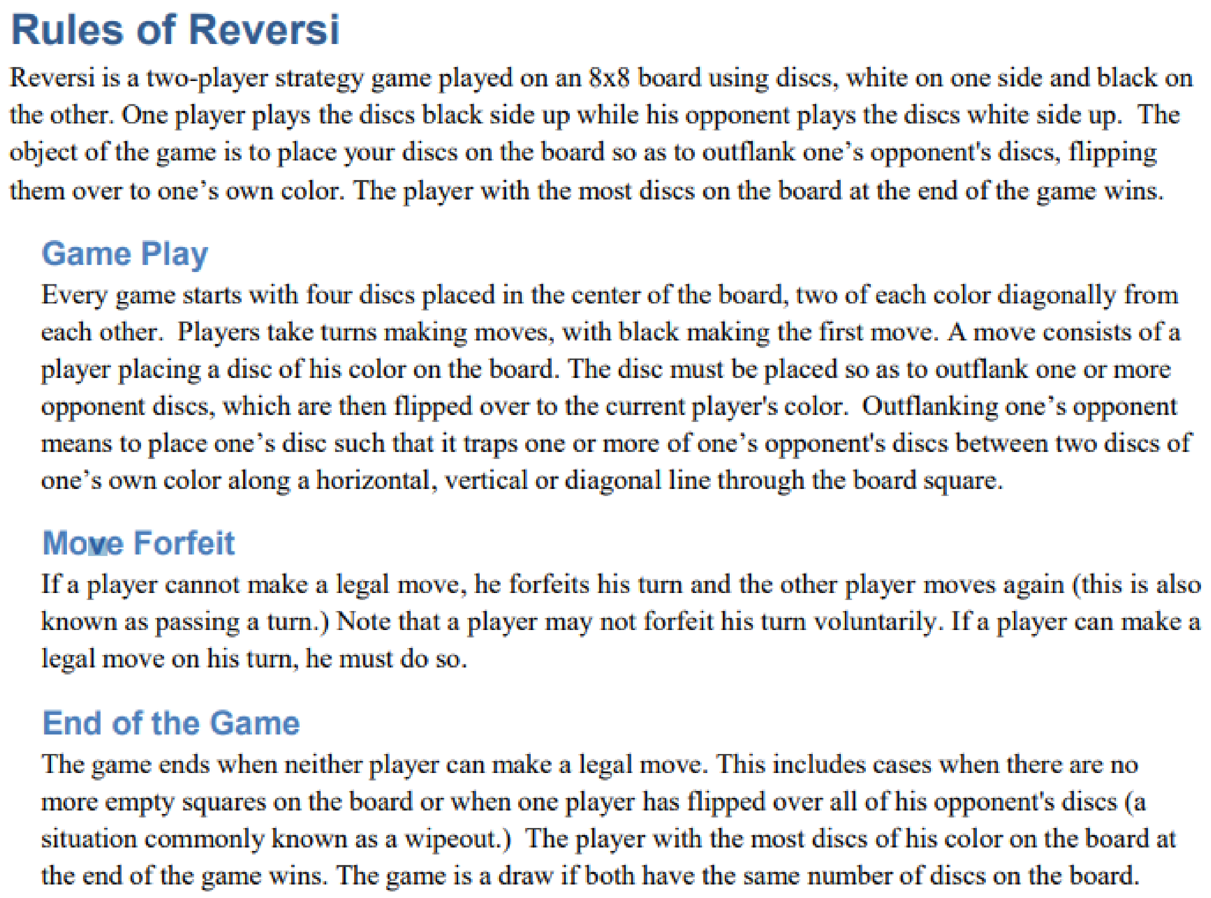
Reversi Minimax

The Minimax Algorithm is a simple, predictive game playing technique where the agent assumes that the opponent makes “perfect” moves. The Minimax Algorithm attempts to maximize the probability of a win but assumes that the opponent will always make the best move and in so doing minimize the probability. This minimizing / maximizing behavior is where Minimax gets its name. Minimax is a type of DepthFirst search where its ply represents a move either by the agent or the opponent. This work involves implementing Minimax Algorithm for an agent built to play the game Reversi (Othello.) As Minimax is a Depth-First search, it would take a long time to complete without some limitations, so a depth limit will be employed to prevent the algorithm from taking too much time.



Rules of Reversi

Reversi is a two-player strategy game played on an 8x8 board using discs, white on one side and black on the other. One player plays the discs black side up while his opponent plays the discs white side up. The object of the game is to place your discs on the board so as to outflank one’s opponent's discs, flipping them over to one’s own color. The player with the most discs on the board at the end of the game wins.

Game Play

Every game starts with four discs placed in the center of the board, two of each color diagonally from each other. Players take turns making moves, with black making the first move. A move consists of a player placing a disc of his color on the board. The disc must be placed so as to outflank one or more opponent discs, which are then flipped over to the current player's color. Outflanking one’s opponent means to place one’s disc such that it traps one or more of one’s opponent's discs between two discs of one’s own color along a horizontal, vertical or diagonal line through the board square.

Move Forfeit

If a player cannot make a legal move, he forfeits his turn and the other player moves again (this is also known as passing a turn.) Note that a player may not forfeit his turn voluntarily. If a player can make a legal move on his turn, he must do so.

End of the Game

The game ends when neither player can make a legal move. This includes cases when there are no more empty squares on the board or when one player has flipped over all of his opponent's discs (a situation commonly known as a wipeout.) The player with the most discs of his color on the board at the end of the game wins. The game is a draw if both have the same number of discs on the board.

Data Structures

This lab project is principally concerned with the implementation of the Minimax Algorithm for determining the "best" move. To that end, the following data structures will be used. (Note: not all methods / properties / fields are listed – see file for details.)

Board

This class represents a Reversi board configuration.

Relevant Fields

`public static readonly int Black, White, Empty;`
These fields should be used to differentiate between players and their pieces.

`public static readonly int Width, Height;`
These fields should be used for bounds checking any board operations.

Relevant Methods

`public void Copy(Board board)`
Use this method instead of the copy constructor to make one board a copy of another.

`public int GetSquareContents(int row, int col)`
Get the value of a given square. Return values correspond to the Black, White or Empty field values.

`public void MakeMove(int color, int row, int col)`
Places a disc for the player on the board and flips any outflanked opponents. Note: For performance reasons, it does NOT check that the move is valid.

`public bool HasAnyValidMove(int color)`
Use this function to determine if a player has a valid move to make. This will be useful for forfeiture in the minmax algorithm.

`public bool IsTerminalState()`
Returns true if the game is over.

`public bool IsValidMove(int color, int row, int col)`
Returns true if a particular move to a row and column is valid.

ComputerMove

This class represents a move made by an AI agent.

Relevant Fields

`public int row, col, rank;`
These fields store the row and column of a move as well as a rank for comparison purposes.

Relevant Methods

`public ComputerMove(int row, int col)`
This is the complex constructor for a ComputerMove object. It is also the only constructor available.

You will build a complete Reversi AI player. A skeleton class is provided.

StudentAI Class

Students will build the `StudentAI` class which resides in the **`StudentAI_TODO.cs`** file. There are no restrictions on the architecture of the class except that its public API must consist of exactly one method, the **`Run()`** method, as outlined below.

Required Method

This method is required of student classes implementing the Reversi AI.

```
public ComputerMove Run(int  nextColor, Board  board, int 
 lookAheadDepth)
```

Implements the Minimax algorithm to determine the best move for this player and return it.

Suggested Helper Methods

These methods are not required but are a recommended part of the construction of the class.

```
private ComputerMove GetBestMove(int color, Board board, int depth)
```

This function is called by Run() with exactly the same arguments as Run(). This is to separate the initial call to your program from recursive calls. Aids debugging and visualization.

NOTE: For alpha-beta pruning, you will have to add alpha, beta, and the parentId of a node to this function's arguments. This is detailed in the pseudocode for alpha-beta in this document.

```
private int Evaluate(Board board)
```

This function determines the value of the board based on position and color of pieces.

You may use the EvaluateTest method within ExampleAI.MinimaxAFI to focus on problems that lie solely in their Minimax algorithm. Any calls to this method should not be left in the code when it's completed. All calls to the EvaluateTest method will output the calculated value to the console.

The EvaluateTest method uses the following rules to determine board value:

1. Squares contain a white piece, black piece, or are empty: white = 1, black = -1, empty = 0
2. If the square is on a corner multiply that square value by 100, if not on a corner but on a side multiply by 10, multiply by 1 for all other squares.
3. Total the values for all squares on the board
4. If the game is over, add 10000 if the previous total is positive or subtract 10000 otherwise.

There are six test cases that will be evaluated:

- Beginner (black) vs. Beginner (white)
- Intermediate (black) vs. Beginner (white)
- Intermediate (black) vs. Intermediate (white)
- Advanced (black) vs. Beginner (white)
- Advanced (black) vs. Intermediate (white)
- Advanced (black) vs. Advanced (white)

Your code will always be tested as the **black** player.

If Full Sail AI vs Full Sail AI has black winning at 34 to 30, your black should win with 34 or more pieces. If Full Sail AI vs Full Sail AI was 20 to 44, you only have to match 20 or more. Winning or losing the game doesn't matter, it's the amount of pieces black captures that matter. If your AI finishes with less pieces than the Full Sail AI, then you will receive proportionally less credit. (If the game finishes before the board is filled, the ratio of white to black pieces is used for evaluating.)

Delivery

Upload a compressed (Zipped) file which should include:

```
StudentAI (folder)
      \
      StudentAI_TODO.cs
```

Place this folder in the ROOT of your zip file, not in a subfolder. DO NOT SUBMIT EXECUTABLES, LIBRARIES, OR OBJECT FILES.