Inteligencja obliczeniowa w analizie danych cyfrowych

Kacper Cienkosz kcienkosz@student.agh.edu.pl
Miłosz Dubiel miloszdubiel@student.agh.edu.pl

10 marca 2024

EasyAI

Clumsy connect four

Gra logiczna dla dwóch osób opierająca się na ułożeniu wertykalnie, horyzontalnie lub po skosie 4 żetonów na planszy o wymiarach 7 x 6 pól. Żetony są umieszczane na planszy, poprzez wrzucanie ich od góry, co powoduje, że żeton zajmuje pierwsze wolne miejsce w danym rzędzie planszy.

Implementacja tej gry znajduje się w pythonowej bibliotece EasyAI.

Naszym zadaniem było rozszerzenie standardowej wersji tej gry o aspekt probabilistyczny. Wybierany przez gracza rząd, do którego powinien trafić żeton, mógł zostać zmieniony na rząd poprzedzający bądź następujący.

Fragment kodu ConnectFour.py, który pokazuje sposób wprowadzenia elementu probabilistycznego do oryginalnej gdy *Connect Four*:

```
def make_move(self, column):
    possible_moves = [column]
    if column > 0:
        possible_moves.append(column - 1)
    if column < len(self.board[0]) - 1:
        possible_moves.append(column + 1)

    chosen_column = random.choice(possible_moves)
    if chosen_column in self.possible_moves():
        self._make_move(chosen_column)

def _make_move(self, column):
        line = np.argmin(self.board[:, column] != 0)
        self.board[line, column] = self.current_player
```

Podstawowe porównanie wersji deterministycznej i probabilistycznej

Pierwsze porównanie zostało przeprowadzone poprzez przeprowadzenie **stu** rozgrywek dwóch graczy napędzanych algorytmem *Negamax z pruningiem alfa-beta* zarówno dla deterministycznej, jak i probablistycznej wersji gry Connect Four. Ponadto, zostały zastosowane różne głębokości przeszukiwania drzewka możliwych kroków przez graczy AI.

Wyniki prezentujemy w postaci tabeli:

		Deterministyczne Connect Four		Probabilistyczne Connect Four	
	Głębokość	Wygrane	Czas podejmowania decyzji	Wygrane	Czas podejmowania decyzji
Gracz 1	3	0	0.057	51	0.064
Gracz 2	5	100	0.326	49	0.422
Gracz 1	6	0	0.733	59	0.958
Gracz 2	3	100	0.054	40	0.062

Przyglądając się powyższym wynikom, można zauważyć, że dla deterministycznej wersji gdy mamy całkowitą przewagę gracza drugiego, nawet pomimo mniejszej głębokości przeszukiwania względem przeciwnika. Jedyną zastanawiającą rzeczą jest, że ta gra posiada strategię wygrywającą dla gracza rozpoczynającego rozgrywkę, dlatego niezrozumiałym dla nas jest, dlaczego wygrywał gracz drugi, a nie pierwszy.

Dla probabilistycznej wersji gry można zauważyć, że gracz rozpoczynający z głębokością mniejszą (3 vs 5) i tak ma więcej zwycięstw, co potwierdzałoby strategię pierwszego gracza wygrywającego. Gdy gracz pierwszy miał większą głębokość przeszukiwania (6 vs 3) odsetek jego zwycięstw uwydatnia się jeszcze bardziej, ale dodatkowo pojawia się jeden remis.

Dodatkowo podane zostały czasy podejmowania decyzji przez algorytmy o różnych głębokościach przeszukiwania drzewka kolejnych stanów.

Wprowadzenie zależności od odcinania alfa-beta

Ze względu na brak domyślnego dostępu do zarządzania alfa-beta pruningiem w implementacji algorytmu *Negamax* w bibliotece *EasyAI* - alfa-beta pruning jest na stałe włączony - dokonaliśmy modyfikacji wspomnianego wcześniej algorytmu, dzięki której jesteśmy w stanie zapanować nad występowaniem odcinania alfa-beta.

Modyfikacje kodu w Negamax.py (kod skrócony w miejscach `[...]`):

```
def negamax(game, depth, origDepth, scoring, alpha=+inf, beta=-inf,
    tt=None, alpha_beta=True):
    [...]

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

        if lookup["depth"] >= depth:
            [...]

            if alpha_beta is False:
                return value

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

    [...]

    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, -alpha,
            tt, alpha_beta=alpha_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 alpha < move_alpha:
                alpha = move_alpha
                # best_move = move
                if depth == origDepth:
                    state.ai_move = move
                if alpha_beta is True and alpha >= beta:
                    break

    [...]

    return bestValue

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

def __call__(self, game):
    [...]

    self.alpha = negamax(
        game,
        self.depth,
        self.depth,
        scoring,
        -self.win_score if self.alpha_beta else -inf,
        +self.win_score if self.alpha_beta else +inf,
        self.tt,
        self.alpha_beta,
    )
    return game.ai_move
```

Próba pierwsza - błędna

Podczas pierwszej próby testowej musieliśmy mieć błąd w kodzie odcinającym alfa-beta, przez co algorytm nie zadziałał poprawnie. Następne dwie tabele i komentarz do nich można pominąć, zostawiamy je jednak jako udokumentowanie prób.

Wyniki poszczególnych testów wraz z parametrami prezentują się następująco:

			Deterministyczne Connect Four		Probabilistyczne Connect Four	
	Głębokość	Alfa-beta	Wygrane	Czas pod. dec.	Wygrane	Czas pod. dec.
Gracz 1	6	Tak	0	0.172	60	0.217
Gracz 2	3	Tak	100	0.013	39	0.014
Gracz 1	6	Tak	0	0.173	56	0.231
Gracz 2	3	Nie	100	0.013	44	0.015
Gracz 1	6	Nie	0	0.177	58	0.227
Gracz 2	3	Tak	100	0.013	42	0.015
Gracz 1	6	Nie	0	0.182	57	0.230
Gracz 2	3	Nie	100	0.013	41	0.015

			Deterministyczne Connect Four		Probabilistyczne Connect Four	
	Głębokość	Alfa-beta	Wygrane	Czas pod. dec.	Wygrane	Czas pod. dec.
Gracz 1	3	Tak	0	0.013	59	0.015
Gracz 2	5	Tak	100	0.078	41	0.100
Gracz 1	3	Tak	0	0.014	51	0.015
Gracz 2	5	Nie	100	0.078	49	0.100
Gracz 1	3	Nie	0	0.014	51	0.015
Gracz 2	5	Tak	100	0.078	49	0.100
Gracz 1	3	Nie	0	0.014	58	0.015
Gracz 2	5	Nie	100	0.078	42	0.100

Powyższe wyniki nie pokazują różnic ani pomiędzy czasami podejmowania decyzji przez algorytmy, ani w rezultatach gier. Może oznaczać to, że nasze pierwsze próby dodania poprawek do wyłączenia obcinania alfa-beta okazały się niepoprawne i nie zadziałały.

0.0.1 Próba druga

Z premedytacją zmniejszyliśmy tutaj głębokość przeszukiwania drzewka kolejnych stanów ze względu na drastycznie rosnący czas wykonywania obliczeń.

Wyniki prezentują się następująco:

			Deterministyczne Connect Four		Probabilistyczne Connect Four	
	Głębokość	Alfa-beta	Wygrane	Czas pod. dec.	Wygrane	Czas pod. dec.
Gracz 1	3	Tak	10	0.00787	6	0.00797
Gracz 2	1	Tak	0	0.000845	4	0.000892
Gracz 1	3	Tak	10	0.00797	5	0.00768
Gracz 2	1	Nie	0	0.000859	5	0.000865
Gracz 1	3	Nie	10	0.0337	7	0.0320
Gracz 2	1	Tak	0	0.000851	3	0.000861
Gracz 1	3	Nie	10	0.0334	6	0.0306
Gracz 2	1	Nie	0	0.000846	4	0.000840

Już po pierwszym teście możemy zauważyć różnicę w czasie podejmowania decyzji dla głębokości 3 z alfa-beta pruningiem i bez niego - odpowiednio ok. 0.00787 [s] vs ok. 0.0337 [s].

Wyniki drugiego testu po zmianie głębokości:

			Deterministyczne Connect Four		Probabilistyczne Connect Four	
	Głębokość	Alfa-beta	Wygrane	Czas pod. dec.	Wygrane	Czas pod. dec.
Gracz 1	4	Tak	0	0.0117	6	0.0194
Gracz 2	2	Tak	10	0.00173	4	0.00243
Gracz 1	4	Tak	0	0.0116	7	0.0162
Gracz 2	2	Nie	10	0.00361	3	0.00511
Gracz 1	4	Nie	0	0.121	4	0.228
Gracz 2	2	Tak	10	0.00173	6	0.00243
Gracz 1	4	Nie	0	0.121	6	0.223
Gracz 2	2	Nie	10	0.00362	4	0.00569

Tutaj również wyraźnie widać, że dla wyłączonego alfa-beta pruningu czas podejmowania decyzji znacznie wzrasta, zarówno dla głębokości 4 (np. 0.0117 [s] vs 0.121 [s]) jak i dla głębokości 2 (np. 0.00173 [s] vs 0.00361 [s]).

Kod do wielokrotnego testowania

Dla każdej z gier, każdej z wcześniej ustawionych głębokości i każdego ustawienia alpha/beta za pomocą tej funkcji jesteśmy w stanie zmierzyć czas działania dla zadanej liczby gier oraz wybranego algorytmu (w tym wypadku Negamax).

```
def run(algo, num_of_games: int) -> None:
    games = (ProbabilisticConnectFour, ConnectFour)
    ultimate_results = {ProbabilisticConnectFour.__name__: [], ConnectFour.__name__: []}
    depths = [(3, 5), (6, 3), (6, 1), (8, 1)]
    alpha_betas = [(True, True), (True, False), (False, True), (False, False)]

    for game in games:
        for depth in depths:
            for alpha_beta in alpha_betas:
                ai_algo_1 = algo(depth[0], alpha_beta=alpha_beta[0])
                ai_algo_2 = algo(depth[1], alpha_beta=alpha_beta[1])
                player1 = AI_Player(
                    ai_algo_1,
                    name=f"first_{algo.__name__}_d{ai_algo_1.depth}_ab"
                    +f"{'T' if ai_algo_1.alpha_beta else 'F'}"
                )
                player2 = AI_Player(
                    ai_algo_2,
                    name=f"second_{algo.__name__}_d{ai_algo_2.depth}_ab"
                    +f"{'T' if ai_algo_2.alpha_beta else 'F'}"
                )

                results = []
                for _ in range(num_of_games):
                    winner = play_game(game, player1, player2)
                    results.append(winner)

                counter = Counter(results)
                print(game.__name__)
                ultimate_results[game.__name__].append(
                    {
                        f"d{depth[0]}{depth[1]}_ab"
                        +f"{'T' if ai_algo_1.alpha_beta else 'F'}"
```

```
                    +f"{'T' if ai_algo_2.alpha_beta else 'F'}": {
                        player1.name: player1.avg_of_times(),
                        player2.name: player2.avg_of_times(),
                        "results": counter
                    }
                }
            )
```

Dodatkowo przygotowaliśmy kod do mierzenia czasu potrzebnego na wykonanie jednego ruchu i znalezienie optymalnego rozwiązania.

```
def avg_of_times(self):
    return sum(self.times) / len(self.times)

def ask_move(self, game):
    start = perf_counter()
    move = self.AI_algo(game)
    end = perf_counter()
    self.times.append(end - start)

    return move
```