Connect-4: Technical Document

Ucizi Mafeni

April 4, 2017

1 Time Taken

I spent approximately 30 hours coding up and testing my implementation of the coding exercise. Upon completing the basic tasks, I dedicated most of my efforts into implementing the AI player.

2 Core Classes

2.1 Game Engine

The Game Engine contains the core logic of the game. I modelled it a simple state machine with a set of public methods used to alter its internal state.

At any point in time, external objects can query the state of the Game Engine in order to determine which actions to take. The three main states were:

INIT The starting state of the game-engine. Methods used to setup the game (i.e. register players, initialise the board) can be successfully called while in this state. All other method calls (i.e. related to making moves) will be ignored.

PLAYING This state is triggered by the `startGame()` method. once in the PLAYING state, all methods move-making methods can be successfully called.

GAMEOVER This state is triggered once a winner has been determined. The only callable method at this point is `resetGame()`

The game-engine's main job is to maintain a single instance of the Board. After each move, it checks if the player who made the move has won by scanning the array vertically, horizontally and diagonally from the point where the last piece was placed. Each scan has a running time of $O(k)$ where k is the number of connected pieces needed to win.

2.2 Board

This is the object used to keep track of the moves that have been made so far. The core part of the board is represented as a 2D integer array, symbolising the "co-ordinates" of each position on the board. The array uses zero-indexing so, as an example, the index `[5][4]` would be used to access the slot on the 6th column going left to right and the 5th row going horizontally.

The integer stored at each index represents the piece placed by player n. (i.e. a `1` stored at index `[0][0]` means player `1` placed a piece on the slot in the first row and first column). Empty slots are represented using a zero.

Using integers means players can be added to the board by simply incrementing a number.

2.3 GUI

The board was printed as a square grid of ASCII characters, with each piece represented by the player number. The columns were labelled with integers corresponding to the column index in the 2D board array. Users can make a move by selecting a column number from the dropdown-list and then clicking the "make-move" Button.

The board extended the `Observable` class. This meant I could link up the `JTextPane` containing the visual representing the board to an `Observer` class so it automatically refreshes every time a move is made.

2.4 AI Player

My AI player made its move decisions by searching a minimax game-tree. Alpha-Beta pruning was used to improve search speed by cutting off useless branches in the game-tree.

The maximum search depth was estimated as the depth which would result in a tree of no more than 150,000 nodes. This meant my AI could simulate up to 6 moves ahead of the current state on a standard 7×6 board.

The core component of the search heuristic measures the quality of a move by taking the sum of the squares of the number of connected components in each direction (See Figure 1). This value is negated when calculated for an opponents move, and the "Overall" game state quality is calculated by summing all the individual move quality heuristic values along the path to the root.

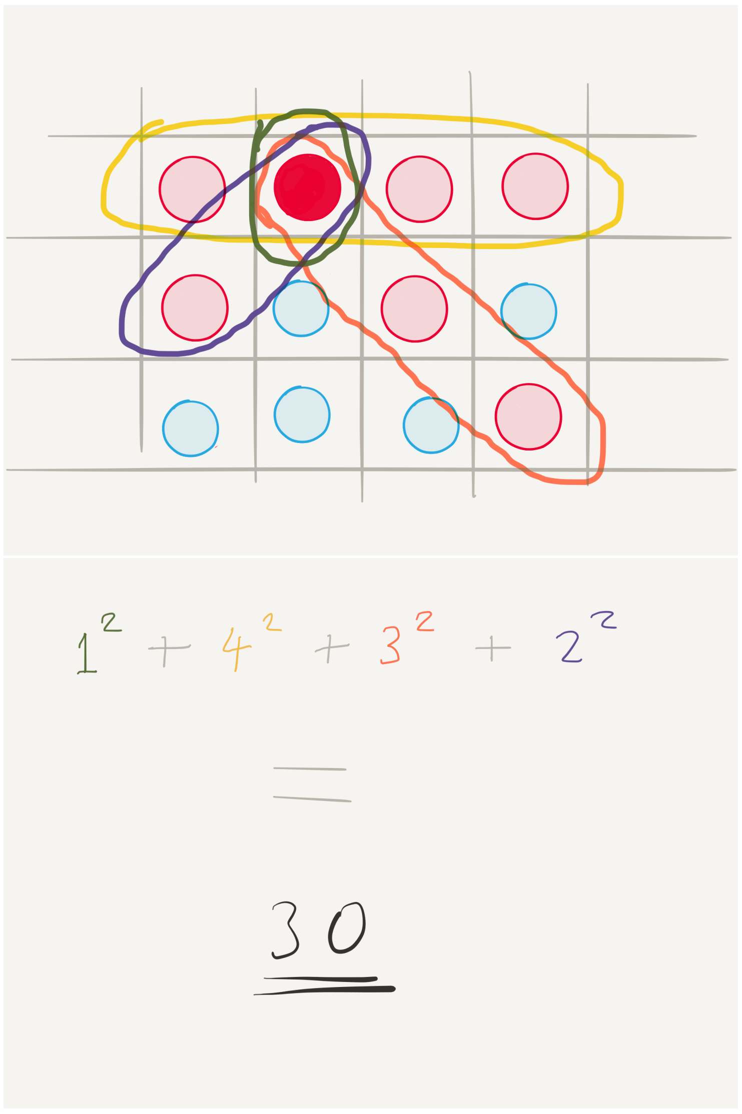

Figure 1: Example showing how the move quality heuristic value would be calculated from the solid red piece

3 Problems And Limitations

3.1 Exception Handling

Due to the time-sensitive nature of the task, there were times when I overlooked throwing exceptions at potential error points. This could make it harder to unearth any unexpected errors which may occur when changes are made to the code.

3.2 Usability

As stated earlier, I placed a lot of focus on implementing the AI player and the quality of the GUI suffered as a result. Though the dropdown-list move selector does an okay job, I would've preferred to place buttons above each column, allowing the user to click above the column they wish to place a piece, as I feel this would be more intuitive than the dropdown-list.

3.3 MVC Compromise

I implemented a makeshift MVC pattern where the View and "Controller" were implemented in the same class. This lack of modularity could make it harder to translate the model to different GUI representations.