Ludo like UCSC
Assignment report

231046 - P.T. Athsara Fernando

September 1, 2024

Contents

1 Overview

This is the report gives an overall explanation of the LUDO game which has written in C. This report mainly consist of The structures used in the code, Justification fort the used structures and Discussion of the efficiency of the program written.

2 File structure

There are 3 files. `types.h` - consist of the user defined functions declaration and the function prototypes, `logic.c` - consist of all the functions the game is using to run, `main.c` - consist of the main function calling to run the game.

3 The Structures Used

There are two main structs used in the code - struct Piece and struct Player.

3.1 struct Piece

to hold the variables of each piece of each player. This is declared in the types.h and initialized in the logic.c .

3.1.1 structs Piece in `types.h`

```
// Define the Piece struct
struct Piece {
    int pieceid;              // 0, 1, 2, 3 (ID of the piece)
    int current_position;     // 0-51 (Position on the board)
    int base_position;         // initial base position (just a negative value to deviate)
    int traveled_cells;        // number of traveled cells
    int starting_position;     // Starting positions: Y=2, B=15, R=28, G=41
    int home_start;     // Starting positions: Y=2, B=15, R=28, G=41
    int no_captured;           // Number of pieces captured by this piece, initialized to 0
    int clockwise;             // Moving direction (1 heads for clockwise, 0 is tail for cou
    int approach_position;     // Approach cell position: R=26, G=39, Y=0, B=13
    int six_counter;           // Counter for the number of times a piece has rolled a 6, in
    int in_base;               // 1 if still in base, 0 if out of the base
    int land_same_color_cell;
    int capturable;            // Means vulnarable to be captured by another player
    int capture_ability;       // Does have the ability to capture another player's piece
    int is_rolling_piece;
    //int energized;
    //int sick;
    //int kotuwa_paused;
    int finished;
};
```

3.2 struct Player

this is to hold the variables related to each player. the piece struct is also a part inside this. This is also declared in the types.h.

3.2.1 structs Player in `types.h`

```
// Define the Player struct, which contains an array of 4 Piece structs
struct Player {
    char playerid;             // 'R', 'G', 'Y', 'B' (ID of the player)
    int player_rank ;
    struct Piece pieces[4];    // Array of 4 pieces for each player
};
```

4 Justification for the Used Structures

4.1 struct Piece

The `Piece` struct holds all the properties which a piece should hold for each player. These are the list of property variables it hold.

- **pieceid:** each piece has an id of 0-3.
- **current_position:** each piece has an id of 0-3.
- **current_position:** holds the current cell number of a piece (0-51).

- **base_position:** holds a value to represent that a piece is in the base (can be negative values).
- **traveled_cells:** holds the number of cells traveled through the board,
- **starting_position:**
- **home_start:** starting cell number of each piece is stored here.
- **no_captured:** number of pieces captured by each player is stored here.
- **clockwise:** the moving direction of each piece is stored here.
- **approach_position:** approach cell number of each player
- **six_counter:** number of consecetive 6s of of each player is stored here
- **in_base:** identifier for pieces to check if it is in the base or not (0 or 1).
- **land_same_color_cell:** to check if the piece will land on a same color piece cell.
- **capturable:** if this piece is vulnarable to captured is stored here.
- **capture_ability:** this stores if this piece does have the ability to capture another color piece on the board.
- **rolling_piece:** identifier to show the currently moving piece
- **energized:** energized mystery affect of Bhawana is stored here
- **sick:** sick mystery affect of Bhawana is stored here
- **kotuwa_paused:** mystery affect of kotuwa is stored here
- **finished:** check if the piece has arived to the home or not.

4.2 Player Structure

The `Player` struct holds all the properties which a player should hold. Also we create a array for pieces for the Piece struct inside of this.

- **playerid:** sick mystery affect of Bhawana is stored here
- **player_rank:** mystery affect of kotuwa is stored here
- **Piece pieces:** check if the piece has arived to the home or not.

4.3 Player players[SIZE]; Structure

The `Player players[SIZE]` Represent an array where each element is a 'player' structure,This structure allows we to manage multiple players in the game

5 Main Functions used in the game

5.1 initialize_random

The `initialize_random` function sets up the random number generator to ensure that each run of the program produces different sequences of random numbers. It does this by using the current time as a seed, which helps in generating unpredictable values like dice rolls. This initialization is crucial for ensuring that the random elements in the game, such as dice rolls or random events, are not the same every time the program is executed, making the game fair and engaging. The function operates in constant time, O(1), because it only involves setting up the random seed.

5.2 initialize_players

The `initialize_players` function sets up the initial state of the game by positioning each player's pieces at their designated starting locations on the board. It assigns an ID to each player and places their pieces in the correct starting positions based on the game rules. This setup is essential for ensuring that each player begins the game with their pieces correctly placed and ready for play. The function runs in linear time, O(P), where P is the number of players, as it involves initializing each player and their associated pieces.

5.3 take_out_base

The `take_out_base` function moves a player's piece from the base to the board when the player rolls a 6, allowing the piece to start moving. This function updates the piece's status to indicate that it has been brought into play and is now on the board. This action is vital for allowing pieces to begin their journey according to the game rules. The function operates in constant time, O(1), as it involves a fixed number of operations regardless of the number of pieces or players.

5.4 move_piece_forward

he `move_piece_forward` function advances a piece on the board by the number of steps determined by the dice roll. It handles special cases such as moving pieces past the approach cell and wrapping around the board. This function ensures that pieces move correctly according to the game rules, including handling transitions from normal cells to special color-specific cells. The function's time complexity is O(1) because the operations performed are constant-time checks and updates for each move.

5.5 move_piece_backward

The `move_piece_backward` function moves a piece backward on the board by a specified number of steps, which is useful for reversing moves or correcting positions. It updates the piece's position accordingly and ensures that the piece stays within valid board boundaries. This function is essential for handling specific game rules or conditions that require moving pieces in the opposite direction. The time complexity is O(1), as it performs a fixed set of operations for each move regardless of the board's state or number of pieces.

5.6 roll_dice

The `roll_dice` function simulates the roll of a dice, generating a random number between 1 and 6. This function is used to determine the number of steps a piece should move forward or backward on the board. It provides the randomness required for gameplay, ensuring that the outcome of each roll is unpredictable and fair. The function operates in constant time, O(1), as it generates a random number through a single operation.

5.7 land_on_same_cell

The `land_on_same_cell` function checks if a player's piece lands on a cell already occupied by another piece. If so, it handles the interaction between the two pieces according to the game rules, such as capturing or or triggering the land same color cell variable to 1(on) . This function ensures that the game correctly processes situations where pieces occupy the same position, affecting gameplay and strategy. The time complexity is O(1) because the function performs a constant number of operations to check the position and apply the game rules.

5.8 capture_piece

function activates the process of capturing an opponent's piece that lands on the same cell as one of the player's pieces. It moves the captured piece back to its base and updates the game state to reflect the capture. This function is critical for managing the game's rules related to capturing and resetting pieces, adding a strategic element to gameplay. The function operates in constant time, O(1), as it involves a fixed set of actions to handle the capture process.

5.9 coin_toss

The `coin_toss` function simulates the flipping of a coin to make decisions or determine the outcome of certain game events, such as deciding the direction of a piece or which Mystery affect affect on a piece on Bhawana. It generates a random result of either heads or tails, which is used to make a fair and unbiased decision. This function ensures that the game has a random element for decision-making, adding fairness and unpredictability. The function's time complexity is O(1), as it performs a single random generation operation. .

5.10 print_game_state

The `print_game_state` function displays the current status of the game, including the positions of all pieces, player statuses, and any relevant game information. This function provides a visual representation of the game state to the players, helping them understand the current situation and make informed decisions. It is essential for debugging and for players to follow the game's progress. The time complexity is O(P * N), where P is the number of players and N is the number of pieces, because it involves iterating through each player and piece to print their states.

5.11 determine_starting_player

The `determine_starting_player` function decides which player will start the game, typically based on a random process or a predefined rule, such as the player who rolls the highest number on the dice. This function ensures that the game begins fairly and that the starting order is set according to the rules. It is crucial for initiating gameplay and ensuring that all players have an equal chance to start. The time complexity is O(P), where P is the number of players, as it involves evaluating the starting criteria for each player.

5.12 determine_player_order

The `determine_player_order` function establishes the order in which players will take their turns based on specific criteria, such as dice rolls or predefined rules. It organizes the players in a sequence that will be followed throughout the game, ensuring a fair and consistent turn-taking process. This function is important for maintaining the flow of the game and ensuring that players act in the correct order. The time complexity is O(P), where P is the number of players, as it involves sorting or arranging players based on the criteria. .

5.13 check_game_on

The `check_game_on` function verifies if the game is still active and has not ended. It checks conditions such as whether there are any winning pieces or if any players have reached the home. This function ensures that gameplay continues only while the game is still in progress, allowing players to take their turns and make moves until the game concludes. It is vital for managing the game's lifecycle and ensuring proper termination. The time complexity is O(1) as it involves checking a set of predefined conditions to determine the game status.

5.14 mystery_cell_generate

The `mystery_cell_generate` function is responsible for creating or generating mystery cells on the game board. This function randomly selects positions on the board to place these mystery cells, ensuring they are distributed in a way that adds an element of surprise and strategy to the game. It is essential for adding variability and excitement to the game by introducing unpredictable elements. The time complexity is O(N), where N is the number of cells on the board, as it involves iterating through the board to assign mystery cells to specific locations. .

5.15 activate_mystery

Currently Deactivated (within multi-line comments). The `activate_mystery` function handles the activation and processing of mystery cells when a player's piece lands on one. When a piece lands on a mystery cell, this function determines the type of effect or action associated with that cell and applies it to the game. This could include things like changing the piece's direction, moving pieces, or other game-altering effects. The function ensures that the special rules or effects of mystery cells are applied correctly, impacting gameplay according to the game's design. The time complexity is O(1) because it involves checking and applying the effects based on the mystery cell type, with a constant number of operations.).

5.16 play

The `play` function orchestrates the entire flow of a game turn. It acts as the central control mechanism, ensuring that each turn progresses smoothly by calling various other functions as needed. Initially, it determines the current player's turn and rolls the dice to decide the number of steps a piece should move. The function then calls `move_piece_forward` to update the position of the player's piece based on the dice roll. After the move, it checks for any special conditions such as landing on a mystery cell or capturing an opponent's piece by invoking `activate_mystery` or `land_on_same_cell`, respectively. The function also updates and prints the game state using `print_game_state` to reflect the changes. Additionally, it determines if the game should continue or end by calling `check_game_on`. Overall, `play` integrates the game's mechanics into a cohesive sequence, ensuring that all aspects of the turn are executed correctly. The time complexity of `play` is O(1) as it involves a series of function calls and operations that execute in constant time relative to the turn's complexity. .

5.17 run_game

The `run_game` function manages the overall game loop, guiding the game from start to finish. It begins by initializing the game state, setting up the board, and determining the starting player using `determine_starting_player`. It then enters a loop where it repeatedly calls the `play` function for each player's turn, managing the flow of the game turn by turn. After each turn, it checks if the game should continue or end using `check_game_on`. The function ensures that the game progresses correctly, updating the game state and player positions while handling special events and conditions as they arise. If the game ends, `run_game` handles any final actions or declarations, such as announcing the winner. The time complexity of `run_game` is O(n), where n is the number of turns or rounds played, as it depends on the total number of game turns required to complete the game.

6 Discussion of the Efficiency of the program

6.1 Space Complexity

The space complexity of the game is O(1), indicating constant space usage that does not scale with the size of the input or the number of moves. This efficiency stems from the game's use of fixed-size data structures, such as arrays for player pieces and their attributes. Each player has a predefined number of pieces and attributes that are statically allocated in memory. For instance, the game employs a fixed array for player pieces and their positions, which remains constant in size regardless of gameplay progress. Thus, memory consumption remains stable, irrespective of the number of turns or actions performed during the game.

6.2 Time Complexity

The time complexity of the game operations is O(1), which signifies that the execution time for game-related functions is constant and independent of the number of moves or game state changes. This is achieved by using direct indexing into fixed-size arrays and performing a constant number of operations for tasks such as updating piece positions or checking win conditions. For example, each dice roll and piece move involves a constant number of arithmetic and conditional operations, ensuring that the time required for these operations does not vary with gameplay dynamics. Consequently, the game performs efficiently even as it progresses through different states.

6.3 Efficiency Justification

The game's efficiency, characterized by O(1) time and space complexities, is largely due to its reliance on fixed-size data structures and operations. By utilizing arrays with fixed capacities for player pieces and positions, the game avoids the overhead associated with dynamic memory allocation and resizing. This design ensures that both time and space resources are used optimally, as each game function operates within constant time bounds and does not incur additional memory usage based on gameplay progression. The choice of fixed-size structures and operations provides consistent performance, making the game well-suited for environments with limited computational resources and ensuring reliable and predictable gameplay.

7 Issues and Clarifications

7.1 Not all the player behaviours are added

only the player behaviour of the Blue color has added and it is applied for the other players too.

7.2 Block logic is not implemented

7.3 Mystery cell generation is implemented but activation is not implemented

Most of the mystery cell functions are completed but due to in-completion activate mystery function has kept inside multi-line comments to not to activate.

7.4 Only the Winner is printing in the end not the Ranks

In the windows gcc compiler the player rank is also working but due to the ranks generate non expected values in linux gcc that rank printing few lines has been kept inside multiline comments.

8 Conclusion

In conclusion, this report provides a comprehensive overview of the LUDO game implementation, detailing the structure, justification, and efficiency of the code. By utilizing well-defined structures such as `Piece` and `Player`, and maintaining a clear file organization, the game ensures a modular and manageable codebase. The functions employed efficiently handle various aspects of gameplay, from initialization to turn management, with constant time complexity ensuring smooth performance. The fixed-space requirements and constant-time operations contribute to the program's efficiency, making it robust and scalable. Overall, the design choices and implementation strategies outlined in this report reflect a well-structured approach to creating a functional and efficient LUDO game in C.