Inteligencja Obliczeniowa 1

Krzysztof Ferda, Krzysztof Czerenko

March 2025

1 Gra Nim

1.1 Cel gry

Gra oparta jest na klasycznej grze Nim. Na planszy znajdują się trzy stosy o rozmiarach: 1, 3 i 5. Celem gry jest doprowadzenie do sytuacji, w której wszystkie stosy będą puste. W zależności od przyjętych reguł, gra może zakończyć się zwycięstwem gracza, który wykona ostatni ruch lub jego przegraną. W tej implementacji przyjmujemy uproszczoną wersję, gdzie gra kończy się, gdy wszystkie stosy są puste, a zwycięża gracz który nie wykonał ostatniego ruchu.

1.2 Struktura gry

Gra została zaimplementowana w Pythonie przy użyciu biblioteki `easyAI`, która ułatwia tworzenie gier dwuosobowych. Główne elementy gry to:

Klasa `Piles`
Klasa ta zarządza stanem stosów. Jej główne zadania to:

- Przechowywanie liczby elementów w poszczególnych stosach.
- Sprawdzanie, czy dany stos jest pusty.
- Pobieranie elementów ze stosu (pod warunkiem, że ruch jest legalny).
- Sprawdzanie, czy wszystkie stosy są puste (co oznacza zakończenie gry).

Klasa `Nimby`
Klasa dziedziczy po `TwoPlayerGame` z biblioteki `easyAI` i reprezentuje główną logikę gry. Odpowiada za:

- Inicjalizację gry z trzema stosami o rozmiarach $[1, 3, 5]$.
- Przechowywanie informacji o graczach i ustalanie, który gracz wykonuje ruch.
- Generowanie listy wszystkich możliwych ruchów – każdy ruch jest reprezentowany jako ciąg znaków w formacie "`<numer stosu> <liczba pobranych elementów>`".

- Wykonywanie ruchu – metoda `make_move` aktualizuje stan planszy, a w wersji probabilistycznej (jeśli jest aktywna) z 10% szansą zmienia liczbę pobranych elementów.
- Sprawdzanie, czy gra się zakończyła (czy wszystkie stosy są puste).

1.3 Mechanika gry

W każdej turze gracz wybiera jeden z dostępnych ruchów, który polega na pobraniu określonej liczby elementów z jednego ze stosów. Lista ruchów jest generowana dynamicznie na podstawie aktualnego stanu planszy. Wersja probabilistyczna gry wprowadza losowy element, w którym z 10% szansą liczba pobieranych elementów zostaje zmniejszona o 1. Gra kończy się, gdy wszystkie stosy są puste, co może służyć jako warunek zwycięstwa lub przegranej w zależności od przyjętych reguł.

2 Wyniki eksperymentów

2.1 Wyniki algorytmu Negamax w deterministycznym i probabilistycznym wariancie gry

Game Type	Depth	Player 1 Wins	Player 2 Wins
Deterministic	5	50	50
Probabilistic	5	42	58
Deterministic	13	50	50
Probabilistic	13	46	54

2.2 Podsumowanie wyników

Jak widać w tabeli powyżej przy deterministycznej wersji gry obaj gracze uzyskiwali zbliżone liczby wygranych i porażek, natomiast przy wariancie z losowością uzyskiwaliśmy różne wyniki rozgrywek.

2.3 Napotkane trudności

Największą trudnością była słaba dokumentacja biblioteki easyAI przez co część funkcji było przestarzałych lub w ogóle nie były opisane.

3 Porównanie różnych modeli AI

Algorithm	Depth	Game Type	Player 1 Wins	Player 2 Wins	Avg Time
Negamax	5	Deter	5	5	0.02048
Negamax	5	Probab	7	3	0.02637
Negamax	13	Deter	5	5	0.04946
Negamax	13	Probab	7	3	0.41604
Negamax Alpha-Beta	5	Deter	5	5	0.00446
Negamax Alpha-Beta	5	Probab	5	5	0.00560
Negamax Alpha-Beta	13	Deter	5	5	0.02393
Negamax Alpha-Beta	13	Probab	5	5	0.09899
SSS*	5	Deter	5	5	0.01127
SSS*	5	Probab	6	4	0.00789
SSS*	13	Deter	5	5	0.05945
SSS*	13	Probab	4	6	0.17755

3.1 Uzyskane wyniki

Jak widać Negamax bez ucięcia Alpha-Beta poradził sobie znacznie gorzej z wyszukiwaniem najlepszych posunięć, co pokazuje jak zoptymalizowany jest oryginalny algorytm Negamax. SSS* również poradził sobie lepiej w tym zadaniu od zwykłego Negamax, niemniej jednak gorzej niż oryginalna wersja z ucięciem.