Sztuczna inteligencja i inżynieria wiedzy

Lista 2

Konrad Kiełczyński 260409

1.Implementacja metod dotyczących gry Reversii

W implementacji gry użyłem kilka przydatnych zmiennych:

BOARD_SIZE - przechowuje rozmiar planszy (8 w tym przypadku)

ALL_POSSIBLE_DIRECTIONS - przechowuje wszystkie możliwe kierunki ruchu, w których mogą zmieniać się kolory pionków na planszy (8 kierunków na planszy kwadratowej)

BOARD - reprezentuje początkowy stan gry dla reversii o wymiarach 8x8

SYMBOL_X - stała przechowująca wartość reprezentującą czarny kolor pionka i jest równoważny **1**

SYMBOL_O - stała przechowująca wartość reprezentującą biały kolor pionka i jest równoważny **2**

SYMBOL_NONE - stała przechowująca wartość reprezentującą puste pole

Wykorzystane funkcje:

calculate_score(board) - funkcja ta oblicza wynik na podstawie aktualnej planszy. Zwraca słownik zawierający ilość symboli X i O na planszy.

is_game_over(board) - funkcja ta sprawdza, czy gra się skończyła. Zwraca True, jeśli nie ma możliwych ruchów dla żadnego z symboli.

is_on_board(x, y) - funkcja ta sprawdza, czy podane współrzędne (x, y) są na planszy. Zwraca True, jeśli są na planszy, w przeciwnym razie zwraca False.

print_board(board) - funkcja ta wypisuje planszę na konsoli w formie graficznej wykorzystując znaki emotikona czyli białych i czarnych pionów.

calc_possible_moves(board, symbol) - funkcja ta oblicza wszystkie możliwe ruchy dla danego symbolu na danym planszy. Wykorzystuje funkcję is_valid_move dla każdego pola na planszy. Zwraca listę możliwych ruchów w postaci krotek (x, y, cords_to_flip), gdzie x i y to współrzędne ruchu, a cords_to_flip to lista krotek z współrzędnymi pól, które należy zmienić, aby dokonać ruchu, nie ma sensu liczyć tego 2 raz.

is_valid_move(board, symbol, x, y) sprawdza, czy ruch symbolu o podanych współrzędnych (x, y) na planszy jest poprawny.

Aby to zrobić, funkcja przeszukuje planszę wzdłuż każdego kierunku z listy ALL_POSSIBLE_DIRECTIONS, szukając sąsiadów o symbolu przeciwnika. Jeśli znajdzie takiego sąsiada, to zapisuje go do listy valid i kontynuuje przeszukiwanie w tym kierunku dopóki znajduje kolejnych sąsiadów o symbolu przeciwnika. Kiedy natrafi na puste pole lub wyjdzie poza planszę, przeszukiwanie w danym kierunku zostaje zakończone.

Jeśli na końcu danego kierunku funkcja znajdzie symbol gracza, to oznacza to, że ruch jest poprawny i funkcja dodaje listę valid do listy change_symbol. W ten sposób uzyskujemy pozycje, na których przeciwnik zmienił swoje symbole na symbole gracza.

Jeśli lista change_symbol nie jest pusta, to funkcja zwraca pozycję (x, y) na której gracz postawił swój symbol, oraz listę change_symbol z pozycjami, na których przeciwnik zmienił swoje symbole na symbole gracza. W przeciwnym przypadku funkcja zwraca wartość logiczną False.

make_move(board, symbol, move) - funkcja ta dokonuje ruchu na planszy dla danego symbolu na podstawie podanej krotki move (x, y, cords_to_flip). Zwraca planszę po dokonaniu ruchu.

2.Heurystyki

Corners: polega na ocenianiu liczby zajętych narożników przez każdego gracza. Zajęcie narożnika uważane jest za korzystne, ponieważ jest to pozycja stała i przeciwnik nie może odwrócić pionka w narożniku. Dlatego warto starać się zajmować narożniki jak najwcześniej w grze. Narożniki to także punkty kontrolne na planszy, ponieważ od nich zaczyna się budowanie stabilnych pozycji.

Edges: polega na ocenianiu liczby zajętych krawędzi przez każdego gracza. Krawędzie to mniej korzystna pozycja niż narożniki, ale wciąż są cennymi punktami, ponieważ zapewniają stabilną pozycję i utrudniają przeciwnikowi wykonanie ruchu.

Coin parity: polega na ocenianiu różnicy w liczbie pionków gracza i przeciwnika na planszy. Jeśli gracz ma więcej pionków niż przeciwnik, to jego pozycja jest korzystna

Mobility: polega na ocenianiu liczby możliwych ruchów dla każdego gracza w danej sytuacji. Posiadanie większej liczby możliwych ruchów uważane jest za korzystne, ponieważ daje graczowi więcej opcji do wykonania strategicznych ruchów. Dlatego warto starać się utrzymać dużą liczbę możliwych ruchów i utrudnić przeciwnikowi wykonanie ruchu.

Stability: polega na ocenianiu stabilności pionków na planszy. Stabilny pionek to taki, którego przeciwnik nie może przewrócić w żadnym kierunku. Dlatego warto starać się stawiać pionki w taki sposób, aby utrudnić przeciwnikowi wykonanie ruchu, który zniszczyłby stabilne pozycje na planszy.

Do każdej z tych heurystyk wykorzystywana jest różnica procentowa w celu porównania wyników obu graczy i uzyskania miary ich względnej siły. Dzięki temu można szybko ocenić, który gracz ma przewagę w danym aspekcie gry, np. w ilości posunięć, które może wykonać, ilości pionków na planszy czy stabilności ich ustawienia.

Różnica procentowa jest preferowana wobec innych miar, takich jak różnica bezwzględna lub stosunek, ponieważ pozwala ona na bezpośrednie porównanie wartości obu graczy, niezależnie od skali i wartości tych wartości. Innymi słowy, **różnica procentowa skupia się na proporcjach, a nie na wartościach bezwzględnych**, co jest bardziej adekwatne do porównań względnych między graczami w grze, w której ilości posiadanych pionków, czy możliwości ruchu, są stałe i znane

3.Strategie

- STATIC STRATEGY
  Strategia w której każde pole ma przypisaną wagę np. cornery są najbardziej punktowane 4 a pola obok nich najmniej, następnie sumowanie tych wartości dla każdego z symboli na planszy (gracz X i gracz O) w celu obliczenia wyniku.

```
STATIC_BOARD = [
    [ 4, -3,  2,  2,  2,  2, -3,  4],
    [-3, -4, -1, -1, -1, -1, -4, -3],
    [ 2, -1,  1,  0,  0,  1, -1,  2],
    [ 2, -1,  0,  1,  1,  0, -1,  2],
    [ 2, -1,  0,  1,  1,  0, -1,  2],
    [ 2, -1,  1,  0,  0,  1, -1,  2],
    [-3, -4, -1, -1, -1, -1, -4, -3],
    [ 4, -3,  2,  2,  2,  2, -3,  4]
]
```

  Ta strategia jest prosta i łatwa do zaimplementowania, ale ma kilka wad. Po pierwsze, nie bierze pod uwagę rzeczywistych możliwości ruchów graczy na planszy, co może prowadzić do błędnych wyników w przypadku, gdy jeden gracz ma więcej możliwości ruchu niż drugi. Po drugie, wartości w tablicy STATIC_BOARD są ustalone na podstawie pewnych założeń i mogą nie odzwierciedlać rzeczywistych warunków na planszy w trakcie gry.

- STABLE STRATEGY
  Kolejną strategią jest Stable strategy która nadaje pewne wagi konkretnym heurystyką z punktu 2. Biorąc pod uwagę ważność tych heurystyk podczas gry. Wagi do tych statystyk zostały wzięte z pracy naukowej.
  https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf

```
def stable_strategy(board, max_symbol):
    weights = {"corners": 30,
               "mobility": 5,
               "coin_parity": 25,
               "edges": 10,
               "stability": 25}
    return board_current_score(board, max_symbol, weights)
```

- ADAPTIVE STRATEGY
  Ta strategia jest rozwinięciem poprzedniej strategii. W tym podejściu dzielimy grę na fazy ze względu na ilość postawionych pionów na planszy.

  Faza otwarcia: gdy jest mało pionków na planszy, ważne jest zagwarantowanie sobie jak największej mobilności, czyli jak największej liczby dostępnych ruchów, co pozwala na bardziej elastyczną grę i łatwiejsze zajęcie korzystnych pozycji. Również równowaga monet jest w tej fazie ważna, ponieważ pozwala na utrzymanie przewagi w ilości posiadanych pionków, co jest kluczowe dla strategii w dalszej części gry

  Faza środka gry: gdy plansza jest już bardziej zapełniona, wszystkie heurystyki stają się ważne, ale mobilność i stabilność są nieco ważniejsze niż narożniki i równowaga monet.

Mobilność pozwala na zachowanie inicjatywy w grze, a stabilność na kontrolę planszy i blokowanie ruchów przeciwnika

Faza końcowa: gdy na planszy pozostaje tylko kilka pionków, narożniki i stabilność stają się najważniejszymi heurystykami, ponieważ kontrola narożników i zajęcie stabilnych pozycji to klucz do zwycięstwa. Mobilność w tej fazie jest nadal ważna, ale traktowana jako mniej istotna niż w poprzednich fazach gry.

```
if num_pieces < 20:
    # most important mobility and coin_parity
    weights = {"corners": 15, "mobility": 15, "coin_parity": 30, "edges":
10, "stability": 5}
elif num_pieces < 50:
    # all heuristics are important
    weights = {"corners": 20, "mobility": 20, "coin_parity": 20, "edges":
15, "stability": 25}
else:
    # corners and stability are the most important
    weights = {"corners": 35, "mobility": 10, "coin_parity": 20, "edges":
10, "stability": 25}
return board_current_score(board, max_symbol, weights)
```

Porównanie

```
('Random Move', {'wins': 1, 'losses': 5, 'ties': 0, 'matches': ['Random
('Alpha-Beta (Adaptive)', {'wins': 5, 'losses': 1, 'ties': 0, 'matches'
('Alpha-Beta (Stable)', {'wins': 3, 'losses': 2, 'ties': 1, 'matches':
('Alpha-Beta (Static)', {'wins': 2, 'losses': 3, 'ties': 1, 'matches':
```

Wyniki pokazują, że algorytm "Alpha-Beta (Adaptive)" okazał się najskuteczniejszy, wygrywając 5 z 6 rozegranych gier, a "Random Move" był najmniej skuteczny, wygrywając tylko jedną grę. "Alpha-Beta (Stable)" i "Alpha-Beta (Static)" okazały się być podobnie skuteczne, z wynikami 3 zwycięstw, 2 porażek i 1 remisu dla "Alpha-Beta (Stable)" oraz 2 zwycięstw, 3 porażek i 1 remisu dla "Alpha-Beta (Static)".

Analizując wyniki, można zauważyć, że algorytm "Alpha-Beta (Adaptive)" okazał się bardziej skuteczny niż inne, co sugeruje, że adaptacyjna strategia wyboru ruchu jest bardziej efektywna niż statyczna strategia, która nie zmienia swojego podejścia w zależności od zmieniających się warunków gry. Wgłębiając się w wyniki algorytm " Alpha-Beta (Stable) " jest lepszy pokonał on lidera korzystającego z adaptacyjnej heurystyki. Podczas gdy heurytyka oparta na statycznej tablicy przegrała z randomowym wyborem ruchów

4.Minmax

Algorytm minmax to popularny algorytm w dziedzinie sztucznej inteligencji, wykorzystywany przede wszystkim w grach dwuosobowych o sumie zerowej, takich jak szachy, warcaby, czy go. Jego celem jest znalezienie optymalnej strategii dla jednego z graczy w sytuacji, gdy drugi gracz będzie starał się utrudnić mu osiągnięcie tego celu.

Algorytm ten opiera się na metodzie przeszukiwania drzewa gry, w którym wierzchołkami są różne stany gry, a krawędzie reprezentują możliwe ruchy graczy. W każdym wierzchołku algorytm wybiera ruch, który maksymalizuje szansę na wygraną pierwszego gracza, przy założeniu, że drugi gracz będzie dążył do minimalizacji tej szansy.

get_move(board, symbol, score_heuristic) jest punktem wejścia do algorytmu. Przyjmuje bieżącą planszę gry, symbol aktualnego gracza i funkcję heurystyki punktacji. Funkcja ta wyznacza listę możliwych ruchów dla danego symbolu i dla każdego z tych ruchów uruchamia funkcję _minimax().

_minimax(board, symbol, depth, is_max_round, score_heuristic) to prywatna metoda i właściwy algorytm minimax. Przyjmuje bieżącą planszę gry, symbol aktualnego gracza, bieżącą głębokość przeszukiwania drzewa, flagę określającą, czy to jest runda maksymalna, czy minimalna oraz funkcję heurystyczną. Algorytm ten rekurencyjnie przeszukuje drzewo stanów gry, wybierając najlepszy ruch dla każdego symbolu.

Jeśli dany symbol nie ma możliwych ruchów, algorytm przechodzi do następnego symbolu i wykonuje tę samą procedurę dla niego. Jeśli nie ma już żadnych możliwych ruchów dla wszystkich symboli, oznacza to koniec gry. Wtedy algorytm wywołuje się ponownie dla przeciwnika, zwiększając głębokość o 1 i zmieniając flagę na przeciwną.

Jeśli algorytm osiągnie maksymalną głębokość drzewa, zostanie wywołana funkcja heurystyczna, która zwraca wartość punktową dla bieżącej planszy.

5.Alpha–beta pruning

Algorytm minimax przeszukuje całe drzewo gry, co jest niepraktyczne dla większości gier, rozwiązaniem jest modyfikacja tego algorytmu znana Alpha-beta pruning.

Idea alpha-beta pruning polega na tym, że algorytm przeszukiwania drzewa przerywa przeszukiwanie poddrzewa, gdy już wiadomo, że nie może ono prowadzić do lepszego wyniku. W tym celu używa dwóch parametrów - alfy i bety.

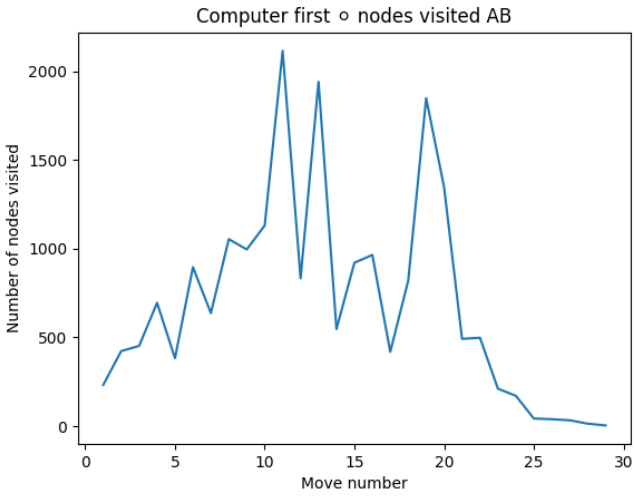
- **Alpha**: Najlepszy (najwyższej wartości) wybór, jaki znaleźliśmy dotychczas w dowolnym punkcie na ścieżce maksymalizującej. Początkowa wartość alfa wynosi -∞.
- **Beta**: Najlepszy (najniższej wartości) wybór, jaki znaleźliśmy dotychczas w dowolnym punkcie na ścieżce minimalizującej. Początkowa wartość beta wynosi +∞.

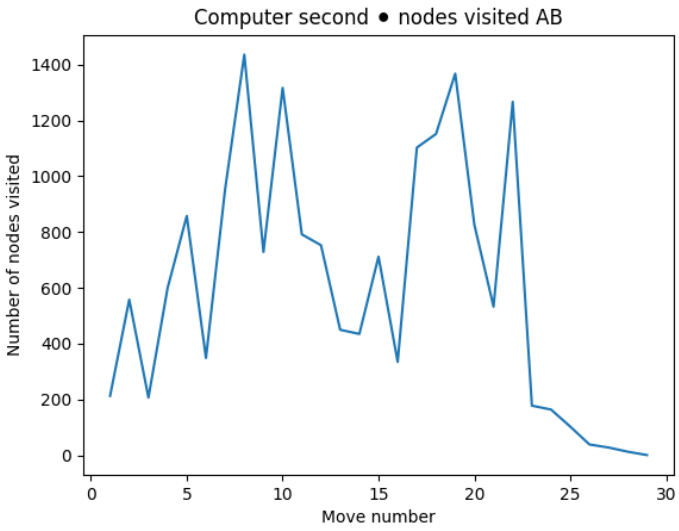
Algorytm alpha-beta pruning przeszukuje drzewo rekurencyjnie, wywołując się na kolejnych węzłach. Dla każdego węzła wyznacza wartość alfa i beta, na podstawie której decyduje, czy dalsze przeszukiwanie jest potrzebne.

```
if alpha >= beta:
    break
```

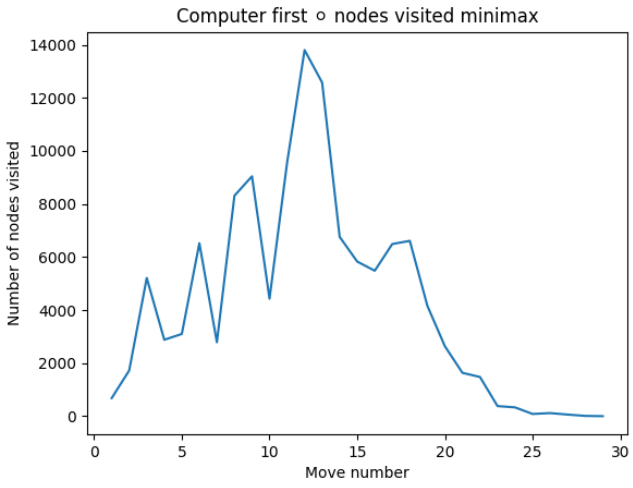
W takiej sytuacji algorytm alpha-beta pruning przerywa dalsze przeszukiwanie gałęzi, ponieważ nie ma już sensu szukać dalej - minimalizujący gracz nie osiągnie już lepszego wyniku, a maksymalizujący gracz już osiągnął najlepszy możliwy wynik.

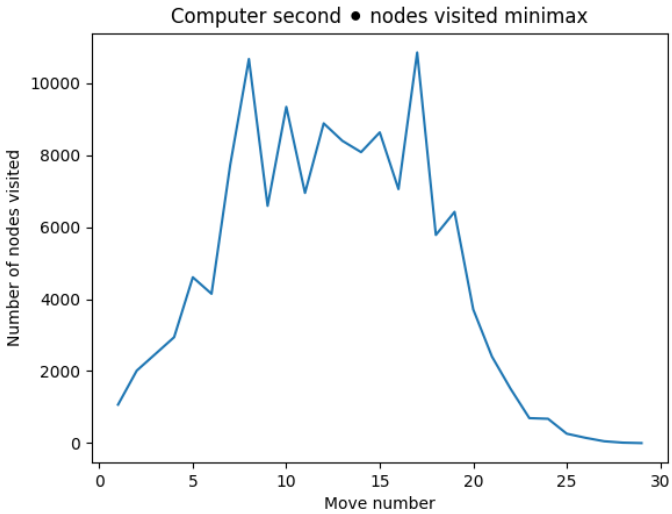
Porównanie liczby odwiedzonych węzłów AB vs Minmax dla głębokości 4





Alpha Beta





Minimax

Algorytm Alpha-beta umożliwia przyspieszenie przeszukiwania drzewa poprzez cięcie krawędzi, które nie mają wpływu na wynik, co jest szczególnie widoczne na wykresach (głębokośc drzewa była ustawiona na 4).

Jednakże, w miarę wzrostu głębokości drzewa, różnica w wydajności pomiędzy algorytmem Alpha-beta a klasycznym podejściem będzie coraz większa. Alpha-beta jest więc preferowanym algorytmem do przeszukiwania drzew decyzyjnych, ponieważ pozwala na szybsze przeszukiwanie drzewa przy jednoczesnym zachowaniu wyniku optymalnego.

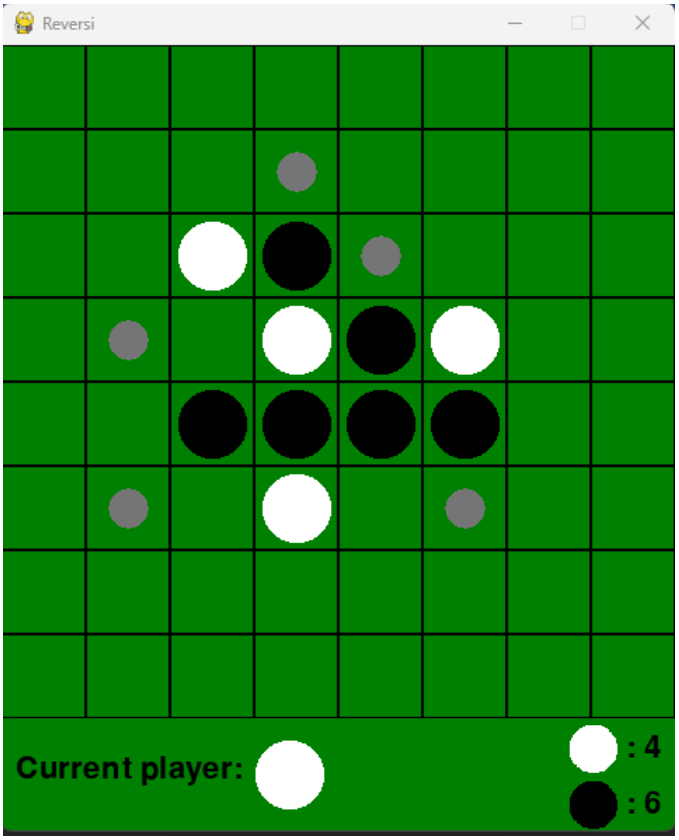
6.Zaimplementowane gry

CLI:

Gra w formacie comnad line interface gdzie można poprzez zmiany w kodzie grać albo PvP, CvC oraz PvC

GUI:

Gra w formacie graphical user interface zapimplementowana w pyGameA gdzie można poprzez zmiany w kodzie grać albo PvP, CvC oraz PvC



Wersja wykonująca jeden ruch:

W linii poleceń podaje się dwa argumenty: symbol gracza (którym gra komputer) oraz nazwę pliku, w którym zapisane jest aktualne ustawienie planszy.

Program wczytuje ustawienie planszy z pliku JSON, a następnie wywołuje funkcję **play_one_move** z modułu **reversi** i **alpha_beta**, która wyznacza ruch dla komputera, korzystając z algorytmu alfa-beta cięcia i adaptacyjnej strategii heurystycznej. Jeśli wyznaczony ruch jest możliwy do wykonania, program wykonuje ten ruch i zwraca zmodyfikowaną planszę.

Na końcu program zapisuje zmodyfikowaną planszę do pliku JSON, nadpisując poprzednie ustawienie planszy.

W ten sposób program umożliwia grę w Reversi z komputerem, gdzie kolejne ruchy podejmuje tylko komputer.

```
if __name__ == '__main__':
    if len(sys.argv) < 3:
        print("Usage: python one_move.py <symbol> <board file>")
        exit()

    board_file = sys.argv[2]
    current_player = int(sys.argv[1])
    with open(board_file, "r") as f:
        board = json.load(f)

    board = play_one_move(board, current_player)
    with open(board_file, 'w') as f:
        json.dump(board, f)
```

Dodatkowy koordynator dla jednego ruchu:

Ten program jest koordynatorem gry w Reversi pomiędzy dwoma komputerami.

Program tworzy plik board.json na podstawie pliku clean_board.json, który zawiera początkowe ustawienie planszy. Następnie program wywołuje funkcję is_game_over z modułu reversi, aby sprawdzić, czy gra nie została zakończona.

Jeśli gra nie jest jeszcze skończona, program wywołuje skrypt one_move.py z argumentami current_player (oznaczającym symbol gracza) i board_file (nazwa pliku z ustawieniem planszy).

Po wykonaniu ruchu przez komputer, program sprawdza, czy gra nie została zakończona. Następnie program zmienia gracza, który wykonuje następny ruch, a także wczytuje aktualne ustawienie planszy z pliku board.json.

Gdy gra się kończy, program wyświetla końcowy wynik gry oraz informację o zwycięzcy lub remisie.

```
if __name__ == '__main__':
    clear_board_file = 'clean_board.json'
    board_file = 'board.json'

    shutil.copy2(clear_board_file, board_file)
    current_player = reversi.SYMBOL_X

    with open(board_file, 'r') as f:
        board = json.load(f)
    while not reversi.is_game_over(board):
        subprocess.run(['python', 'one_move.py', str(current_player),
board_file])
```

```
        current_player = reversi.SYMBOL_X if current_player ==
reversi.SYMBOL_O else reversi.SYMBOL_O
        with open(board_file, 'r') as f:
            board = json.load(f)

    # Print the final score
    reversi.print_board(board)
    score = reversi.calculate_score(board)
    if score[reversi.SYMBOL_X] > score[reversi.SYMBOL_O]:
        print("Player ◯ wins!")
    elif score[reversi.SYMBOL_X] < score[reversi.SYMBOL_O]:
        print("Player ● wins!")
    else:
        print("Tie game!")
```

Solution:
https://github.com/Golden3x11/AI/tree/main/decision_games_reversi