ENGG1110 Course Project Report

Chan Cheuk Ka (1155174356)

Table of Contents

Feature A: Computer Strategy

A single-player mode is implemented where the user can play against a strategic computer opponent.

Game initialisation

Due to having a choice between single- and multi-player, the program now has to go through an initialisation phase.

Difficulty choice

The user is given the freedom to choose the AI's difficulty level. The intricacies of difficulty level will be discussed more below.

```
Enter player count: 1
Enter desired difficulty setting (0-100, higher is harder): 60
...
```

Player number choice

The user is also given a choice to play as either player.

```
Enter player count: 1
Enter desired difficulty setting (0-100, higher is harder): 60
Enter which player you will play as (Player 1 will go first): 2
!!! Computer is player 1 - You are player 2 !!!
...
```

Notice that a notification is also displayed to denote who is playing as which player number.

Game tree progress tracker

The game tree search takes a noticeable amount of time to complete. Hence, a progress tracker is implemented to inform the user.

```
Enter player count: 1
Enter desired difficulty setting (0-100, higher is harder): 60
Enter which player you will play as (Player 1 will go first): 2
!!! Computer is player 1 - You are player 2 !!!
Doing game tree search: 2426721/3516986 (69%)
```

The gameplay can truly start after the game tree search has been completed.

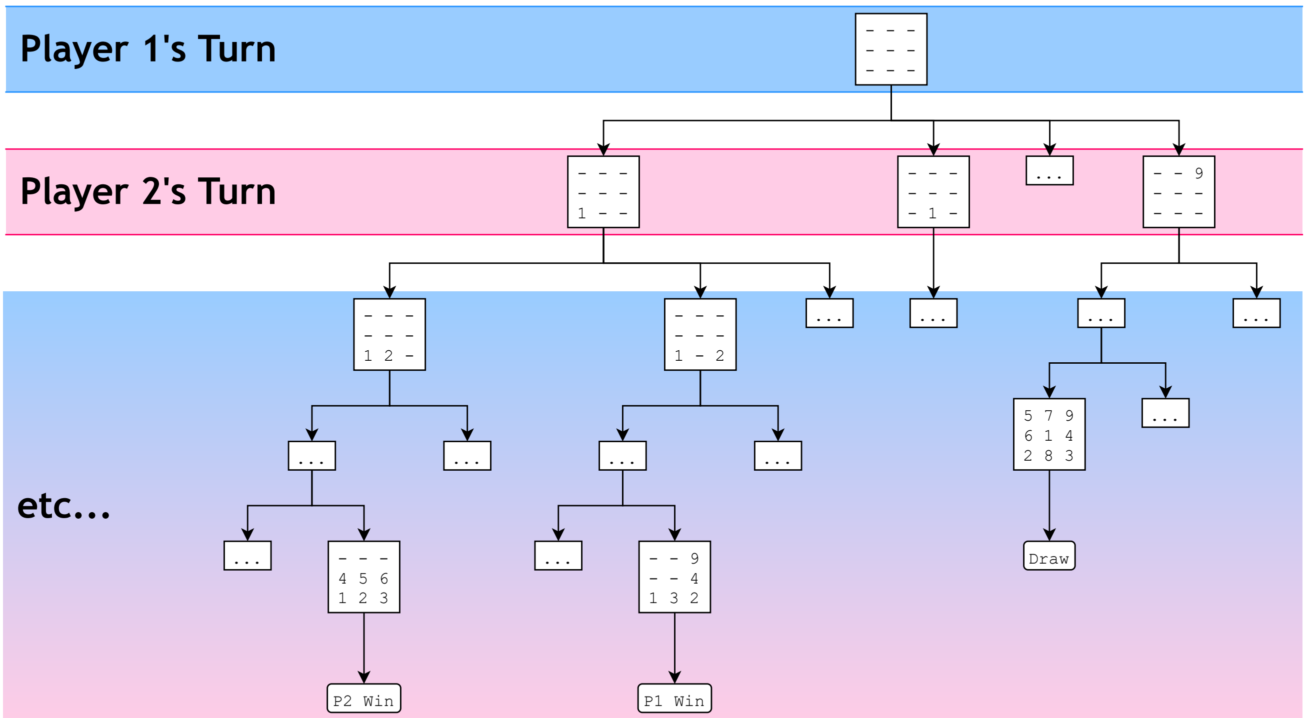
```
...
Doing game-tree search: 3516986/3516986 (100%)
Done.
### Player 1's turn ###
| | | |
| | | |
| | | |
Unused numbers:
1 2 3 4 5 6 7 8 9
...
```

AI

This program implemented a strategic AI designed to play theoretically perfectly. The AI does a game tree search in a depth-first search manner. During the search, it determines whether a board is advantageous to which player, then proceeds to find the best move in the current board position.

Game tree

A game tree is a tree of possible moves that both players can make in a given board position. For each move made, a new board position would be attained, with its tree of possible moves.



Move enumeration

Moves are enumerated for each board position to generate a list of valid moves.

> The source code segment responsible is as follows:
>
> ```
> char enumMoves(
>     char moves[45][2] /*by reference*/,
>     char board[3][3],
>     char nums[2][5],
>     char isPlayer1Turn  //? bool
> ) {                     //? add to moves and return moveCount
>     char moveCount = 0;
>
>     for (char pos = 0; pos < 9; pos++) {
>         if (board[pos / 3][pos % 3]) {
>             continue;
>         }
>
>         for (char i = 0; i < 5; i++) {
>             char num = nums[1 - isPlayer1Turn][i];
>             if (num == 0) {
>                 continue;
>             }
>             moves[moveCount][0] = pos;
>             moves[moveCount][1] = num;
>             moveCount++;
>         }
>     }
>
>     return moveCount;
> }
> ```
>
> Each board tile is iterated through to first check if they are empty, then all the remaining unused numbers are pushed to a valid move list.

Depth-first search

A new board position is then generated and assessed for each move. This process is done recursively.

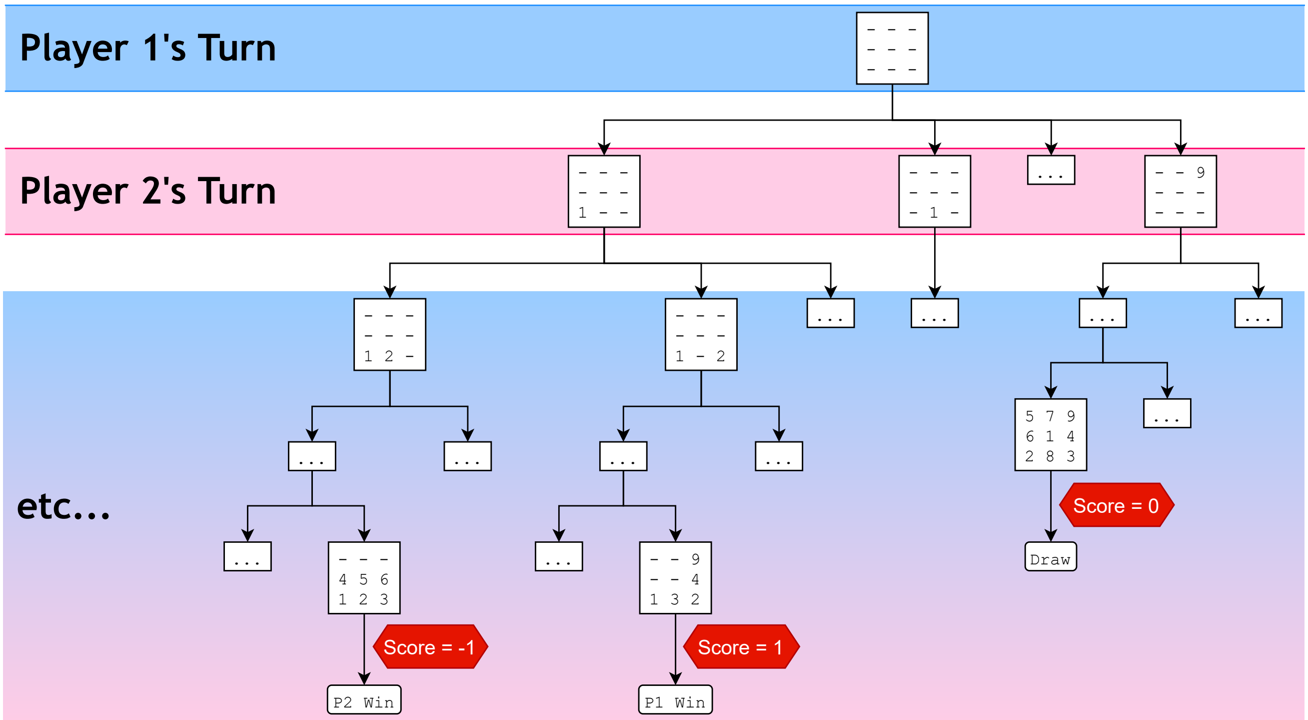
> The source code segment responsible is as follows:

```
// add to moves and return moveCount
char moves[45][2] = {0};  //? [position, number]
char moveCount = enumMoves(moves, board, nums, isPlayer1Turn);
//
char moveScores[45] = {-10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10
, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10
, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10
, -10};
//? -10: uninitiliased
char moveScoresLength = 0;
//
for (char i = 0; i < 45; i++) {  //? moves.forEach
    char movePosition = moves[i][0];
    char moveNumber = moves[i][1];
    if (!moveNumber) {
        continue;
    }
    //
    char newBoard[3][3];  // clone
    for (char j = 0; j < 3; j++) {
        newBoard[j][0] = board[j][0];
        newBoard[j][1] = board[j][1];
        newBoard[j][2] = board[j][2];
    }
    char newNums[2][5];  // clone
    for (char j = 0; j < 2; j++) {
        newNums[j][0] = nums[j][0];
        newNums[j][1] = nums[j][1];
        newNums[j][2] = nums[j][2];
        newNums[j][3] = nums[j][3];
        newNums[j][4] = nums[j][4];
    }
    //
    char newIsPlayer1Turn = !isPlayer1Turn;
    newBoard[movePosition / 3][movePosition % 3] = moveNumber;

    // remove moveNumber from newNums
    for (char j = 0; j < 5; j++) {
        if (newNums[1 - isPlayer1Turn][j] == moveNumber) {
            newNums[1 - isPlayer1Turn][j] = 0;
            break;
        }
    }

    //
    char treeScore = tree(pScore, newBoard, newNums, newIsPlayer1Turn, searched);
    moveScores[moveScoresLength] = treeScore;
    moveScoresLength++;
```

> Using the move list given by `enumMoves`, each possible future board positions is simulated and iterated through recursively via `tree`.

Game termination

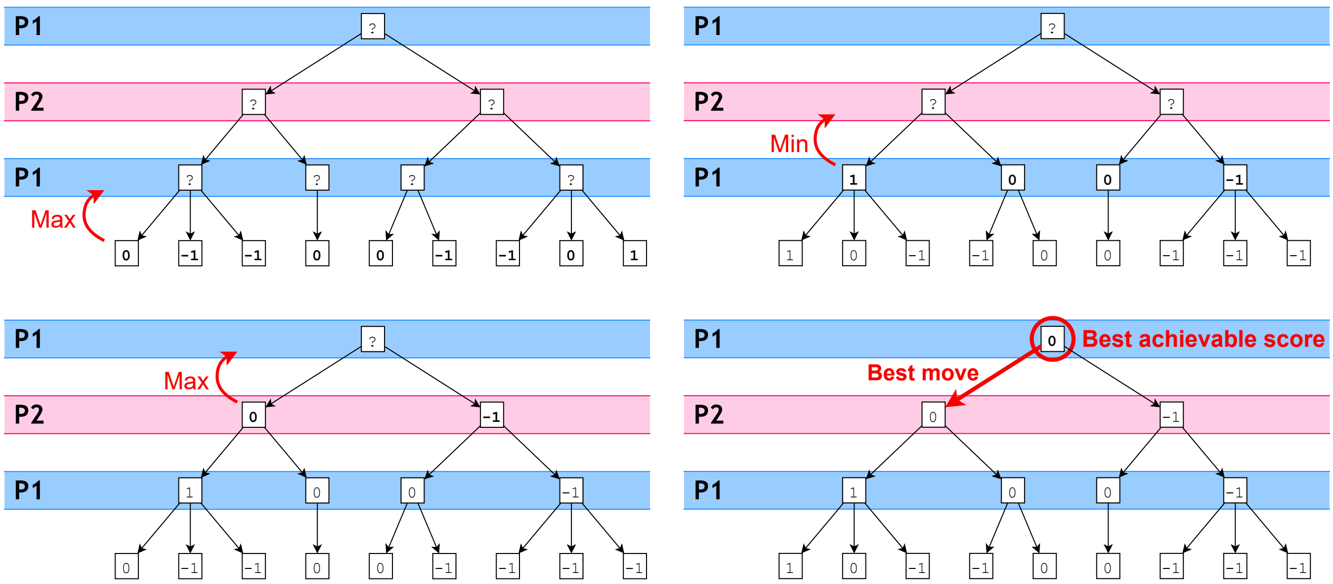
If a board qualifies for game termination (win/lose/draw), a score is assigned to the board depending on the termination mode: `1` denotes player 1 wins; `-1` denotes player 2 wins; `0` denotes a draw.



> The source code segment responsible is as follows:
>
> ```c
> char finalScore = 0;
> if (aiOnlyHasWinner(board)) {
>     finalScore = 1 - 2 * isPlayer1Turn;
> } else if (aiOnlyIsFull(board)) {
>     finalScore = 0;
> } else {
> ```
>
> If a game termination condition is met, `finalScore` is set directly and the branch is terminated. Notice that {`1 - 2 * isPlayer1Turn`} will give `1` or `-1` depending on whose turn it was.

Minimax

Consider a branch and its children nodes. Recall that a higher score corresponds to a larger advantage for player 1, and a lower score corresponds to that for player 2. Logically, player 1 would choose to play a move to maximise the score, and player 2 would choose to play a move to minimise the score. Assuming both players play logically, they would only choose moves that benefit them the most; hence, the score of a non-terminal board position would be either the minimum or maximum of the scores of its children nodes. Assuming both players play logically, children nodes with maximal and minimal scores will be chosen every other turn, since the players alternate turns playing. By this, the score of non-terminal board positions can be found through back-propagation.



Memoisation

After the score of a board is calculated, it is stored in memory with an index derived from the board position. During gameplay, the AI can enumerate the valid moves and board positions, read their scores, and determine the best move.
This chunk of memory is also used during the game tree search process for memoisation to optimise the search time since a board position can potentially be reached via multiple move orders.

> For instance, the moves `1@1, 2@2; 3@3` would achieve the same board position as `3@1, 2@2; 1@3`.

Memoisation also offers other optimisation opportunities via symmetry.

Index

The board index is calculated by the numbers present on the board. The numbers on the board starting from that in position 1, are concatenated into a nine-digit board index. Empty spaces are treated as the number `0`.

> Consider the board:
>
> ```
> 5|7|8
> 2| |
>  |1|3
> ```
>
> Its board index would be `013200578`.

In practice, leading zeros will be dropped since board indices are stored as regular integers, but it does not affect anything.

> The source code segment responsible for computing index is as follows:
>
> ```
> short r0 = sConcat(board[0][0], board[0][1], board[0][2]);
> short r1 = sConcat(board[1][0], board[1][1], board[1][2]);
> short r2 = sConcat(board[2][0], board[2][1], board[2][2]);
>
> int boardIndex = lConcat(r0, r1, r2);
> ```
>
> Two auxiliary functions are called to condense the process:
>
> ```
> short sConcat(char a, char b, char c) {
>     return a * 100 + b * 10 + c;
> }
> int lConcat(short a, short b, short c) {
>     return a * 1000000 + b * 1000 + c;
> }
> ```

Storage format

Consider the score of any given position: there are only 4 different states we have to store.

Binary value	Score	State
`00`	-	Uninitialised
`01`	`-1`	Player 2 wins
`10`	`0`	Draw
`11`	`1`	Player 1 wins

It can be observed that 2 bits are sufficient to store the score and state of each board position.

Considering the board index algorithm, the maximum index would be `987654321`.
It can be observed that 246913581 bytes (= 987654321 * 2 bits = 1975308642 bits) are necessary to store all board positions. (Note that a large majority of indices would correspond to invalid board positions; however, there is no efficient algorithm to index only valid board positions; hence, it is decided that it should be a reasonable compromise considering the ease and convenience.)

> The source code segment responsible is as follows:
>
> ```c
> unsigned char* pScore;  //? For AI, to be initialised later
> ```
>
> ```c
> pScore = malloc(246913600);  //? in bytes = 987654321 * 2 / 8
> ```
>
> A chunk of memory is allocated for memoisation.

For each byte (8 bits), the scores for four board positions can be stored. The score values can be found at offset {`pScore + 2 * index`}.

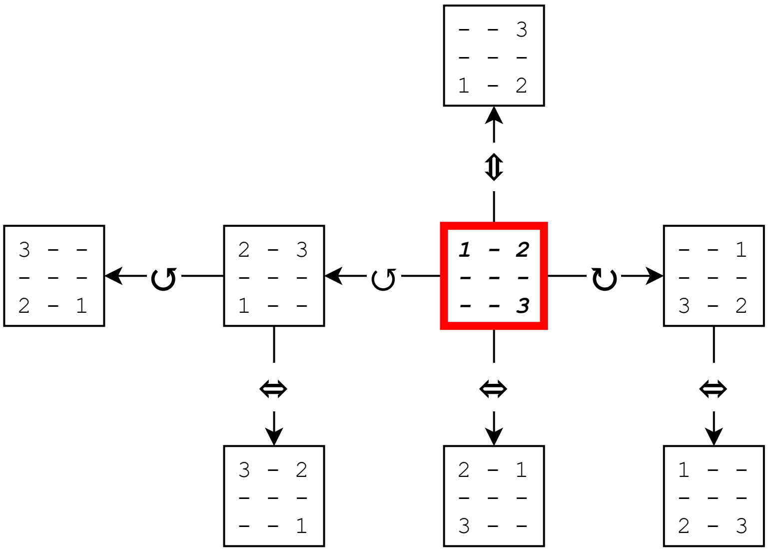
> Consider the following memory chunk.
>
> | `pScore`+ | `7` | `6` | `5` | `4` | `3` | `2` | `1` | `0` |
> |---|---|---|---|---|---|---|---|---|
> | `0x00` | `1` | `0` | `0` | `0` | `1` | `0` | `1` | `1` |
> | `0x08` | `1` | `0` | `0` | `0` | `1` | `0` | `0` | `0` |
> | `0x10` | `0` | `0` | `1` | `1` | `1` | `0` | `0` | `0` |
> | `0x18` | `0` | `0` | *`1`* | *`1`* | `0` | `0` | `1` | `1` |
>
> For a board with index `000000014`, the score state can be found at offset `0x1C`. The value is `11`, which would correspond to an advantage for player 1.

> The source code segment responsible for reading the data values is as follows:
>
> ```c
> char getScore(unsigned char* pointer, int index) {
>     char shiftCount = index % 4 * 2;
>     char value = *(pointer + index / 4);
>     return (value >> shiftCount) - (value >> (shiftCount + 2) << 2) - 2;
> }
> ```
>
> `getScore` returns the score value (`0` / `1` / `-1`) for a given `index`. Bit-shift operations are used to isolate only the bits enquired.
> Note that the data format is designed in a way that the raw data value subtracted by 2 would be the corresponding score value.

> The source code segment responsible for writing the data value is as follows:
>
> ```c
> void initialiseScore(unsigned char* pointer, int index, signed char value) {
>     //! Cannot overwrite!
>     *(pointer + index / 4) += (value + 2) << (index % 4 * 2);
> }
> ```
>
> Note that there is no implementation for overwriting values since under normal circumstances, the data should be only written once. In fact, `initialiseScore` would cause data corruption if it is called twice on the same `index`.

Symmetry

Considering the symmetrical nature of the board, boards can be rotated and/or reflected to obtain new board positions without affecting their respective scores. Note that at most 2 operations can be done on each board before duplicates are introduced. This property is exploited to reduce necessary computation by a factor of 8.




```
//
short s0 = sConcat(board[2][0], board[1][0], board[0][0]);
short s1 = sConcat(board[2][1], board[1][1], board[0][1]);
short s2 = sConcat(board[2][2], board[1][2], board[0][2]);
//
short t0 = sConcat(board[2][2], board[2][1], board[2][0]);
short t1 = sConcat(board[1][2], board[1][1], board[1][0]);
short t2 = sConcat(board[0][2], board[0][1], board[0][0]);
//
short u0 = sConcat(board[0][2], board[1][2], board[2][2]);
short u1 = sConcat(board[0][1], board[1][1], board[2][1]);
short u2 = sConcat(board[0][0], board[1][0], board[2][0]);

//
int boardIndices[] = {
    lConcat(r0, r1, r2), lConcat(r2, r1, r0),
    lConcat(s0, s1, s2), lConcat(s2, s1, s0),
    lConcat(t0, t1, t2), lConcat(t2, t1, t0),
    lConcat(u0, u1, u2), lConcat(u2, u1, u0)};
for (char i = 0; i < 8; i++) {                      //? boardIndices.forEach
    if (getScore(pScore, boardIndices[i]) != -2) {  //! already initiliased
        continue;
    }
    initialiseScore(pScore, boardIndices[i], finalScore);
}
```

`r0`, `r1`, `r2` are as above.
Note that it is necessary to check that the value is not already initialised, since it is possible that a board remains unchanged after rotations and/or reflections.

Gameplay

During gameplay, the AI will first enumerate the legal moves in the current position, then look up the calculated move score table it made during memoisation, and then find a move with the best score (highest score if it is player 1; lowest score if it is player 2).

Move randomisation

Notice that since the scores only denote if the given position is favourable to either player 1 or player 2, it is possible for multiple moves to have the same score. The AI will choose a random move out of a list of favourable moves to avoid game repetition.

> The source code segment responsible is as follows:
>
> ```c
> //! play best move
> int bestScore;
> if (computerPlayerNumber == 1) {
>     bestScore = max(moveScores, moveScoresLength);
> } else {
>     bestScore = min(moveScores, moveScoresLength);
> }
>
> // TODO: debug info
> // printf("Best score: %d\n", bestScore);
> // for (int i = 0; i < 45; i++) {
> //     printf("%d | Pos: %d | Num: %d | Score: %d\n", i, moves[i][0], moves[i][1], moveScores[i]);
> // }
> //
>
> int bestMoveIndices[45];
> int bestMoveIndicesLength = 0;
> for (int i = 0; i < 45; i++) {
>     if (!moves[i][1]) {
>         continue;
>     }
>     if (moveScores[i] != bestScore) {
>         continue;
>     }
>     bestMoveIndices[bestMoveIndicesLength] = i;
>     bestMoveIndicesLength++;
> }
> //
> decidedMoveIndex = bestMoveIndices[rand() % bestMoveIndicesLength];
> ```
>
> The best achievable score within `moveScores` is first computed. All indices corresponding to moves with the best score is pushed to a list, then `rand` is called to choose a random move among the best moves.

Respect of difficulty

As demonstrated above, the user is prompted to choose a difficulty level in the range of [0-100]. The difficulty level is treated as a % of when the computer will choose to play the best move, otherwise, a random move.

> For example, a difficulty level of 40 would entice the AI to make the best moves 40% of the time, and random moves 60% of the time.

> The source code segment responsible is as follows:
>
> ```
> if (rand() % 100 >= computerDifficultyLevel) {  //? 0-99 < 100 (max)
>     //! player random move
>     decidedMoveIndex = rand() % moveScoresLength;
> } else {
>     //! play best move
> ```
>
> `rand` is first called to determine whether the AI would proceed to play a random move or the best move. If it is compelled to play a random move, another `rand` call would determine which random valid move it would make.

Move notification

The AI will notify the player what move it had made.

```
...
### Player 1's turn ###
| | | |
|1|8|2|
|5| | |
Unused numbers:
3 4 6 7 9
The computer played the number 7 at position 2
### Player 2's turn ###
| | | |
|1|8|2|
|5|7| |
Unused numbers:
3 4 6 9
...
```

Advantages

This AI utilises a full game tree search to determine its moves. During this search process, it goes through every possible move and game to find the best next move. With this approach, it can solve the game in its entirety. Therefore, it is a theoretically perfect strategy. Additionally, since the game is solved, it can be confidently said that player 1 will always win with perfect play, as shown from the starting empty board position having a score in favour of player 1.

Feature B: Invalid User Input Checking

This feature aims to sanitise invalid user inputs and prompt the user to re-enter said inputs.

Game initialisation

As demonstrated in the previous section, a new game initialisation phase is implemented to account for the two game modes and their settings.

Game mode choice

The user is prompted to choose between a single- or multi-player game. The program will warn and re-prompt the user if their input is out of range [1-2].

```
Enter player count: -50
Player count must be 1 or 2!
Enter player count:
```

> The source code segment responsible is as follows:
>
> ```c
> int inputIsInvalid = 1;
> while (inputIsInvalid) {
>     inputIsInvalid = 0;
>     int playerCount = 0;
>     //
>     printf("Enter player count: ");
>     scanf("%d", &playerCount);
>     if (playerCount != 1 && playerCount != 2) {
>         printf("Player count must be 1 or 2!\n");
>         inputIsInvalid = 1;
>     }
>     playWithCPU = playerCount % 2;
> }
> ```

Difficulty choice

The user is also prompted to choose a difficulty level for the AI. The program will warn and re-prompt the user if their input is out of range [0-100].

```
Enter player count: 1
Enter desired difficulty setting (0-100, higher is harder): -10
Difficulty must be between 0-100!
Enter desired difficulty setting (0-100, higher is harder):
```

> The source code segment responsible is as follows:
>
> ```c
> if (rand() % 100 >= computerDifficultyLevel) {  //? 0-99 < 100 (max)
>     //! player random move
>     decidedMoveIndex = rand() % moveScoresLength;
> } else {
>     //! play best move
> ```

Player number choice

The user is also prompted to choose whether to play first in single-player mode. The program will warn and re-prompt the user if their input is out of range [1-2].

```
Enter player count: 1
Enter desired difficulty setting (0-100, higher is harder): 60
Enter which player you will play as (Player 1 will go first): 9
You must input 1 or 2!
Enter which player you will play as (Player 1 will go first):
```

> The source code segment responsible is as follows:
>
> ```c
> //* player number choice
> inputIsInvalid = 1;
> while (inputIsInvalid) {
>     inputIsInvalid = 0;
>     //
>     int answer;
>     printf("Enter which player you will play as (Player 1 will go first): ");
>     scanf("%d", &answer);
>     if (answer != 1 && answer != 2) {
>         printf("You must input 1 or 2!\n");
>         inputIsInvalid = 1;
>         continue;
>     }
>     //
>     computerPlayerNumber = 3 - answer;
>     if (answer == 1) {
>         printf("!!! You are player 1 - Computer is player 2 !!!\n");
>     } else {
>         printf("!!! Computer is player 1 - You are player 2 !!!\n");
>     }
> }
> ```

Gameplay

During regular gameplay, the program will check the validity of the entered position and number. If either or both of them are invalid, the program will warn the user and re-prompt them to enter.

```
...
|6| | |
|9|2| |
| |1| |
Unused numbers:
3 4 5 7 8
### Player 1's turn ###
Input the position: 20
Input the number: 14
Position can only be an integer between 1-9!
Number can only be an integer between 1-9!
Input the position: 4
Input the number: 6
Position is occupied!
Player 1 can only input odd integers!
Input the position: 6
Input the number: 1
1 is already used!
Input the position:
```

> User entered both an out-of-range position (20) and number (14), and was warned.
> User then entered an occupied position (4) and an invalid number (6). Note that 6 is not in the pool of usable number for player 1, and it is also already used in the game. Notice that the program decided to only issue a warning for the incorrect odd/even parity since it is more important.
> User then entered a used number, and was warned.

As demonstrated above, the program can issue warnings about the entered position and number independently. The above snippet also illustrated the precedences of the warnings.

Priority	Position	Number
1	Out of range [1-9]	Out of range [1-9]
2	Occupancy	Wrong number parity (odd/even)
3		Used number

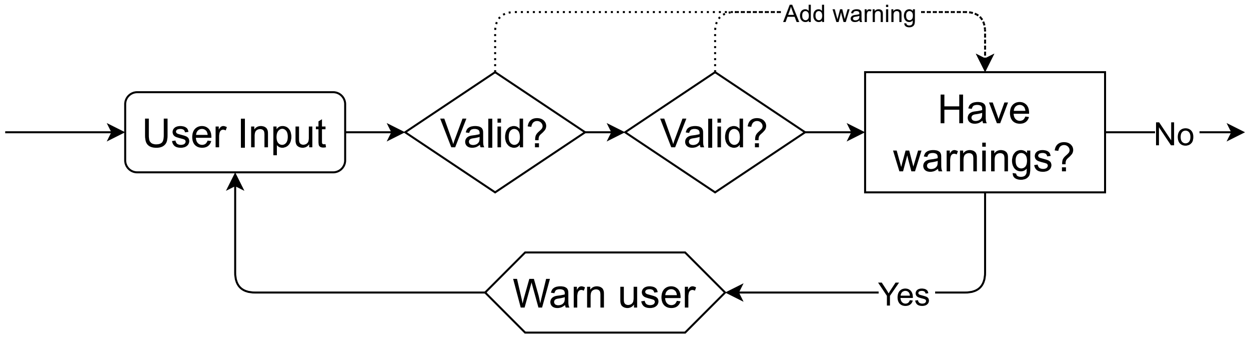
The source code segment responsible is as follows:

```c
    //! player input
    int inputIsInvalid = 1;
    while (inputIsInvalid) {
        inputIsInvalid = 0;

        //?
        printf("Input the position: ");
        scanf("%d", &position);

        printf("Input the number: ");
        scanf("%d", &number);

#pragma region Validity check  //!
        if (position < 1 || position > 9) {
            printf("Position can only be an integer between 1-9!\n");
            inputIsInvalid = 1;
        } else {
            if (gameBoard[(position - 1) / 3][(position - 1) % 3]) {
                printf("Position is occupied!\n");
                inputIsInvalid = 1;
            }
        }
        if (number < 1 || number > 9) {
            printf("Number can only be an integer between 1-9!\n");
            inputIsInvalid = 1;
        } else {
            if (currentPlayer % 2 != number % 2) {
                if (currentPlayer == 1) {
                    printf("Player 1 can only input odd integers!\n");
                } else {
                    printf("Player 2 can only input even integers!\n");
                }
                inputIsInvalid = 1;
            } else {
                if (numberUsed[number - 1]) {
                    printf("%d is already used!\n", number);
                    inputIsInvalid = 1;
                }
            }
        }
#pragma endregion
    }
```

Notice the warning priorities as described above.

Validity check rationale

The above three demonstrated validity checks use the same rationale.



This model allows the program to issue multiple warnings at the same time while being able to determine warning priorities.

Documentary

This section documents the attempts and workarounds in order to overcome various challenges in this project. Most of the challenge stems from the AI implementation, as one might be able to intuit.

Writing a full game tree search algorithm is not as hard as I had imagined, but it definitely comes with a lot of annoying problems especially considering the language I had to work with. I deliberately did not go on to research algorithms since that would defeat the fun of coming up with my own approach. I have some prior knowledge about how chess engines work, but apart from their general gist, that was about it.
To add to the challenge, *C* is not a very versatile language, in the sense that it lacks a lot of useful modern language features. Nonetheless, I attempted to write the algorithm in *C* in the beginning, but I quickly got frustrated by its limitations and verbosity in even simple operations.
I decided to switch to *JavaScript (JS)* to flesh out the logic first before transpiling back into *C*, considering I am exponentially more fluent in *JS*, not to mention it has significantly more debugging tools available. Despite this, I still had to hold myself back from using too many modern language features lest the transpilation be tedious. Along the way, considering the transpilation process, I switched to *TypeScript (TS)*, which is a typed superset of *JS*, since I thought it would lessen my confusion at the later stages.
The bulk of the logic is rather easy, and I got to transpiling. I have never transpiled before, but it went a lot more smoothly than I had imagined, although a lot of simple one-liners in *TS* can only be expressed in 10+ lines of *C* code, which killed the elegance quite a bit.

For the actual logic itself, I had considered doing short-circuit evaluations instead of full game tree searches since they are significantly faster, and it would also provide an easy parameter (depth) to be tweaked as the difficulty setting. However, it is very difficult to write a good evaluation algorithm, since this game is arguably more reliant on pure logic than visual patterns like chess does.
Eventually, I decided to go with the full game tree search as I originally intended to. I didn't realise it at the time, but this decision also opened up a huge optimisation opportunity. Unlike in short-circuit evaluations where scores span a huge range, scores in full tree searches can only be `0`, `1`, or `-1`. I went ahead and tried to use this property and store the scores as the smallest addressable type in *C*, which would be `char`. However, I quickly found out that even with `char`, the memoisation array would be too large for *C*. Despite only having 8 bits, `char` is still too large of a type. I was stuck for a very long time until I had the idea to invent a custom data format. I noticed that I really only needed 2 bits for each board position, and hence birthed the memory chunk method I introduced above.
When I was stuck, I also went ahead and shortened all the variable types to `char` and `short` since I figured the program rarely needs to handle large numbers anyway.

As mentioned above, I had intended to implement a difficulty setting from the very beginning, but the simplified approach I used does not seem to allow for it. While I could change the depth of the search, the challenge is to also write a good algorithm for it, as I have mentioned above. Alongside this, the maximum depth of this game is too shallow for any meaningful difficulty setting anyway. Therefore, as a compromise, I implemented difficulty by mixing random moves among the best moves, and it just so happens to also be the easiest option to implement. Although I would argue that this is not what true difficulty level would and should look like, I genuinely cannot be bothered to conceive of a better way.