Politechnika Wrocławska

Projektowanie i analiza algorytmów

Projekt 4

Freddy-Ms

Poniedziałek 17:05 - 18:45

4 czerwca 2024

1 Wstęp

Algorytmy sztucznej inteligencji odgrywają kluczową rolę w rozwiązywaniu problemów decyzyjnych, zwłaszcza w grach komputerowych. Jednym z podstawowych i najbardziej znanych algorytmów stosowanych w grach dwumianowych, takich jak kółko i krzyżyk, jest algorytm minimax. Algorytm ten umożliwia znalezienie optymalnej strategii gry, zakładając, że obaj gracze podejmują decyzje w sposób racjonalny i dążą do maksymalizacji swoich szans na wygraną. W celu zwiększenia efektywności działania algorytmu minimax, stosuje się technikę alfa-beta cięć, która znacząco redukuje liczbę analizowanych stanów gry.

2 Fragmenty kodu

Kod odpowiedzialny za interfejs graficzny użytkownika (GUI) został celowo pominięty, ponieważ nie odgrywa on kluczowej roli w analizie algorytmu. Cała implementacja gry została zrealizowana w sposób obiektowy, co umożliwia bardziej zorganizowane i modularne podejście do kodowania. Poniżej przedstawione zostaną wszystkie elementy składowe klasy odpowiedzialnej za logikę gry. Niektóre z tych elementów zostaną jedynie opisane pod kątem ich funkcji, natomiast dla innych zostanie zaprezentowana pełna implementacja kodu.

2.1 Deklaracja klasy

```
class GameGUI
{
   public:
        GameGUI(int size, int wincondition); // Constructor
        void run(); // Process events and draw
        ~GameGUI(); // Destructor
    private:
        typedef pair<int, int> bestMove;
        bool AItoMove = false;
        bool firstMove = true;
        bool running = true;
        sf::RenderWindow window; // Window from SFML
        size_t size; // Size of the board
        size_t wincondition; // Number of signs in a row,column or diagnolas to win
        size_t maxDepth; // Maximum depth for the minimax algorithm
        char** board; // Board of the game
        void EventHandler(); // Process events
        void Draw(); // Draw the board
        bool isValidMove(int x, int y); // Check if the move is valid
        bool isWin(char sign); // Check if the player with the sign has won
        bool isDraw(); // Check if the game is a draw
        void AITurn(char AIsign, char playerSign); // AI turn
        int minimax(char AIsign, char playerSign, size_t depth, bool isMaximizingPlayer
            , int alpha, int beta); // Minimax algorithm with alpha-beta pruning
};
```

Warianty zmiennych typu boolowskiego stanowią istotny element implementacji interfejsu graficznego, wpływając na sposób rozpoczęcia rozgrywki. Użytkownik ma możliwość wyboru pomiędzy dwoma modalnościami: rozpoczęcia gry przez człowieka lub przez sztuczną inteligencję. W przypadku wyboru rozpoczęcia gry przez człowieka, wystarczy dokonać zaznaczenia wybranej pozycji za pomocą lewego przycisku myszy. Natomiast, aby rozgrywkę rozpoczęła sztuczna inteligencja, wystarczy dokonać dowolnego naciśnięcia prawego przycisku myszy.

2.2 Ruch sztucznej inteligencji

```
void GameGUI::AITurn(char AIsign, char playerSign)
{
    int bestScore = INT_MIN;
    vector<bestMove>* Moves = new vector<bestMove>();

    for (size_t i = 0; i < this->size; i++)
    {
        for (size_t j = 0; j < this->size; j++)
        {
            if (isValidMove(i,j)) {
                this->board[i][j] = AIsign;
                int score = minimax(AIsign, playerSign, 0, true, INT_MAX, INT_MIN);
                this->board[i][j] = ' ';
                if (score > bestScore)
                {
                    bestScore = score;
                    Moves->clear();
                    Moves->emplace_back(i, j);
                }
                else if(score == bestScore)
                {
                    Moves->emplace_back(i, j);
                }
            }
        }
    }
    srand(time(0));
    int randomNumber = rand() % Moves->size();
    bestMove randomBestMove = (*Moves)[randomNumber];
    delete Moves;
    this->board[randomBestMove.first][randomBestMove.second] = AIsign;
}
```

Sztuczna inteligencja, w celu podejmowania optymalnych decyzji w grze, korzysta z algorytmu minimax, który analizuje potencjalne ruchy i przewiduje ich skutki na przestrzeni kolejnych etapów gry. Początkowo, implementacja komputera była deterministyczna, co oznaczało, że dla określonej sekwencji ruchów gracza, sztuczna inteligencja zawsze odpowiadała tym samym zestawem ruchów. Aby zniwelować ten efekt determinizmu i wprowadzić większą zróżnicowanie w zachowaniu komputera, zastosowano strategię przechowywania wszystkich najlepszych możliwych ruchów w wektorze danych. Na zakończenie analizy, spośród tych potencjalnych ruchów wybierany jest losowo jeden z nich. Takie postępowanie nie tylko eliminuje przewidywalność zachowania komputera, lecz także dodaje element losowości, co przyczynia się do zwiększenia atrakcyjności i dynamiki rozgrywki.

2.3 Algorytm minimax z alfa-beta cięciami

```
int GameGUI::minimax(char AIsign, char playerSign, size_t depth, bool
    isMaximizingPlayer, int alpha, int beta)
{
    if(isWin(AIsign))
    {
        return 1 + this->maxDepth - depth;
    }
    else if(isWin(playerSign))
    {
        return -1 - this->maxDepth + depth;
    }
    else if(isDraw())
    {
        return 0;
    }
    else if (depth == maxDepth)
    {
        return 0;
    }

    if (isMaximizingPlayer)
    {
        int bestScore = INT_MAX;
        for (size_t i = 0; i < this->size; i++)
        {
            for (size_t j = 0; j < this->size; j++)
            {
                if (isValidMove(i, j))
                {
                    this->board[i][j] = playerSign;
                    int score = minimax(AIsign, playerSign, depth + 1, false, alpha,
                        beta);
                    this->board[i][j] = ' ';
                    bestScore = min(bestScore, score);
                    alpha = min(alpha, bestScore);
                    if (alpha <= beta)
                        break;
                }
            }
            if (alpha <= beta)
                break;
        }
        return bestScore;
    }
    else
    {
        int bestScore = INT_MIN;
        for (size_t i = 0; i < this->size; i++)
        {
            for (size_t j = 0; j < this->size; j++)
            {
                if (isValidMove(i, j))
                {
                    this->board[i][j] = AIsign;
                    int score = minimax(AIsign, playerSign, depth + 1, true, alpha, beta
                        );
```

```
                this->board[i][j] = ' ';
                bestScore = max(bestScore, score);
                beta = max(beta, bestScore);
                if (alpha <= beta)
                    break;
            }
        }
        if (alpha <= beta)
            break;
    }
    return bestScore;
  }
}
```

Na początku funkcja przeprowadza wstępną analizę w celu ustalenia, czy nie zostało osiągnięte zwycięstwo któregoś z graczy lub remis. Jeśli żaden z tych warunków nie został spełniony, a algorytm osiągnął maksymalną dopuszczalną głębokość analizy, oznacza to, że aktualny stan planszy kończy się remisem.

Początkowo algorytm, gdy oszacował, że przegrał, rezygnował z dalszej próby obrony, co ograniczało jego zdolność do kontynuowania gry. W celu zmiany tego zachowania, funkcja oceniająca stan planszy została dostosowana tak, aby w przypadku zwycięstwa gracza zwracała sumę jedności oraz maksymalnej głębokości analizy pomniejszonej o aktualną głębokość. Natomiast, dla bota, zwraca sumę jedności wraz z aktualną głębokością analizy pomniejszoną o maksymalną głębokość. Taka adaptacja implementacji sprawia, że bot nie poddaje się w obliczu potencjalnej porażki i kontynuuje walkę aż do końca partii. Dzięki takiemu podejściu, efektywność bota została usprawniona, co przekłada się na bardziej zrównoważoną i konkurencyjną rozgrywkę.

Przedstawiony przykład ilustruje istotną zmianę w zachowaniu bota, która wynikała z poprawy procedury oceny stanu planszy. W pierwotnej wersji algorytmu, gdy bot oszacował, że jego przegrana jest nieunikniona, poddawał się bez próby obrony. Nawet w sytuacji, gdy mógłby zablokować ruch gracza, który doprowadziłby do jego zwycięstwa, bot nie podejmował działań obronnych. Efektem tego podejścia było szybkie zakończenie gry.

Po wprowadzeniu zmian w ocenie stanu planszy, bot zaczął podejmować bardziej aktywne działania w celu obrony. Zauważył możliwość blokowania ruchów gracza prowadzących do zwycięstwa, co pozwoliło mu przedłużyć grę i utrudnić osiągnięcie zwycięstwa przeciwnikowi. Warto podkreślić, że nawet jeśli gracz nie zauważyłby potencjalnie zwycięskiej sekwencji, bot teraz podejmował działania w celu zablokowania tej możliwości, co mogło prowadzić do bardziej zrównoważonej i dłuższej rozgrywki.

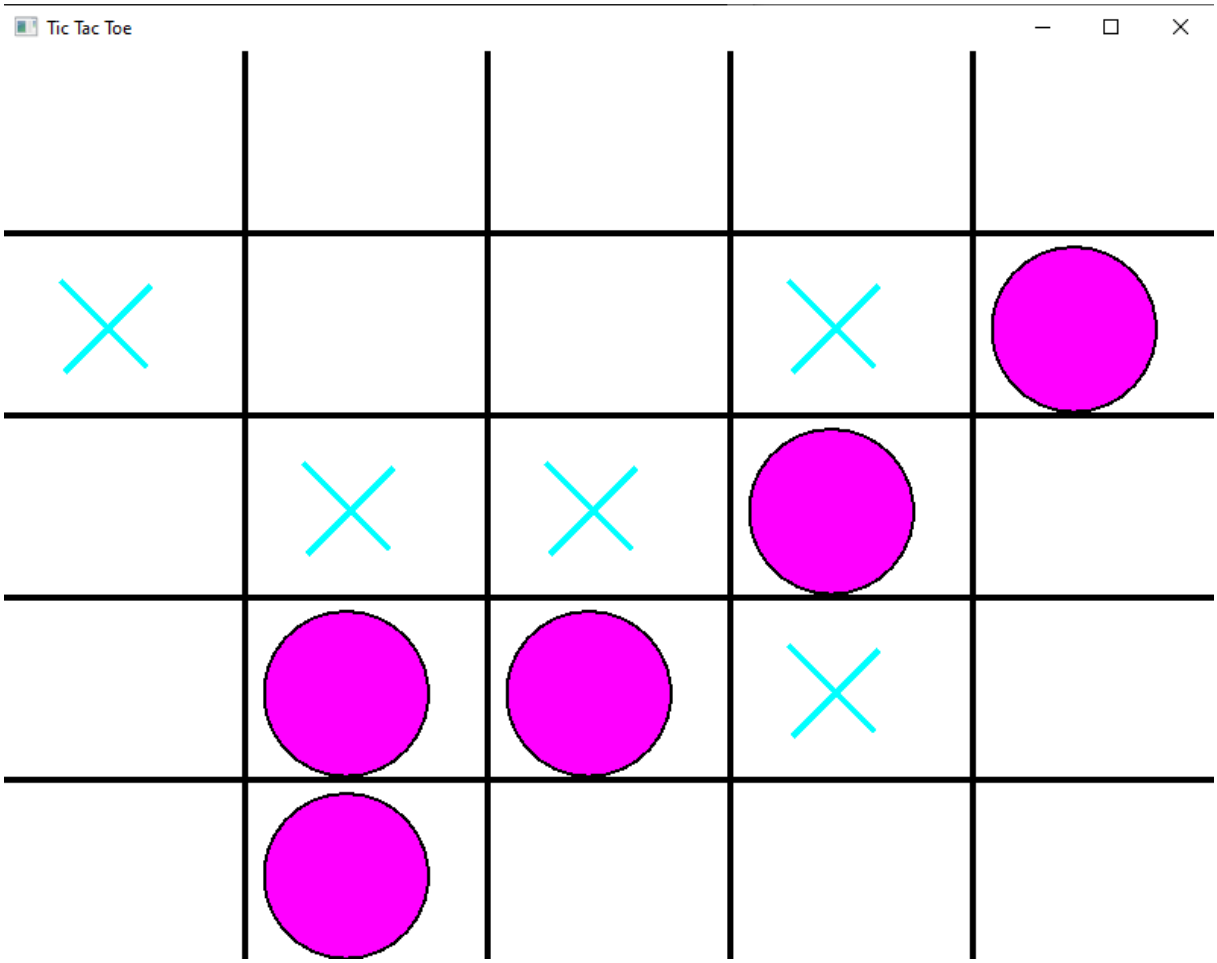
2.4 Funkcja sprawdzająca zakończenie gry oraz ocenę planszy

```
bool GameGUI::isWin(char sign)
{
    size_t rows = 0;
    size_t columns = 0;
    // Searching rows and columns
    for (size_t i = 0; i < this->size; i++)
    {
        rows = 0;
        columns = 0;
        for (size_t j = 0; j < this->size; j++)
        {
            if (this->board[i][j] == sign)
            {
                rows++;
                if (rows == this->wincondition) {
                    return true;
                }
            }
            else
            {
                rows = 0;
            }
            if (this->board[j][i] == sign)
            {
                columns++;
                if (columns == this->wincondition) {
                    return true;
                }
            }
            else
            {
                columns = 0;
            }

        }
    }
    size_t diagonal1 = 0;
    size_t diagonal2 = 0;
    // Searching diagonals
    for (size_t i = 0; i <= this->size - this->wincondition; i++) {
        for (size_t j = 0; j < this->size; j++) {
            int diagonal1 = 0;
            int diagonal2 = 0;
            for (size_t k = 0; k < this->wincondition; k++) {
                if (i + k < size && j + k < size && board[i + k][j + k] == sign) {
                    diagonal1++;
                    if (diagonal1 == this->wincondition)
                        return true;
                }
                else {
                    diagonal1 = 0;
                }

                if (i + k < size && j >= k && board[i + k][j - k] == sign) {
                    diagonal2++;
                    if (diagonal2 == this->wincondition)
```

```
                    return true;
                }
                else {
                    diagonal2 = 0;
                }
            }
        }
    }
    return false;
}
```

Funkcja oceniająca stan gry przyjmuje jako parametr wejściowy znak, który reprezentuje jednego z graczy ('X' lub 'O'). Następnie przeprowadza analizę stanu planszy, sprawdzając, czy warunek zwycięstwa został spełniony w którymś z wierszy, kolumn lub przekątnych.

3 Wygląd interfejsu graficznego



Rysunek 1: Wygląd interfejsu graficznego (GUI)

4 Wnioski

Nawet w przypadku prostszych implementacji algorytmów, które można zaklasyfikować jako sztuczną inteligencję, można zaobserwować, jak potężnym narzędziem jest ta technologia. W skończonym, relatywnie krótkim czasie komputer jest w stanie przeanalizować ogromną liczbę możliwych przyszłych stanów gry i wybrać optymalny ruch, prowadzący do zwycięstwa. Przeprowadzenie takich samych obliczeń przez człowieka zajęłoby zdecydowanie więcej czasu. Warto zauważyć, że im większa głębokość analizy wykonywanej przez sztuczną inteligencję, tym lepsza jest jej efektywność w podejmowaniu decyzji. Niemniej jednak, zwiększenie głębokości analizy ma istotny wpływ na wydajność obliczeniową algorytmu.

5 Bibliografia

- https://www.youtube.com/watch?v=SLgZhpDsrfc
- https://www.youtube.com/watch?v=l-hh51ncgDI
- https://www.youtube.com/watch?v=Wlor5pu_Ivw
- Cormen T.; Leiserson C.E.; Rivest R.L.; Stein C.: Wprowadzenie do algorytmów, WNT