

Wykorzystanie algorytmu MiniMax do gry w Reversi

Karolina Łukasik

19 maja 2023

Spis treści

1	Wstęp	2
2	Stan gry oraz możliwe ruchy dla gracza	2
3	Strategie - zbiór heurystyk	3
3.0.1	Największa liczba dysków gracza	3
3.0.2	Największa liczba stabilnych dysków gracza	3
3.0.3	Unikanie stawiania krążków na polach przyległych do pustego narożnika .	4
3.0.4	Mobilność	4
3.0.5	Bezpieczne pola	4
3.0.6	Strategie odwrotne	4
4	Implementacja algorytmu Minimax	5
4.1	Alpha Beta cięcie	5
5	Symulacje gry w Othello	6
6	Podsumowanie	7

1 Wstęp

Poniższy raport zawiera teoretyczne opisy idei, metod i algorytmów za pomocą których podeszłam do problemu wykorzystania algorytmu Minimax do gry w Reversi. Jest on uzupełnieniem **notatnika Jupyter**, w którym znajdują się funkcje i algorytmy wraz z przykładami ich użycia.

Zacniemy od zdefiniowania stanu gry oraz sposobu generowania możliwych ruchów dla graczy. Następnie zastanowimy się, jakie strategie oceny stanu gry mogą przyjąć gracze, by wybierać możliwie najlepsze ruchy. Dalej przejdziemy do samego algorytmu Minimax, zaimplementujemy go z perspektywy gracza 1 i wykorzystamy do zasymulowania rozgrywki. Na koniec ulepszymy algorytm za pomocą alfa-beta cięcia i porównamy działanie obu wersji.

2 Stan gry oraz możliwe ruchy dla gracza

By móc poprawnie zinterpretować stan gry, musimy przyjąć następujące podstawowe założenia:

- Rozważamy wersję Othello gry Reversi
- Gracz czarny odpowiada graczowi 1 i odpowiedniej liczbie '1' w polu
- Gracz biały odpowiada graczowi 2 i odpowiedniej liczbie '2' w polu
- Pole z zawartością '0' jest puste
- Układ początkowy odpowiada czterem dyskom w środku planszy, na zmianę 1 i 2
- Przyjmijmy, że $L1$ = liczba krążków gracza 1 na planszy, $L2$ = liczba krążków gracza 2 na planszy. Jeśli obaj gracze nie mają możliwości wykonania ruchu, to $L1 > L2$ oznacza wygraną gracza 1, a $L2 > L1$ wygraną gracza 2.

W celu znalezienia wszystkich możliwych ruchów dla konkretnego stanu planszy i danego gracza zaimplementowałam funkcję *PossibleMoves(player, game_array)*. Algorytm w niej zawarty polega na przejściu kolejno po każdym polu na planszy i sprawdzeniu, czy gracz może położyć na tym polu swój dysk. Warunkiem koniecznym, choć niewystarczającym jest, by to pole było puste. Dalej sprawdzamy, czy w którymś z 8 kierunków (dół, góra, boki i po skosie) jest seria dysków przeciwnika zakończona dyskiem gracza. Za pomocą pętli działającej, póki nie wyjdziemy poza planszę, oddalamy się co jedno pole, analizując je:

- jeśli trafiliśmy na puste pole, oznacza to, iż ruch nie jest możliwy w tę stronę i przerywamy pętlę dla tego kierunku ruchu,
- jeśli trafiliśmy na swój własny dysk i nie jest to pole przyległe do sprawdzanego, oznacza to, iż znaleźliśmy możliwy ruch, możemy przerwać szukanie dla tego pola i dodać ruch do listy możliwych ruchów,
- jeśli trafiliśmy na swoje pole, ale jest ono przyległe do sprawdzanego, oznacza to brak możliwości ruchu w tym kierunku.

Na koniec zwracamy listę możliwych ruchów w formie listy krotek.

W celu uzyskania nowych plansz dla każdego ruchu stworzyłam funkcję *PossibleBoards(player, game_array)*. Zwraca ona słownik, w którym kluczami są krotki z możliwymi ruchami, a wartościami odpowiadające im plansze. Funkcja korzysta z wyznaczonych za pomocą poprzedniej funkcji możliwych ruchów. Dla każdego z nich kopiuję aktualną planszę i sprawdza w każdym z 8 kierunków, czy ruch umożliwił przewrócenie dysków. Jeśli tak to zmienia liczby w polach na numer gracza.

```

1  exampl_board= np.array([
2      [0, 0, 0, 0, 0, 0, 0, 0],
3      [0, 0, 0, 0, 0, 0, 0, 0],
4      [0, 0, 2, 1, 0, 0, 0, 0],
5      [0, 0, 0, 1, 2, 0, 0, 0],
6      [0, 0, 1, 1, 2, 2, 0, 0],
7      [0, 1, 1, 2, 0, 2, 0, 0],
8      [0, 0, 2, 0, 0, 2, 0, 0],
9      [0, 0, 0, 0, 0, 0, 0, 0]])
10
[10] ✓ 0.0s

▷ 1 PossibleMoves(2, exampl_board)
[11] ✓ 0.0s
... [(1, 2), (1, 3), (2, 4), (3, 1), (3, 2), (4, 0), (4, 1), (5, 0), (6, 1)]

1 PossibleBoards(2, exampl_board)
[12] ✓ 0.0s
... {(1,
2): array([[0, 0, 0, 0, 0, 0, 0, 0],
           [0, 0, 2, 0, 0, 0, 0, 0],
           [0, 0, 2, 2, 0, 0, 0, 0],
           [0, 0, 0, 1, 2, 0, 0, 0],
           [0, 0, 1, 1, 2, 2, 0, 0],
           [0, 1, 1, 2, 0, 2, 0, 0],
           [0, 0, 2, 0, 0, 2, 0, 0],
           [0, 0, 0, 0, 0, 0, 0, 0]]),
(1,
3): array([[0, 0, 0, 0, 0, 0, 0, 0],
           [0, 0, 0, 2, 0, 0, 0, 0],
           [0, 0, 2, 2, 0, 0, 0, 0],
           [0, 0, 0, 1, 2, 0, 0, 0],
           [0, 0, 1, 1, 2, 2, 0, 0],
           [0, 1, 1, 2, 0, 2, 0, 0],
           [0, 0, 2, 0, 0, 2, 0, 0],
           [0, 0, 0, 0, 0, 0, 0, 0]])}

```

Rysunek 1: Funkcje wyznaczające możliwe ruchy i odpowiadające im posunięcia

3 Strategie - zbiór heurystyk

Poniżej zdefiniuję kilka rodzajów heurystyk, za pomocą których algorytm będzie oceniał stan gry dla poszczególnych graczy.

3.0.1 Największa liczba dysków gracza

Pierwszą, tylko intuicyjnie dobrą strategią jest wybieranie ruchów, które odwrócą największą ilość dysków. Niemniej jest to popularna strategia, która znajdzie się jako jedna ze składowych funkcji heurystycznej.

3.0.2 Największa liczba stabilnych dysków gracza

Jest to bardzo ważna strategia, faworyzująca ruchy zapewniające największą liczbę stabilnych pozycji, niemożliwych do przejścia przez przeciwnika. Była ona również najtrudniejsza w implementacji.

3.0.3 Unikanie stawiania krążków na polach przyległych do pustego narożnika

Strategia pozwalająca uniknąć przejścia narożnika przez przeciwnika.

3.0.4 Mobilność

Strategia faworyzująca ruchy zapewniające większą liczbę możliwych ruchów w kolejnym posunięciu.

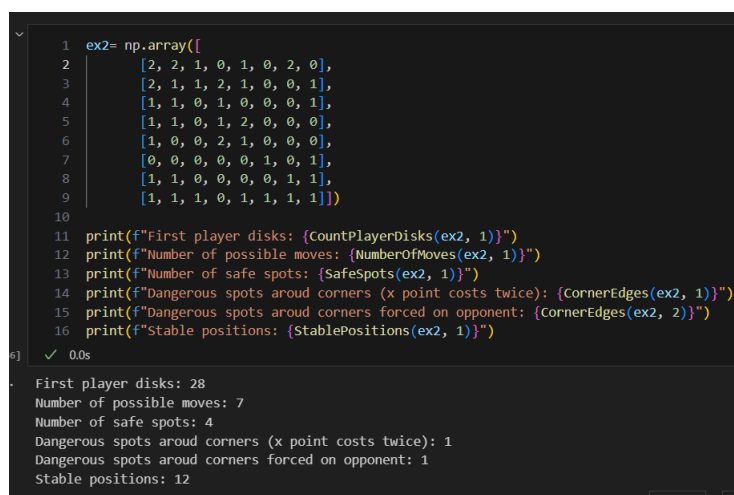
3.0.5 Bezpieczne pola

Strategia premiująca pozostawianie pól, których przeciwnik nie jest w stanie zająć, do wykorzystania później.

3.0.6 Strategie odwrotne

Powyższe strategie charakteryzują się tym, że gracze dążą do jak najczęstszego występowania powyższych cech u siebie i jednocześnie jak najrzadszego u przeciwnika. Poza tym mamy do czynienia z grą o sumie zerowej. Z tego względu gracz drugi będzie minimalizował wartość heurystyk opisujących sytuację gracza 1. Możemy również zastosować powyższe strategie do opisu sytuacji gracza 2, przypisując heurystykom ujemne wagi. W ten sposób uzyskujemy nowe odwrotne strategie:

- Najmniejsza liczba dysków przeciwnika.
- Najmniejsza liczba stabilnych dysków przeciwnika.
- Przeciwnik posiada krążki wokół narożników.
- Ograniczenie liczby możliwych ruchów przeciwnikowi.
- Ograniczenie bezpiecznych pól przeciwnika.



```
1 ex2= np.array([
2     [2, 2, 1, 0, 1, 0, 2, 0],
3     [2, 1, 1, 2, 1, 0, 0, 1],
4     [1, 1, 0, 1, 0, 0, 0, 1],
5     [1, 1, 0, 1, 2, 0, 0, 0],
6     [1, 0, 0, 2, 1, 0, 0, 0],
7     [0, 0, 0, 0, 0, 1, 0, 1],
8     [1, 1, 0, 0, 0, 0, 1, 1],
9     [1, 1, 1, 0, 1, 1, 1, 1])
10
11 print(f"First player disks: {CountPlayerDisks(ex2, 1)}")
12 print(f"Number of possible moves: {NumberOfMoves(ex2, 1)}")
13 print(f"Number of safe spots: {SafeSpots(ex2, 1)}")
14 print(f"Dangerous spots around corners (x point costs twice): {CornerEdges(ex2, 1)}")
15 print(f"Dangerous spots around corners forced on opponent: {CornerEdges(ex2, 2)}")
16 print(f"Stable positions: {StablePositions(ex2, 1)}")
```

First player disks: 28
Number of possible moves: 7
Number of safe spots: 4
Dangerous spots around corners (x point costs twice): 1
Dangerous spots around corners forced on opponent: 1
Stable positions: 12

Rysunek 2: Działanie funkcji heurystycznych dla przykładowej tablicy (przykład obrazowy, niemożliwy do wystąpienia w grze)

4 Implementacja algorytmu Minimax

W celu wybrania najlepszego ruchu każdy z graczy będzie korzystał z algorytmu MiniMax stosując inną kombinację funkcji heurystycznych. Drzewo decyzyjne, które będziemy przeszukiwać, składa się z możliwych od danego momentu sytuacji na planszy. Nie jesteśmy w stanie rozwinąć tego drzewa do końca, więc ustawimy punkt odcięcia, maksymalną głębokość, do której będziemy przeszukiwać. U nas:

- CUTTINGPOINT = 4

Każdy z graczy będzie dążył do minimalizacji maksymalnej możliwej straty, której może doznać, biorąc pod uwagę ruchy przeciwnika. Co istotne, **gracz1 będzie starał się maksymalizować wartość funkcji heurystycznej, a gracz2 minimalizować**. Wynika to z założeń gry o sumie zerowej. Sama implementacja polega na rekurencyjnym przejściu wierzchołków od korzenia, aż do maksymalnej głębokości lub liścia oznaczającego koniec gry. Funkcja MinMax przyjmuje na wejściu nr gracza, stan planszy, głębokość oraz wersję funkcji heurystycznej. W ciele funkcji kolejno:

- generujemy słownik możliwych ruchów dla gracza na danej planszy,
- jeśli ani gracz ani jego przeciwnik nie mają możliwości wykonania ruchu z danej pozycji, to zwracamy informację o wyniku gry tzn.: 640 punktów jeśli gracz1 ma większą liczbę dysków na planszy, -640 w przeciwnym wypadku,
- jeśli doszliśmy do maksymalnej głębokości, to zwracamy wartość funkcji heurystycznej dla danej planszy,
- w przeciwnym wypadku zwracamy maksymalną (jeśli $gracz == gracz1$) lub minimalną (jeśli $gracz == gracz2$) wartość z wywołania algorytmu MiniMax na każdym z dzieci. Co istotne, algorytmowi MiniMax dla dzieci-wierzchołków podajemy w parametrach: numer przeciwnika, planszę dla potencjalnego ruchu, informację o głębokości powiększoną o 1 oraz wersję heurystyki.

4.1 Alpha Beta cięcie

W celu optymalizacji algorytmu MiniMax zaimplementujemy alpha-beta cięcie. Pozwoli nam to wyeliminować gałęzie drzewa, których przeszukiwanie na pewno nie poprawi wartości dla korzenia. Jak się okaże, pozwoli to kilkukrotnie skrócić czas działania algorytmu.

Konstrukcja algorytmu jest analogiczna, z tą różnicą, iż przeszukując węzły dla kolejnych dzieci, odcinamy całe gałęzie, kiedy okaże się, że nie poprawimy tym węzłem wartości dla całego algorytmu. Skorzystamy w tym celu z dwóch parametrów, które będziemy na bieżąco aktualizować:

- Alpha = najlepsza dotychczas znaleziona opcja na drodze z korzenia dla maksymalizera
- Beta = najlepsza dotychczas znaleziona opcja na drodze z korzenia dla minimalizera

Implementacja zgodna jest z poniższym pseudokodem z wykładu:

```

alpha-beta(player,board,alpha,beta)
    if(game over in current board position)
        return winner

    children = all legal moves for player from this board
    if(max's turn)
        for each child
            score = alpha-beta(other player,child,alpha,beta)
            if score > alpha then alpha = score (we have found a better best move)
            if alpha >= beta then return alpha (cut off)
        return alpha (this is our best move)
    else (min's turn)
        for each child
            score = alpha-beta(other player,child,alpha,beta)
            if score < beta then beta = score (opponent has found a better worse move)
            if alpha >= beta then return beta (cut off)
        return beta (this is the opponent's best move)

```

Rysunek 3: Pseudokod algorytmu alpha beta

5 Symulacje gry w Othello

Kolejnym krokiem będzie symulacja gry w Othello. Dwóch graczy będzie kolejno podejmować decyzje, co do wyboru ruchu na podstawie swojej wersji funkcji heurystycznej i algorytmu MiniMax. Gra będzie się toczyć póki nie dojdzie do wygranej jednego z graczy tzn. obaj gracze nie będą mieli możliwości wykonania ruchu i jeden z nich będzie miał większą liczbę dysków na planszy. Symulacje rozpoczynamy od zainicjalizowania planszy początkowej oraz dwóch graczy, z których standardowo czarny (z numerem 1) rozpoczyna. Następnie w pętli, póki chociaż jeden gracz będzie miał możliwość ruchu, będziemy analizować kolejne posunięcia. Jedno posunięcie składa się z ruchu gracza1 i następnie ruchu gracza2. Może się oczywiście zdarzyć, że gracz nie wykona ruchu w danym posunięciu. Funkcja symulująca grę w Othello zwraca na wyjściu liczbę dysków poszczególnych graczy oraz końcową planszę. Może też printować aktualny stan planszy po każdym ruchu.

```

1 p1_wins, p2_wins, times = 0, 0, 0
2 for i in range(5):
3     start = time.time()
4     _, player1_count, player2_count = OthelloSimulationAlphaBeta()
5     end = time.time()
6     times+=end-start
7     if player1_count>player2_count:
8         p1_wins+=1
9     elif player1_count<player2_count:
10        p2_wins+=1
11
12 print(p1_wins,p2_wins)
13 print(times/5)
✓ 35.1s

0 5
7.0185966968536375

1 p1_wins, p2_wins, times = 0, 0, 0
2 for i in range(5):
3     start = time.time()
4     _, player1_count, player2_count = OthelloSimulation()
5     end = time.time()
6     times+=end-start
7     if player1_count>player2_count:
8         p1_wins+=1
9     elif player1_count<player2_count:
10        p2_wins+=1
11
12 print(p1_wins,p2_wins)
13 print(times/5)
✓ 2m 38.0s

0 5
31.577995014198673

```

Rysunek 4: Porównanie czasów działania algorytmów

```

Both players loop number: 0
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 1 1 0 0 0]
 [0 0 0 2 1 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]]
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 1 1 0 0 0]
 [0 0 0 2 2 2 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]]
Both players loop number: 1
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 1 1 0 0 0]
 [0 0 0 1 2 2 0 0]
 [0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0]]
...
[2 2 2 2 2 2 2 2]]
Both players loop number: 32
Player 2 wins!

```

(a) Początkowe ruchy i info o wygranej gracza

```

Both players loop number: 30
[[2 2 2 2 2 2 2 2]
 [1 1 1 2 2 2 2 2]
 [2 1 2 2 2 2 2 2]
 [2 1 1 2 2 2 2 2]
 [2 1 2 2 2 2 2 2]
 [2 1 2 2 2 2 2 2]
 [2 1 2 2 2 2 2 0]
 [2 2 1 1 2 2 2 1]
 [2 2 2 2 2 2 2 0]]
Both players loop number: 31
[[2 2 2 2 2 2 2 2]
 [1 1 1 2 2 2 2 2]
 [2 1 2 2 2 2 2 2]
 [2 1 1 2 2 2 2 2]
 [2 1 2 2 2 2 2 2]
 [2 1 1 1 1 1 1 1]
 [2 2 1 1 2 2 2 1]
 [2 2 2 2 2 2 2 0]]
[[2 2 2 2 2 2 2 2]
 [1 1 1 2 2 2 2 2]
 [2 1 2 2 2 2 2 2]
 [2 1 1 2 2 2 2 2]
 [2 1 2 2 2 2 2 2]
 [2 1 1 1 1 2 1 2]
 [2 2 1 1 2 2 2 2]
 [2 2 2 2 2 2 2 2]]

```

(b) Końcowe ruchy

Rysunek 5: Symulacja gry w Reversi

6 Podsumowanie

Przy rozwiązywaniu listy korzystałam jedynie z wykładu oraz załączonej do listy literatury, głównie stron z zasadami oraz strategiami gry w Reversi. Kod jest w całości mój. Biblioteki, które wykorzystałam to jedynie **numpy** do operacji na macierzy oraz **copy** i **random**. Najtrudniejszą implementacyjnie była strategia zliczająca stabilne pozycje na planszy, ponieważ zwinięcie jej w kompaktowy sposób okazało się wyzwaniem.