# Ludo - CS Assignment Report

Manimehalan

September 1, 2024

# Contents

**1 Introduction** **3**

**2 Structures Used** **4**
2.1 Game Struct . . . 4
2.2 Player Struct . . . 5
2.3 Piece Struct . . . 5
2.4 Board structure . . . 5
2.5 Mystery Effects Struct . . . 6
2.6 Priority Structs . . . 6
2.6.1 Red Priority Struct . . . 6
2.6.2 Green Priority Struct . . . 7
2.6.3 Yellow Priority Struct . . . 7
2.6.4 Blue Priority Struct . . . 7
2.6.5 Piece Priority Union . . . 8

**3 Justifications for the Structures** **9**

**4 Functions** **10**
4.1 Initialization Functions . . . 10
4.2 Error handling functions . . . 10
4.3 Helper functions . . . 11
4.4 Game action functions . . . 11
4.5 Behavior functions . . . 11
4.5.1 Player AI . . . 12
4.5.2 Initial piece analysis . . . 12
4.5.3 Individual piece analysis . . . 12
4.5.4 Blockade analysis . . . 12
4.5.5 Piece Importance validation . . . 12
4.5.6 Finalizing the choice of player AI . . . 13
4.6 Display functions . . . 13
4.7 Game loop functions . . . 13
4.8 Endgame functions . . . 14

4.9 Play game function . . . . . . . . . . . . . . . . . . . . . . . . 14

**5 Program Efficiency** **15**
5.1 Time complexity analysis . . . . . . . . . . . . . . . . . . . . . 15
5.2 Space complexity analysis . . . . . . . . . . . . . . . . . . . . . 15

**6 Conclusion** **17**

# 1. Introduction

This report includes detailed description of the code related to Ludo - CS assignment including the structures used in the code, justification for said structures, brief description of the functions used in the program and further discussions on program efficiency.

The program consist of 3 sections of file explained below:

- Header files: Contains types.h and game.h header files
  - `"types.h"`: Contains definitions for all the structs, unions, macros and enums used in the program
  - `"game.h"`: Contains all the declarations for the functions used in the program
- Game logic file (`"game.c"`): Contains the definitions for the functions declared in `"game.h"`
- Main file (`"main.c"`): Contains the execution of the main game function

# 2. Structures Used

The brief descriptions of the structures used will be explained here along with the purpose of these structures. Structures are defined in the "types.h" file

## 2.1 Game Struct

The **Game** struct contains all the variables needed to keep track of various events in the game such as rounds, mystery cell number, player order array, winner order array, etc. This allows for easy manipulation and access of several key parts of the game without needing to pass multiple variables to the functions.

```c
struct Game
{
  int rounds;
  int mysteryCellNo;
  int mysteryRounds;
  int roundsTillMysteryCell;
  int winIndex;
  int order[PLAYER_NO];
  int winners[PLAYER_NO];
  int prevMysteryCell;
} __attribute__((aligned(4)));
```

## 2.2 Player Struct

The **Player** struct represent individual players of the game. It consists of member variables such as startIndex to keep track of the starting point of the player, pieces array which consists of all the pieces of the relavant player and color to indicate the current color of the player.

```
struct Player
{
  int startIndex;
  struct Piece pieces[4];
  enum Color color;
} __attribute__((aligned(4)));
```

## 2.3 Piece Struct

**Piece** struct represents the individual pieces in the game and consists of several member variables to store the state of each piece as it progresses in the game.

```
struct Piece
{
  int cellNo;
  int captured;
  int noOfApproachPasses;
  bool clockWise;
  bool blockClockWise;
  uint8_t reserved : 6;
  char name[3];
  struct MysteryEffects effect;
} __attribute__((aligned(4)));
```

## 2.4 Board structure

The board of the game is represented in a 2-Dimensional array of pointers to `struct Piece` which are initialized to `NULL`. The outer array consists of

52 inner arrays and each inner array is of length 4. This 2D array acts as a one to one simulation of the real standard cells in the board. The inner array represents the maximum number of players that can be in a cell at a time. This allows for easy emulation for game functions such as **blockades**.

```c
struct Piece *standardCells[MAX_STANDARD_CELL][
   PIECE_NO] = {NULL};
```

## 2.5 Mystery Effects Struct

The **Mystery effect** struct represents the mystery effects of a particular piece at a time. Each **Piece** struct has a **MysteryEffect** struct as its member variable.

```c
struct MysteryEffects
{
  bool effectActive;
  bool pieceActive;
  int effectActiveRounds;
  int diceMultiplier;
  int diceDivider;
} __attribute__((aligned(4)));
```

## 2.6 Priority Structs

Priority structs are used in the program to track the priority status of individual pieces during the player AI/behavior phase. This enables for easily tracking of the priority status of a particular piece before deciding to move it

### 2.6.1 Red Priority Struct

```c
struct RedPriority
{
  bool canMoveFromBase;
  bool canFullMove;
  bool canPartialMove;
  bool canAttack;
  bool canFormBlock;
  bool canExitBlock;
} __attribute__((aligned(4)));
```

### 2.6.2 Green Priority Struct

```
struct GreenPriority
{
  bool canMoveFromBase;
  bool isBlockMovable;
  bool canFullMove;
  bool canPartialMove;
  bool canFormBlock;
} __attribute__((aligned(4)));
```

### 2.6.3 Yellow Priority Struct

```
struct YellowPriority
{
  bool canMoveFromBase;
  bool canFullMove;
  bool canPartialMove;
  bool canAttack;
  bool canExitBlock;
} __attribute__((aligned(4)));
```

### 2.6.4 Blue Priority Struct

```
struct BluePriority
{
  bool canFullMove;
  bool canPartialMove;
  bool preferToMove;
  bool canExitBlock;
} __attribute__((aligned(4)));
```

### 2.6.5 Piece Priority Union

Piece Priority union allows for accessing of priority of each piece of each player from a single point. This greatly helps when abstracting away different player behaviors to a singular function since at a time only one player behavior is analyzed.

Piece priority union stores only the pointers to the array which holds the relavant priority structs of **each piece** for a particular player

```
union PiecePriority
{
  struct RedPriority *redPriority;
  struct GreenPriority *greenPriority;
  struct YellowPriority *yellowPriority;
  struct BluePriority *bluePriority;
} __attribute__((aligned(8)));
```

# 3. Justifications for the Structures

The structures used are properly padded and aligned to relavant bytes of memory. The design of the structs in this project emphasizes efficient memory usage and proper alignment to meet system requirements.

The `struct MysteryEffects` is designed with straightforward integer fields, aligned to 4 bytes to ensure efficient access and compatibility with typical system architectures. The `struct Piece` maintains its original integer and character fields, aligning to 4 bytes and including padding where necessary to ensure correct alignment and performance. The `struct Game` and `struct Player` use consistent 4-byte alignment for their integer fields and arrays, ensuring efficient storage and access. Priority-related structs are also aligned to 4 bytes, maintaining clarity and optimizing memory usage. The `union PiecePriority` is aligned to 8 bytes to accommodate pointers on 64-bit systems, ensuring correct alignment and efficient memory usage. Overall, this ensure the structs are well-aligned and efficiently manage memory without triggering any unecessary fetches due to padding.

The `struct Piece *standardCells[MAX_STANDARD_CELL][PIECE_NO]` which represents the board of the game takes up considerable space making the program less space efficient however on modern systems the space allocated is very miniscule that not much impact can be seen. Defining the array in this way also allows for easy lookup of available pieces in the board reducing the need for several conditional checks. This improves the runtime efficiency. Since the array only points to the pointer of the struct Piece, the space taken is only 8 bytes for one element in the array.

# 4. Functions

This section explains about the various functions defined in the `game.c` file and the purpose of these functions. Since the list of functions is very large, several related functions have been grouped under the following sub-categories.

## 4.1 Initialization Functions

The initialization functions include all the functions related to initializing several structures of the game such as:

- `createPiece` function to properly initialize each piece
- `createPlayer` function to create the Player structures
- `initializePlayers` function to initialize the player array
- `initializePlayerOrder` function to initialize the orders of player during the beginning of the game

## 4.2 Error handling functions

Error handling functions primarily deal with handling various errors in the program without causing alarming crash. This is a simplified and very unsophisticated way of handling errors since C does not have a try and catch method to deal with errors. These functions are only used in smaller contexts inside helper functions in this program.

## 4.3 Helper functions

Helper functions contain a lot of functions that assists the game loop in executing several repetitive tasks. These functions primarily do not perform game actions but assist several game action functions in executing the code.

For example: The `int getNoOfPiecesInBase(struct Player *player);` gets the number of pieces of a particular player in base. The returned value can be used by game actions to decide on specific actions or to print the result to the user.

## 4.4 Game action functions

The game action functions primarily contain all the logic related to several actions in the game. Actions such as rolling a dice, moving a piece, teleporting a piece via mystery cells, capturing a piece, forming a block, etc come under this.

Example function:

```
void moveFromBase(struct Player *player, struct Piece *piece, struct
 Piece *cell[PLAYER_NO]);
```

This function moves the particular piece of the player from base to its starting point on the standard cell.

## 4.5 Behavior functions

Behavior functions primarily deal with handling the player AI and deciding to move a particular piece on the board. The behavior functions can be separated into the following categories:

### 4.5.1 Player AI

This is the main function that encapsulates all the logic needed for the player behavior to work. The `moveParse()` function handles this logic and it is abstracted to a single function due to overlap in similar player actions. The player AI operates by doing an initial analysis and then a deep analysis. Finally it grades each piece based on **priority system** which differs for **each player**.

### 4.5.2 Initial piece analysis

The initial piece analysis consists of common functions for all players with slight differences for some specific actions of each player. This check decides whether to continue to deep analysis or directly skip to priority validation based on certain cases (mainly when the piece is at at home-/homestraight, cannot be moved or at the base)

### 4.5.3 Individual piece analysis

This is the starting phase of deep analysis of each piece. Here individual piece's ability to move in the board is analyzed and graded by several functions based on priority system depending on the player type.

### 4.5.4 Blockade analysis

This is the ending phase of deep analysis and this consists of functions that will analyze the blockades of a particular player (if any exists). Similar to individual piece analysis, here blockade's ability to progress is graded on the priority system. Also some players may have differing priority on this particular analysis as a whole.

### 4.5.5 Piece Importance validation

This consists of mainly 4 functions that validates the importance of each piece for all 4 players. The functions calculate the priorities and finally return the piece that is most suitable. The most suitable piece is then send for the final phase of the player AI.

### 4.5.6 Finalizing the choice of player AI

This is a common function that finalizes the choices of all four player AIs. `finalizeMovement()` function handles this logic. Even after analysis of player behaviors, the execution must be verified and appropriate action should be undertaken. This ensures that errors don't occur when the player behavior forces a player to choose a piece it can't move due to several circumstances (Ex: blocked by other piece, unable to move piece to board)

## 4.6 Display functions

Display functions primarily deal with displaying output to the users. These functions do not perform game actions or assist in any other way except displaying output.

## 4.7 Game loop functions

Game loop functions consists of functions that deal with several loops of the game. These functions are executed for the entirety of the program execution and are integral for the connection of several components of the program.

- `initialGameLoop` function deals with the initial player choosing loop where the first player to start the round is chosen
- `handleMysteryCellLoop` function deals with generating mystery cells throughout the program execution
- `mainGameLoop` function primarily deals with main game execution loop where all players play the game.

## 4.8 Endgame functions

Endgame functions only deal with endgame cases where the game is about to be stopped or when winners are calculated.

## 4.9 Play game function

This is a single function that initializes the necessary structures and runs the necessary game loop. This function is called and executed in `main.c`

```c
#include "game.h"

int main()
{
    playGame(); // play game function is executed
    return 0;
}
```

# 5. Program Efficiency

This section discusses the program efficiency mainly in two factors time complexity analysis and space complexity analysis.

## 5.1 Time complexity analysis

The program does not have cases of nested loops that grow based on input size or deep recursion. The maximum a particular loop iterates depends on number of pieces or players which is four. This value never changes therefore the loops do not grow on input size.

Also due to the use of 2D pointer array the number of checks for some cases have been reduced improving the overall time efficiency. This also results in some checks taking only O(1) in terms of time complexity since accessing an element on an array is constant time preventing the need of looping through available pieces to find particular information.

There is only one case of recursion and that is for a specific case of mystery cell teleportation logic in `applyTeleportation` function however as per the game logic this is a very specific scenario and at maximum only 2 consecutive teleportations can be executed. Therefore the recursion is only one level deep in the worst case.

## 5.2 Space complexity analysis

The program also does not have cases where the size of array or other structures grow based on input size. The highest space allocated is for the 2D array that stimulates the standard cells in the board. However this space is fixed for the entirety of the program and never increases. All structs and union type used are properly aligned with proper padding for efficient access as well as less memory usage.

There are also instances of `malloc` being used in the program to dynamically allocate space for arrays such as player array and pointer arrays to piece priority structs. However these are freed upon the end of particular functions using them, especially the pointer arrays that point to piece priority struct since these arrays are dynamically allocated frequently for player behavior calculations.

The large data structures (mostly excluding primitive data types) used in this program are only passed via pointers to the function. This reduces large structures being recreated inside the function since a pointer is always 8 bytes in 64 bit systems.

# 6. Conclusion

As per my report, I have detailed all the structures, the justification for said structures, functions implemented as well as efficiency of my program. This LUDO - CS game implementation presented in this report uses well structured and efficient data representation to run the game. The logic of the game is implemented in an efficient manner considering the system architectures of target machines while balancing both time and space efficiency.