Projektowanie algorytmów i metody sztucznej inteligencji.

Projekt 3 - wersja na ocene 5.0

26.03.2023

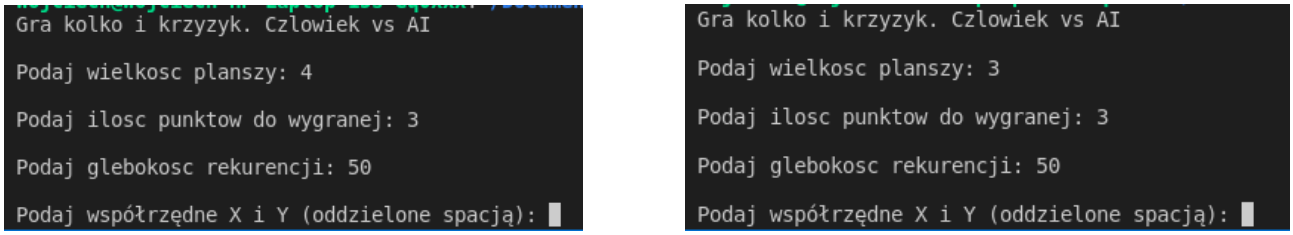
Nazwisko i imię	Wojciech Bukała
Termin zajęć:	Poniedziałek 18:55
Kod grupy projektowej	Y01-27h
Data wykonania ćwiczenia	05.06.2023
Prowadzący kurs	Dr Marek Bazan

1 Treść polecenia.

Należy zaimplementować grę w kółko i krzyżyk z wykorzystaniem algorytmu MinMax z alfa-beta cięciami. Gracz powinien posiadać możliwość definiowana rozmiaru pola (kwadratowego) wraz z ilością znaków w rzędzie.

2 Opis gry

Spośród możliwych do wyboru gier, wybrałem **kółko i krzyżyk**. Program polega na rozgrywce człowieka (użytkownika który wpisuje następne ruchy za pomocą klawiatury) przeciwko graczowi symulowanego przez komputer.
Na początku po uruchomieniu pliku wykonywalnego, gracz ustawia **wymiary planszy** (musi byc ona kwadratowa), **ilość punktów do wygranej**, **głębokość rekurencji algorytmu minimax**. Po ustawieniu parametrów gry, gracz wykonuje ruchy naprzemiennie z komputerem, aż do wygranej gracza lub komputera lub remisu, czyli sytuacji takiej, że cała plansza jest zapełniona i nikt nie spełnił warunków zwycięstwa.



Rys. 1: Przykładowe konfiguracje.

3 Algoytm minimax()

Algorytm dzięki, któremu komputer wykonuje optymalne ruchy to **minimax**. Nadaje się on do gier dwuosobowych takich jak kółko i krzyżyk lub warcaby. Funkcja **minimax()** działa rekurencyjnie i przyjmuje następujące parametry:

- **board** - referencja do obiektu typu **Board**, klasa ta została zdefiniowana w osobnym pliku i odpowiada za funkcjonalność planszy do gry.
- **depth** - zmienna przechowująca aktualną głębokość rekurancji.
- **alpha i beta** - zmienne używane do optymalizacji za pomocą alfa-beta cięć.
- **isMaximizer** - wskazuje, na to który gracz wykonuje ruch, czy jest on maksymalizowny czy minimalizowany.

Algorytm w kolejnych krokach wykonuje następujące czynności:

1. Sprawdzenie wyniku gry. Metoda minimax() zwraca, *WIN* - depth jeśli komputer wygrywa w danej konfiguracji, *LOSS* + depth jeśli komputer przegrywa w danej konfiguracji na planszy, *DRAW* - jeśli plansza jest pełna i nikt nie wygrał lub głębokość rekurencji osiągnęła swoje maksimum podane wcześniej przez użytkownika.
2. W przypadku gdy parametr *isMaximizer* przekazuje wartość *PRAWDA* to funkcja maksymalizuje szanse na wygranie.
    - Inicjalizacja najlepszego wyniku na wartość ujemną graniczną.
    - Dla każdego pustego pola na planszy symulowany jest ruch.
    - Wywołanie rekurencyjne algorytmu *minimax()* do zasymulowania ruchu przeciwnika.

- Aktualizacja najlepszego wyniku na większy z obecnego majlepszego wyniku i obecnego wyniku dla danego ruchu.
- Aktulaizacja wartości *alpha* na większą z *alpha* i najlepszego wyniku.
- Usunięcie zasymulowanego ruchu.
- Jeśli beta jest mniejsze lub równe to następuje alfa-beta odcięcie i zwracana jest wartość *alpha*.

3. W przypadku gdy parametr *isMaximizer* przekazuje wartość *FAŁSZ* to funkcja szuka najlepszego ruchu dla przeciwnika.

   - Funkcja działa bardzo podobnie jak przy maksymalizowaniu, jednak teraz robi to dla symulacji ruchu człowieka.
   - Wywołuje rekurencyjnie *minimax()* dla przeciwnika czyli w tym przypadku komputera.
   - Najlepszy wynik jest aktualizowany mniejszą wartością spośród oboecnego najlepszego wyniku i wyniku dla danego ruchu.
   - Aktualizacja wartości beta na mniejszą z wartości beta i najlepszego wyniku.
   - Usunięcie zasymulowanego ruchu z planszy.
   - Jeśli beta jest mniejsze/równe *alpha*, występuje alpha-beta odcięcie i zwraca jest wartość *beta*.

```
int Simulated_player::minimax(Board& board, int depth, int alpha, int beta, bool isMaximizer)
{
    int score = evaluate(board);
    if (score == WIN)
    {
        return score - depth;
    }

    if (score == LOSS)
    {
        return score + depth;
    }

    if (board.full() || depth == Max_Depth)
    {
        return DRAW;
    }

    if (isMaximizer)
    {
        int bestScore = -max_int;

        for (unsigned int i = 0; i < board.size; i++)
        {
            for (unsigned int j = 0; j < board.size; j++)
            {
                if (board.empty_index(i, j))
                {
                    board.insert('o', i, j);
                    int currentScore = minimax(board, depth + 1, alpha, beta, !isMaximizer);
                    bestScore = std::max(bestScore, currentScore);
                    alpha = std::max(alpha, bestScore);
                    board.insert(' ', i, j);
                    if (beta <= alpha)
                    {
                        return alpha;
                    }
                }
            }
        }

        return bestScore;
    }
```

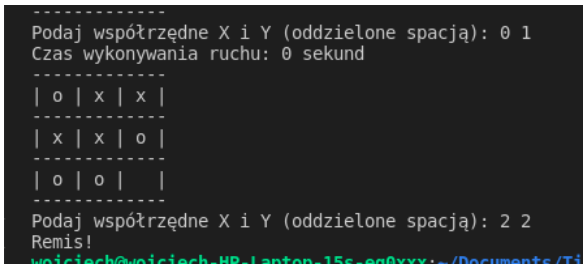
```
    else
    {
        int bestScore = max_int;

        for (unsigned int i = 0; i < board.size; i++)
        {
            for (unsigned int j = 0; j < board.size; j++)
            {
                if (board.empty_index(i, j))
                {
                    board.insert('x', i, j);
                    int currentScore = minimax(board, depth + 1, alpha, beta, !isMaximizer);
                    bestScore = std::min(bestScore, currentScore);
                    beta = std::min(beta, bestScore);
                    board.insert(' ', i, j);
                    if (beta <= alpha)
                    {
                        return beta;
                    }
                }
            }
        }

        return bestScore;
    }
}
```

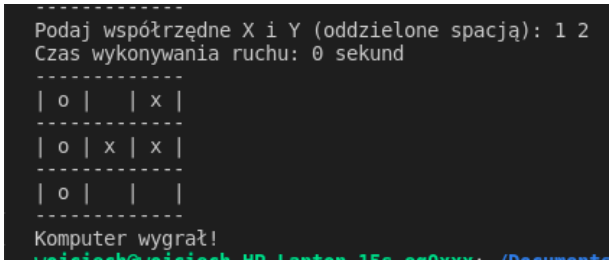
Rys. 2: Minimax

4 Optymalizacja alfa-beta cięciami

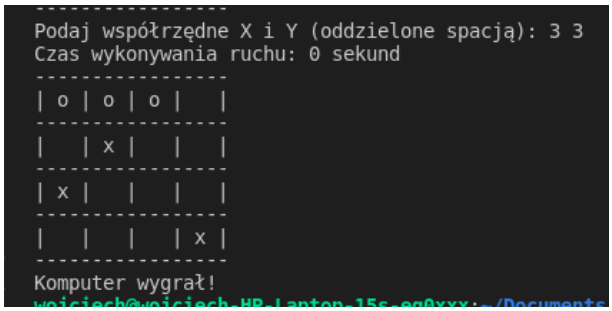
Cięcia alfa-beta to technika optymalizacji algorytmu minimax. W trakcie przeszukiwania drzewa możliwych ruchów i ich ocena, algorytm minimax utrzymuje dwie zmienne *alpha* i *beta*. W przypadku ruchu maksymalizującego *alpha* jest minimalną wartością, jaką gracz jest w stanie osiągnąć, natomiast w przypadku ruchu minimalizującego *beta* jest maksymalna wartością, jaką gracz mozę osiągnąć. Porównując na każdym razem wartości *alpha* i *beta* algorytm jest w stanie pominąć węzły podrzędne węzłu w którym wykryto iż jest nieistotny.

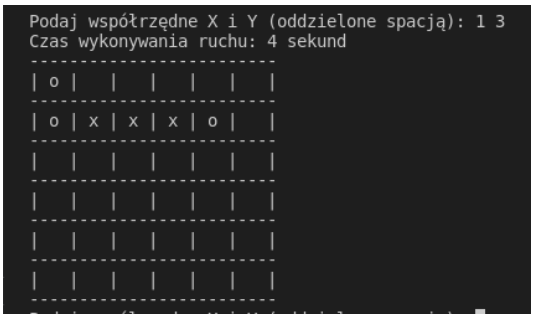
5 Testy gry.





Rys. 3: Rozgrywka 3x3





Rys. 4: Rozgrywka 4x4 i 6x6

6 Wnioski

- Czas wykonywania się algorytmu jest zależny od rozmiaru planszy, ilości punktów potrzebnych do wygranej oraz głębokości rekursji.
- Dla większych plansz (rzędu 5x5, 6x6) głebokość rekursji przy której algorytm wykonuje się płynnie wartości mniejsze od 10.
- Optymalizacja alfa-beta cięciami znacznie poprawiła czas wykonywania ruchu przez komputer.
- Dla planszy 3x3 nie ma żadnych problemów z szybkością wykonywania ruhców przez komputer. Wykonują się one natychmiastowo.
- Dla planszy 3x3 i 3 punktow do wygranej nie da się wygrać z komputerem.