UF UNIVERSITY of FLORIDA
Herbert Wertheim College of Engineering
Computer & Information Science & Engineering

CAP4053

# Manual: Reversi

## Overview

Minimax is a predictive game playing technique. A minimax agent is ***perfectly rational*** – it makes the optimal, rational decision based on the assumption that the opponent is also a perfectly rational agent. This is done by trying to maximize the agent's position by predicting that the opponent will try to minimize that same position.

In this exercise, students will implement a Minimax behavior for the game Reversi (Othello) with a ***depth-limit***. The complete Minimax algorithm traces all possible sequences of actions from the current game state to all possible game outcomes; as such, a complete Minimax search is impractical in typical game scenarios. As such, most such algorithms used in practice employ a depth-limit; the algorithm will search game states only a limited number of actions from the current state, then use a heuristic evaluation as an estimate of the state's value.

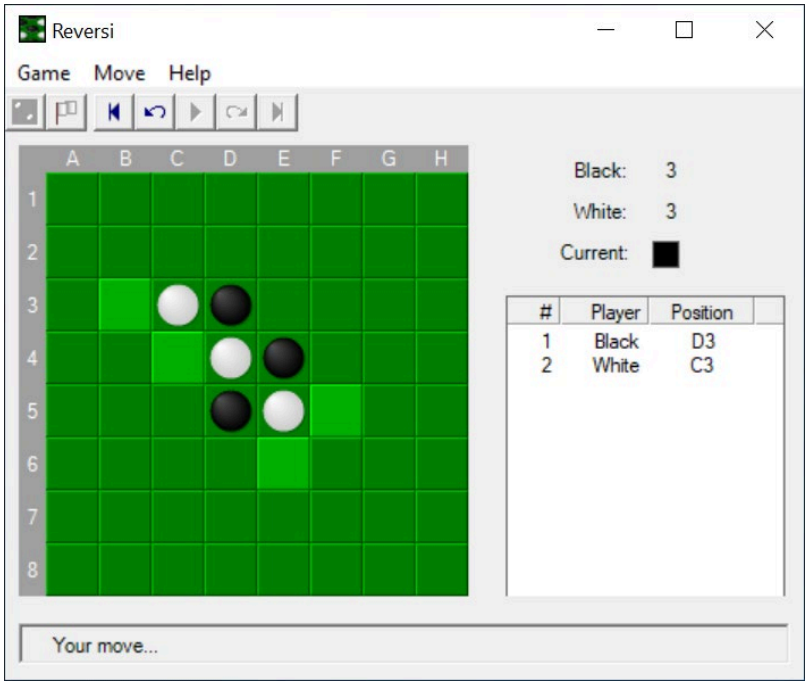


**Figure 1. Reversi C# Application**

## Rules of Reversi

Reversi (**Figure 1**) 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 their opponent plays the discs white side up. The object of the game is to place your discs on the board 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 their color on the board. The disc must be placed 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, they forfeit their turn and the other player moves again; this is also known as passing a turn. Note that a player may not forfeit their turn voluntarily. If a player can make a legal move on their turn, they 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 their opponent's discs – a situation commonly known as a wipeout. The player with the most discs of their 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.

## Structure

In this exercise, students will implement Minimax for Reversi using the interfaces defined in this section.

### Development Interface

The following classes, representing Reversi game elements, are in the `GameAI.GamePlaying.Core` namespace.

#### Board

This class represents the game board state at a given moment in time.

##### *Read-Only Attributes*

```
public static readonly int Black, White, Empty
```
Values used to represent board tile states and identify if they are held by a player or are empty.

```
public static readonly int Width, Height
```
Hold the width and height (bounds) of the game board; used for checking validity of board operations.

##### *Relevant Methods*

```
public void Copy(Board board)
```
Use this method instead of the copy constructor (thus avoiding allocation) to copy `board` over this object.

```
public int GetTile(int row, int column)
```
Returns the state of the game board tile at position (`row, column`), which will be **`Black`**, **`White`**, or **`Empty`**.

```
public void MakeMove(int color, int row, int column)
```
Places a disc for the player specified by `color` on the board and flips any outflanked opponent pieces to `color`. Note: for performance reasons, this method **does not** check that the move in question is valid.

```
public bool HasAnyValidMove(int color)
```
This method will determine if the player of the specified `color` has and valid moves to make. This method is useful in determining whether a player must forfeit a turn when predicting moves.

```
public bool IsTerminalState()
```
Returns **`true`** if this state is an endgame state (i.e., if the game is over and no further moves can be made).

```
public bool IsValidMove(int color, int row, int column)
```
Returns **`true`** if placement of a disc at position (`row, column`) by the player specified by `color` is valid.

### ComputerMove

This class represents a move made by an AI agent.

#### *Read-Write Attributes*

```
public int row, column, rank
```
These attributes hold the `row`, `column`, and `rank` (strategic value) of the potential move.

#### *Relevant Methods*

```
public ComputerMove(int row, int column)
```
Constructor; creates a new object representing placing a piece at the specified `row` and `column`.

## Assignment Interface

Students will implement the AI player behavior, matching the specification outlined in this section. The classes outlined are found in the `GameAI.GamePlaying` namespace.

### StudentAI

Objects of this class represent a unique instance of an AI agent / its game behavior. While implementations must have the methods listed as required, students are also encouraged to write their own private helper methods to deconstruct the problem in order to construct a robust gameplaying agent.

#### *Required Methods*

```
public StudentAI()
```
Constructor; creates a new Minimax AI agent behavior.

```
public ComputerMove Run(int color, Board board, int lookAheadDepth)
```
Implementation of Minimax. Determines the best move for the player specified by `color` on the given `board`, looking ahead the number of steps indicated by `lookAheadDepth`. This method will not be called unless there is at least one possible move for the player specified by `color`.

#### *Recommended Methods*

```
private int Evaluate(Board board)
```
Returns a strategic value estimate for `board` based on position and color of pieces. It is recommended that students implement a method with this functionality to evaluate board states at the lookahead depth cutoff.

## Testing

The `ExampleAI.MinimaxExample` class provides this method so students may incrementally build their agents:

```
public static int EvaluateTest(Board board)
```

This method provides an evaluation of the game board state which is also printed to the console. Please note that this method is **only for testing** and should not be left in student code when submitted; exercise submissions which use this method in the final implementation **will be marked zero**.

The **`EvaluateTest`** method determines a board's strategic value using the following criteria:

1. A square's value is determined by the color of the piece contained: White = 1, Black = -1, Empty = 0.
2. If a square is in a corner, its value is multiplied by 100.
3. If a square is not in a corner but is on the side of the board, its value is multiplied by 10.
4. One the value for each square is identified, all values are added together.
5. If the game is over, value increases by 10,000 if the score is positive / decreases by 10,000 if negative.

The student's implementation should perform at least well as the example AI to receive full credit in all test cases. All tests will be done with the student player as the color **black**. For example, if the Example Intermediate vs Example Beginner results in black winning 33 to 30, then Student Intermediate vs Example Beginner should also result in a win for black with a ratio of 110% (33 to 30 or 22 to 20, for example). At minimum, these tests will be run:

- 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)

# Submissions

Students will submit a **zip file** containing the following file(s) at the end of this exercise on Canvas:

- StudentAI.cs

Place them in the **root directory** of your zip file, not in a subdirectory. Do not submit any other files.