Raport 1 - Hexapawn - Piotr Gąsiorek, Szymon Frączek

In [ ]:
```
import easyAI
```

Na samym początku skupiamy się na modyfikacji istniejącego kodu z pakietu easyAI. Chcemy dodać wersję probabilistyczną gry Hexapawn która umożliwi spawnowanie piona po zbiciu z określonym prawdopodobieństwem.

W konstruktorze klasy Hexapawn dodano nowe pola: starting_positions i captured_pawns. Pole starting_positions przechowuje początkowe pozycje pionów gracza, a captured_pawns przechowuje pozycje pionów, które zostały zbite.

Metoda make_move została zmodyfikowana, aby obsługiwać nową funkcjonalność. Jeśli pion jest zbijany, jest dodawany do listy captured_pawns i usuwany z listy starting_positions. Jeśli prawdopodobieństwo jest większe od 0, istnieje szansa, że zbita pion zostanie "odrodzony" i dodany z powrotem do listy pawns i starting_positions.

In [ ]:
```
from easyAI import TwoPlayerGame
import copy
import random
import config
import time
import pandas as pd

# Convert D7 to (3,6) and back...
to_string = lambda move: " ".join(
    ["ABCDEFGHIJ"[move[i][0]] + str(move[i][1] + 1) for i in (0, 1)]
)
to_tuple = lambda s: ("ABCDEFGHIJ".index(s[0]), int(s[1:]) - 1)


class Hexapawn(TwoPlayerGame):
    """
    A nice game whose rules are explained here:
    http://fr.wikipedia.org/wiki/Hexapawn
    """

    def __init__(self, players, size=(4, 4), probability=0):
        self.size = M, N = size
        self.probability = probability
        p = [[(i, j) for j in range(N)] for i in [0, M - 1]]

        for i, d, goal, pawns in [(0, 1, M - 1, p[0]), (1, -1, 0, p[1])]:
            players[i].direction = d
```

```
            players[i].goal_line = goal
            players[i].pawns = pawns
            players[i].starting_positions = copy.deepcopy(pawns)
            players[i].captured_pawns = []

        self.players = players
        self.current_player = 1

    def possible_moves(self):
        moves = []
        opponent_pawns = self.opponent.pawns
        d = self.player.direction
        for i, j in self.player.pawns:
            if (i + d, j) not in opponent_pawns:
                moves.append(((i, j), (i + d, j)))
            if (i + d, j + 1) in opponent_pawns:
                moves.append(((i, j), (i + d, j + 1)))
            if (i + d, j - 1) in opponent_pawns:
                moves.append(((i, j), (i + d, j - 1)))

        return list(map(to_string, [(i, j) for i, j in moves]))

    def make_move(self, move):
        move = list(map(to_tuple, move.split(" ")))

        pawn_index = self.player.pawns.index(move[0])
        self.player.pawns[pawn_index] = move[1]

        if move[1] in self.opponent.pawns:
            opponent_pawn_index = self.opponent.pawns.index(move[1])
            starting_position = self.opponent.starting_positions.pop(oppo
            self.opponent.pawns.remove(move[1])
            self.opponent.captured_pawns.append(starting_position)

        if move[1][0] == self.opponent.goal_line:
            return

        if self.probability <= 0:
            return

        if random.random() < self.probability and self.opponent.captured_
            resurrected_pawn_index = random.choice(range(len(self.opponen
            resurrected_pawn = self.opponent.captured_pawns.pop(resurrect
            self.opponent.pawns.append(resurrected_pawn)
            self.opponent.starting_positions.append(resurrected_pawn)

    def lose(self):
        return any([i == self.opponent.goal_line for i, j in self.opponen
            self.possible_moves() == []
        )

    def is_over(self):
        return self.lose()
```

```
    def show(self):
        f = (
            lambda x: "1"
            if x in self.players[0].pawns
            else ("2" if x in self.players[1].pawns else ".")
        )
        print(
            "\n".join(
                [
                    " ".join([f((i, j)) for j in range(self.size[1])])
                    for i in range(self.size[0])
                ]
            )
        )
```

Odcinanie alfa-beta to metoda optymalizacji, która pozwala na pominięcie niektórych gałęzi drzewa gry, które nie wypłyną na końcowy wynik. Zgodnie z zadaniem musimy zmodyfikować algorytm Negamax, tak aby dało się go wywołać bez odcinania alfa-beta.

W celu umożliwienia uruchomienia algorytmu Negamax bez odcinania alfa-beta, dodajemy parametr prunning do funkcji negamax i klasy Negamax. Ten parametr określa, czy algorytm powinien stosować odcinanie alfa-beta. Jeśli prunning jest ustawione na True, algorytm działa tak jak wcześniej. Jeśli jest ustawione na False, algorytm pomija odcinanie alfa-beta.

In [ ]:
```
"""
The standard AI algorithm of easyAI is Negamax with alpha-beta pruning
and (optionnally), transposition tables.
"""

import pickle

LOWERBOUND, EXACT, UPPERBOUND = -1, 0, 1
inf = float("infinity")


def negamax(game, depth, origDepth, scoring, alpha=+inf, beta=-inf, tt=No
    """
    This implements Negamax with transposition tables.
    This method is not meant to be used directly. See ``easyAI.Negamax``
    for an example of practical use.
    This function is implemented (almost) acccording to
    http://en.wikipedia.org/wiki/Negamax
    """

    alphaOrig = alpha

    # Is there a transposition table and is this game in it ?
    lookup = None if (tt is None) else tt.lookup(game)
```

```
    if lookup is not None:
        # The game has been visited in the past

        if lookup["depth"] >= depth:
            flag, value = lookup["flag"], lookup["value"]
            if flag == EXACT:
                if depth == origDepth:
                    game.ai_move = lookup["move"]
                return value
            elif flag == LOWERBOUND:
                alpha = max(alpha, value) if prunning else alpha
            elif flag == UPPERBOUND:
                beta = min(beta, value) if prunning else beta

            if prunning and alpha >= beta:
                if depth == origDepth:
                    game.ai_move = lookup["move"]
                return value

    if (depth == 0) or game.is_over():
        # NOTE: the "depth" variable represents the depth left to recurse
        # so the smaller it is, the deeper we are in the negamax recursio
        # Here we add 0.001 as a bonus to signify that victories in less
        # have more value than victories in many turns (and conversely, d
        # after many turns are preferred over defeats in less turns)
        return scoring(game) * (1 + 0.001 * depth)

    if lookup is not None:
        # Put the supposedly best move first in the list
        possible_moves = game.possible_moves()
        possible_moves.remove(lookup["move"])
        possible_moves = [lookup["move"]] + possible_moves

    else:

        possible_moves = game.possible_moves()

    state = game
    best_move = possible_moves[0]
    if depth == origDepth:
        state.ai_move = possible_moves[0]

    bestValue = -inf
    unmake_move = hasattr(state, "unmake_move")

    for move in possible_moves:

        if not unmake_move:
            game = state.copy()  # re-initialize move

        game.make_move(move)
        game.switch_player()
```

```
        move_alpha = -negamax(game, depth - 1, origDepth, scoring, -beta,

        if unmake_move:
            game.switch_player()
            game.unmake_move(move)

        # bestValue = max( bestValue,  move_alpha )
        if bestValue < move_alpha:
            bestValue = move_alpha
            best_move = move

        if prunning and alpha < move_alpha:
            alpha = move_alpha
            # best_move = move
            if depth == origDepth:
                state.ai_move = move
            if alpha >= beta:
                break

    if tt is not None:

        assert best_move in possible_moves
        tt.store(
            game=state,
            depth=depth,
            value=bestValue,
            move=best_move,
            flag=UPPERBOUND
            if (bestValue <= alphaOrig)
            else (LOWERBOUND if (bestValue >= beta) else EXACT),
        )

    return bestValue


class Negamax:
    """
    This implements Negamax on steroids. The following example shows
    how to setup the AI and play a Connect Four game:

        >>> from easyAI.games import ConnectFour
        >>> from easyAI import Negamax, Human_Player, AI_Player
        >>> scoring = lambda game: -100 if game.lose() else 0
        >>> ai_algo = Negamax(8, scoring) # AI will think 8 turns in adva
        >>> game = ConnectFour([Human_Player(), AI_Player(ai_algo)])
        >>> game.play()

    Parameters
    ----------

    depth:
      How many moves in advance should the AI think ?
```

```
    (2 moves = 1 complete turn)

  scoring:
    A function f(game)-> score. If no scoring is provided
       and the game object has a ``scoring`` method it ill be used.

  win_score:
    Score above which the score means a win. This will be
      used to speed up computations if provided, but the AI will not
      differentiate quick defeats from long-fought ones (see next
      section).

  tt:
    A transposition table (a table storing game states and moves)
    scoring: can be none if the game that the AI will be given has a
    ``scoring`` method.

  Notes
  -----

  The score of a given game is given by

  >>> scoring(current_game) - 0.01*sign*current_depth

  for instance if a lose is -100 points, then losing after 4 moves
  will score -99.96 points but losing after 8 moves will be -99.92
  points. Thus, the AI will chose the move that leads to defeat in
  8 turns, which makes it more difficult for the (human) opponent.
  This will not always work if a ``win_score`` argument is provided.

  """

  def __init__(self, depth, scoring=None, win_score=+inf, tt=None, prun
      self.scoring = scoring
      self.depth = depth
      self.tt = tt
      self.win_score = win_score
      self.prunning = prunning

  def __call__(self, game):
      """
      Returns the AI's best move given the current state of the game.
      """

      scoring = (
          self.scoring if self.scoring else (lambda g: g.scoring())
      )  # horrible hack

      self.alpha = negamax(
          game,
          self.depth,
          self.depth,
          scoring,
```

```
            -self.win_score,
            +self.win_score,
            self.tt,
            self.prunning,
        )
        return game.ai_move
```

In [ ]:
```
# If True, the game is deterministic (no random elements)
DETERMINISTIC = False

# The probability that a captured pawn is resurrected
PROBABILITY = [0, 0.1, 0.5, 0.9]

# Number of games to play
NUMBER_OF_GAMES = 5

# Depth of the ai predictions
AI_DEPTHS = [2, 5, 15]
```

Symulacja jest przeporwadzana dla kilku kombinacji głębokości przeszukiwania algorytmu Negamax (AI_DEPTHS), prawdopodobieństwa odrodzenia pionków (PROBABILITY) i uzycia odcinania alfa-beta (prunning).

Dla każdej kombinacji tych parametrów, tworzony jest nowy obiekt gry Hexapawn z dwoma graczami AI. Gracze AI korzystają z algorytmu Negamax z określoną głębokością przeszukiwania i użyciem (lub nie) odcinania alfa-beta. Prawdopodobieństwo odrodzenia pionów jest ustawiane na określoną wartość.

Lista results zawiera wyniki wszystkich gier, wraz z parametrami, które były używane w tych grach.

In [ ]:
```
from easyAI import AI_Player, Human_Player

scoring = lambda game: -100 if game.lose() else 0
results = []

for i in range(NUMBER_OF_GAMES):
    for depth in AI_DEPTHS:
        for probability in PROBABILITY:
            for prunning in [True, False]:
                variant = 'Deterministic' if probability == 0 else 'Proba
                print(f'\n\n ========= Starting game {i+1} at depth {dept
                ai = Negamax(depth, scoring, prunning=prunning)
                game = Hexapawn([AI_Player(ai), AI_Player(ai)], probabili

                start_time = time.time()
                game.play()
                elapsed_time = time.time() - start_time

                winner = game.opponent_index
                results.append([i+1, depth, variant, prunning, probabilit
```

In [ ]:
```
import pandas as pd

df = pd.DataFrame(results, columns=['Game', 'Depth', 'Variant', 'Prunning
print(df)
```

```
     Game  Depth        Variant  Prunning  Probability  Winner       Time
0       1      2  Deterministic      True          0.0       1   0.008092
1       1      2  Deterministic     False          0.0       2   0.009438
2       1      2  Probabilistic      True          0.1       1   0.005668
3       1      2  Probabilistic     False          0.1       2   0.007404
4       1      2  Probabilistic      True          0.5       2   0.005858
..    ...    ...            ...       ...          ...     ...        ...
115     5     15  Probabilistic     False          0.1       2   4.526428
116     5     15  Probabilistic      True          0.5       2  20.530115
117     5     15  Probabilistic     False          0.5       2  20.903446
118     5     15  Probabilistic      True          0.9       1  31.203263
119     5     15  Probabilistic     False          0.9       2  25.756992

[120 rows x 7 columns]
```

Kolumny w tabeli oznaczają:

- Game: Numer gry w symulacji.
- Depth: Głębokość przeszukiwania algorytmu Negamax używana w tej grze.
- Variant: Wariant gry - "Deterministic" oznacza, że prawdopodobieństwo odrodzenia pionów było ustawione na 0, a "Probabilistic" oznacza, że było większe od 0.
- Prunning: Czy w tej grze używano odcinania alfa-beta (True) czy nie (False).
- Probability: Prawdopodobieństwo odrodzenia pionów w tej grze.
- Winner: Który gracz wygrał tę grę - gracz 1 czy 2.
- Time: Czas trwania gry w sekundach.

Analizując wyniki widzimy, ze generalnie czas gry rośnie wraz ze wzrostem głębokości przeszukiwania algorytmu Negamax. Gry z odcinaniem alfa-beta zazwyczaj trwają krócej ni gry bez odcinania, co jest spodziewane, jako iz ta technika optymalizacji ma na celu przyspieszenie przesuzkiwania drzewa gry.

In [ ]:
```
print('\n\nNumber of wins for each player at each depth and variant:')
print(df.groupby(['Probability', 'Depth', 'Prunning', 'Winner']).size())
```

```
Number of wins for each player at each depth and variant:
Probability  Depth  Prunning  Winner
0.0          2      False     2         5
                    True      1         5
             5      False     2         5
                    True      1         5
             15     False     2         5
                    True      1         5
0.1          2      False     2         5
                    True      1         4
                              2         1
             5      False     2         5
                    True      1         2
                              2         3
             15     False     2         5
                    True      1         4
                              2         1
0.5          2      False     2         5
                    True      1         4
                              2         1
             5      False     2         5
                    True      1         2
                              2         3
             15     False     2         5
                    True      1         2
                              2         3
0.9          2      False     2         5
                    True      1         4
                              2         1
             5      False     2         5
                    True      2         5
             15     False     2         5
                    True      1         3
                              2         2
dtype: int64
```

Tabela przedstawia liczbę wygranych dla każdego z graczy w zależności od głębokości przeszukiwania algorytmu Negamax (Depth), prawdopodobieństwa odrodzenia pionów (Probability) i użycia odcinania alfa-beta (Prunning).

Kolumny w tabeli oznaczają:

- Probability: Prawdopodobieństwo odrodzenia pionów w grze.
- Depth: Głębokość przeszukiwania algorytmu Negamax używana w grze.
- Prunning: Czy w grze używano odcinania alfa-beta (True) czy nie (False).
- Winner: Który gracz wygrał grę - gracz 1 czy 2.
- Wartość w ostatniej kolumnie: Liczba wygranych dla danego gracza.

W większości przypadków gracz 2 wygrywał więcej gier, pradopodbieństwo odrodzenia pionów i uzycie odcinanai alfa-beta miało wpływ na wyniki gier - dla

prawdopodobieństwa odrodzenia pionów 0.1 i głębokości przeszukiwania 5, gracz 1 wygrał więcej gier, gdy odcinanie alfa-beta było wyłączone, ale gracz 2 wygrał więcej gier, gdy odcinanie alfa-beta było włączone.

In [ ]:
```
average_times = df.groupby(['Probability', 'Depth', 'Prunning'])['Time'].
print('\n\nAverage times spent by each AI variant:')
print(average_times)
```

```
Average times spent by each AI variant:
Probability  Depth  Prunning
0.0          2      False        0.006232
                    True         0.004718
             5      False        0.048968
                    True         0.037811
             15     False        2.572616
                    True         1.001569
0.1          2      False        0.005675
                    True         0.004440
             5      False        0.049196
                    True         0.039896
             15     False        4.233591
                    True         3.625594
0.5          2      False        0.005430
                    True         0.004732
             5      False        0.054034
                    True         0.044345
             15     False       36.004777
                    True        15.133766
0.9          2      False        0.005371
                    True         0.004697
             5      False        0.055539
                    True         0.043778
             15     False       25.626632
                    True        43.148022
Name: Time, dtype: float64
```

Tabela przedstawia średni czas gry dla różnych kombinacji głębokości przeszukiwania, prawdopodobieństwa odrodzenia pionów i użycia odcinania alfa-beta.

Kolumny w tabeli oznaczają:

- Probability: Prawdopodobieństwo odrodzenia pionów w grze.
- Depth: Głębokość przeszukiwania algorytmu Negamax używana w grze.
- Prunning: Czy w grze używano odcinania alfa-beta (True) czy nie (False).
- Time: Średni czas trwania gry w sekundach.

Z wyników można zauważyć, że średni czas gry generalnie rośnie wraz ze wzrostem głębokości przeszukiwania. Jest to spodziewane, ponieważ większa głębokość

przeszukiwania oznacza, że algorytm musi przeszukać więcej gałęzi drzewa gry.