag is set to True, allowing the block to be moved.
- **Checking Move Validity:**
  - The update_position() function verifies the move by calling is_move_valid(), checking whether the block stays within the board's bounds, and both orientations of blocks check whether there are no obstacles in their movement.
- **Dragging and Updating Position:**
  - While the player is dragging the block, the update_position function constantly updates the block's position based on the current mouse position. Then, the mouse position is converted into grid coordinates again using grid_x and grid_y. Lastly, the block's position is updated to the new grid coordinates and rendered out on the terminal and UI if the move is valid (see figure 6.4).

4.3 Software Engineering Aspects of the Unblock Me Project

- **Modular Design:**
  - By splitting the code into several files and functions, each responsible for handling and acting as a function of the game, the project becomes more organized, maintainable, and easier to extend or use in future development stages. For instance, the "main.py" handles the core game execution and transitions the renders between game states, the "board.py" manages the grid and block positions, and "block.py" defines block properties and movements, and so on. This approach ensures the reduction of errors and makes the code easier to navigate over time.
- **Object-Oriented Programming (OOP):**
  - The project implements object-oriented programming principles, storing the game attributes and logic in classes. For example, in this case I have designed blocks are represented by a "Block" class with attributes such as position, orientation, color, size, and the block's ID. OOP allows the game to be more intuitive by mapping the attributes into an object, making the logic easier to follow and understand.
- **Version Control:**
  - Git is mostly used to track changes, manage branches, and collaborate effectively during the development stage.
  - Using version control systems like GitLab, the project ensures changes and commits are tracked, allowing the developer to create a feature branch to develop new functionality for the project without interfering with the main build.
- **User Interface (UI) and Usability:**
  - My game‘s UI is designed with simplicity, readability and accessibility, aligning with the user-centered design principles. Using Pygame's rendering capabilities for the buttons, text and the animation/effects, the interface is designed into user-friendly and visually appealing.

    One key aspect of accessibility is the adjustable settings that I put in the setting.py. These settings allow the customization of elements such as the HUD and resolution size without causing issues. Issues like rendering prevent the button from shifting or overlapping with other buttons.

4.4 Implementation

The project adopts a modular programming approach, with the code and functions diverged into multiple Python files, and below is an explication of the milestones achieved through this term one and the purpose of each file:

```
+---Assets
|   +---effects
|   |   \---audio
|   |       |---block_move_sound.wav
|   |       |---button_click_sound.wav
|   |       |---win_sound.wav
|   |       |---window_xp_logoff_sound.wav
|   |       |---window_xp_logon_sound.wav
|   +---experimental
|   |   +---A_Star
|   |       |---A_Star_maze.py
|   |   +---BFS
|   |       |---BFS_maze.py
|   |   +---DFS
|   |       |---DFS_maze.py
|   |   +---Dijkstra
|   |       |---dijksta_maze.py
|   |   \---Greedy
|   |       |---Greedy_maze.py
|   +---font
|   |   |---PressStart2P.ttf
|   \---photo
|   |   |---example_levels.png
|   |   |---icon.png
|   |   |---Sound_effect_making_1.png
|   |   |---Sound_effect_making_2.png
+---Documents
|   |---Final_year_project_planning_100999504.pdf
\---Game
|   |---main.py
|   \---src
|   |   |---block.py
|   |   |---board.py
|   |   |---function_buttons.py
|   |   |---levels.py
|   |   |---menu.py
|   |   |---setting.py
|   |   |---setup.py
|   |   |---solver.py
```

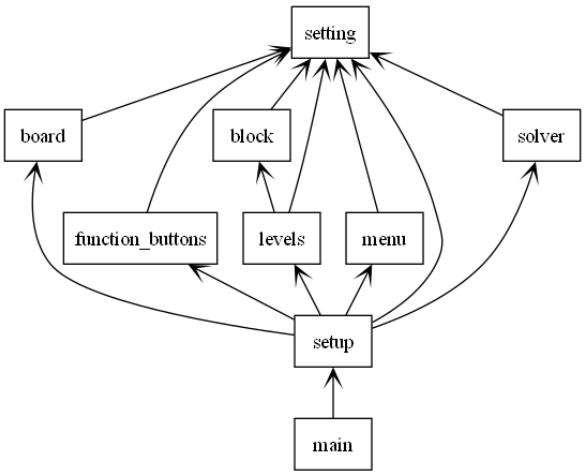


Figure 4.1: General view of variables are being called

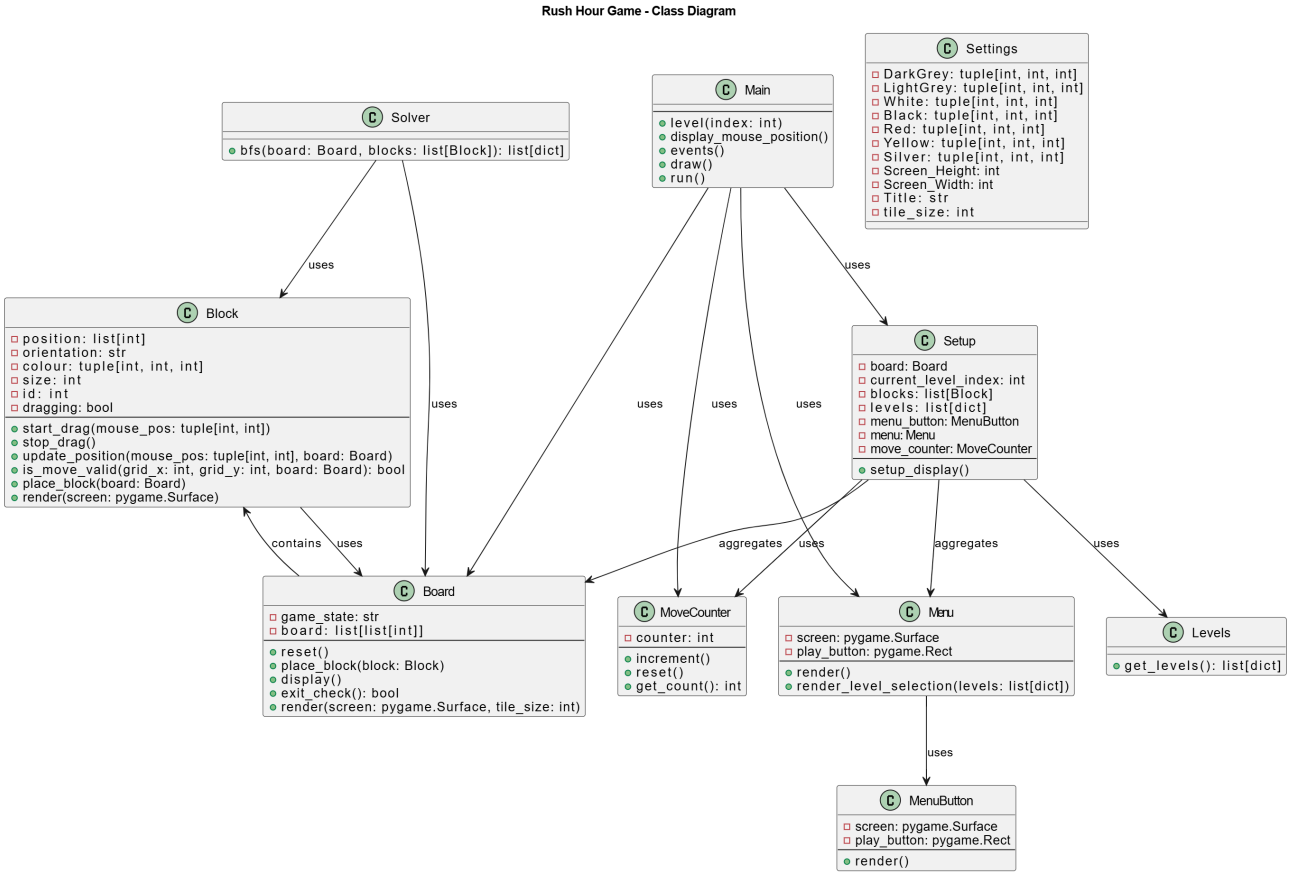


Figure 4.2: UML of the project

- **Milestones Achieved**
  - In this term, I have developed a fully functional game with a stable UI and audio effects that hope to provide a flawless user experience. On the other hand, also conducted research and experiments on the selected algorithms using a terminal-based maze layout, which served as a more straightforward and more intuitive starting point for understanding their behavior. (Detail in Chapter 6)
- **main.py**
  - This file serves as the central controller for the game, manages the main game loop, and handles the render transition between different game states such as menu, gameplay, and winning screens.
  - **def level(index):**
    - The level(index) function is responsible for loading and setting up the game level based on the provided index from another function.
  - **def display_mouse_position():**
    - The display_mouse_position() function is designed to print the mouse cursor's position relative to the game's grid. By dividing the cursor's coordinates by the tile size, it calculates the corresponding grid_x and grid_y positions, which are displayed to allow me to debug.
  - **def events():**
    - The events() function handles all the user input and game events, such as closing the window, handling mouse clicks, moving blocks, and transitioning between game states.
    - When the user clicks the window close button, the function plays a logoff sound, waits for 2 seconds for the audio cue to finish, and then closes the game.

```
    if event.type == pygame.QUIT:
        window_xp_logoff_sound.play()
        pygame.time.delay(2000)
        pygame.quit()
        quit(1)
```

* When the user clicks their left mouse button, the function will check if the game state is equal to stage 0(the main menu). It will also reset the move counter and transition to stage_0_1(the level selection screen) when the mouse button hovers and the "play button" hitbox is clicked. Then, in the stage_0_1 state, it will fetch all the existing data and levels from setting.py and automatically render every level out. It checks which level the user selects and loads that level, then switches to stage_1(gameplay state), ensuring smooth and uncluttered navigation between different parts of the game. (See Figures 6.4)

```
if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:
    mouse_pos = event.pos
    if board.game_state == "stage_0":
        # Reset counter for move
        move_counter.reset()
        if menu.play_button.collidepoint(mouse_pos):
            # Transition to level selection
            button_click_sound.play()
            board.game_state = "stage_0_1"

    elif board.game_state == "stage_0_1":
        for index in range(len(levels)):
            level_rect = pygame.Rect(Screen_Width // 2 - 100,
               Screen_Height // 4 + index * 50 - 20, 200, 40)
            if level_rect.collidepoint(mouse_pos):
                current_level_index = index
                # Load the selected level
                level(index)

                # Starting the game by switching to stage_1
                button_click_sound.play()
                board.game_state = "stage_1"
```

* This section of code handles the mouse and keyboard interaction for the gameplay. When the user presses the left mouse button, they can either drag blocks or return to the menu, then ensure that it resets the board after pressing the menu button. While dragging, block positions are updated dynamically based on the mouse movement, like up or down (horizontally or vertically). Upon releasing the mouse button, the code checks if a block's position has changed, plays a sound cue, and then lastly increments a move counter and renders for what has changed. I have also added debug functions like pressing the "0" key to display the mouse's position on the grid and pressing "s" key to the solver function.

```
                    if board.game_state == "stage_1":
                        if menu_button.play_button.collidepoint(mouse_pos):
                            button_click_sound.play()
                            # Reset the board
                            board.reset()
                            board.game_state = "stage_0"

                        for block in blocks:
                            # Saving up the initial position
                            block.initial_position = block.position[:]
                            block.start_drag(mouse_pos)

                elif event.type == pygame.MOUSEMOTION and board.game_state
                   == "stage_1":
                    # Moving the block
                    for block in blocks:
                        mouse_pos = pygame.mouse.get_pos()
                        block.update_position(mouse_pos, board)

                elif board.game_state == "stage_1":

                    if event.type == pygame.MOUSEBUTTONUP and event.button
                       == 1:
                        # Create a loop for when the block is moved

                        for block in blocks:
                            block.stop_drag()

                            # Check if the block moved
                            if block.position != block.initial_position:

                                block_move_sound.play()

                                move_counter.increment()
                                print(f"Block␣moved␣from␣{block.
                                   initial_position}␣to␣{block.position}")
                                board.display()

                    if event.type == pygame.KEYDOWN and event.key == pygame.
                       K_0:
                        # Excuse check mouse position
                        display_mouse_position()

                    elif event.type == pygame.KEYDOWN and event.key ==
                       pygame.K_s:
                        # Solver stage
                        board.game_state = "stage_1_1"
```

- **main.py**
  - **draw():**

    - This function allows the visual rendering to work in different stages; for instance, stage_0 presents the main menu, where the player can start or navigate the game. stage_0_1 is the level selection screen, allowing the player to choose a specific level. stage_1 is the primary gameplay stage where the puzzle-solving occurs. stage_1_1 is the solver stage executes the solver functions. stage_2: The winning stage, which congratulates the player upon successfully solving the puzzle.
  - **run():**
    - This function ensures the game operates smoothly by continuously managing two core components: the events() and draw() functions.

- **setup.py**
  - **draw():**
    - Setup.py is used to initialize the game environment and set up all necessary components, such as importing the required libraries and modules, font configuration, and sound effects. This sets up the game's foundation, preparing all kinds of essential elements before the main loop begins.

- **setting.py**
  - This code section sets up a collection of reusable colors and game-specific settings to simplify implementation and ensure consistency throughout the game. The color definitions, for instance, DarkGrey = (40, 40, 40) and White = (255, 255, 255), serve as standardized variables to represent RGB values, making it easier to apply consistent styling to game elements.
  - The game-specific settings, such as the screen dimensions (Screen_Height = 800 and Screen_Width = 600), demarcate the overall window size for the game. Those two dimension variables are also used to define the coordinates for how the game object is being rendered. Using this approach ensures flexibility as if the user decides to resize the game, all associated objects and buttons will adapt their positions dynamically, preventing misalignment or objects from going out of place when only using fixed coordinates.

- **menu.py**
  - **class Menu:**
    - The menu class sets the necessary elements for the game's interface, such as buttons for the main menu and level selection. (See Figures 6.1 and 6.2) Then, setup.py will call out the class again so that the main.py can use the Menu class to create the menu button. Menu class uses the Pygame library to dynamically render the visual components on the game screen, creating buttons for interaction and drawing the text labels. For instance, in stage_0, the render function displays the main menu with options like play and exit, while in stage_0_1, the render_level_selection function dynamically generates and positions buttons for the available levels. This modular design allows the menu system to respond flexibly to game state changes and user actions.

- **levels.py**
  - **def get_levels():**
    - This defines and stores the game-level data, each containing a list of blocks placed on a board. Each block is instantiated from the Block class with specific parameters such as position, orientation, color, size, and a unique block ID. The function is stored by creating the first level with a single horizontal red block and progressively adding more levels with increasing complexity, each defined by a dictionary where the "blocks" key holds a list of Block objects. Lastly, the levels are appended to the list and returned at the end of the function.
- **board.py**
  - **board class:**
    - The Board class defines the structure and behavior of the game's board. It is set up with a 2D array representing the board, where each cell can hold a value indicating whether it is empty (0), occupied by the red block (1), or occupied by other blocks identified by their unique IDs(n+1). This class is crucial for managing the game state, visualizing the board, and checking win conditions.
  - **reset():**
    - A reset function has been added. It ensures that the board is reset to its initial state, deep copying to return it to empty(0)s.
  - **place_block():**
    - The place_block method updates the board array with the block's position and size based on its orientation and color.
  - **display():**
    - The display method prints the board state to the console, which helps with debugging and future development.
  - **exit_check():**
    - Method checks if the red block is at the designated exit location.
  - **render():**
    - The method uses Pygame to draw the board to the screen, coloring the tiles based on their value and rendering grid lines.
- **block.py**
  - **Block class:**
    - Firstly, the block class defines the the behavior and properties of each block within the game. Each block is initialized with attributes such as its position on the grid, orientation (horizontal or vertical), size, color, and a unique ID. Those attributes can be found in level.py, as it is mentioned earlier. Additionally, both of the start_drag and stop_drag methods handle the interaction of a block with the mouse, turning the dragging state on and off based on the mouse's position relative to the block. In short, This class plays a crucial role in both gameplay mechanics and visualization.
  - **update_position():**
    - The method is the most critical part of the block movement, as it calculates a new grid position from the mouse's coordinates and validates the move to ensure it stays within bounds, does not overlap other blocks, and adheres to the block's orientation constraints. Furthermore, I have included special

handling for the red block to reach the exit by allowing only the horizontal red block to move through the coordinate (5, 2). If the move is invalid, the block reverts to its previous position on the board.

- **is_move_valid():**
  - The function checks for collisions by verifying that the path is clear of obstacles for both horizontal and vertical movements. If the move passes all of the checks, the block's position updates and then its new coordinate location is marked on the board. Otherwise, the place_block method ensures the block is re-placed at its original location in the grid.
- **render()**
  - This method is responsible for the visual representation of each block. Like the other rendered objects, Pygame draws the block on the screen based on its current position, size, and orientation, adding a distinct outline and adjusting the color for visualization.

- **functional_button.py**
  - **MoveCounter class:**
    - This file is for codes that are for the game UI procedures like the MoveCounter class is designed to keep track of the number of moves made during gameplay. The increment method increases the count by one each time it is called, reflecting a move(n+1), and then the reset method resets the counter back to zero, this reset function is called when starting a new game or returning to the main menu, ensuring the counter goes back to 0. Lastly, The current count can be retrieved using the get_count method, allowing the game to display the rendered counter or use the move count in other features in future development.
    - The MenuButton class creates a button to return to the main menu. It is initialized with the screen object, and the button's position and dimensions are defined using the "pygame.Rect" function. Lastly, the render method is responsible for displaying the button on the screen as it fills the screen with a dark grey background, draws the button as a silver rectangle, and overlays it with the word "menu" in black text. This Menu button is a critical component of the interface because it enables users to navigate back to the menu by changing the game state back to 'state_0' after it is pressed.

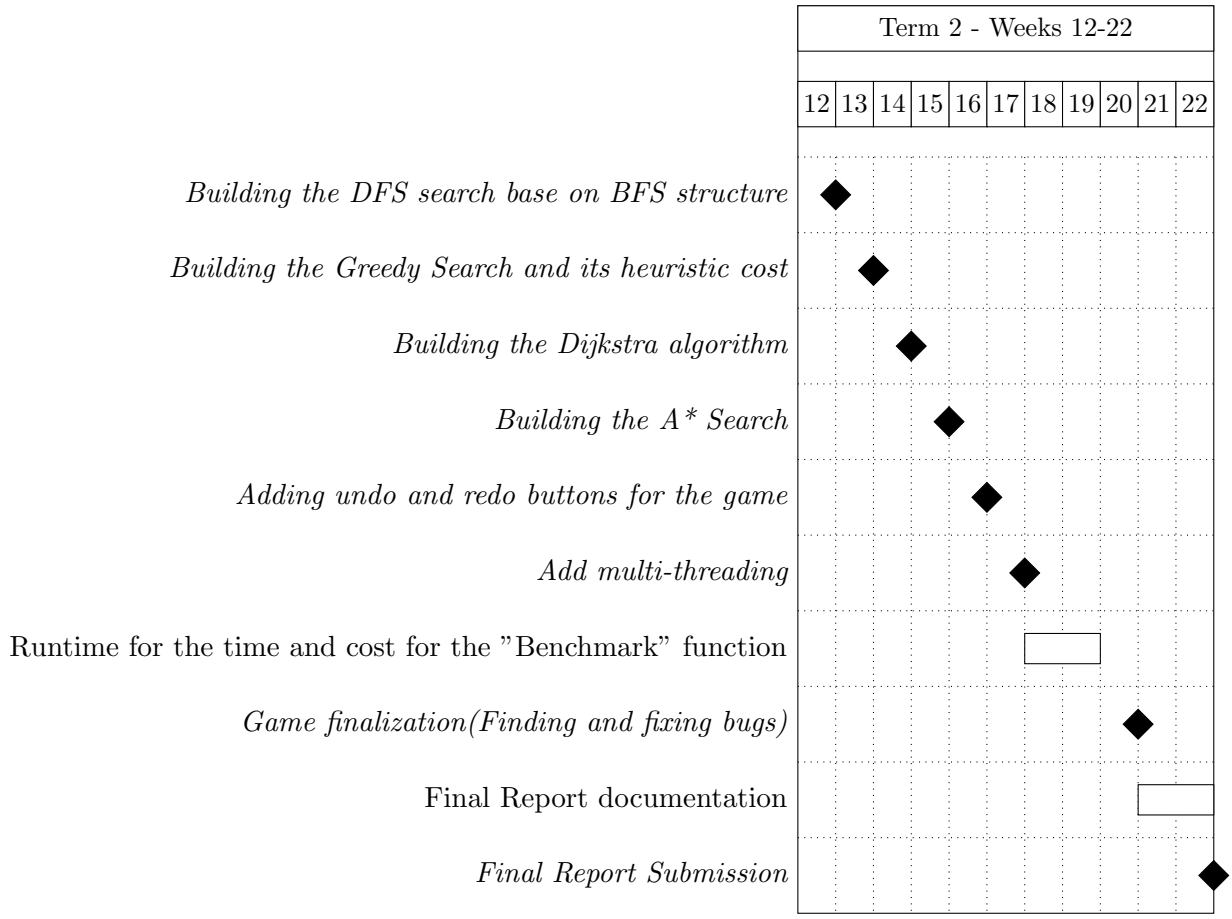
- **solver.py**
  - **bfs(board, blocks):**
    - The first solver I implemented into the game is the Breadth-First Search (BFS) algorithm. Firstly, the state space consists of the Board state, Block positions, and sequence of moves. In order to ensure that the original game state remains unaltered, I used ".deepcopy" to copy the board and block configurations to ensure it explores every possible move. Then, the algorithm tracks visited states to avoid redundant computations, storing them as hashed board representations. I used hashed board representations instead of using entire non-hashed board configurations comparison directly because it is computationally expensive, and it will slow down the computerization as the game's size and complexity increase according to higher levels. Additionally, The BFS queue uses the function dequeue(O(1)) to manage the exploration of the state's process, where each entry contains the current sequence of moves, the board state, and the block positions.

* In the loop, the algorithm dequeues a state, then it will checks if the red block has reached the exit and returns the list of moves if successful. On the other hand, every block have three different checks, its valid movement, stays within the 6x6 grid, and does not collide with other blocks, then it attempts to move either forward or backward based on its orientation. Once it passes all the checks, every valid move creates new copies of the board and block configurations, applies the move, and updates the queue with this new state, provided it has not been visited before. Lastly, those steps ensure that the BFS algorithm evaluates all possible states, guaranteeing it finds the shortest path to the solution if one exists.

Chapter 5: **Term 2 Plan**

5.1 Other algorithm for solver.py

In the next term, I will be focusing on implementing Depth-First Search (DFS), Greedy Search, Dijkstra's, and A* Search Algorithms. After building on the research and knowledge I have gained this term, I now have the basic foundational understanding necessary to develop algorithms for my project. I aim to finish developing all of the solver algorithms in the future of term two within 4 weeks.



5.2 Other functions

I aim to add multithreading to my game to counteract performance challenges because some of the tasks are computationally intensive and involve waiting. A notable example is when the BFS solver is being processed, it often causes a lag in the visualization part. I have tried using macOS and Windows, and the results have highlighted distinct differences in how each handles the issue. On Windows(See Figures5.1), the system will turn to a semi-translucent white blink screen after approximately three seconds of lag when the game tries to process 80+ moves level. In contrast, the macOS(See Figures5.2) system does not have this problem, and it can successfully show the whole visualization of the game being solved smoothly when solving the same complex level. This problem underscores the need for a solution like multithreading, which can ensure consistent performance and prevent this "lag" problem from ever happening on different platforms.

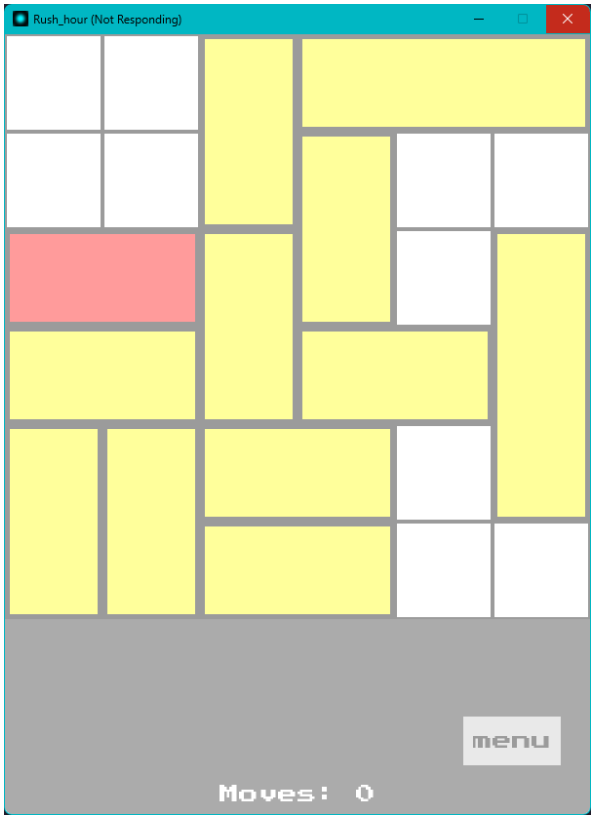


Figure 5.1: Lag handling in Windows

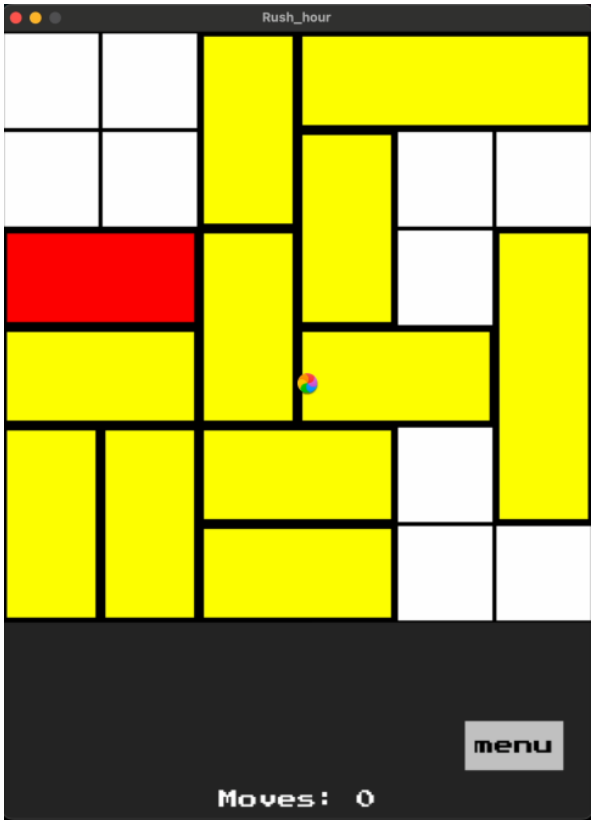


Figure 5.2: Lag handling in MacOS

Chapter 6: **Demo reference**

Here is the video for the demo: https://youtu.be/W16QiZogscs

Figure 6.1: Menu screen

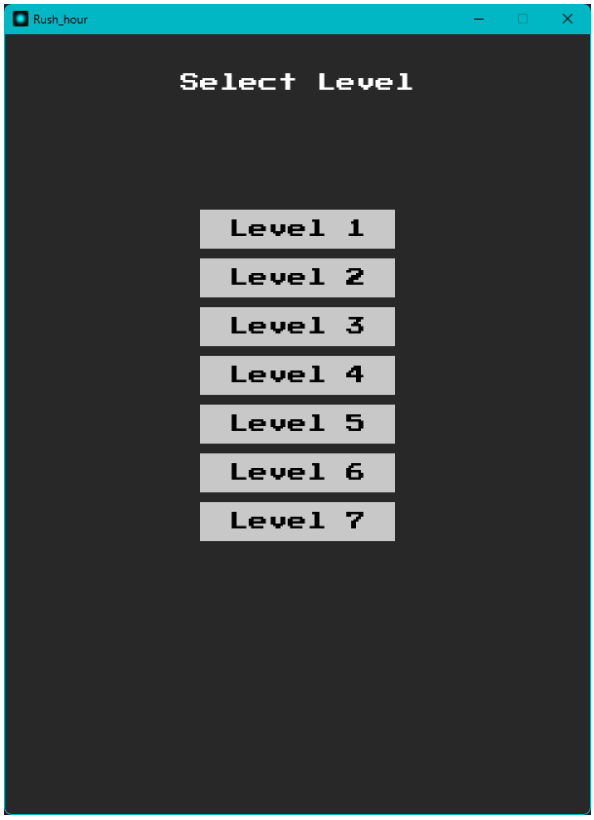


Figure 6.2: level screen after pressed the play button

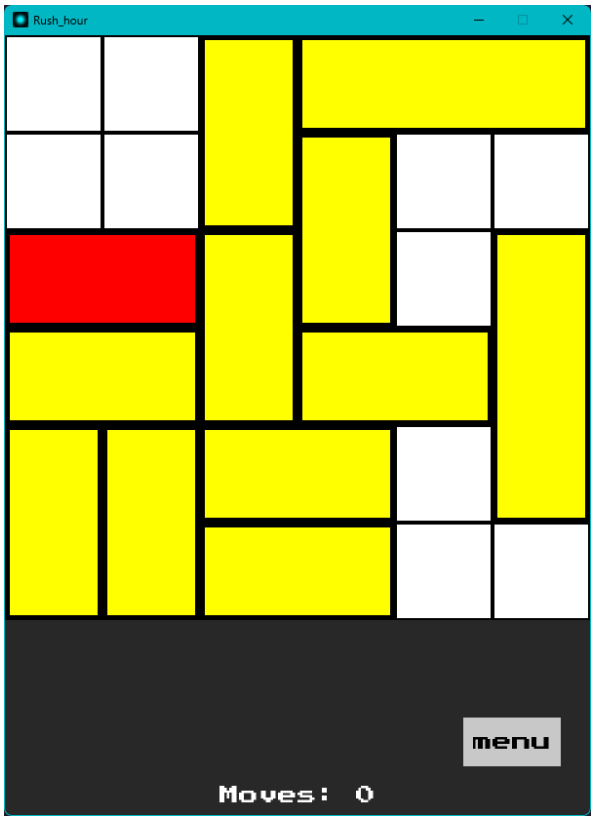


Figure 6.3: Using level 7 as the example

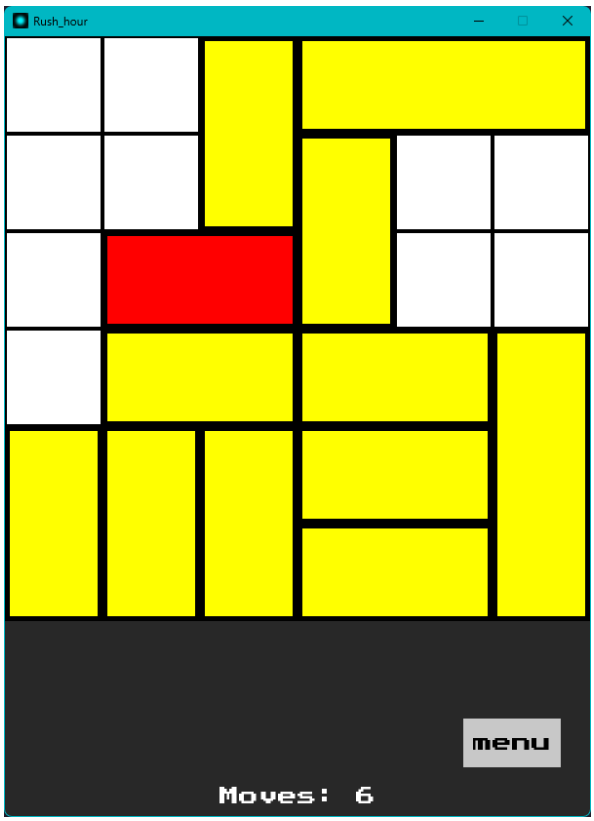


Figure 6.4: Counter function and how the block are moved

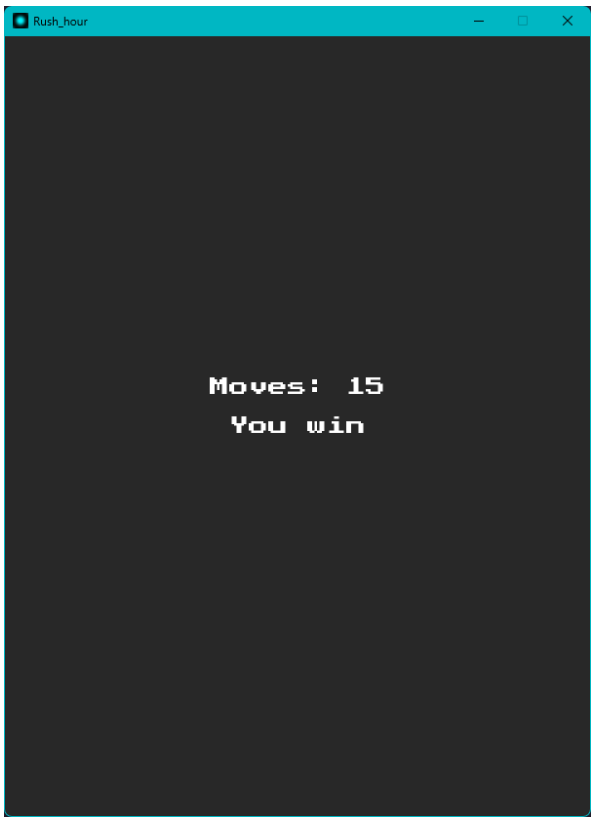


Figure 6.5: Winning screen

Chapter 7: **Diary**

```
# Diary of my final years project

I will be documenting my own work in the following diary. I have also divided
the thing that i did, in each commits to make it more clear.

## 4/10/2024
- Created the basic files for the rush_hour game.
- Created the base function of a windowed app. (Properly closing and opening the game)
- Added some setting for the app
- Added diary.md and I have created a introduction for my initial release.
- Added requirements.txt for the easy and simple download, wrote down the installation
instructions for the future users.
#
- Implementation of the basic code for the pygame to run.
- Implementation of the basic code for the game to work in console first.
#
- Created the basic functions like grid rendering, and winning message for the game.
- Tested if the red car passed the exit and will it appear the winning message and
update the game state act as a soft lock.
## 7/10/2024
- Created the function of the block.py, which contain drag function, their data
position/orientation/colour/size and how they are being rendered

## 8/10/2024
- I figured that due to my function making the block can only move within 6x6 grid.
And i needed my red block to move past the array [5][2] as right now when the red block
moved to [5][2] it counts as victorious.
we doesnt want that to happened, so I recreated the array, making the red block
accessible to [6][2] and fixed some code to expand the grid beyond 6x6, making it
look more appealing.

- I thought of a way to make the game more interesting by adding multiple levels.
It will also give the game a "Gameboy-like" appearance.

## 9/10/2024
- Added icon for the game which i created it in photoshop.
- Fixed some of the code's structures.


## 16/10/2024
- Added levels.py for the game. Now we can design and run different levels
for the menu selection(WIP)
- In order to solve the confusion for each blocks, I added black margin
outlines for each blocks. Changed the way how they being rendered out.
#
- Changed the way hows it going to be printed out in the board terminal
#
- I have found example levels on the internet from
"https://www.reddit.com/r/puzzles/comments/17j3cpj/hardest_rush_hour_puzzle_for_every_pos
sible/"
```

```
#
- Added array/coord checker for me to easier to manually adding the blocks
for each levels
# 17/10/2024
- Yesterday, when i was trying different levels that i have built for the game,
I figured there are some kind of collustions error that causing a block
to "jump" through another block which was blocking the dragging one.
- Example: From this dragging the red block("R R") to the array [2, 4].
‘‘‘
=============
=X X Y Y Y Y=
=X X Y Y X X==
=X R R Y X X X
=Y Y Y Y X Y==
=Y Y Y Y Y Y=
=Y Y Y Y Y Y=
=============
‘‘‘
- As you can see it "jumped" over the the yellow block.
‘‘‘
=============
=X X Y Y Y Y=
=X X Y Y X X==
=X X X Y R R X
=Y Y Y Y X Y==
=Y Y Y Y Y Y=
=Y Y Y Y Y Y=
=============
‘‘‘
- I was spending so much time in the wrong direction, at first i thought
in the first check for the block to move around in their direction
have the wrong logic
‘‘‘
if self.orientation == ’h’:
                # Horizontal blocks can only move horizontally
                if 0 <= grid_x <= len(board[0]) - self.size and grid_y ==
                self.position[1]:
                    if self.is_move_valid(grid_x, grid_y, board):
                        print("1")
                        valid_move = True
            elif self.orientation == ’v’:
                # Vertical blocks can only move vertically
                if self.position[0] == grid_x and 0 <= grid_y <=
                len(board) - self.size:
                    if self.is_move_valid(grid_x, grid_y, board):
                        print("2")
                        valid_move = True
‘‘‘

However, i later figure out the error is located in the
‘‘‘
    def is_move_valid(self, grid_x, grid_y, board):
        if self.orientation == ’h’:
            for i in range(self.size):
```

```
                if board[grid_y][grid_x + i] != "X":
                    return False
        elif self.orientation == 'v':
            for i in range(self.size):
                if board[grid_y + i][grid_x] != "X":
                    return False
        return True
```
```
With the latest check logic the blocks can not be phasing through
other blocks even there are spaces left for them to fit in.

# 18/10/2024
- Create a menu for the game, now the game have a function basic
menu for the level selection, still work in progress.
- Dynamic levels selections, not hard coded.
- Changed event() function for a more logical and better
code structure.

#
- Commented the debug codes, will remove dead code later on.
- I have sampled a few founds for the game.

# 21/10/2024
- Supervisor has suggested the material related
to my work https://aima.cs.berkeley.edu/

# 25/10/2024
- Conducting experment on DFS, array Maze solving is used
to let me understand more about it.
- It is used data structures known as **"Stack"**.
- A "stack" is a data structure used for the collection of
the objects and based on the principle of **"LIFO"**
known as **Last In First Out**.
- Now in a maze there are North, East, South, West.
- Marking the inital point as visited.
- Push current position to the stack.
- Pop() last element from the stack out.
- Check each neighboring cell in the North, East, South
, and West directions.
- If a neighboring cell is open and unvisited, mark it as
visited, push it onto the stack, and continue.
- If the neighboring cell is the maze exit, the solution is found.
#
- In this case what is DFS
- If a path is blocked or no unvisited neighbors, DFS
automatically backtracks by popping the stack, "reversing"
the path and trying alternative routes until exit is find.

# 29/10/2024
- Conducting experment on BFS, array Maze solving is
used to let me understand more about it.
- It is used data structures known as **"Queue"**.
- A "Quene" is a data structure used for the collection
of the objects and based on the principle of **"FIFO"**
```

```
known as **First In First Out**.
- Now in a maze there are North, East, South, West.
- Marking the inital point as visited.
- Push current position to the quene.
- Pop(0) first element from the quene out.
- Check each neighboring cell in the North, East, South
, and West directions.
- If a neighboring cell is open and unvisited, mark it
as visited, enqueue it, and continue.
- If the neighboring cell is the maze exit, the shortest
solution path is found, as BFS explores layer-by-layer.
#
- Path tracing in BFS
- Ensuring the shortest path by exploring all possible
nodes at the current "layer" before moving on to nodes at the next layer.
- Therefore, the first time the exit is reached, it will
be by the shortest path.
#
- I will be using ‘collections‘ library for the ‘deque‘
in Python instead of **pop(0)**, they work the same way
but **pop(0)** is inefficient because it requires
shifting all remaining elements one position to the
left, resulting in an **O(n)** time complexity. ‘Deque‘
, on the other hand, is more optimized for appending
and popping from both ends with **O(1)** time complexity.

# 1/11/2024
- Dijkstra’s algorithm
- It ensures that the shortest path is found
- Dijkstra’s algorithm requires each cells act like
"node" in a graph and it searches for the shortest
path with the lowest costs
- **Priority Queue** implemented as a min-heap,
keeps track of cells with the lowest accumulated costs.
This allows the algorithm to expand the lowest-cost paths first.


# 8/11/2024
- Greedy search
- ‘Manhattan distance‘ as the heuristic

# 9/11/2024
- Changed the whole grid from string format into integer
format. It was suggested by the supervisor.
- Edited how the output in the terminal is shown.

# 10/11/2024
- Created ‘Greedy search‘ in maze format
- Just uses the heuristic function: ‘f(n) = h(n)‘
- ‘Manhattan distance‘ estimates how far a point is from
the goal, by summing the absolute differences in ‘row‘
and ‘column‘ indices
- Once again, ‘priority queue‘ (min-heap) to ‘store‘
and ‘retrieve positions‘ based on their heuristic values
```

```
- Going to do A* next

# 11/11/2024
- Creating the ‘A* search‘
- Using the heuristic function: ‘f(n) = g(n) + h(n)‘

# 16/11/2024
- Created the ‘A* search‘
- Estimateing the distance between the current
position x, y and the goal
- Then sum of the absolute differences of row and
column indices
- Priority Queue is to maintain a min-heap based on
‘f_cost = g_cost + heuristic‘
- g_cost means cost to reach the current node
- f_cost means the estimated total cost

# 17/11/2024
- I tried to create a "path" that stores every
infomation for the algorithms to work

‘‘‘
File "d:\projects\Year_3_Program\Final year assignment
\project\rush_hour_final_year_project
\PROJECT\Game\block.py", line 97, in is_move_valid
    if board[grid_y][i] != 0:
       ~~~~~~~~~~~~~
TypeError: ’Board’ object is not subscriptable
‘‘‘

- This error ocurred when I am trying to make the def path,
later found out that i forgot to index into the board
object like a list
- but the board object is not behaving like a list or array,
hence it is not directly subscriptable
- from board to board.board it fixed the problem
- Now it can access the array

# 20/11/2024
- After i tired to make the path for the algorithms
to work properly, i figured that saving everything
within a block can not save resources and making it
overcomplicated, instead i will be changing how the
path work in my next commit, which is saving the grid as the path

# 25/11/2024
‘‘‘  File "d:\projects\Year_3_Program\
Final year assignment\project\rush_hour_final_year_project
\PROJECT\Game\block.py", line 101, in is_move_valid
    if board.board[grid_y][i] != 0:
       ~~~~~~~~~~~~~~~~~~~~~
IndexError: list index out of range"
‘‘‘
```

```
- This bug is encountered when i tried to make the
solver for BFS. After fixing is_move_valid()
function, making it unable to go out bounds the problem is fixed.

- Finished working on the BFS algothm solver.

# 27/11/2024
- Added visualization for the BFS solver, for
now whenever pressed the "a" button, it will excuse the function.

# 29/11/2024
- Moved all of the functions other than the main.py to .Game/src files.
- Added a blocky font for the game, the font is found on google font.
"https://fonts.google.com/specimen/Press+Start+2P"
- Added a counter for the game.


# 30/11/2024
- Added a menu button for the game
- Fixing smelly code
- Fixed render errors

# 12/1/2024
- Fixed an error where when it changes back to stage 0
from stage 1 after loading the level, it didn't reset
the board state so it bugged the whole rendering
and gameplay. Now when the user switches back to
stage 0 using the menu button, it will use a
deepcopy of the levels so it won't change the
level.py block coordinates, ALSO I added a
reset block function to ensure that the state is fully reset.
#
- I used Ableton, which is a music software
from Germany, and Audacity, a free and
open-source audio editing and recording tool, to create the game audio.
- Fixed a smelly code
#
- Sound Effects Attribution:
- The Windows XP login and logoff sound
effects are used in this project for
educational purposes only. These sound
effects are copyrighted by Microsoft Corporation
and are not intended for commercial use.
- Credits: Windows XP Login and Logoff
Sound Effects { Courtesy of Microsoft Corporation.

# 12/5/2024
- Fixed more smelly codes
```

Bibliography

[1] H. S. A. H. C. E. Y. B. Badr Elkari, Loubna Ourabah and K. E. Moutaouakil. Exploring maze navigation: A comparative study of DFS, BFS, and A* search algorithms. *Statistics Optimization Information Computing*, 12(3):761–781, Feb 2024.

[2] V. Chugani. What is manhattan distance?, Jul 2024.

[3] Collections — container datatypes — python 3.8.3 documentation. Deques are used for efficient O(1) append and pop operations.

[4] Copy — shallow and deep copy operations — python 3.9.0 documentation. Deepcopy is used in this project for duplicating nested data structures.

[5] heapq — heap queue algorithm — python 3.10.0 documentation. heapq is used in this project for priority queues.

[6] S. Collette, J.-F. Raskin, and F. Servais. On the symbolic computation of the hardest configurations of the rush hour game. In H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers, editors, *Computers and Games*, pages 220–233, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, third edition.* MIT Press, Jul 2009.

[8] M. Fogleman. Michael fogleman, 2018.

[9] S. Lantinga. Simple directmedia layer - homepage.

[10] A. Liu, G. Sicherman, and T. Yoshigahara. *The Puzzles of Nobuyuki Yoshigahara.* Springer Nature, Dec 2020.

[11] X. Liu and D. Gong. A comparative study of a-star algorithms for search and rescue in perfect maze. *2011 International Conference on Electric Information and Control Engineering*, Apr 2011.

[12] M. Malkauthekar. Analysis of euclidean distance and manhattan distance measure in face recognition. *Third International Conference on Computational Intelligence and Information Technology (CIIT 2013)*, 2013.

[13] Pygame. About - pygame wiki, 2019. "Pygame is a set of Python modules designed for writing games. It is written on top of the excellent SDL library."
—Value to the project: Pygame is the main engine for the creation of "Unblock Me" game.

[14] V. N. Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, Apr 1993.

[15] R. Sedgewick and K. Wayne. *Algorithms.* Addison-Wesley, Upper Saddle River, Nj, 2011.