Sprawozdanie

Michał Szczygieł Antoni Zięciak

Cel ćwiczenia

Celem ćwiczenia jest analiza działania alogrymu Negamax (wersji zwykłej oraz *alpha-beta pruning*) w różnych wariantach gry *Nim* - probabilistycznym (z 10% prawdopodobieństwem gracz, który się porusza, musi wziąć z wybranego stosu o jeden element mniej niż zamierzał) i deterministycznym. Przeanalizowane są średni czas wykonania ruchu oraz liczba zwycięstw.

Opis problemu - Nimby

Nimby to dwuosobowa gra, w której każdy z graczy podczas ruchu usuwa z jednego ze stosów krążek/krążki. Stos zawiera określoną liczbę krążków, z których gracz może zabrać od 1 do *liczby dostępnych krążków na stosie* podczas tury. Gracz, który usunie ostatni dostępny krążek przegrywa.

In [ ]:
```
from copy import deepcopy
from random import random
from easyAI import TwoPlayerGame
from time import time
from easyAI import AI_Player, Negamax
```

Realizacja rozwiązania

Klasa *Nim* dziedziczy po klasie *TwoPlayerGame* z biblioteki *EasyAI*. Jest ona implementacją gry Nim dla dwóch graczy. Istnieje opcja, aby gracze mieli 10% szans na zmniejszenie liczby ściąganych krążków ze stosu o 1. Przyjmuje ona również atrybut *times*, do którego zapisywane są czasy wykonywania ruchu obu graczy i na podstawie, którego jest liczony średni czas ruchu.

In [ ]:
```
class Nim(TwoPlayerGame):
    def __init__(self, players=None, max_removals_per_turn=None, piles=(5, 5, 5,
        """ Default for `piles` is 5 piles of 5 pieces. """
        self.players = players
        self.piles = list(piles)
        self.times = times
        self.max_removals_per_turn = max_removals_per_turn
        self.current_player = current_player  # player 1 starts.
        self.prob = prob

    def possible_moves(self):
        return [
            "%d,%d" % (i + 1, j)
            for i in range(len(self.piles))
            for j in range(
                1,
```

```
                self.piles[i] + 1
                if self.max_removals_per_turn is None
                else min(self.piles[i] + 1, self.max_removals_per_turn),
            )
        ]

    def make_move(self, m):
        move = list(map(int, m.split(",")))
        num_pieces = move[1]

        # Check if the current player is affected by the 10% probability
        if self.prob and random() < 0.1:
            num_pieces -= 1

        self.piles[move[0] - 1] -= num_pieces

    def show(self):
        print(" ".join(map(str, self.piles)))

    def win(self):
        return max(self.piles) == 0

    def is_over(self):
        return self.win()

    def scoring(self):
        return 100 if self.win() else 0

    def play(self, nmoves=1000, verbose=True):
        history = []

        if verbose:
            self.show()

        for self.nmove in range(1, nmoves + 1):

            if self.is_over():
                break

            t1 = time()
            move = self.player.ask_move(self)
            t2 = time()
            total = t2-t1
            self.times[self.player].append(total)
            history.append((deepcopy(self), move))
            self.make_move(move)

            if verbose:
                print(
                    "\nMove #%d: player %d plays %s :"
                    % (self.nmove, self.current_player, str(move))
                )
                self.show()

            self.switch_player()

        history.append(deepcopy(self))

        return history
```

```
    def ttentry(self):
        return tuple(self.piles)  # optional, speeds up AI
```

Funkcja Negamax

W przypadku funkcji Negamax z alpha-beta pruning została wykorzystana klasa z biblioteki *EasyAI*, dostarczas ona wrapper na funkcję *negamax* z tej samej bibliteki, która pozwala na wykorzystanie jej przez gracza AI. Klasa *Negamax2* jest wrapperem na funkcję *negamax2*, która jest modyfikacją wcześniej wspomnianej funkcji, która różni się tylko tym, że nie przerywa algorytmu, gdy *alpha* >= *beta*.

In [ ]:
```
LOWERBOUND, EXACT, UPPERBOUND = -1, 0, 1
inf = float("infinity")


def negamax2(game, depth, origDepth, scoring, alpha=+inf, beta=-inf, tt=None):
    alphaOrig = alpha

    if (depth == 0) or game.is_over():
        return scoring(game) * (1 + 0.001 * depth)

    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 = -negamax2(game, depth - 1, origDepth, scoring, -beta, -alph

        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 alpha < move_alpha:
            alpha = move_alpha
            # best_move = move
            if depth == origDepth:
                state.ai_move = move

    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 Negamax2:
    def __init__(self, depth, scoring=None, win_score=+inf, tt=None):
        self.scoring = scoring
        self.depth = depth
        self.tt = tt
        self.win_score = win_score

    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 = negamax2(
            game,
            self.depth,
            self.depth,
            scoring,
            -self.win_score,
            +self.win_score,
            self.tt,
        )
        return game.ai_move
```

Funkcja do rozegrania rozgrywki

Rozgrywa grę między podanymi graczami w wariancie deterministycznym, oraz niedeterministycznymm, oraz zlicza zwycięstwa obu graczy i mierzy ich średni czas podjęcia decyzji.

In [ ]:

```
def play_game(p1, p2):
    print("-----Deterministic-----")
    wins = {
        1: 0,
        2: 0
    }

    times = {
        p1: [],
        p2: []
    }
```

```
# deterministc
for i in range(10):
    game = Nim([p1, p2], current_player=(i % 2 + 1), times=times)
    game.play(verbose=False)

    wins[game.current_player] += 1

print(f'Player 1 won: {wins[1]} times')
print(f'Player 2 won: {wins[2]} times')

print(f"Player 1 avg time: {sum(game.times[p1]) / len(game.times[p1])}")
print(f"Player 2 avg time: {sum(game.times[p2]) / len(game.times[p2])}")

print("-----Probabilistic-----")
wins = {
    1: 0,
    2: 0
}

times = {
    p1: [],
    p2: []
}

# probabilitsic
for i in range(10):
    game = Nim([p1, p2], current_player=(i % 2 + 1), prob=True, times=times)
    game.play(verbose=False)

    wins[game.current_player] += 1

print(f'Player 1 won: {wins[1]} times')
print(f'Player 2 won: {wins[2]} times')

print(f"Player 1 avg time: {sum(game.times[p1]) / len(game.times[p1])}")
print(f"Player 2 avg time: {sum(game.times[p2]) / len(game.times[p2])}")
```

Podjęte próby rozwiązania

Do powyższej funkcji zostały wysłane dwie pary graczy:

- korzystający z algorytmy *Negamax* z odcięciem
  - pierwszy gracz ma głębokość 8
  - drugi gracz ma głębokość 6
- korzystający z algorytmy *Negamax* bez odcięcia
  - pierwszy gracz ma głębokość 4
  - drugi gracz ma głębokość 3

W przypadku algorytmu *Negamax* bez odcięcia próbowaliśmy sprawdzić zestawienie graczy o głębokości 8 i 6, lecz po upływie 30 minut i braku rozwiązania nawet dla przypadku deterministycznego zaprzestaliśmy eksperymentu.

Gracze z alfa-beta prunningiem

In [ ]:
```
p1 = AI_Player(
    Negamax(8)
)
p2 = AI_Player(
    Negamax(6)
)
play_game(p1, p2)
```

```
-----Deterministic-----
Player 1 won: 10 times
Player 2 won: 0 times
Player 1 avg time: 2.181157112121582
Player 2 avg time: 0.23351802323993884
-----Probabilistic-----
Player 1 won: 7 times
Player 2 won: 3 times
Player 1 avg time: 2.7088718800931364
Player 2 avg time: 0.28632613232261256
```

Gracze bez alfa-beta prunning

In [ ]:
```
p1 = AI_Player(
    Negamax2(4)
)
p2 = AI_Player(
    Negamax2(3)
)
play_game(p1, p2)
```

```
-----Deterministic-----
Player 1 won: 10 times
Player 2 won: 0 times
Player 1 avg time: 0.8876371972701129
Player 2 avg time: 0.09054574171702066
-----Probabilistic-----
Player 1 won: 4 times
Player 2 won: 6 times
Player 1 avg time: 1.0115687305277044
Player 2 avg time: 0.10571060509517274
```

Liczba wygranych, przegranych

- gracze z alfa-beta prunning - wariant deterministyczny

	Gracz 1	Gracz 2
Liczba wygranych	10	0
Średni czas ruchu	2.18	0.23

- gracze z alfa-beta prunning - wariant probabilistyczny

	Gracz 1	Gracz 2
Liczba wygranych	7	3

	Gracz 1	Gracz 2
Średni czas ruchu	2.71	0.29

- gracze bez alfa-beta prunning - wariant deterministyczny

	Gracz 1	Gracz 2
Liczba wygranych	10	0
Średni czas ruchu	0.89	0.09

- gracze bez alfa-beta prunning - wariant probabilistyczny

	Gracz 1	Gracz 2
Liczba wygranych	4	6
Średni czas ruchu	1.01	0.11

Przemyślenia i wnioski

Zauważyć można, że jedynym sensownym wariantem algorytmu *Negamax* jest zasosowanie alpha-beta pruningu. Wariant ten jest o rząd wielkości szybszy od podstawowej wersji. Akceptowalny czas wykonywania jednego ruchu w wersji alogrytmu bez obcięcia kończy się w momencie, w którym głębokość przekracza 4. W deterministycznym przypadku grę wygra zawodnik z większą głębokością. W przypadku wariantu probabilistycznego zwycięzca będzie losowy (a przynajmniej takie wrażenie odnieśliśmy po uruchomieniu rozgrywki kilka razy).