tictactoe

December 3, 2022

1 Podejmowanie decyzji

1.1 Wstęp

Celem tego laboratorium jest zapoznanie się z kilkoma metodami wykorzystywanymi do podejmowania decyzji w kontekście, w którym dwa podmioty mają sprzeczne cele, a konkretnie w sytuacji, w której wygrana jednego podmiotu oznacza przegraną drugiego podmiotu. Wykorzystamy do tego grę w kółko i krzyżyk, która posiada bardzo proste zasady. Zaprezentowane metody mają jednak znacznie szersze zastosowanie i mogą być wykorzystywane w szerokim spektrum problemów decyzyjnych.

Zastosowane metody to: * losowe przeszukiwanie przestrzeni decyzji, * heurystyczne przeszukiwanie przestrzeni decyzji, * algorytm minimax, * algorytm alpha-beta, * przeszukiwanie w oparciu o metodę Monte Carlo.

Na końcu laboratorium zrobimy turniej graczy zaimplementowanych z użyciem tych metod i zobaczymy, która metoda radzi sobie najlepiej w grze w kółko i krzyżyk.

2 Gra w kółko i krzyżyk

Zaczniemy od implementacji gry w kółko i krzyżyk. Możemy podejść do tego na kilka różnych sposobów. Jednym z najprostszych jest gracz wykonujący losowy ruch i liczący na szczęście. Trochę bardziej skomplikowanym graczem będzie gracz wykorzystujący jakąś heurystykę (można taką znaleźć na Wikipedii), nas będą bardziej interesowali gracze oparci o przeszukiwanie drzewa rozgrywki np. algorytmem minimax czy A*, a w bardziej skomplikowanych problemach znajdują zastosowanie również algorytmy probabilistyczne typu Monte Carlo, a w prostych - klasyczne grafowe przeszukiwanie.

Na początek zaiplemętujemy samą planszę i metody niezbędne do rozgrywki takie jak sprawdzenie, czy ktoś wygrał, wyświetlające stan planszy itd. Poniżej znajduje się implementacja gry, która wyświetla stan gry w fomie tekstowej oraz informuje, kto wygrał grę.

[ ]:
```
PLAYER_1 = -1
PLAYER_2 = 1


class TicTacToe:
    def __init__(self, player1, player2):
        self.board = [
```

```python
            [0, 0, 0],
            [0, 0, 0],
            [0, 0, 0],
        ]
        self.player1 = player1
        self.player2 = player2
        self.chars = {0: " ", PLAYER_1: "O", PLAYER_2: "X"}

    def play(self, verbose=True):
        self.print_board(verbose)
        for i in range(4):
            self.player1.move(self.board, PLAYER_1)
            self.print_board(verbose)
            if check_for_end(self.board, PLAYER_1):
                if verbose:
                    print("player 1 wins")
                return "win"
            self.player2.move(self.board, PLAYER_2)
            self.print_board(verbose)
            if check_for_end(self.board, PLAYER_2):
                if verbose:
                    print("player 2 wins")
                return "loss"
        self.player1.move(self.board, PLAYER_1)
        self.print_board(verbose)
        if check_for_end(self.board, PLAYER_1):
            if verbose:
                print("player 1 wins")
            return "win"
        else:
            if verbose:
                print("draw")
            return "draw"

    def print_board(self, verbose):
        if not verbose:
            return
        str_line = "---------------"
        print("\n" + str_line)
        for row in self.board:
            for cell in row:
                symbol = self.chars[cell]
                print(f"| {symbol} |", end="")
            print("\n" + str_line)


def check_for_end(board, player):
```

```python
    return (
        check_rows(board, player)
        or check_cols(board, player)
        or check_diagonals(board, player)
    )

def check_rows(board, player):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] == player:
            return True
    return False


def check_cols(board, player):
    for i in range(3):
        if board[0][i] == board[1][i] == board[2][i] == player:
            return True
    return False


def check_diagonals(board, player):
    return (board[0][0] == board[1][1] == board[2][2] == player) or (
        board[0][2] == board[1][1] == board[2][0] == player
    )

def check_if_board_empty(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] != 0:
                return False
    return True

def check_if_board_full(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return False
    return True
```

2.1 Interfejs gracza

Skoro mamy już zdefiniowaną grę to teraz potrzebny jest nam gracz. Poniżej zdefiniowany jest interface PlayerInreface. Należy go wykorzystywać implementując swoich graczy. Interfejs ma metodę `move`, która reprezentuje pojedynczy ruch gracza a także metodę `name`, która zwraca nazwę zaimplementowanego algorytmu.

[ ]:
```python
from abc import ABC, abstractmethod

class PlayerInterface(ABC):
    @abstractmethod
    def move(self, board, player):
        pass

    @property
    def name(self):
        return type(self).__name__
```

Zobaczmy, jak wyglądałby prawdopodobnie najprostszy gracz do zaimplementowania, a więc gracz losowy. Wybiera on losowo wiersz i kolumnę, a następnie sprawdza, czy pole jest puste. Jeżeli tak, to stawia tam swój znak.

[ ]:
```python
import random

class RandomPlayer(PlayerInterface):
    def move(self, board, player):
        while True:
            row = random.randint(0, 2)
            col = random.randint(0, 2)
            if self.__can_make_move(board, row, col):
                board[row][col] = player
                return

    def __can_make_move(self, board, row, col):
        return board[row][col] == 0
```

Sprawdźmy, czy wszystko działa.

[ ]:
```python
game = TicTacToe(RandomPlayer(), RandomPlayer())
game.play()
```

```
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------

---------------
|   ||   || O |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
```

```
---------------

---------------
|   ||   || O |
---------------
|   || X ||   |
---------------
|   ||   ||   |
---------------

---------------
|   || O || O |
---------------
|   || X ||   |
---------------
|   ||   ||   |
---------------

---------------
|   || O || O |
---------------
|   || X ||   |
---------------
| X ||   ||   |
---------------

---------------
|   || O || O |
---------------
|   || X || O |
---------------
| X ||   ||   |
---------------

---------------
|   || O || O |
---------------
|   || X || O |
---------------
| X ||   || X |
---------------

---------------
|   || O || O |
---------------
|   || X || O |
---------------
| X || O || X |
```

```
---------------

---------------
| X || O || O |
---------------
|   || X || O |
---------------
| X || O || X |
---------------
player 2 wins
```

[ ]: 'loss'

Napiszemy teraz kilku kolejnych graczy, podejmujących decyzje nieco inteligentniej, a na koniec zrobimy mały turniej i sprawdzimy, który jest najlepszy.

3 Podejmowanie decyzji oparte o heurystykę

W każdym nowym problemie warto sprawdzić proste podejście heurystyczne, wykorzystujące elementarną wiedzę o problemie. Sama gra w kółko i krzyżyk, jak wiadomo, nie jest zbyt skomplikowana. Jest w niej najwyżej 9 możliwych ruchów, mniej niż $3^9 - 1$ możliwych końcowych stanów planszy, a naiwnie licząc, grę można rozegrać na 9! sposobów. Dzięki tak mocnemu uproszczeniu istnieje tu strategia optymalna, gwarantująca w najgorszym wypadku remis:

Dla tej gry istnieje optymalna strategia tzn. w najgorszym przypadku zremisujemy.

1. Win: If the player has two in a row, they can place a third to get three in a row.
2. Block: If the opponent has two in a row, the player must play the third themselves to block the opponent.
3. Fork: Cause a scenario where the player has two ways to win (two non-blocked lines of 2).
4. Blocking an opponent's fork: If there is only one possible fork for the opponent, the player should block it. Otherwise, the player should block all forks in any way that simultaneously allows them to make two in a row. Otherwise, the player should make a two in a row to force the opponent into defending, as long as it does not result in them producing a fork. For example, if "X" has two opposite corners and "O" has the center, "O" must not play a corner move to win. (Playing a corner move in this scenario produces a fork for "X" to win.)
5. Center: A player marks the center. (If it is the first move of the game, playing a corner move gives the second player more opportunities to make a mistake and may therefore be the better choice; however, it makes no difference between perfect players.)
6. Opposite corner: If the opponent is in the corner, the player plays the opposite corner.
7. Empty corner: The player plays in a corner square.
8. Empty side: The player plays in a middle square on any of the four sides.

Implementacja takiego bota byłaby jednak niezbyt ciekawa, a na dodatek specyficzna tylko dla tego konkretnego problemu. Dlatego my zajmiemy się eksploracją drzewa gry, a więc możliwych stanów oraz przejść między nimi. Na dobry początek ulepszymy naszego losowego bota trywialną heurystyką - jeżeli to możliwe, ma wybrać ruch wygrywający, a jeżeli nie, to losowy.

Można by też zadać pytanie, czemu nie wykorzystamy tutaj ciekawszego problemu, jak np. szachy. Odpowiedź jest prosta - nie mamy na to czasu, a konkretnie wieczności. Shannon obliczył dolną granicę złożoności drzewa gry na 10^{120}.

3.1 Wygraj, jeśli to możliwe w kolejnym kroku

Zadanie 1 (1 punkt)

Zaimplementuj ulepszonego losowego bota tak, aby wybierał ruch wygrywający, jeżeli to możliwe, a jeżeli nie, to losowy.

[ ]:
```
class RandomPlayerWinIfCan(PlayerInterface):
    def move(self, board, player):
        possible_moves = []

        for row in range(3):
            for col in range(3):
                if not self._can_make_move(board, row, col):
                    continue
                if self._move_if_can_win(board, player, row, col):
                    return
                possible_moves.append((row, col))

        self._random_move(board, player, possible_moves)

    def _can_make_move(self, board, row, col):
        return board[row][col] == 0

    def _move_if_can_win(self, board, player, row, col):
        board[row][col] = player
        if check_rows(board, player) or check_cols(board, player) or␣
 ↪check_diagonals(board, player):
            return True
        board[row][col] = 0
        return False

    def _random_move(self, board, player, possible_moves):
        row, col = random.choice(possible_moves)
        board[row][col] = player
```

[ ]:
```
game = TicTacToe(RandomPlayerWinIfCan(), RandomPlayer())
game.play()
```

```
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
```

```
---------------
|   ||   ||   |
---------------
|   ||   || O |
---------------
|   ||   ||   |
---------------

---------------
|   ||   || X |
---------------
|   ||   || O |
---------------
|   ||   ||   |
---------------

---------------
|   ||   || X |
---------------
|   ||   || O |
---------------
|   ||   || O |
---------------

---------------
|   ||   || X |
---------------
|   ||   || O |
---------------
|   || X || O |
---------------

---------------
|   ||   || X |
---------------
|   || O || O |
---------------
|   || X || O |
---------------

---------------
| X ||   || X |
---------------
|   || O || O |
---------------
|   || X || O |
---------------
```

```
---------------
| X ||   || X |
---------------
| O || O || O |
---------------
|   || X || O |
---------------
player 1 wins
```

[ ]: 'win'

3.2 Blokuj kolejny krok wygrywający przeciwnika

Skoro w poprzednim zadaniu wygrywamy, kiedy możemy to zrobić w jednym kroku, to spróbujmy ulepszyć naszą strategię wcześniej. Możemy to zrobić, minimalizując swoje straty, czyli sprawdzamy dodatkowo, czy przeciwnik może skończyć grę. Jeżeli tak, to go blokujemy.

Zadanie 2 (1 punkt)

Zaimplementuj ulepszenie bota, w którym dodatkowo jeżeli nie możemy wygrać w danym ruchu, a przeciwnik tak, to go blokujemy. A jeżeli ani my, ani przeciwnik nie może wygrać w kolejnym ruchu, to wykonujemy losowe posunięcie.

[ ]:
```python
class Blocking(RandomPlayerWinIfCan):
    def move(self, board, player):
        super()
        possible_moves = []
        blocking_moves = []
        other_player = -player

        for row in range(3):
            for col in range(3):
                if not self._can_make_move(board, row, col):
                    continue
                if self._move_if_can_win(board, player, row, col):
                    return
                if self._blocks_opponent(board, other_player, row, col):
                    blocking_moves.append((row, col))
                possible_moves.append((row, col))

        if blocking_moves:
            self._random_move(board, player, blocking_moves)
        else:
            self._random_move(board, player, possible_moves)

    def _blocks_opponent(self, board, other_player, row, col):
        if self._move_if_can_win(board, other_player, row, col):
            board[row][col] = 0
```

```
            return True
        return False
```

[ ]:
```
game = TicTacToe(RandomPlayerWinIfCan(), Blocking())
game.play()
```

```
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------

---------------
|   ||   ||   |
---------------
| O ||   ||   |
---------------
|   ||   ||   |
---------------

---------------
| X ||   ||   |
---------------
| O ||   ||   |
---------------
|   ||   ||   |
---------------

---------------
| X ||   ||   |
---------------
| O ||   ||   |
---------------
| O ||   ||   |
---------------

---------------
| X ||   ||   |
---------------
| O || X ||   |
---------------
| O ||   ||   |
---------------

---------------
```

```
| X ||   ||   |
---------------
| O || X ||   |
---------------
| O || O ||   |
---------------

---------------
| X ||   ||   |
---------------
| O || X ||   |
---------------
| O || O || X |
---------------
player 2 wins
```

[ ]: 'loss'

4 Algorytm minimax

Zaiplemetujemy teraz algorytm minimax. Minimalizuje on nasze maksymalne straty lub maksymalizuje minimalne zyski (maksmin). Poprzednie 2 kroki nas do tego zbliżały. Algorytm wywodzi się z teorii gier o sumie zerowej (gdzie wygrana ma wartość 1, przegrana -1, a remis 0 - w takich grach wygrana jednego gracza, oznacza, że drugi gracz przegrał, czyli nie ma strategii, która prowadziłaby do sytuacji *win-win*).

Twierdzenie o minimaksie (minimax theorem) mówi, że dla każdej dwuosobowej gry o sumie zerowej istnieje wartość V i mieszana strategia dla każdego gracza, takie, że:

- biorąc pod uwagę strategię gracza drugiego, najlepszą możliwą spłatą dla gracza pierwszego jest V,
- biorąc pod uwagę strategię gracza pierwszego, najlepszą możliwą spłatą dla gracza drugiego jest $-V$.

Każdy gracz minimalizuje maksymalną możliwą spłatę dla swojego przeciwnika – ponieważ gra jest grą o sumie zerowej, wynikiem tego algorytmu jest również maksymalizowanie swojej minimalnej spłaty.

Powyższa ilustracja przedstawia fragment analizy końcowej partii gry w kółko i krzyżyk. **Max** oznacza turę, w której gracz wybiera ten spośród dostępnych ruchów, który da maksymalną spłatę, natomiast **min** oznacza turę, w której gracz wybiera ruch, który da minimalną spłatę. Inaczej - *max* to ruch, z punktu wiedzenia gracza, dla którego chcemy, żeby wygrał, a *min* ruch gracza, który chcemy żeby przegrał.

Minimax jest algorytem rekurencyjnym, w którym liście drzewa możliwych ruchów oznaczają zakończenie gry, z przypisaną do nich wartością z punktu wiedzenia gracza, którego ruch jest w korzeniu drzewa możliwych ruchów. We wcześniejszych węzłach - w zależności czy jest tura gracza max, czy min, będą wybierane te gałęzie, które maksymalizują, bądź minimalizują wartość spłaty.

Przykładowo - w pierwszym wierszu mamy trzy strzałki i gracza *max*, dlatego gracz ten wybierze pierwszy możliwy ruch, bo daje on maksymalną spłatę. W drugim wierwszu w środkowej kolumnie mamy dwie strzałki. Gracz *min* wybierze zatem strzałkę (ruch) prawy, ponieważ minimalizuje on spłatę (0). Analogicznie w drugim wierszu po prawej stronie wybierana jest prawa strzałka, ponieważ daje ona wartość mimalną (-1).

Poniższy diagram zawiera to samo drzewo analizy ruchów, gdzie pozostawiono wyłącznie wartości spłaty dla poszczególnych węzłów.

Algorytm *minimax* jest następujący: 1. zainicalizuj wartość na $-\infty$ dla gracza którego spłata jest maksymalizowana i na $+\infty$ dla gracza którego spłata jest minimalizowana, 2. sprawdź czy gra się nie skończyła, jeżeli tak to ewaluuj stan gry z punktu widzenia gracza maksymalizującego i zwróć wynik, 3. dla każdego możliwego ruchu każdego z graczy wywołaj rekurencyjnie *minimax*: 1. przy maksymalizacji wyniku, zwiększ wynik, jeśli otrzymany wynik jest większy od dotychczas największego wyniku, 2. przy minimalizacji wyniku, pomniejsz wynik, jeśli otrzymany wynik jest mniejszy od dotychczas najmniejszego wyniku, 3. zwróć najlepszy wynik.

Algorytm ten wymaga dodatkowo funkcji ewaluującej, która oceni stan gry na końcu. Uznajemy, że zwycięstwo to +1, przegrana -1, a remis 0.

Zadanie 3 (3 punkty)

Zaimplementuj gracza realizującego algorytm minimaks.

[ ]:
```
from math import inf


class MinimaxPlayer(Blocking):
    def move(self, board, player):
        if check_if_board_empty(board):
            board[1][1] = player
            return

        row, col = self._get_best_move(board, player)
        board[row][col] = player

    def _get_best_move(self, board, player):
        return self.__minimax(board, player, 1)[0]

    def _check_end_payoff(self, board, player, payoff):
        if check_for_end(board, player): return payoff
        if check_for_end(board, -player): return -payoff
        if check_if_board_full(board): return 0
        return None
```

```python
    def __minimax(self, board, player, payoff):
        end_payoff = self._check_end_payoff(board, player, payoff)
        if end_payoff is not None: return None, end_payoff

        best_payoff = -inf * payoff
        best_move = None

        for row in range(3):
            for col in range(3):
                if not self._can_make_move(board, row, col):
                    continue

                board[row][col] = player
                _, value = self.__minimax(board, -player, -payoff)

                if value * payoff > best_payoff * payoff:
                    best_payoff = value
                    best_move = row, col
                board[row][col] = 0

        return best_move, best_payoff
```

[ ]:

```python
%%time
game = TicTacToe(MinimaxPlayer(), Blocking())
game.play()
```

```
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------

---------------
|   ||   ||   |
---------------
|   || O ||   |
---------------
|   ||   ||   |
---------------

---------------
|   ||   ||   |
---------------
|   || O ||   |
---------------
```

```
|   ||   || X |
---------------

---------------
| O ||   ||   |
---------------
|   || O ||   |
---------------
|   ||   || X |
---------------

---------------
| O || X ||   |
---------------
|   || O ||   |
---------------
|   ||   || X |
---------------

---------------
| O || X ||   |
---------------
| O || O ||   |
---------------
|   ||   || X |
---------------

---------------
| O || X ||   |
---------------
| O || O || X |
---------------
|   ||   || X |
---------------

---------------
| O || X ||   |
---------------
| O || O || X |
---------------
| O ||   || X |
---------------
player 1 wins
CPU times: user 24.7 ms, sys: 1.8 ms, total: 26.5 ms
Wall time: 27 ms
```

[ ]: 'win'

Wróć na chwilę do implementacji minimaxu i zastanów się, co się dzieje, jeżeli ten algorytm wykonuje ruch na pustej planszy i jak to wpływa na jego czas działania. Może coś można poprawić?

5 Alpha-beta pruning

Widzimy, że nasz poprzedni gracz jest właściwie idealny, bo w końcu sprawdza całe drzewo gry. Ma tylko w związku z tym wadę - dla większości problemów wykonuje się wieczność. Dlatego też zastosujemy ważne ulepszenie algorytmu minimax, nazywane alfa-beta pruning.

W przypadku klasycznego minimaxu ewaluujemy każdą możliwą ścieżkę gry. Alfa-beta pruning, jak i wiele innych metod, opiera się na "przycinaniu" drzewa, czyli nie ewaluujemy tych odnóg drzewa, co do których wiemy, że nie da ona lepszego wyniku niż najlepszy obecny. W niektórych wypadkach, np. w dobrej implementacji dla szachów, potrafi zredukować liczbę rozważanych ścieżek nawet o 99.8%.

Do poprzedniego algorytmu dodajemy 2 zmienne, α i β:

- α przechowuje najlepszą wartość dla gracza maksymalizującego swój wynik,
- β przechowuje najlepszą wartość dla gracza minimalizującego swój wynik.

Dzięki tej informacji możemy przerwać sprawdzanie danej gałęzi, kiedy α jest większa od β. Oznacza to bowiem sytuację, w której najlepszy wynik gracza maksymalizującego jest większy niż najlepszy wynik gracza minimalizującego.

Pseudokod:

```
function alphabeta(node, depth,  ,  , maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1,  ,  , FALSE))
              := max( , value)
            if      then
                break (*   cutoff *)

        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1,  ,  , TRUE))
              := min( , value)
            if      then
                break (*   cutoff *)
        return value
```

Zadanie 4 (2 punkty)

Zaimplementuj gracza realizującego algorytm alfa-beta pruning.

```
[ ]: class AlphaBetaPlayer(MinimaxPlayer):
         def _get_best_move(self, board, player):
             return self.__alpha_beta(board, player, 1, -inf, inf)[0]

         def __alpha_beta(self, board, player, payoff,  ,  ):
             end_payoff = self._check_end_payoff(board, player, payoff)
             if end_payoff is not None: return None, end_payoff

             best_payoff = -inf * payoff
             best_move = None

             for row in range(3):
                 for col in range(3):
                     if not self._can_make_move(board, row, col):
                         continue

                     board[row][col] = player
                     _, value = self.__alpha_beta(board, -player, -payoff,  ,  )
                     board[row][col] = 0

                     if value * payoff > best_payoff * payoff:
                         best_payoff = value
                         best_move = row, col

                         if payoff > 0:
                              = max( , value)
                         else:
                              = min( , value)

                         if  >=  :
                             return best_move, best_payoff

             return best_move, best_payoff
```

```
[ ]: %%time
     game = TicTacToe(AlphaBetaPlayer(), MinimaxPlayer())
     game.play()
```

```
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
```

```
---------------
|   ||   ||   |
---------------
|   || O ||   |
---------------
|   ||   ||   |
---------------

---------------
| X ||   ||   |
---------------
|   || O ||   |
---------------
|   ||   ||   |
---------------

---------------
| X || O ||   |
---------------
|   || O ||   |
---------------
|   ||   ||   |
---------------

---------------
| X || O ||   |
---------------
|   || O ||   |
---------------
|   || X ||   |
---------------

---------------
| X || O ||   |
---------------
| O || O ||   |
---------------
|   || X ||   |
---------------

---------------
| X || O ||   |
---------------
| O || O || X |
---------------
|   || X ||   |
---------------
```

```
---------------
| X || O || O |
---------------
| O || O || X |
---------------
|   || X ||   |
---------------

---------------
| X || O || O |
---------------
| O || O || X |
---------------
| X || X ||   |
---------------

---------------
| X || O || O |
---------------
| O || O || X |
---------------
| X || X || O |
---------------
draw
CPU times: user 197 ms, sys: 8.32 ms, total: 205 ms
Wall time: 287 ms
```

[ ]: 'draw'

5.1 Monte Carlo Tree Search (MCTS)

Metody Monte Carlo polegają na wprowadzeniu losowości i przybliżaniu za jej pomocą rozwiązań dla trudnych problemów. Dla gry w kółko i krzyżyk nie jest to co prawda niezbędne, ale dla bardziej skomplikowanych gier, jak szachy czy go, już zdecydowanie tak.

Ogólna metoda MCTS składa się z 4 etapów: * selekcji - wybieramy najlepsze dziecko, aż dotrzemy do liścia, * ekspansji - jeżeli nie możemy dokonać selekcji, rozwijamy drzewo we wszystkich możliwych kierunkach z węzła, * symulacji - po ekspansji wybieramy węzeł do przeprowadzenia symulacji gry aż do końca, * wstecznej propagacji - kiedy dotrzemy do końca, ewaluujemy wynik gry i propagujemy go w górę drzewa.

W naszym wypadku wystarczy nieco prostszy algorytm Pure Monte Carlo Tree Search (Pure MCTS), w którym realizujemy tylko symulację i wsteczną propagację. Dla każdego możliwego ruchu w danej rundzie przeprowadzamy N symulacji oraz obliczamy prawdopodobieństwo zwycięstwa/remisu/przegranej dla każdego z możliwych ruchów, a następnie wybieramy najlepszy ruch.

Zadanie 5 (2 punkty)

Zaimplementuj gracza realizującego algorytm Pure MCTS.

[ ]:
```
from copy import deepcopy

class MonteCarloPlayer(MinimaxPlayer):
    def __init__(self, simulation_count = 100):
        super()
        self.__simulation_count = simulation_count

    def move(self, board, player):
        best_result = -inf
        best_move = None

        empty_fields = [
            (row, col) for row in range(3) for col in range(3)
            if self._can_make_move(board, row, col)
        ]

        for row, col in empty_fields:
            result = 0

            for _ in range(self.__simulation_count):
                # Add 1 if player won
                # Remove 1 if player lost
                # Add 0 if a game ended with a draw
                result += self.__get_simulation_result(
                    deepcopy(board), player, row, col, empty_fields[:]
                )

            if result > best_result:
                best_result = result
                best_move = row, col

        row, col = best_move
        board[row][col] = player

    def __get_simulation_result(self, board, player, r, c, empty_fields):
        empty_fields.remove((r, c))
        board[r][c] = player
        current_player = -player

        while empty_fields:
            if check_for_end(board, player): return 1
            if check_for_end(board, -player): return -1

            field = random.choice(empty_fields)
            empty_fields.remove(field)
```

```
            row, col = field
            board[row][col] = current_player
            current_player *= -1

        return 0
```

[ ]:
```
%%time
game = TicTacToe(MonteCarloPlayer(), MinimaxPlayer())
game.play()
```

```
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------

---------------
|   ||   ||   |
---------------
|   || O ||   |
---------------
|   ||   ||   |
---------------

---------------
| X ||   ||   |
---------------
|   || O ||   |
---------------
|   ||   ||   |
---------------

---------------
| X ||   || O |
---------------
|   || O ||   |
---------------
|   ||   ||   |
---------------

---------------
| X ||   || O |
---------------
|   || O ||   |
---------------
```

```
| X ||   ||   |
---------------

---------------
| X ||   || O |
---------------
| O || O ||   |
---------------
| X ||   ||   |
---------------

---------------
| X ||   || O |
---------------
| O || O || X |
---------------
| X ||   ||   |
---------------

---------------
| X || O || O |
---------------
| O || O || X |
---------------
| X ||   ||   |
---------------

---------------
| X || O || O |
---------------
| O || O || X |
---------------
| X || X ||   |
---------------

---------------
| X || O || O |
---------------
| O || O || X |
---------------
| X || X || O |
---------------
draw
CPU times: user 227 ms, sys: 7.39 ms, total: 234 ms
Wall time: 326 ms
```

[ ]:
```
'draw'
```

6 Turniej

Teraz przeprowadzimy turniej w celu porównania zaimplementowanych metod. Każdy algorytm będzie grał z każdym po 10 razy.

[ ]:
```
from collections import defaultdict
from IPython.display import display
import pandas as pd


def print_scores(scores, names):
    win = {}
    for name in names:
        win[name] = [scores["win"][name][n] for n in names]
    loss = {}
    for name in names:
        loss[name] = [scores["loss"][name][n] for n in names]
    draw = {}
    for name in names:
        draw[name] = [scores["draw"][name][n] for n in names]

    df = pd.DataFrame.from_dict(win, orient="index", columns=names)
    display(df)
    df2 = pd.DataFrame.from_dict(loss, orient="index", columns=names)
    display(df2)
    df3 = pd.DataFrame.from_dict(draw, orient="index", columns=names)
    display(df3)
```

[ ]:
```
number_of_rounds = 10
players = [
    RandomPlayer(),
    Blocking(),
    RandomPlayerWinIfCan(),
    MinimaxPlayer(),
    AlphaBetaPlayer(),
    MonteCarloPlayer(),
]
scores = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))

for player in players:
    for adversary in players:
        for i in range(number_of_rounds):
            game = TicTacToe(player, adversary)
            score = game.play(False)
            # print("player name {} adversary name {} score {} ".format(player.
↪name, adversary.name, score))
            scores[score][player.name][adversary.name] += 1
```

```
print_scores(scores, [player.name for player in players])
```

```
                    RandomPlayer  Blocking  RandomPlayerWinIfCan  \
RandomPlayer                   6         0                     4
Blocking                      10         4                     9
RandomPlayerWinIfCan           8         0                     7
MinimaxPlayer                 10         3                     9
AlphaBetaPlayer               10         4                    10
MonteCarloPlayer              10         5                    10

                    MinimaxPlayer  AlphaBetaPlayer  MonteCarloPlayer
RandomPlayer                    0                0                 1
Blocking                        0                0                 4
RandomPlayerWinIfCan            0                0                 1
MinimaxPlayer                   0                0                 7
AlphaBetaPlayer                 0                0                 4
MonteCarloPlayer                0                0                 6
                    RandomPlayer  Blocking  RandomPlayerWinIfCan  \
RandomPlayer                   3         7                     5
Blocking                       0         2                     0
RandomPlayerWinIfCan           1         7                     1
MinimaxPlayer                  0         0                     0
AlphaBetaPlayer                0         0                     0
MonteCarloPlayer               0         0                     0

                    MinimaxPlayer  AlphaBetaPlayer  MonteCarloPlayer
RandomPlayer                   10                6                 8
Blocking                        2                4                 1
RandomPlayerWinIfCan            8               10                 9
MinimaxPlayer                   0                0                 0
AlphaBetaPlayer                 0                0                 0
MonteCarloPlayer                1                0                 1
                    RandomPlayer  Blocking  RandomPlayerWinIfCan  \
RandomPlayer                   1         3                     1
Blocking                       0         4                     1
RandomPlayerWinIfCan           1         3                     2
MinimaxPlayer                  0         7                     1
AlphaBetaPlayer                0         6                     0
MonteCarloPlayer               0         5                     0

                    MinimaxPlayer  AlphaBetaPlayer  MonteCarloPlayer
RandomPlayer                    0                4                 1
Blocking                        8                6                 5
RandomPlayerWinIfCan            2                0                 0
MinimaxPlayer                  10               10                 3
AlphaBetaPlayer                10               10                 6
```

`MonteCarloPlayer`	9	10	3

6.1 Pytania kontrolne (1 punkt)

1. Jaki algorytm z omawianych mógłby się sprawdzić w grze go i dlaczego?
2. Czy minimax jest optymalny dla gry w szachy i dlaczego?
3. Jakie są możliwe zastosowania podanych algorytmów poza grami?
4. Jaka jest przewaga obliczeniowa MCTS nad pozostałymi metodami?

6.1.1 Odpowiedzi

1. Ze względu na bardzo duże drzewo gry, zastosowanie algorytmu minimax oraz alpha-beta pruning nie jest rozsądne, ponieważ wykonanie algorytmu trwałoby wieczność. Zamiast tych algorytmów, można wykorzystać MCTS (Monte Carlo Tree Search), ponieważ ten algorytm nie sprawdza całego drzewa gry, a jedynie wykonuje kilkukrotnie symulację możliwego zakończenia gry dla danego ruchu, ograniczając tym samym znacząco wymaganą do wykonania liczbę operacji, przez co możliwe jest wyznaczenie dobrego ruchu w rozsądnym czasie.

2. Nie, ponieważ algorytm minimax wymaga sprawdzenia całego drzewa gry, które, w przypadku szachów, jest niemożliwe do przejrzenia w rozsądnym czasie, z wykorzystaniem obecnie dostępnych komputerów. Shannon obliczył dolną granicę złożoności drzewa gry na 10^{120}, a tak dużego drzewa nie da się przejrzeć w krótkim czasie.

3. Omówione algorytmy przydają się wszędzie tam, gdzie chcemy zmaksymalizować pewien wynik, jednocześnie minimalizując inny wynik. Wykorzystuje się je np. podczas projektowania systemów bezpieczeństwa oraz w systemach kolejkowania.

4. MCTS nie przeszukuje drzewa gry, a jedynie symuluje możliwe stany gry, z wykorzystaniem losowo grających przeciwników. Mimo, że to podejście nie daje nam gwarancji dokonania optymalnego wyboru, ponieważ decyzja jest podejmowana w oparciu o prawdopodobieństwo zwycięstwa, wykonanie algorytmu zwykle trwa znacznie krócej niż sprawdzenie drzewa gry przez algorytm minimax, czy algorytm alpha-beta pruning.

6.2 Zadanie dodatkowe

Rozwiń kod algorytmu MCTS tak, aby działał z ogólnymi zasadami, a nie w formie uproszczonego Pure MCTS.

Zastosuj prosty, ale bardzo skuteczny sposób selekcji UCT (*Upper Confidence Bound 1 applied to trees*), będący wariantem bardzo skutecznych metod UCB, stosowanych m.in. w podejmowaniu decyzji, uczeniu ze wzmocnieniem i systemach rekomendacyjnych. Polega na wyborze tego węzła, dla którego następujące wyrażenie ma maksymalną wartość:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

gdzie: * w_i to liczba zwycięstw dla węzła po rozważeniu i-tego ruchu * n_i to łączna liczba symulacji przeprowadzonych dla węzła po i-tym ruchu, * N_i oznacza całkowitą liczbę symulacji przeprowadzoną po i-tym ruchu dla węzła-rodzica aktualnie rozważanego węzła, * c to hiperparametr, wedle teorii powinien mieć wartość $\sqrt{2}$.

O uzasadnieniu tego wzoru możesz więcej przeczytać tutaj.

Jeżeli chcesz, możesz zastosować ten algorytm dla bardziej skomplikowanej gry, jak np. warcaby czy szachy.