Title:	Creating a Connect-4 Bot Using the Minmax Algorithm
Date:	07/07/2023
By:	Ethan Corgatelli

1.0 Table of Contents

2.0 Abstract

This paper describes the Connect-4 bot I designed using the Minmax algorithm. I used Rust for this project. My code can be found here: github.com/ETBCOR/cs470/tree/master/proj2

- In the Algorithm section, I describe how the bot works.
- In the Conclusions section I discuss the performance of the bot.
- In the Appendix I have included one full game of Connect-4 played against a “perfect” bot (that I didn’t make).

3.0 Algorithm

This section describes how I used the Minmax algorithm to implement my Connect-4 bot.

3.1 Building Blocks

Spot is an enum that takes one of the following values: *Empy*, *O*, or *X*, representing a single piece position in a game of Connect-4.

Score is an enum that is similar to *Spot*, except instead of an *Empty* variant there is an *InProgress* variant that stores an integer that represents the score of some board state, where negative values are good for the human ("O") and positive values are good for the bot ("X").

Move is a struct used in the *best_move()* function to explore possible future board states. It has the following fields:

- ***col*** (integer — represents the column that a piece was dropped into to get to the associated *GameBoard*),
- ***score*** (integer — represents intermediate Minmax score calculations),
- ***depth*** (integer — represents the maximum depth that was used to get the associated score),
- and ***board*** (*GameBoard* — stores the board that the move creates).

GameBoard is a struct that stores a 2D vector of *Spot*s which is initialized to be completely *Spot::Empty*. The game logic, the scoring system, and the Minmax algorithm are all implemented as associated functions[1] of *GameBoard*.

- ***open_spot()*** takes no inputs and returns a boolean representing whether there is an open spot on the board (or if the board has been filled).
- ***play()*** handles the main game loop. It takes no inputs and returns a *Score* representing the outcome of the game. It starts by asking who should play first. If the human decides that the bot should go first, a single call to *turn_bot()* is made, since the main game loop starts with a human turn. The main loop is two copies of some very similar code. Before making a turn call, it checks that there's an *open_spot()* to play in. If there isn't, the current *InProgress* score is returned (the game is a draw). If there is, it calls either *turn_human()* or *turn_bot()* (alternating), then checks the board's current *score()* (described in Section 3.2). If it is *O* or *X*, then the game has a winner, which is returned (nothing is done if *score()* returns *InProgress*).
- ***turn_human()*** simply validates input from the human for their turn, checking that the value entered is in bounds and that there is space to play in the column chosen.
- ***turn_bot()*** is basically a wrapper function for *best_move()* (described in Section 3.3). It does the printing and the actual dropping of the piece that was determined to be the best move.
- ***drop_piece()*** takes two inputs: a *Spot* and an integer representing a column (0-6) to drop a piece into. It updates the current game board, dropping a piece and returning

[1] In Rust, an associated function for some type is a function whose first parameter is a reference to an object of that type, and is called like so: *object.function()*.

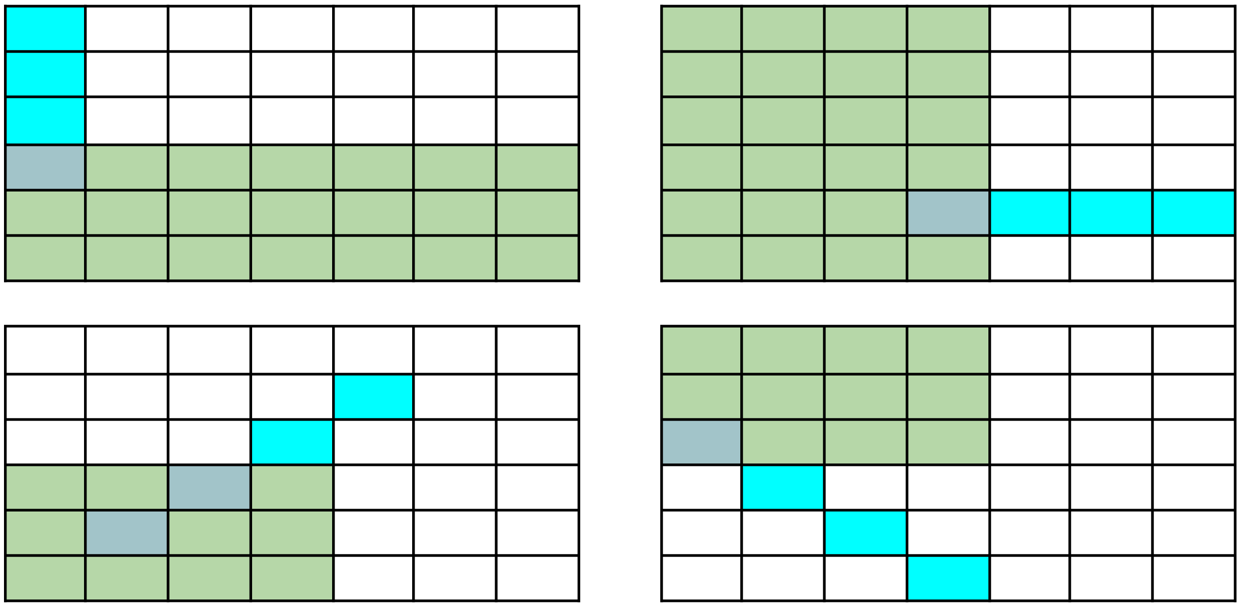
true if the placement is valid, and leaving the board unchanged and returning false if the placement is invalid.

- ***drop_piece_new_board()*** is similar to *drop_piece()*. However, instead of modifying the current "actual" *GameBoard* with the new move, it returns an *Option<GameBoard>* which is either *Some(GameBoard)* if the placement is valid, or *None* if the placement is invalid. This is helpful for the Minmax implementation.

3.2 Scoring System

The scoring system is broken into several functions, each of which calls the function defined directly after it multiple times in such a way that the end result is the sum of the scores for each possible line of 4 cells on the board. Because they all return the *Score* type, any winning line (4 X's or 4 O's in a line) discovered will be recursed up to the initial call, ignoring all the *InProgress* scores calculated from the other positions.

score() is the highest level scoring function that gets called in other parts of the code. It returns the sum of the scores calculated by four calls to *score_area()* using different areas and directions to check (shown in the figure below; green is the area of each call, and blue is one example of a line that gets scored in that area). Note: this part of my design was inspired by Reference 1.



score_area() takes two inputs: an *Area* struct (two *Vec2*s that define a bounding box) to be scored, and a *CheckDir* enum (either *N*, *E*, *NE*, or *SE*) defining the direction to be checked. It iterates over every position in the area, calling *score_pos()* with the correct position and direction, and returns the sum.

score_pos() takes two inputs: a *Vec2* for the location to be scored, and a *CheckDir* for the direction of the line. It then generates a vector containing the correct *Spot*s taken from the *GameBoard*. It also generates a vector of bools representing whether the spots under the line are solid (this might make certain positions more valuable). It then clones both vectors, reverses the clones, then returns the sum of two calls to *score_line()*, one with the forward *Spot* and *bool* vectors, the other with the reversed vectors.

score_line() takes two inputs, both slices[2]: *line* (a *Spot* slice) and *under* (a *bool* slice). This is where the core scoring logic lives. It uses a single match expression that checks for certain combinations of *Spot*s and the equalities between adjacent *Spot*s. In a Rust match expression, the first arm that matches the input will be run. In this example, the first two arms are for four *Spot::O*s / *Spot::X*s in a row, which return *Score::O* and *Score::X* respectively. I came up with the scoring logic partially by considering my own thought process while playing Connect-4. Here is a non-exhaustive list of the cases that contribute to scoring in my program:

- Four in a row => return the appropriate *Score* (described above)
- Three in a row; 4th spot is empty with a solid spot below it => return +/-16 (either positive if the three in a row are *Spot::X* or negative if they are *Spot::O*).
- Two in a row; 3rd and 4th spots are empty with solid spots below => +/- 8
- Two in a row; 3rd is empty with solid below, 4th is ally piece => +/-16
- Two in a row; 3rd is empty with solid below, 4th is enemy piece => +/-4
- Etcetera… (here is *score_line()*'s definition for an exhaustive list).

3.3 Minmax Implementation

best_move() takes two inputs: *spot* (the *Spot* that should be evaluated), and *depth* (an integer representing the depth of the current calculation). It returns an *Option<Move>* (either *Some(Move)* if a valid move was found or *None* if not).

The base call to *best_move()* is in *turn_bot()*, passing *Spot::X* and the constant *MAX_DEPTH*, 4. For every call to *best_move()* it first creates an empty vector of *Move*s called *moves*, used to assess the scores of the board states resulting from each possible move from the current board state. To do so it uses a for loop, iterating *col* from zero to the number of columns in the grid minus one, calling *self.drop_piece_new_board(spot, col)* for each value of *col*. If it returns *Some<GameBoard>* then a new *Move* will be pushed to *moves*. The new *Move*'s *col* will be the current value of *col*, and its *board* will be the new *GameBoard*. If the *new_board*'s score is a winning score, then the new *Move*'s *score* will be set to either the *MAX* possible value of the integer type used if the winner is *Score::X* or the *MIN* if the winner is *Score::O*, and its *depth* will be set to the current value of *depth*.

Otherwise, if the new board's score is *InProgress*, it checks *depth*. If it's 0, the new *Move*'s *score* will be the *InProgress score* from *new_board* and its *depth* will be the current *depth* (0). If *depth > 0*, it recurses, calling *new_board.best_move([spot's opposite], depth - 1)*. If the call returns *None*, the new *Move*'s *score* and *depth* will come from *new_board* (like in the *depth == 0* case). If it returns *Some(Move)*, then the new *Move*'s *score* and *depth* will come from that *Move*.

After the for loop, either the max or min *Move* from *moves* will be returned, depending on *spot*. The "greater" *Move* here is the one with the greater *score*, or if their scores are equal, the *Move* with the greater *depth* (this ensures that immediate wins will be taken when possible, instead of sometimes opting for a setup in which the bot knows it can win in the future, such as *Empty, X, X, X, Empty* – a bug that took much thought to solve).

[2] In Rust, a slice is a reference into some kind of contiguous memory (for example, a vector).

4.0 Conclusions

Considering that I'm still a beginner in the language I chose to use for the bot, I'm quite happy with its level of ability. I probably played at least a couple hundred games of Connect-4 against my bot while creating it, often quickly discovering a bug that led me to rethink how I was designing it to do things. Originally I had it coded as Negamax, but I later changed it to regular Minmax.

As a first priority, if the bot can win on its current move, it will always do so. Second, if the human can win on their next move, the bot will always block them. Beyond that, the bot has a surprising ability to set up favorable positions for itself (although I will admit that I am not the best Connect-4 player in the world). Although it's not perfect, I learned a lot from this project and created a decently competent Connect-4 AI! See the Appendix for a full game played against a "perfect" Connect-4 solver.

I spent a few days trying to implement a version of the bot that used alpha-beta pruning, so that I could increase the depth beyond 4. But for some reason it was always difficult for me to fully wrap my head around it – no matter how I coded it, it seemed like it would surely miss some branches of the possibility tree that could have contributed to the final decision! And indeed, no matter how I coded it, it was very bad at Connect-4. My alpha-beta pruning implementation was a bit difficult to debug, but I spent many hours debugging it anyway. I definitely learned from the time I spent on this; I can imagine how I might apply alpha-beta pruning in other problems…. Sadly, however, I couldn't work it out properly for Connect-4. That being said, here is as far as I got (which was frustratingly close, I suspect).

5.0 References

1. Connect 4 Algorithm (roboticsproject.readthedocs.io)
2. Artificial Intelligence at Play — Connect Four (Mini-max algorithm explained) | by Jonathan C.T. Kuo | Analytics Vidhya | Medium
3. Creating the (nearly) perfect connect-four bot with limited move time and file size | by Gilles Vandewiele
4. Connect 4 Solver (connect4.gamesolver.org)

Appendix: Full Game

Below is a full game of Connect-4 played against this Connect-4 solver (that I didn't make). Here is a link to the final state of the game board on the solver's website. In this output, my bot is X's and the solver is O's ("you"). On the solver's website, my bot is red and the solver is yellow.

```
Starting a game of connect 4!
Who should go first? (O: human,
X: bot) x
Bot's move: 4
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |X| | | |
Your move (1-7 then ⏎): 4
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |O| | | |
| | | |X| | | |
Bot's move: 5
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |O| | | |
| | | |X|X| | |
Your move (1-7 then ⏎): 6
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |O| | | |
| | | |X|X|O| |
Bot's move: 4
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |X| | | |
| | | |O| | | |
| | | |X|X|O| |
Your move (1-7 then ⏎): 4
| | | | | | | |
| | | | | | | |
| | | |O| | | |
| | | |X| | | |
| | | |O| | | |
| | | |X|X|O| |
Bot's move: 6
| | | | | | | |
| | | | | | | |
| | | |O| | | |
| | | |X| | | |
| | | |O| |X| |
| | | |X|X|O| |
Your move (1-7 then ⏎): 6
| | | | | | | |
| | | | | | | |
| | | |O| | | |
| | | |X| |O| |
| | | |O| |X| |
| | | |X|X|O| |
Bot's move: 6
| | | | | | | |
| | | | | | | |
| | | |O| |X| |
| | | |X| |O| |
| | | |O| |X| |
| | | |X|X|O| |
Your move (1-7 then ⏎): 4
| | | | | | | |
| | | |O| | | |
| | | |O| |X| |
| | | |X| |O| |
| | | |O| |X| |
| | | |X|X|O| |
Bot's move: 2
| | | | | | | |
| | | |O| | | |
| | | |O| |X| |
| | | |X| |O| |
| | | |O| |X| |
| |X| |X|X|O| |
Your move (1-7 then ⏎): 3
| | | | | | | |
| | | |O| | | |
| | | |O| |X| |
| | | |X| |O| |
| | | |O| |X| |
| |X|O|X|X|O| |
Bot's move: 3
| | | | | | | |
| | | |O| | | |
| | | |O| |X| |
| | | |X| |O| |
| | |X|O| |X| |
| |X|O|X|X|O| |
```

```
Your move (1-7 then ⏎): 2
| | | | | | | |
| | | |O| | | |
| | | |O| |X| |
| | | |X| |O| |
| |O|X|O| |X| |
| |X|O|X|X|O| |
```

```
Bot's move: 5
| | | | | | | |
| | | |O| | | |
| | | |O| |X| |
| | | |X| |O| |
| |O|X|O|X|X| |
| |X|O|X|X|O| |
```

```
Your move (1-7 then ⏎): 2
| | | | | | | |
| | | |O| | | |
| | | |O| |X| |
| |O| |X| |O| |
| |O|X|O|X|X| |
| |X|O|X|X|O| |
```

```
Bot's move: 1
| | | | | | | |
| | | |O| | | |
| | | |O| |X| |
| |O| |X| |O| |
| |O|X|O|X|X| |
|X|X|O|X|X|O| |
```

```
Your move (1-7 then ⏎): 1
| | | | | | | |
| | | |O| | | |
| | | |O| |X| |
| |O| |X| |O| |
|O|O|X|O|X|X| |
|X|X|O|X|X|O| |
```

```
Bot's move: 2
| | | | | | | |
| | | |O| | | |
| |X| |O| |X| |
| |O| |X| |O| |
|O|O|X|O|X|X| |
|X|X|O|X|X|O| |
```

```
Your move (1-7 then ⏎): 2
| | | | | | | |
| |O| |O| | | |
| |X| |O| |X| |
| |O| |X| |O| |
|O|O|X|O|X|X| |
|X|X|O|X|X|O| |
```

```
Bot's move: 6
| | | | | | | |
| |O| |O| |X| |
| |X| |O| |X| |
| |O| |X| |O| |
|O|O|X|O|X|X| |
|X|X|O|X|X|O| |
```

```
Your move (1-7 then ⏎): 2
| |O| | | | | |
| |O| |O| |X| |
| |X| |O| |X| |
| |O| |X| |O| |
|O|O|X|O|X|X| |
|X|X|O|X|X|O| |
```

```
Bot's move: 1
| |O| | | | | |
| |O| |O| |X| |
| |X| |O| |X| |
|X|O| |X| |O| |
|O|O|X|O|X|X| |
|X|X|O|X|X|O| |
```

```
Your move (1-7 then ⏎): 3
| |O| | | | | |
| |O| |O| |X| |
| |X| |O| |X| |
|X|O|O|X| |O| |
|O|O|X|O|X|X| |
|X|X|O|X|X|O| |
```

```
Bot's move: 3
| |O| | | | | |
| |O| |O| |X| |
| |X|X|O| |X| |
|X|O|O|X| |O| |
|O|O|X|O|X|X| |
|X|X|O|X|X|O| |
```

```
Your move (1-7 then ⏎): 3
| |O| | | | | |
| |O|O|O| |X| |
| |X|X|O| |X| |
|X|O|O|X| |O| |
|O|O|X|O|X|X| |
|X|X|O|X|X|O| |
```

```
Bot's move: 7
| |O| | | | | |
| |O|O|O| |X| |
| |X|X|O| |X| |
|X|O|O|X| |O| |
|O|O|X|O|X|X| |
|X|X|O|X|X|O|X|
Your move (1-7 then ⏎): 5
┃   │ O │   │   │   │   │   ┃
┃   │ O │ O │ O │   │ X │   ┃
┃   │ X │ X │ O │   │ X │   ┃
┃ X │ O │ O │ X │ O │ O │   ┃
┃ O │ O │ X │ O │ X │ X │   ┃
┃ X │ X │ O │ X │ X │ O │ X ┃
You won!
```