Sprawozdanie z Projektu 1: EasyAI

Bartosz Zielinski, Sergey Zeliuk

9 marca 2025

1 Wstęp

Celem niniejszego sprawozdania jest przedstawienie implementacji prostej gry w *kółko i krzyżyk* (ang. *Tic-Tac-Toe*), wykorzystującej algorytm `Negamax` do wyznaczania najlepszego ruchu dla gracza AI, a także zaprezentowanie eksperymentów, w których porównano efektywność działania sztucznej inteligencji z włączonym i wyłączonym przycinaniem alfa-beta (*Alpha-Beta Pruning*). Gra posiada również wersję probabilistyczną, w której istnieje 20% szansa, że wybrany ruch nie zostanie wykonany (tzw. *pudło*). W eksperymentach mierzono m.in. średni czas podejmowania decyzji przez każdą ze stron oraz liczbę wygranych, przegranych i remisów.

2 Opis gry i implementacji

2.1 Klasa `TicTacToe`

Podstawowa logika gry została zamknięta w klasie `TicTacToe`, dziedziczącej z `TwoPlayerGame` z biblioteki `easyAI`. Najważniejsze elementy:

- `board`: lista dziewięciu pól (0 – puste, 1 – gracz pierwszy, 2 – gracz drugi),
- `possible_moves()`: zwraca listę indeksów pól, na które można wykonać ruch,
- `make_move(move)`: wykonuje ruch na planszy; jeśli gra jest w trybie probabilistycznym i nie jest to symulacja (AI obliczające *minimax*), istnieje 20% szansy na "pudło",
- `lose()`: sprawdza, czy aktualny gracz przegrał (tj. czy przeciwnik ułożył rząd 3 symboli),
- `is_over()`: sprawdza, czy gra się zakończyła wskutek wygranej któregoś z graczy lub wyczerpania pól,
- `scoring()`: funkcja oceniania do algorytmu `Negamax` (kara -100 za przegraną).

Wyróżniono fazę symulacyjną (do wyboru najlepszego ruchu przez `Negamax`) oraz fazę rzeczywistą (`probabilistic`), gdzie może wystąpić prawdopodobieństwo chybienia ruchu.

2.2 Klasa `TimedAI_Player`

Klasa `TimedAI_Player` dziedziczy po `AI_Player` i dodatkowo mierzy czas poświęcony na obliczanie ruchów przez algorytm. W ten sposób każdy ruch dodaje się do sumarycznego czasu `total_time`, a na końcu można wyliczyć średni czas na posunięcie (`avg_time`).

2.3 Funkcja `run_matches`

Funkcja ta pozwala na uruchomienie wielu partii (`n_matches`) pomiędzy dwoma przeciwnikami AI i zwraca statystyki:

- liczba wygranych `p1_wins` i `p2_wins`,
- liczba remisów `draws`,
- szczegółowe rozbicie wygranych z perspektywy rozpoczynającego (`p1_starting_wins`, `p2_starting_wins`) i nie-rozpoczynającego,
- średni czas podejmowania decyzji (`p1_avg_time`, `p2_avg_time`).

Wyniki są następnie zwracane w postaci słownika, co pozwala na łatwą konwersję do tabeli w pakiecie `pandas`.

2.4 Funkcja `main()`

1. Definiuje pary głębokości przeszukiwania: $(3, 3)$, $(9, 9)$ oraz $(3, 9)$.
2. Dla każdej pary tworzy dwa obiekty AI z zadanymi głębokościami i włączonym (`win_score=float('inf')`) lub wyłączonym (`win_score=1e9`) mechanizmem alpha-beta.
3. Wykonuje serię rozgrywek bez trybu probabilistycznego (deterministycznie) oraz w trybie probabilistycznym (z 20% szansą na chybienie).
4. Zapisuje zebrane wyniki (liczba wygranych, przegranych, remisów, średni czas ruchu itp.) do pliku `results.xlsx`.

Poniżej przedstawiono kluczowe fragmenty kodu:

```
class TicTacToe(TwoPlayerGame):
    def __init__(self, players, probabilistic=True):
        self.players = players
        self.current_player = 1
        self.board = [0]*9
        self.probabilistic = probabilistic
        self.simulation = True  # Oznacza, e ruchy wykonywane s
           tylko symulacyjnie
    ...

class TimedAI_Player(AI_Player):
    def __init__(self, ai_algo):
        super().__init__(ai_algo)
        self.total_time = 0.0
        self.n_moves = 0

    def ask_move(self, game):
        start = time.time()
        move = super().ask_move(game)
        end = time.time()

        self.total_time += (end - start)
        self.n_moves += 1
        return move
```

```
def run_matches(n_matches, ai1, ai2, probabilistic=False, verbose=False)
    :
    ...
    return results

def main():
    depth_pairs = [(3,3), (9,9), (3,9)]
    ...
    for (d1, d2) in depth_pairs:
        for ab_setting in [True, False]:
            ...
            # deterministic
            res_det = run_matches(n_matches, ai1, ai2, probabilistic=
                False)
            ...
            # probabilistic
            res_prob = run_matches(n_matches, ai1, ai2, probabilistic=
                True)
            ...
    df = pd.DataFrame(results_list)
    df.to_excel("results.xlsx", index=False)
```

3 Eksperymenty

3.1 Konfiguracja

- Liczba rozgrywek: `n_matches = 100` dla każdej pary głębokości i ustawień alpha-beta.
- Pary głębokości: $(3, 3)$, $(9, 9)$, $(3, 9)$.
- *Alpha-Beta Pruning*: włączone (ustawione `win_score=float('inf')`) lub wyłączone (`win_score=1e9`).
- Tryb gry: deterministyczny i probabilistyczny (20% szans na chybienie ruchu).

Przeprowadzono łącznie:

3 (pary głębokości) $\times$ 2 (alpha-beta on/off) $\times$ 2 (det/prob) $\times$ 100 (partii) = 1200 gier.

4 Wyniki i analiza

Wyniki zostały zapisane w pliku `results.xlsx` w postaci następujących kolumn:

- `AlphaBeta`: czy użyto alpha-beta (`AB`) czy nie (`NoAB`),
- `Depth1`, `Depth2`: głębokości przeszukiwania dla obu AI,
- `Probabilistic`: wartość `True` lub `False`,
- `P1_Wins`, `P2_Wins`, `Draws`: łączna liczba wygranych i remisów,
- `P1_Starting_Wins`, `P1_NonStarting_Wins` (analogicznie dla P2),

- `P1_AvgTime`, `P2_AvgTime`: średni czas na ruch w sekundach.

Na podstawie otrzymanych danych można sformułować następujące obserwacje:

1. **Rola głębokości:** AI przeszukujące głębiej (np. $d = 9$) ma zazwyczaj wyższą skuteczność (większy odsetek wygranych) kosztem dłuższego czasu obliczeń. Przy głębokościach typowych dla *kółka i krzyżyka* ($d = 9$ pokrywa cały stan gry) można spodziewać się braku przegranych lub bardzo nielicznych, jeśli algorytm jest poprawny i nie występują błędy.

2. **Znaczenie *Alpha-Beta Pruning*:** Dla większych głębokości widać istotne skrócenie czasu obliczeń dzięki przycinaniu alfa-beta. Szczególnie zauważalne jest to przy $d = 9$.

3. **Tryb probabilistyczny:** Wersja z 20% szansą na chybienie ruchu zwiększa prawdopodobieństwo nieoczekiwanych rozstrzygnięć. W efekcie można zaobserwować większy rozrzut wyników, a liczba remisów może rosnąć. Czas obliczeń natomiast zmienia się nieznacznie (algorytm w fazie *minimax* dalej symuluje deterministycznie).

4. **Różnica czasu przy włączonym i wyłączonym alpha-beta:** Niezależnie od trybu (deterministyczny vs. probabilistyczny) *pruning* w większości przypadków skracał czas obliczeń, co widać w kolumnach `P1_AvgTime` i `P2_AvgTime`.

5 Wnioski

Przeprowadzone testy pokazują, że:

- **Alpha-Beta Pruning** wyraźnie optymalizuje działanie algorytmu `Negamax` dla głębokich przeszukiwań, przy zachowaniu tych samych wyników jakościowych (liczby wygranych i remisów).

- **Głębokość 9** pozwala niemal zawsze na grę bezbłędną, co w tradycyjnym *kółku i krzyżyku* najczęściej kończy się remisami (od strony teorii jest to gra z wynikiem remisowym przy optymalnej grze obu stron).

- **Wprowadzenie czynnika losowości (prawdopodobieństwo chybienia ruchu)** zwiększa atrakcyjność rozgrywki i powoduje większą liczbę wygranych/przegranych, nawet dla silniejszych graczy, ponieważ perfekcyjny ruch nie zawsze zostanie poprawnie wykonany.

Podsumowując, zaimplementowana gra w *kółko i krzyżyk* z wykorzystaniem algorytmu `Negamax` i przycinania alpha-beta pozwala na przeprowadzenie licznych eksperymentów, które wykazują przewagę optymalizacji alpha-beta w zakresie czasu obliczeń. Dodatkowo wariant probabilistyczny czyni rozgrywkę bardziej zróżnicowaną i stanowi ciekawe rozszerzenie standardowej wersji gry.