Ludo-CS Implementation Report

Sahan Gajanayake - 23000481

September 1, 2024

1 Introduction

This report describes the implementation of a Ludo game using the C language. The game is designed for up to four players, each with four pieces.These players name as RED , GREEN,YELLOW and BLUE.The report includes little introduction about game behavior,details on the data structures used to represent the game board and pieces, the justification for choosing these structures, and a discussion on the efficiency of the program.

2 Game Behavior

In this traditional ludo game, First check who's from start the game then after it very player roll dice in their round .when rolling ,if roll SIX, again player can roll dice then after roll SIX that value may be canceled. Every player can cut their opponent pieces,then that piece go to Base, In this game every players looking to enter home, When 4 pieces come home of one player, this player named as Winner.In addition there are more rules in this game.Each player has inherent behavior, So player behave according to their behaviours, Ex: Red (like to capture), Green (like to blocking), Blue (like to Misty shell story), yellow (like to enter home).

3 Structures Used

3.1 Piece Structure

The 'Piece' structure represents a single game piece and contains the following fields:

```
struct Piece
{
    int position;
    int isHome;
    // in the begin os approch shell
    int isBoard;
    // peice in the standed path
    int endAtHome;
    // in the home
    int isXplace;
    int isInblock;
    int direction;
```

```
    /// clockwise - heads[1](1) counterclockwise - tail[2](-1)
    int forToend;
    // far to end
    int cutCount;
    // count of cutted piece by each piece
    float mistryShellPower;
    // 1 mean not get super power or low powers yet
    int mistryBrief;
    // 0 means not brief by  mistry shell
    int mistryPowerStatus;
    // if it is 0 mistry shell not allow
};
```

Explanation:

- `position`: Tracks the current position of the piece on the board. This is necessary to determine the piece's location and movement.Furthermore when land on opponent piece's location (capturing), position of each 2 pieces are necessary.
- `isHome`: Flags whether the piece has reached the home area. This helps in determining if the piece has completed its journey.When rolling dice, player can Decide to whether the piece which is in home, not to move.
- `isboard`: Indicates if the piece is currently on the board or still in the base. This is crucial for movement.When roll SIX if there are any pieces which have isboard = 0 and isHome = 0, it means, this piece can move base to board.
- `endAtHome`: Indicates if the piece is currently on the Home or not.By using this player can decide whether piece is Home or not there for programm haven't check it's position.
- `isXplace`: Indicates if the piece is currently on the X or not.By using this player can decide whether piece is X or not there for programm haven't check it's position.
- `isInblock`: Indicates if the piece is block or not,By using this when program do movement, it can check piece is in block or not.
- `direction`: Indicates, piece direction, as (clockwise or not) when do movement of each piece, this is more use full.

- `forToend`: Indicates, how mach far to finish the game, it 'll be use for check piece whether come to home or not.
- `cutCount`: Indicates, how many pieces have been captured.
- `mistryShellPower`: Indicates, how mach increase or discrete dice value by putting mistry shell(BAWANA)
- `mistryBrief`: Indicates, One situation of by putting mistry shell.
- `mistryPowerStatus`:Indicates, check whether the piece with mistry shell powers or not.

3.2 Block Structure

The 'Block' structure represents a block system (though it is not yet used in the current implementation):

```
struct Blocks {
    int block_1Position;  // Position of block 1
    int block_1Active;    // Block 1 active or not
    int block_1count;     // Number of pieces in block 1
    int block_1direction; // Direction of block 1, can be 1 or -1
    int block_2Position;  // Position of block 2
    int block_2Active;    // Block 2 active or not
    int block_2count;     // Number of pieces in block 2
    int block_2direction; // Direction of block 2, can be 1 or -1
};
```

Explanation:

- `block_1Position`: The position of block 1. Each player can make two blocks at one time, and this indicates the position of block 1.
- `block_1Active`: Indicates whether block 1 is active (1 for active, 0 for inactive).
- `block_1count`: Represents the number of pieces in block 1. This helps to easily calculate moves when dividing the dice roll.
- `block_1direction`: The direction of block 1; can be 1 (forward) or -1 (backward).
- `block_2Position`: The position of block 2.

- `block_2Active`: Indicates whether block 2 is active (1 for active, 0 for inactive).
- `block_2count`: Represents the number of pieces in block 2, similar to block 1.
- `block_2direction`: The direction of block 2; can be 1 (forward) or -1 (backward).

3.3 Player Structure

The 'Player' structure represents a single player and includes:

```
struct Player {
    struct Piece pieces[PIECE_FOR_PLAYER];
    // Array of pieces for the player.
    char color[8];
    // Color representing the player.
    int playerId;
    // Unique ID for the player.
    int mistryPowerStatus;
    // piece of this player has mistryshell powers
    struct Blocks blocks;
};
```

Explanation:

- `pieces`: An array of 'Piece' structures representing all the pieces for the player. This allows easy management and access to each piece's attributes.
- `color`: A string representing the player's color. This is used for display purposes to distinguish players.It named as [R,G,Y,B]
- `playerId`: A unique identifier for the player, which helps in when player use piece, check it group in the running programs. [R = 0, G = 1, Y = 2, B = 3]
- `mistryPowerStatus`: piece of this player has mistryshell powers. when do move program can check whether it has powers or not.

4 Justification for Used Structures

The selected data structures effectively represent the game's needs. The 'Piece' structure would show the compact representation of the status and location of each piece that can be easily updated and checked while the game progresses. The 'Player' structure totally all the pieces and information relevant to each player, making it easy to handle the players further in the game logic.Futhermore finally check whether who win , those data Structures very use full.

- **Modularity**: The use of structures allows for modular and organized code. Each structure use for a specific aspect of the game like check piece status, which pieces can capture like that.With out struct programs has to call pieces one by one.And also structures making the code easier to understand and maintain.
- **Efficiency**: Structures are used to group related data, which improves both memory usage and access speed. Accessing a player's pieces or status involves direct indexing, which is efficient.

5 Efficiency Discussion

The efficiency of the program can be evaluated based on several factors:

5.1 Time Complexity

- **Movement Operations**: The time complexity for moving a piece or performing game actions (like capturing) is defined below,
  - **Cutting function(red)**: Big O($p*n^2$)
    - **Outer Loop**: - Big O(p)

      ```
      for (int index1 = 0; index1 < PLAYERS; index1++)
      ```

    - **First Inner Loop**: - Big O(n)

      ```
      for (int index2 = 0; index2 < PIECE; index2++)
      ```

    - **Second Inner Loop**: - Big O(n)

      ```
      for (int index3 = 0; index3 < PIECE; index3++)
      ```

- **Check Win Condition**: The time complexity for checking if a player has won is linear, $O(n)$, where n is the number of pieces. This involves iterating over all pieces to verify if they have all reached home.

O(n3̂ * m), where n is the maximum number of pieces on the board and m is the number of rounds played before a winner is determined.

5.2 Space Complexity

- **Memory Usage**:Most of the memory usage of the Ludo game code is from the Player structure, which will hold information about the pieces of each player and color and player ID. Assuming the total estimated amount with respect to memory use concerning the game state, it is about 336 bytes. This is a rather minimel amount of memory, and this game should just run fine on any computers.(calculate from using AI)

5.3 Overall Performance

The program is efficient for this game. Operations on pieces and players are executed in executing time, and the game can handle multiple rounds efficiently. The structure of the code allows for quick updates and checks, contributing to the overall performance of the game.

6 Conclusion

The Traditional Ludo game program is efficient due to its modular design and effective data management. The modular approach enhances code readability, maintainability, making the program scalable. The use of appropriate data structures ensures efficient storage and reuse of game information, contributing to the program's overall performances.