In the name of God

Shahid Beheshti University_Tehran

Faculty of computer science and engineering

Othello Game Implementation With AI Player

Artificial Intelligence Course

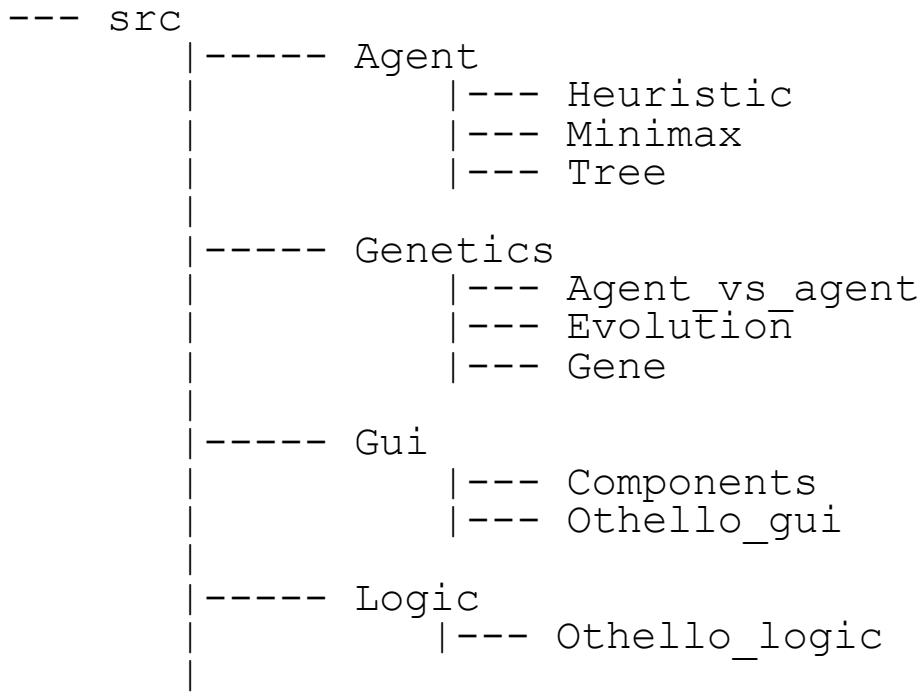
Morteza Kazemi

Shiva Zeymaran

Winter 2021

1. Overview

This project is a complete Othello game that has an intelligent agent to play with users. The project implements in python. The source file of code contains the following packages and python files:



2. Description

2.1. Logic and GUI

Phase one was about implementing the logic of the code and GUI. For this purpose, the “Logic” and “Gui” packages were implemented.

The “Othello_logic.py” contains everything about the logic of the Othello game such as initializing the game board, checking valid movement requests, performing a special movement, and everything that we need to understand the end of the game and calculate the scores. It also considers the rules of the Othello game in every movement.

The “Othello_gui.py”, as it’s clear from its name, contains everything about games’ graphical design using the “TKinter” library of python. This file has an implementation of displaying everything that the user can see when he/she runs the code and also it has a handler for valid clicks on board. The components that are shown in the final window are defined in the “Components.py”; Such as the gray board game, black and white disks, scoreboard, and the color of the player whose turn it is. The components are defined in this file and used to display in the “Othello_gui.py”.

2.2. Agent Strategy

Phase two was about the implementation of an intelligent agent to play and decide like a human.

2.2.1. Minimax and Game Tree

The agent plays using Minimax with alpha-beta pruning to decide its next movement. Minimax is implemented as its traditional algorithm in "Minimax.py"; The extra parts added to the algorithm are as follows:

- The utility of a state is the subtraction of AI player and human player scores(weighted summation of the player's feature values). Feature weights were selected manually in this phase, but they are optimized through the learning algorithm in the third phase.
- To have a deeper search in Minimax tree with the same response time, the width of the tree is reduced using the following strategy:

  The **three** most promising actions(if there are not three actions, chooses as many as possible) causing greater utility values are selected to generate the next level Minimax tree nodes.

- When near the end of the game(game tree depth > 54), instead of choosing the three most promising actions, all possible actions are chosen to continue the Minimax algorithm for the next nodes. Therefore, the entire Minimax tree is traversed and the best possible action would be taken for final movements.

  Note that response time would not be much affected while there are fewer than nine Empty cells left so that the maximum number of actions for each player is less than nine.

The proper depth that gives us good and logical results considering the response time is **nine** for this code, which was obtained by testing various numbers; Accordingly, the Minimax will dive into the game tree up to nine levels and then decide which action is the best one among all possible actions.

Each node of the game tree is an othello_logic by itself that keeps the special state of the game board. The structure of each Node is defined in the “Tree.py”.

2.2.2. Heuristic

To define a proper heuristic function, it's important to extract proper features. Here we have **nine** features that are mapped to each position on the board; Such as, corners, edges, and 4*4 middle square with more details for each of them. To find the number of each feature for each state of the board, we should count the number of disks that are placed in each position and add it to the appropriate feature. Details of setting features are clear in the "Heuristic.py" and for more information, the comments and documentation are available.

The important thing about this part is to **normalize the features** at the end. It's remarkable because features have different valid ranges since the number of corner positions, edge positions, etc. are different from each other.

Features have been normalized by dividing their values by normalization factors so that the ranges will become identical.

2.3. Genetic Algorithm

Phase three focused on evolution to reach the best estimation of weights for the feature vector. The first step was to define the structure of genes. Gene is a class that is implemented in “Gene.py”. Each gene has a list of weights, number of games it plays, number of games it wins or ties(winning has two scores for the winner and a tie has one score for each player), and the fitness of the gene. **The fitness is calculated from the division of total wins by total games.**

The movement logic for playing AI vs AI by using the Minimax algorithm is implemented in “Agent_vs_agent.py” making a few changes to the previous movement logic we had before.

The main part of Genetics implementation is in “Evolution.py”. To run the learning mechanism, the “Main” function in this class must be run. By running this code, it started to generate an **initial population of genes with random weights**. The weights are always between **1-200** and the **population is fixed** in every generation.

The next step is to divide the population into **5 leagues** and let genes play two by two in each league. In this process, the number of wins and

total games will be calculated for each gene. After this stage is completed, it's time to generate the next generation. This work is started by **sorting all genes according to their fitness value.** Now, the logic of generating the next generation is:

- Transferring some of the **best genes** selected **randomly** from the last generation directly to the next
- Transferring **a few worst genes** selected **randomly** from the last generation directly to the next
- Performing crossover and mutation on some of the **best genes** and transferring their children to the next generation
- Performing crossover and mutation on some of the **worst genes** and transferring their children
- Performing crossover and mutation on some of the **best and worst pairs** and transferring their children

Creating the next generation and repeating all the steps above, will continue until we **reach the limit of the number of generations.**

The last remarkable note is about the crossover and mutation procedure. Crossover is started by choosing two **random parents** and a **random "alpha"** which is the parameter that helps us get the **weighted average**

of every two features in two-parent genes. As a result, each feature of a child will be calculated from the following statement:

$$child\,feature = F_{child} = floor(\alpha * F_{parent1} + (1 - \alpha) * F_{parent2})$$

A mutation will affect a few child genes. We have a constant **mutation probability factor** equal to **0.2** that is decreased in every generation (for the first generation this probability is equal to 0.2, 0.2/2 for the next one, 0.2/3, and so on). To perform mutation on each child, first, we get a **random number** between 0 and 1 and then compare the random value with mutation probability. If it is **less** than the probability, the mutation will be done in the following steps:

- Get a random number between 0 and 8 as the index of the feature selected to do mutation on
- Get a random value in the range -50 to 50 as a bias. This bias will be added to the selected feature to reach a new value for a single weight.
- Finally, check the biased value with the valid range of weights (that was 1-200) and change the mutated feature to become in the range.