Projekt 3 - Gry i SI

Filip Sanowski

Projektowanie i Analiza Algorytmów

`Pełny kod źródłowy, w repozytorium GitHub`

1 Wstęp

Celem projektu było napisanie gry komputerowej wykorzystującej elementy sztucznej inteligencji (SI) za pomocą algorytmu MinMax oraz z graficznym interfejsem użytkownika. Wybrałam grę Warcaby, w których gracz może zmierzyć się zarówno z drugim graczem, jak i z komputerem. Głównym celem projektu było nie tylko opracowanie funkcjonalnej i intuicyjnej gry, ale także implementacja przeciwnika sterowanego przez SI, zdolnego do rozgrywania partii na wysokim poziomie trudności.

2 Zasady Gry

Projektowana gra to klasyczne warcaby rozgrywane na planszy 8×8, zgodnie z uproszczonymi zasadami stosowanymi w wielu wersjach międzynarodowych. Gracze poruszają się naprzemiennie, a każdy z nich kontroluje swoje pionki: gracz biały rozpoczyna rozgrywkę i gra od dołu planszy, natomiast gracz czarny – od góry. Celem gry jest zbicie wszystkich pionków przeciwnika lub uniemożliwienie mu wykonania ruchu.

Poniżej przedstawiono główne reguły obowiązujące w grze:

- **Ruchy pionków:**
  Pionki poruszają się ukośnie o jedno pole do przodu – białe w górę, czarne w dół. Nie mogą poruszać się w tył, chyba że zostaną promowane na damkę.
- **Bicie:**
  Pionek może przeskoczyć ukośnie przeciwnika, jeśli za nim znajduje się wolne pole. Bicie jest obowiązkowe – jeżeli którykolwiek z pionków może wykonać bicie, gracz musi to zrobić. W przypadku, gdy istnieje możliwość wykonania kilku bić jednym pionkiem (tzw. bicie sekwencyjne lub combo), gracz musi wykonać wszystkie możliwe bicia w jednej turze tym samym pionkiem.
- **Damka:**
  Pionek, który dotrze do ostatniego rzędu przeciwnika, zostaje promowany na damkę. Damka może poruszać się o jedno pole w dowolnym kierunku (zarówno w przód, jak i w tył), co zwiększa jej możliwości ruchu oraz bicia.

- **Bicie damką:**
  Damka może również wykonywać bicie, przeskakując przeciwnika o dwa pola w dowolnym kierunku – podobnie jak zwykły pionek, ale z możliwością ruchu wstecz.

- **Zakończenie gry:**
  Gra kończy się w momencie, gdy jeden z graczy nie ma już dostępnych ruchów – albo z powodu braku pionków, albo ich zablokowania. W takiej sytuacji przeciwnik wygrywa partię.

3 Algorytm SI - MinMax

3.1 Opis działania

Algorytm **MinMax** to klasyczna metoda podejmowania decyzji w grach dwuosobowych o sumie zerowej, takich jak warcaby. Opiera się na założeniu, że gracz maksymalizujący (np. AI) próbuje osiągnąć możliwie najlepszy wynik, podczas gdy przeciwnik (gracz minimalizujący) stara się ten wynik zminimalizować. Przeglądając drzewo możliwych ruchów do określonej głębokości, algorytm wybiera ten ruch, który w najgorszym scenariuszu daje najlepszy rezultat.

Działanie algorytmu opiera się na rekurencyjnym przeszukiwaniu drzewa gry. Na przemian wywoływane są dwie funkcje:

- **maksymalizująca** – gdy tura należy do AI,
- **minimalizująca** – gdy tura przeciwnika.

Dla każdego możliwego ruchu tworzony jest nowy stan gry i oceniany rekurencyjnie. Po osiągnięciu ustalonej głębokości algorytm nie kontynuuje analizy, tylko ocenia dany stan funkcją heurystyczną (**evaluate**).

3.2 Implementacja

W kodzie projektu logika MinMax została zaimplementowana w funkcji:

```cpp
int Board::minimax(int depth, int alpha, int beta, bool maximizingPlayer, bool whiteTurn) {
    if (isGameOver(whiteTurn))
        return whiteTurn ? INT_MIN + 1 : INT_MAX - 1;

    if (depth == 0)
        return evaluate();

    auto moves = generateAllMoves(whiteTurn);
    if (moves.empty())
        return maximizingPlayer ? -10000 : 10000;

    int bestEval = maximizingPlayer ? INT_MIN : INT_MAX;

    for (auto [x1, y1, x2, y2] : moves) {
        MoveBackup backup = applyMove(x1, y1, x2, y2, whiteTurn);

        // Czy po tym ruchu AI ma kontynuować combo?
        bool continueCombo = inCombo && comboRow == x2 && comboCol == y2;

        int eval;
        if (continueCombo) {
            // Ten sam gracz gra dalej - głębokość nie spada
            eval = minimax(depth, alpha, beta, maximizingPlayer, whiteTurn);
        }
        else {
            // Następny gracz - przełącz kolor i zmniejsz głębokość
            eval = minimax(depth - 1, alpha, beta, !maximizingPlayer, !whiteTurn);
        }

        undoMove(backup);

        if (maximizingPlayer) {
            bestEval = std::max(bestEval, eval);
            alpha = std::max(alpha, eval);
        }
        else {
            bestEval = std::min(bestEval, eval);
            beta = std::min(beta, eval);
        }

        if (beta <= alpha)
            break;
    }

    return bestEval;
}
```

Rysunek 1: Implementacja algorytmu MinMax

Dla każdego ruchu możliwego w danym stanie gry:

- funkcja **applyMove(...)** symuluje ruch bez kopiowania całej planszy.
- Wywoływana jest rekurencyjnie funkcja **minimax(...)**, zmniejszając głębokość i zamieniając turę gracza.
- Po zakończeniu analizy ruch jest cofany funkcją **undoMove(...)**, przywracając planszę do stanu sprzed symulacji.

Dzięki temu osiągnięto dużą efektywność bez konieczności ciągłego kopiowania tablicy planszy.

3.3 Funkcja oceny (evaluate)

Wierzchołki drzewa gry (czyli konkretne stany planszy) są oceniane przez heurystykę **evaluate()**. Zwraca ona liczbę całkowitą wyrażającą przewagę jednego z graczy:

- **Pionek**: $+100$ dla gracza, -100 dla przeciwnika
- **Damka**: $+200$ / -200
- **Zagrożona figura** (może być zbita): kara -100

Dzięki temu algorytm może przewidzieć np. czy warto poświęcić pionka w zamian za damkę lub wykonać bicie, które prowadzi do lepszej pozycji.

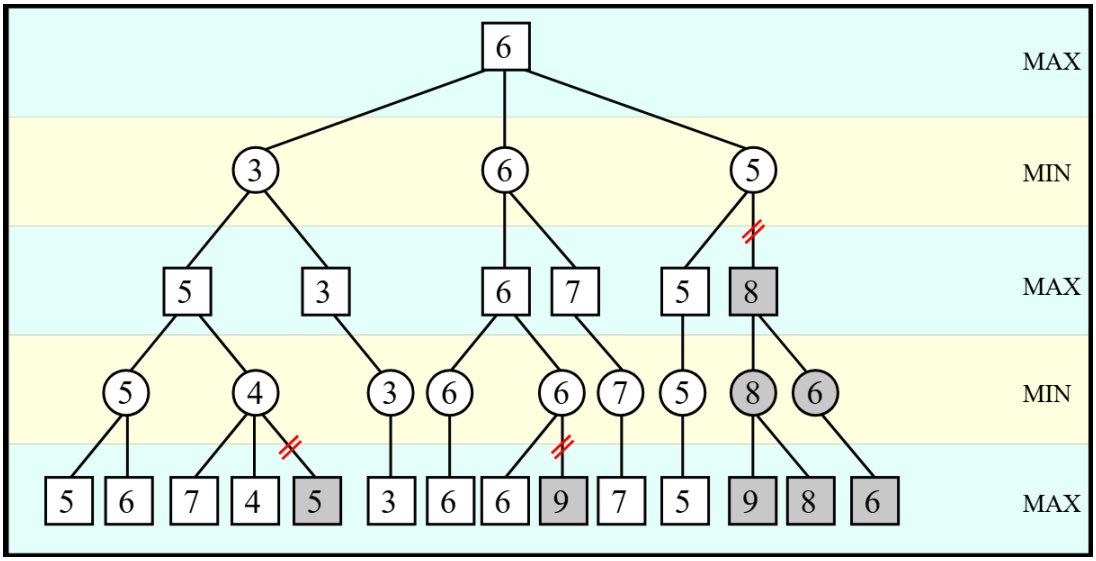
3.4 Modyfikacje – cięcia alfa-beta

Aby przyspieszyć działanie algorytmu MinMax, zastosowano klasyczną optymalizację zwaną **cięciami alfa-beta** (*alpha-beta pruning*). Jej głównym celem jest ograniczenie liczby przeszukiwanych gałęzi drzewa gry, bez wpływu na ostateczny wynik algorytmu.

Parametry cięć:

- α – najlepszy (największy) wynik, jaki może zagwarantować gracz maksymalizujący,
- β – najlepszy (najmniejszy) wynik, jaki może zagwarantować gracz minimalizujący.

W dowolnym momencie rekursji, jeśli zachodzi warunek $\alpha \geq \beta$, oznacza to, że dalsze przeszukiwanie tej gałęzi drzewa nie przyniesie lepszego wyniku i może zostać przerwane. Takie przerwanie nosi nazwę **cięcia** i jest kluczowe dla poprawy efektywności przeszukiwania.



Rysunek 2: Przykład działania algorytmu alfa-beta

Dzięki tej technice, zamiast przeszukiwać wszystkie możliwe ruchy w pełnej głębokości, AI może pominąć całe poddrzewa, które nie mają wpływu na ostateczny wybór najlepszego ruchu.

3.5 Złożoność obliczeniowa

W najgorszym przypadku, bez żadnych optymalizacji, algorytm MinMax ma złożoność:

$$O(b^d)$$

gdzie:

- b – średni *branching factor*, czyli liczba możliwych ruchów w danym stanie gry,
- d – maksymalna *głębokość drzewa gry*, czyli liczba tur w przód, które analizuje algorytm.

Zastosowanie cięć alfa-beta znacząco redukuje liczbę analizowanych pozycji. W idealnych warunkach (przy optymalnym porządku przeszukiwania), złożoność maleje do:

$$O(b^{d/2})$$

co oznacza, że algorytm może analizować znacznie większą głębokość przy tej samej liczbie operacji. W praktyce przekłada się to na możliwość stosowania głębokości rzędu 8–12 w czasie akceptowalnym dla gracza.

3.6 kod źródłowy implementacji

Link do repozytorium github

4 Podsumowanie i Wnioski

Zastosowanie algorytmu MinMax z optymalizacją alfa-beta oraz zaawansowanych metod oceny stanu gry pozwoliło na stworzenie inteligentnego przeciwnika w grze **Warcaby**. Dzięki tym technikom, SI jest stanie analizować różne scenariusze gry i podejmować mądre decyzje, co zapewnia graczowi lepsze wrażenia z gry.

4.1 Ciekawe obserwacje

- algorytm z **alfa-beta** znacznie przyśpiesza działanie programu.
- Wysoka głębokość algorytmu przeszukiwania znacznie wydłuża czas trwania ruchu SI ale równie dobrze wpływa na jego jakość.
- Drobna zmiana w zasadach (np. zakaz bicia do tyłu pionkami) znacząco wpływała na zachowanie AI – zmieniała się struktura drzewa gry i liczba kombinacji do sprawdzenia.
- Przy dużych głębokościach (10–12), nawet prosta funkcja oceny dawała bardzo dobre rezultaty. AI potrafiło przewidywać konsekwencje kilku ruchów do przodu i unikać pułapek bez potrzeby skomplikowanych heurystyk.

4.2 Trudności w realizacji projektu

- poprawne zastosowanie GUI (SFML 3.0),
- implementacja ograniczenia związanego z biciem i kombinacjami,
- optymalizacja działania programu, przy głębokości przeszukiwania powyżej 10 czas oczekiwania jest większy niż 10 sekund co psuje wrażenia z gry,
- synchronizacja AI z logiką gry (cofanie stanu planszy).