Connect 4 Coursework Report

Introduction

The purpose of this coursework was to create a working implementation of the game Connect Four in any programming language. This could only be done through a command line console, i.e. GUIs could not be used. For this task, I have chosen to implement the game in C# as my work environment makes use of C#. Moreover, I found that it would be a good opportunity to further my knowledge of the language.

The size of the board was set to 5 columns by 4 rows, similar to the traditional 7x6 game by HASBRO[1]. A series of GIFs showing the console game in action are available in the coursework repository[2]. The code gives the option to play against another human player; human player against an "AI"; or an "AI" against "AI". Each player in turn has control over the next move. The programme checks for a valid move and determines if the game has been won or tied. Before a move is actioned, the player has the option to change the move. Once the game has been completed, a replay option is included.

Game Design

To implement the game there are some basic structures and functions that need to be considered:

1) The board is a 5x4 2D-array which needs to be updated at each stage. In its simplest form, the board states are stored in a *Queue* and the moves are stored in a *Stack*.
2) An initial screen output is needed to describe the game and to give users the options to play against the computer or another player.
3) To show the current state of play, the 2D-array has to be displayed on the console. A PrintBoard() method was implemented for this task.
4) A series of methods are then needed to identify which player is making a move; if the player's move is allowed; and check if the player has won, or if the board is full to declare a tie.

The structure of the code is described in detail. The main method shows the initial information on the game and initialises the 2D-array to all 0 values displaying it using PrintBoard(). The player is asked to choose the mode of play and control is handed over to one of the following methods.

PvP() – PvPC() – PCvPC() methods are called when either "h", "c" or "a" (respectively) is entered by the user at the start of the game. These functions pool together all the other methods in order to create the game that is then played. All the methods make use of a *while* loop that is only exited once a game

[1] https://shop.hasbro.com/en-us/product/connect-4-game:80FB5BCA-5056-9047-F5F4-5EB5DF88DAF4

[2] https://github.com/CatB1794/Brandani_Caterina_ADS/tree/master/WinGifs

has either been won or tied. In order to switch between players, a Boolean is used, which is initiated as player = true and then flips to false when it is the next player's turn, and vice versa. The player column input has a try-catch that validates that the input is numerical. GetPosition() is then used to check if a valid entry has been made. Once the move is accepted and a check for a win or a tie is applied, PrintBoard() displays the new state of play.

PrintBoard() – this method prints the game board, as shown in Fig. 1, by iterating through a nested *for* loop. To display the state of play, 0 identifies an empty slot; 1 corresponds to a tile of player 1; and 2 corresponds to a tile of player 2. The outer loop iterates through the rows and the inner loop iterates through the columns of the 2D-array. To help the player identify the move, a column with the row number is added on both sides of the array, and a row with the column number is included both above and below the board.

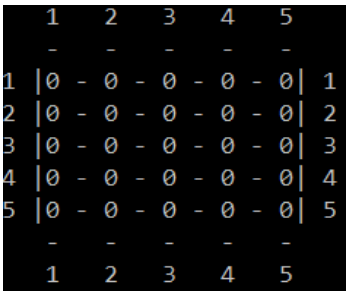

Figure 1. Initialised board.

GetPosition() – this method determines the corresponding row to the user's column input, and only returns the row value if the input is valid. This is achieved through an *if* condition, that tests whether the column is full or not and that the entry is within the board boundaries. The row number is returned for a valid move, otherwise the method returns a 0 value.

GameWon() – this is implemented through an iterative algorithm, consisting of nested *for* loops with *if* conditionals. Figure 2 below shows the possible winning outcomes in a 4x4 sub-array. These are horizontal, vertical, and ascending and descending diagonal chains of 4 tiles.

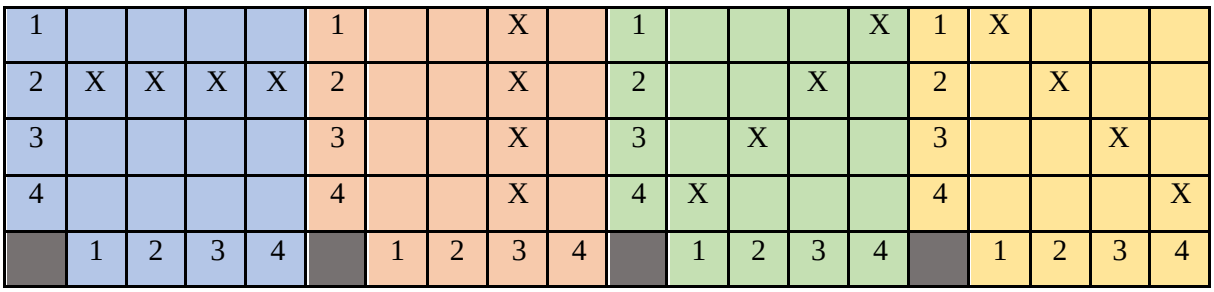

Figure 2. Four possible winning chains identified by GameWon().

As an example, the line for the ascending diagonal chain is shown below.

```
if (Board[x, y] == move && Board[x + 1, y - 1] == move && Board[x + 2, y - 2] == move && Board[x + 3, y - 3] == move)
```

GameTied() – this was originally implemented as a method through an iterative algorithm, consisting of nested *for* loops with an *if* conditional that checks whether there was an empty space i.e. a 0. After evaluation on larger board sizes, this has been converted to counting the total number of moves and comparing it to the size of the board.

PCPlayer() – This method is a very simplistic AI player. It uses a *while* loop to generate a random integer between 1 and the number of columns (normally 5) and checks for a valid move using GetPosition().

MoveUndo() – this method is implemented using an Undo *Stack* data structure. With the last-in first-out feature of *Stack* structures it is relatively easy to go back one or more moves. In its current form only one move can be undone, but this structure would allow also to implement multiple "undo". One could also opt to use *pointer* variables of the entire board at each step and implement the undo method replacing the entire 2D-array by the previous value. The use of stacks is more efficient given that only one array element is changed at each move.

GameReplay() – this method allows the user to review the moves of the game that has just been completed by making use of a *Queue* data structure. Each time a move is made the new board state is enqueued into the Replay *Queue.* A *for* loop will *Dequeue* each move and display the corresponding state of play.

GameMoves{} – this class is a *Struct* data structure, which when used in conjunction with *Push*() for the *Stack* and *Enqueue*() for the *Queue*, retrieves and stores the most recent move made by the player via getters and setters, storing three values: the player's column input and corresponding row value, and which player made the move.

Evaluation

Implementing the game board via the use of a multi-dimensional array data structure, a 2D-array for this task, made it straightforward to write code that is easier to read compared to a one dimensional array, which would need a conversion of rows/columns into a position [(RowN–1)*NCol+ColN] at each occurrence. The 2D-array provides a one to one correspondence with the board and is easy to visualise. This helps also when considering how to implement the win algorithm, which was the first major task to resolve.

The win algorithm that was used for this task presents both advantages and disadvantages. An advantage is that the logic implemented will work for the winning four in a row regardless of the game board size, due to the *if* conditional that checks for the 4 possible winning moves shown Figure 2. However, it is a nested *for* loop which runs in quadratic time, as the board size gets bigger. Therefore, it will take longer

to eventually return true for a winning conditional being met (as shown in Table A1 in the appendix). A more efficient option would be to keep track of possible chains originating from the current move. A simple adaptation would be to check only a sub-array of ±3 positions, with respect to the current move position, in both rows and columns limiting the maximum size of the problem to a 7x7 array for larger boards. This has not been implemented given the small size of boards normally used by players.

A similar issue was found in case of a game that has been tied, with the execution time being $O(n^2)$ for the original nested loop algorithm. Eventually, it would become near impossible to play the game due to the ever-increasing time taken for the methods to finish looping, and potentially could take up too many resources - as even a board of only 1000x1000 averaged at ~15% CPU usage. Another thing to take into consideration is that as the board size becomes bigger, the chance of a tied game becomes smaller, meaning that looping through GameTied() would have been a significant waste of resources and this is the reason why this was changed to simply tracking the total number of moves.

The simple way the "AI" player was implemented works relatively well with a smaller board and keeps the game interesting for both the player vs AI and AI vs AI games, and it is the reason why the board size 5x4 was chosen. The traditional 7x6 size resulted in games that had too many moves. A more advanced computer player could have been developed, for example through the use of a Tree data structure, which could map out all the possible moves and would then select the most efficient path towards a winning conditional. Another simpler defensive approach could have been implemented to check if a move would allow to break the opponent's chains of 2 tiles and then block 3 tiles in a row. This would lead in most cases to tied games.

The undo feature was implemented allowing the player to undo only the move they have just made, rather than being able to undo the entirety of the moves made. If multiple undo/redo had been implemented, to ensure a fair game, each player should be asked to accept the undo/redo move. This would mean slowing the game significantly or could potentially bring the game to a standstill. Instead allowing the player to undo only their own moves is a simpler and fairer way.

The table of results reported in the appendix includes the execution times for different board sizes. The tests used the C# System.Diagnostics Stopwatch feature which gave a reliable execution time result compared to attempting to manually time results.

An additional test was carried out to evaluate PrintBoard(). Setting the board size to 10,000x10,000 resulted in the executable taking 1hr-1-mins-24s (3684404ms) to print just one instance of the board, whereas with the normal 5x4 it only takes 2ms.

Appendix

Board size: 100x100		
AI	Moves	Win:Lose:Tie ratio
1	52, 60, 39, 52, 45	4:1:0
2	51, 59, 38, 52, 44	1:4:0
Total time (s): 44.506, 51.092, 33.694, 44.729, 38.613		
Board size: 50x50		
AI	Moves	Win:Lose:Tie ratio
1	23, 18, 27, 26, 20	1:4:0
2	23, 17, 27, 26, 20	4:1:0
Total time (s): 5.948, 4.529, 6.718, 6.591, 5.145		
Board size: 25x25		
AI	Moves	Win:Lose:Tie ratio
1	18, 11, 20, 16, 27	2:3:0
2	18, 11, 20 15, 26	3:2:0
Total time (s): 1.248, 0.762, 1.392, 1.075, 1.844		
Board size: 5x5		
AI	Moves	Win:Lose:Tie ratio
1	9, 12, 11, 8, 13	2:2:1
2	9, 11, 10, 8, 12	2:2:1
Total time (s): 0.074, 0.096, 0.092, 0.068, 0.106		

Board size	**Average Moves (1)**	**Average Moves (2)**	**Average time (s)**
100x100	49.6	48.8	42.5268
50x50	22.8	22.6	5.7862
25x25	18.4	18	1.2642
5x5	10.6	10	0.0872

NB: All tests done using the PCvPC() method to avoid human reaction time lag..
CPU used: Intel i7-7700K @ 4.20GHz.

Table of results