

Nicolas DESCAMPS

Charles CROON

Techniques of Artificial Intelligence

Report: Othello-playing AI

To start the project, run the following command in the root folder in a basic python environment:

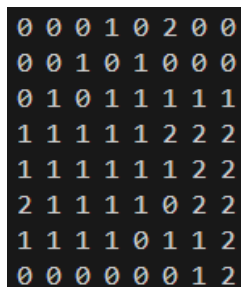
```
> python main.py
```

Rules of the game

Othello is a two-player board game played on an 8x8 grid. Players take turns placing tokens, colored black on one side and white on the other, with the objective of outflanking their opponent's tokens. A move consists of placing a token so that one or more of the opponent's tokens are trapped in a straight line (horizontal, vertical, or diagonal) between the new token and another token of the player's color, then flipping the trapped tokens to the player's color. The game ends when neither player can make a legal move, usually when the board is full. The player with the most tokens of their color on the board at the end of the game wins.

The implementation of the game

First, what we must do is to implement the game itself in python. To do this, we use the class « Game ». It initiates itself with a beginning board represented by an 8*8 matrix, with value from 0 to 2 to describe each square (empty, black token, white token, with player 1 having the black tokens).



0	0	0	1	0	2	0	0
0	0	1	0	1	0	0	0
0	1	0	1	1	1	1	1
1	1	1	1	1	2	2	2
1	1	1	1	1	1	2	2
2	1	1	1	1	0	2	2
1	1	1	1	0	1	1	2
0	0	0	0	0	0	1	2

Example of board

Then many methods are defined to allow a game to be played: get the list of the playable squares for a player, get the number of points that a player wins by playing a token for a certain direction. Basically, those methods are used to implement the main method « play », that allows the player to play a token on a board's square and updates the board according to the play.

If a player's list of playable squares of one player is empty, the game ends, the number of occupied squares is counted, and the winner is displayed. In the previous example, it is player 1's turn and there is no legal play to do: the first player wins.

Running the project

Then, the idea is to make two artificial players play against each other, and compare their win, lose, and draw rate, depending on the type of player they are (their strategy). To rate the efficiency of one type of player, we will make it play 100 games against another type of player. The execution of the "main.py" file displays the following battles:

-First, two random players play against each other. When a player is part of the "randomplayer" class, it chooses a random action in the list of the playable squares when it's its turn. As we can expect it, the result of this battle is very balanced, with an approximately 40% win-rate for the first player, not perfectly balanced because the player going second gets a little advantage.

```
100 games played: 45 wins, 51 losses, 4 draws
The game is quite balanced, though player 2 gets a slight advantage when playing randomly
Press Enter to continue to the next bot...
```

Possible result of the first encounter when running the 'main' file

-What comes next is the "naive" player. While playing, this type of player always chooses the play that makes it win the maximum of points. It's a short-term strategy, but at least it seems better than a random choice. Thus, when the second battle opposing a naive player to a random player comes, this is the type of result you can observe:

```
100 games played: 70 wins, 28 losses, 2 draws, duration: 0.0593 seconds per game
Even a naive player wins a lot against a random player
```

Possible result of the second encounter when running the 'main' file

-Then comes the next and last type of player, that uses the minimax algorithm to decide its plays. For the first use of this player type, we will make it use a tree with a depth of 2. So, it considers its own turn and the other player's next turn. To test its performance, it plays against the precedent naive player:

```
100 games played: 80 wins, 19 losses, 1 draws, duration: 0.5223 seconds per game
With the depth of 2, the minimax bot already has a great win rate against a naive player
Let's see if a minimax bot with a depth of 3 has a greater win rate against the same naive player
```

Possible result of the third encounter when running the 'main' file

-Finally, we make a minimax player with a depth of 3 encounter the naive player, and the result speaks for itself:

```
100 games played: 91 wins, 7 losses, 2 draws, duration: 3.6571 seconds per game
The depth-of-3 minimax bot crushes the naive player!
```

Possible result of the final encounter when running the 'main' file

Project architecture

The project is made of several classes:

- The "Game" class allows to play the game, store a game state and update the game state based on player inputs.
- The "GamePlayer" class allows to start a new game and is made to be used by other classes to interact with the game: it features methods to produce trees of the next possible turns. This class uses the other class "TreeNode" which represents a tree of possible game states and allows to manipulate this tree.
- Then, every other class is a class representing a strategy: random, naive and minimax strategies. Every class allows to simulate N games using a given strategy against another one. The random strategy is tested against another random strategy, the naive strategy is tested against the random strategy and the minimax strategy is tested against the naive one.
- Each strategy class features a function that plays the next turns based on the current state and the chosen strategy.

To output different games at each simulation, every game player needed to feature some randomness. In naive and minimax strategies, in case there are several value-equivalent strategies, the game player picks a random one.

Conclusion

The goal of the project was to achieve a significant win rate at the Othello game using the minimax algorithm. Using the minimax bot with a depth of 3, we have achieved a 90% winrate against a naive player (the typical human Othello player), and the goal is fulfilled.