Analysis of Ludo CS Game Simulation in C

R. K. Kalana Jinendra
Index number - 23000821

August 31, 2024

Contents

1 Introduction

This report presents a analysis of a C program developed to simulate a multiplayer Ludo CS game. The program models the game board, players, and pieces using custom data structures. This document explores the rationale behind the design decisions, the efficiency of the implementation, and the overall game logic.

2 Data Structures

The program defines several structures that represent the main elements of the game: board cells, players, pieces and mystery cells. These structures are crucial for managing the state of the game and ensuring smooth gameplay.

2.1 Cell Structure

The `Cell` structure represents each cell on the game board:

```
typedef struct{
    unsigned short int id;
    unsigned short int blockSize;
    short int blockId ;
    bool isEmpty;
    bool isBlock;

}Cell;
```

2.1.1 `id`

The `id` member is used for store the value of square location on the board The cell identified in the yellow approach cell is considered zero (0).

2.1.2 `blockSize`

The `blockSize` member is used to store the value of block size that is created by piece.

2.1.3 `blockId`

The `blockId` member is used to store the value of block number that is created by piece.

2.1.4 `isEmpty`

The `isEmpty` member is used to identify either cell is empty or not.

2.1.5 `isBlock`

The `isBlock` member is used to identify either cell is block or not.

2.2 Player Structure

The `Player` structure groups a player's details, including the pieces they control:

```
typedef struct{
    char name[8];
    unsigned short int startingPoint;
    unsigned short int approchPoint;
    unsigned short int onplaying;
    unsigned short int homePieces;
    bool isWin;
    Piece pieces[PIECECOUNT];

}Player;
```

2.2.1 name

The `name` member is used to store player's name as a string.

2.2.2 startingPoint

The `startingPoint` member is used to store player's X cell value.

2.2.3 approchPoint

The `approchPoint` member is used to store player's approach cell value.
The

2.2.4 onplaying

`onplaying` member is used to store number of pieces currently playing in the board which are neither home , base or straight line.

2.2.5 homePieces

The `homePieces` member is used to store number of pieces reached to home.

2.2.6 isWin

The `isWin` member is used to identify either player is win or not.

2.2.7 pieces[PIECECOUNT]

The `pieces[PIECECOUNT]` member is used to store array of pieces controlled by player.

2.3 Piece Structure

The `Piece` structure is used to represent each individual game piece on the board. Below is the definition and a detailed explanation of its members:

```
typedef struct {
    unsigned short int name;
    unsigned short int position;
    short int positionCounter;
    short int wice;
    unsigned short int approchLength;
    unsigned short int strightPosition;
    unsigned short int prevSteps;
    unsigned short int consecutiveStepCounter;
    unsigned short int blockId;
    short int mysteryId;
    float mysteryWeight;
    unsigned short int mysteryRound;
    bool isMystery;
    bool isPlaying;
    bool isStright;
    bool isHome;
    bool canIntoHome;

}Piece;
```

2.3.1 name

The `name` member is used to store number of pieces as an integer.

2.3.2 position

The `position` member is used to store position of pieces. The position identified in the yellow approach cell is considered zero (0)

2.3.3 positionCounter

The `positionCounter` member is used to store number of position relative to the piece.

2.3.4 wice

The `wice` member is used to store direction of piece.If it is clockwice then equal to 1 and if it is counter-clockwice then equal to -1.

2.3.5 approchLength

The `approchLength` member is used to store number of cells has to goes approach cell of that player.

2.3.6 strightPosition

The `strightPosition` member is used to store number of position of home straight line relative to the piece.

2.3.7 prevSteps

The `prevSteps` member is used to store number of previous movement. Used for mystery effect of Kotuwa.

2.3.8 consecutiveStepCounter

The `consecutiveStepCounter` member is used to store count of piece. Used for mystery effect of Kotuwa.

2.3.9 blockId

The `blockId` member is used to store id of the block which is piece in.

2.3.10 blockId

The `blockId` member is used to store id of the block which is piece in.

2.3.11 mysteryId

The `mysteryId` member is used to store id of the mystery which is piece in.

- 1 for Bhawana
- 2 for Kotuwa
- 3 for Pita-Kotuwa
- 4 for Base
- 5 for X of the piece colour
- 6 for Approach of the piece colour

2.3.12 mysteryWeight

The `mysteryWeight` member is used to store effect of mystery cell. Used for mystery effect of Bhawana and Kotuwa.

2.3.13 mysteryRound

The `mysteryRound` member is used to store number of rounds Used for mystery effect.

2.3.14 isMystery

The `isMystery` member is used to identify either player is in mystery effect or not.

2.3.15 isPlaying

The `isPlaying` member is used to identify player is in standard path or is in home straight.

2.3.16 isStright

The `isStright` member is used to identify either player is in home straight path or not.

2.3.17 isHome

The `isHome` member is used to identify either player is win or not.

2.3.18 canIntoHome

The `canIntoHome` member is used to identify either player can go home straight or not.

2.4 MysteryCell structure

The `MysteryCell` structure is used to represent mystery cell details. Below is the definition and a detailed explanation of its members:

```
typedef struct{
    unsigned short int cell;
    short int mysteryId;
    float mysteryWeight;
}MysteryCell;
```

2.4.1 cell

The `cell` member is used to store location of mystery cell generated.

2.4.2 mysteryId

The `mysteryId` member is used to store id of mystery effect generated.

- 1 for Bhawana
- 2 for Kotuwa
- 3 for Pita-Kotuwa
- 4 for Base
- 5 for X of the piece colour
- 6 for Approach of the piece colour

2.4.3 mysteryWeight

The `mysteryWeight` member is used to store effect of generated mystery cell.Used for mystery effect of Bhawana and Kotuwa.

3 Justification for the Used Structures

3.1 Cell Structure

The `Cell` structure is crucial for maintaining the state of the board. It allows the program to quickly check if a move is valid (e.g., if a cell is occupied) and to manage interactions between pieces. The structure is minimalistic, containing only the information necessary for these operations, which helps to optimize memory usage.

3.2 Player Structure

The `Player` structure is for grouping player specific information. This design simplifies the handling of player actions such as checking if a player has won. The array of pieces within the player structure helps that each player's pieces are managed together, facilitating operations like iterating over all pieces of a player.

3.3 Piece Structure

The `Piece` structure is designed to encapsulate all attributes and behaviors of a game piece. By addressing the piece's state (e.g., position, mystery effects, blocks), the program can efficiently manage each piece's status. This design simplifies the handling of piece actions, such as moving pieces. The structure allows for easy extension, such as adding new game mechanics or rules.

3.4 MysteryCell

The `MysteryCell` structure is designed to encapsulate all attributes of mystery cell.By addressing this structure allow to get all behaviours of mystery cell.The structure is minimalistic, containing only the necessary information. Therefore that helps to optimize memory usage.

4 Efficiency Analysis

4.1 Time Complexity

The program is designed to efficiently manage the operations required for gameplay.
The primary operations, such as checking win conditions or player selection, are performed in constant time (O(1)).Because the program uses direct indexing for players.
The secondary operation, such as choosing piece to play takes O(n) time complexity.
Complex operations such as capturing performed in O(n2) or O(n3) time complexity. However it is not a big issue for the program because this program n is always lower than or equal 4.

4.2 Space Complexity

The space complexity is determined by the number of players, pieces, and board cells. Each player has a fixed number of pieces, and the board has a fixed number of cells, resulting in predictable and manageable memory usage. The structures are designed to minimize unnecessary data storage, ensuring efficient use of memory.

4.3 Scalability

While the program is currently designed for up to four players, it can be extended to support more players, pieces, stranded cells, with minimal changes. The data structures are flexible to accommodate additional pieces or more complex game rules, though care must be taken to ensure that time complexity remains manageable for the above increases.

5 Game Logic and Flow

The main game loop is responsible for managing the flow of the game, including player turns, piece movements, and checking win conditions. Key functions include:

5.1 movePiece Function

This function handles the movement of a piece based on the dice roll. It checks if the move is valid and updates the piece's position accordingly. Special conditions, such as mystery effects or blocks are also managed within this function.

5.2 playingPlayer Function

This function determines the next player to play based on the current game state.

5.3 game Function

The `game` function is the main loop that continues until a player wins. It coordinates the setup of the board, the flow of turns, and the overall game state. The function ensures that all players get an equal opportunity to play, and that the game progresses smoothly.

5.4 Handling Edge Cases

The program includes checks for edge cases, such as what happens when a player wins or when no valid moves are available. These checks ensure that the game can handle unexpected situations gracefully, without crashing or entering an invalid state.

6 Conclusion

The C program for this Ludo CS game simulation demonstrates a well-thought-out design, with structures that effectively manage the game's complexity. The program is efficient in terms of both time and space, and the logic is designed. This game can be extend to support additional players or more complex rules.