Project 3, Fascicle 2: Thinking Deep

Suggested completion date for this Fascicle: May 4

Artificial Intelligence

An AI opponent in a board game is responsible for determining a best move to apply for any given board state. Since we have taken great care to write our application so that it is agnostic to the **exact kind** of game being played, we will be able to write an AI that is agnostic to the exact rules of a particular game, instead using methods like `GetPossibleMoves`, `ApplyMove`, and `UndoMove` to compute an optimal move for a board state.

General AI patterns:

In general, a two-player game AI will take an existing game state and iterate through all possible moves that the AI player could take. For each possible move, it computes a **weight** of the board after applying that move. **Weight** is similar to **advantage** in that it measures "better" board states, but whereas advantage indicates who is **currently winning**, weight shows **who has the better board position** (perhaps having sacrificed in the near-term to secure a long-term advantage). The AI compares the weight after each applying each possible move, and selects the move that results in the most advantageous weight.

We can see how this strategy is short-sighted with an easy thought experiment. Suppose the AI is controlling Black in a chess game, and all its possible moves result in no change of weight, except for one move in which Black's queen captures White's pawn, increasing Black's weight. The AI will select this move to apply, **even if the move leaves the queen in danger**; the AI saw an advantage and took it, without examining its opponent's possible responses. This leads to a simple adjustment: when examining possible moves, the AI should then look at all of the opponent's possible replies, and assume that the opponent will reply with the **best possible move for the opponent**. The AI then selects its own move to apply so that the best possible response for the opponent is of little weight as possible.

But this just delays the problem: now the AI will examine White's possible moves without considering what the Black AI itself would do in response to each of White's possible moves. So prior to picking White's best response to one of Black's initial moves, the AI will consider all of Black's possible responses to each of White's possible responses, selecting the move with the greatest advantage for Black, so that White will select the move minimizing Black's advantage, so that Black will select the move minimizing White's advantage, ...

A theoretically optimal AI would continue this pattern until reaching the end game state for each of the possible responses to each of the possible responses to each of the possible responses to each of the possible responses to .... each of the original possible moves. This is feasible for some games; tic-tac-toe has 9 initial moves, each with 8 responses, each with 7 responses, etc., giving a theoretical maximum of $9! = 362880$ game boards to consider (but not all of those are valid, i.e., some series of moves result in the game ending without all squares filled). That's an easy task for a modern computer to examine. But a chess board has 20 moves in the initial state, with 20 moves in response, and consistently dozens of possible moves for any given state... examining every single response until end of game is simply impossible.

So we compromise. We will program our AI with a **maximum search depth**, giving an upper limit on how many "response to a response to a response" levels to go down searching for a best move. Simple games can use a relatively high depth; more complex games will use a lower depth. All games will use the same style of AI, which we call a **minimax AI**, since it alternates between minimizing the opponent's advantage and maximizing your own advantage.

Minimax:

We will actually program an algorithm called **minimax with alpha-beta pruning**, but that's just a variant of the **minimax** algorithm, so we will use that colloquial name. In minimax, we use a **search tree** over all possible moves, selecting moves for a given player such that the best responses from the opponent are minimized. A psuedocode version of the algorithm looks like this (via Wikipedia):

```
function minimax(board, depth, isMaximizing)
  if depth == 0 or board is finished
    return {board weight, null}

  bestWeight = $-\infty$ if maximizingPlayer, otherwise $+\infty$
  bestMove = NULL
  foreach possibleMove of board:
    apply possibleMove
    w = minimax(board, depth - 1, not isMaximizing)
    undo possibleMove
    if isMaximizing and w > bestWeight
      bestWeight = w
      bestMove = possibleMove
    else if not isMaximizing and w < bestWeight
      bestWeight = w
      bestMove = possibleMove

  return {bestWeight, bestMove}
```

To start the search for a given board, we call

```
minimax(currentBoard, MAX_DEPTH, currentPlayer == 1)
```

to search for a maximum-weight move for player 1, or a minimum-weight move for player 2. The return value from the function is a pair of values: the best move found by the function, and the weight of the board that resulted.

Getting Started

1. Start by finding a partner and agreeing on whose Project 2 you will extend.
2. Examine the `Project3-Updates` repository on GitHub. Find these files, and copy them to the appropriate places in your solution, overriding the old files:
   (a) From **WpfView**: IWpfGameFactory, IGameViewModel, NumberOfPlayers (make sure to Add this existing item to your WpfView project in Visual Studio)
   (b) From **BoardGames.Model**: IGameBoard
   (c) From **Othello**: OthelloBoard, OthelloViewModel, OthelloGameFactory
   (d) From **TicTacToe**: TicTacToeBoard, TicTacToeViewModel, TicTacToeGameFactory
3. Add a new **Class Library (.NET Standard)** project to your solution named `Cecs475.BoardGames.ComputerOpponen`
   Add copies of **IGameAi.cs** and **MinimaxAi.cs** from the repository to your project. Add a reference to `Cecs475.BoardGames.Model`.
4. Examine the commits of the repository, in particular "Wire AI foundations into Othello and TTT views" and "Integrate AI into TTT and Othello" . The commits will show you how to update your solution so that it:
   (a) allows the user to select either a human opponent or an AI opponent when choosing a game; (GameChoiceWindow and GameWindow)

(b) upgrades `ChessViewModel` to use a computer opponent to play for the black player when chosen by the user. (OthelloViewModel as a reference)

5. Build the and run the WPF application. Choosing a computer opponent should currently do nothing, as your `MinimaxAi` class returns `null` from its `FindBestMove` method.
6. Finish the skeleton `MinimaxAi` class, using the pseudocode above to implement a basic minimax algorithm.
7. Test your AI using TicTacToe at a max depth of 9. You should not be able to defeat the AI, and if you aren't careful, you should outright lose.
8. Test your AI against Othello at a max depth of 6. There should be some lag between moves. You will most likely lose here as well.