INSTITUTO POLITECNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO

ALGORITMO MINIMAX Y PODA ALFA BETA

ALUMNA:

MIRELES SILVESTRE JOSELYN GUADALUPE

INTELIGENCIA ARTIFICIAL

GRUPO: 6CV2

NOMBRE DEL PROFESOR: Andrés García Floriano

Juego de Gato 4x4 con Algoritmo Minimax y Poda Alfa-Beta

1. Introducción

El presente proyecto consiste en el desarrollo de un juego de Gato (Tic-Tac-Toe) utilizando un tablero de 4x4. A diferencia del juego tradicional de 3x3, el aumento del tamaño del tablero incrementa significativamente la complejidad del problema, lo que hace necesario el uso de algoritmos de búsqueda optimizados.

Para la toma de decisiones de la computadora se implementó el algoritmo **Minimax con poda alfa-beta**, técnica ampliamente utilizada en el área de Inteligencia Artificial para juegos adversariales.

2. Objetivo

Desarrollar un juego de Gato en un tablero de 4x4 que permita enfrentar a un jugador humano contra la computadora, utilizando el algoritmo Minimax con poda alfa-beta, implementado en al menos dos lenguajes de programación.

3. Reglas del juego

- El tablero está compuesto por una matriz de 4x4.
- Participan dos jugadores:
    - Jugador humano: X
    - Computadora: O
- Gana el jugador que logre colocar cuatro símbolos iguales de manera consecutiva en:
    - Una fila
    - Una columna
    - Una diagonal
- Si todas las casillas son ocupadas y no existe un ganador, el juego termina en empate.

4. Algoritmo Minimax

El algoritmo Minimax es un método de búsqueda que analiza todas las posibles jugadas futuras del juego. Se basa en dos tipos de jugadores:

- **MAX**: intenta maximizar el valor de la jugada (computadora).
- **MIN**: intenta minimizar el valor de la jugada (jugador humano).

Cada estado del juego es evaluado con un valor numérico que representa qué tan favorable es para la computadora.

5. Poda Alfa-Beta

La poda alfa-beta es una optimización del algoritmo Minimax que permite reducir el número de nodos evaluados en el árbol de búsqueda. Utiliza dos valores:

- **Alfa (α):** el mejor valor encontrado hasta el momento para el jugador MAX.
- **Beta (β):** el mejor valor encontrado hasta el momento para el jugador MIN.

Cuando se detecta que una rama no puede mejorar el resultado actual, dicha rama es descartada, mejorando considerablemente el rendimiento.

6. Optimización para tablero 4x4

Debido al crecimiento exponencial del árbol de búsqueda en un tablero de 4x4, se implementó un **límite de profundidad** junto con una **función heurística** para evaluar estados intermedios del juego.
Esta estrategia permite conservar el uso del algoritmo Minimax con poda alfa-beta sin afectar el desempeño del programa.

7. Lenguajes de programación utilizados

- **Python:** implementación del juego en modo consola utilizando Minimax con poda alfa-beta y profundidad limitada.
- **C++:** implementación alternativa del mismo algoritmo, demostrando la portabilidad de la solución.

8. Resultados

La computadora es capaz de tomar decisiones competitivas, respondiendo de forma óptima a los movimientos del jugador humano.
Gracias a la poda alfa-beta, el tiempo de respuesta se mantiene adecuado incluso con el aumento de complejidad del tablero.

Python:

```python
import math

PLAYER = 'X'
AI = 'O'
EMPTY = ' '

MAX_DEPTH = 4

def create_board():
    return [[EMPTY for _ in range(4)] for _ in range(4)]

def print_board(board):
    print("\n")
    for row in board:
        print(" | ".join(row))
        print("-" * 13)

def check_winner(board):
    lines = []

    for i in range(4):
        lines.append(board[i])
        lines.append([board[j][i] for j in range(4)])

    lines.append([board[i][i] for i in range(4)])
    lines.append([board[i][3 - i] for i in range(4)])

    for line in lines:
        if all(cell == AI for cell in line):
            return AI
        if all(cell == PLAYER for cell in line):
            return PLAYER

    if all(board[i][j] != EMPTY for i in range(4) for j in range(4)):
        return "EMPATE"

    return None

def heuristic(board):
    score = 0
    lines = []

    for i in range(4):
        lines.append(board[i])
        lines.append([board[j][i] for j in range(4)])

    lines.append([board[i][i] for i in range(4)])
    lines.append([board[i][3 - i] for i in range(4)])

    for line in lines:
        if PLAYER not in line:
            score += line.count(AI)
        if AI not in line:
            score -= line.count(PLAYER)

    return score

def minimax(board, depth, alpha, beta, is_max):
    result = check_winner(board)
    if result == AI:
        return 100
    if result == PLAYER:
        return -100
    if result == "EMPATE":
        return 0
    if depth == MAX_DEPTH:
        return heuristic(board)

△0
```

```python
def minimax(board, depth, alpha, beta, is_max):
    if is_max:
        best = -math.inf
        for i in range(4):
            for j in range(4):
                if board[i][j] == EMPTY:
                    board[i][j] = AI
                    value = minimax(board, depth + 1, alpha, beta, False)
                    board[i][j] = EMPTY
                    best = max(best, value)
                    alpha = max(alpha, best)
                    if beta <= alpha:
                        break
        return best
    else:
        best = math.inf
        for i in range(4):
            for j in range(4):
                if board[i][j] == EMPTY:
                    board[i][j] = PLAYER
                    value = minimax(board, depth + 1, alpha, beta, True)
                    board[i][j] = EMPTY
                    best = min(best, value)
                    beta = min(beta, best)
                    if beta <= alpha:
                        break
        return best

def best_move(board):
    best_score = -math.inf
    move = None

    for i in range(4):
        for j in range(4):
            if board[i][j] == EMPTY:
                board[i][j] = AI
                score = minimax(board, 0, -math.inf, math.inf, False)
                board[i][j] = EMPTY
                if score > best_score:
                    best_score = score
                    move = (i, j)

    return move

def main():
    board = create_board()

    while True:
        print_board(board)

        r, c = map(int, input("Ingresa fila y columna (0-3): ").split())
        if board[r][c] != EMPTY:
            print("Casilla ocupada")
            continue

        board[r][c] = PLAYER

        if check_winner(board):
            break

        ai = best_move(board)
        board[ai[0]][ai[1]] = AI

        if check_winner(board):
            break

    print_board(board)
```

```python

def main():
    board = create_board()

    while True:
        print_board(board)

        r, c = map(int, input("Ingresa fila y columna (0-3): ").split())
        if board[r][c] != EMPTY:
            print("Casilla ocupada")
            continue

        board[r][c] = PLAYER

        if check_winner(board):
            break

        ai = best_move(board)
        board[ai[0]][ai[1]] = AI

        if check_winner(board):
            break

    print_board(board)
    print("Resultado:", check_winner(board))

if __name__ == "__main__":
    main()
```

```
  |   |   |
------------
  |   |   |
------------
  |   |   |
------------
  |   |   |
------------
Ingresa fila y columna (0-3): 0 0

X | O |   |
------------
  |   |   |
------------
  |   |   |
------------
  |   |   |
------------
Ingresa fila y columna (0-3): 1 0

X | O |   |
------------
X |   | O |
------------
  |   |   |
------------
  |   |   |
------------
Ingresa fila y columna (0-3): 3 0

X | O |   |
------------
X |   | O |
------------
O |   |   |
------------
X |   |   |
------------
Ingresa fila y columna (0-3): 3 3

X | O |   | O
------------
X |   | O |
------------
O |   |   |
------------
X |   |   | X
------------
```

```
------------
X |   |   | X
------------
Ingresa fila y columna (0-3): 0 3
Casilla ocupada


X | O |   | O
------------
X | O | O |
------------
O |   | X |
------------
X |   |   | X
------------
Ingresa fila y columna (0-3): 0 2


X | O | X | O
------------
X | O | O | O
------------
O |   | X |
------------
X |   |   | X
------------
Ingresa fila y columna (0-3): 1 3
Casilla ocupada


X | O | X | O
------------
X | O | O | O
------------
O |   | X |
------------
X |   |   | X
------------
Ingresa fila y columna (0-3): 2 3


X | O | X | O
------------
X | O | O | O
------------
O | O | X | X
------------
X |   |   | X
------------
Ingresa fila y columna (0-3): 3 2


X | O | X | O
------------
X | O | O | O
------------
O | O | X | X
------------
X | O | X | X
------------
Resultado: O
```

C++:

```cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

const char PLAYER = 'X';
const char AI = 'O';
const char EMPTY = ' ';
const int SIZE = 4;
const int MAX_DEPTH = 4;

vector<vector<char>> board(SIZE, vector<char>(SIZE, EMPTY));

void printBoard() {
    cout << "\n";
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            cout << board[i][j];
            if (j < SIZE - 1) cout << " | ";
        }
        cout << "\n";
        cout << "----------------\n";
    }
}

char checkWinner() {
    for (int i = 0; i < SIZE; i++) {
        if (board[i][0] != EMPTY &&
            board[i][0] == board[i][1] &&
            board[i][1] == board[i][2] &&
            board[i][2] == board[i][3])
            return board[i][0];

        if (board[0][i] != EMPTY &&
            board[0][i] == board[1][i] &&
            board[1][i] == board[2][i] &&
            board[2][i] == board[3][i])
            return board[0][i];
    }

    if (board[0][0] != EMPTY &&
        board[0][0] == board[1][1] &&
        board[1][1] == board[2][2] &&
        board[2][2] == board[3][3])
        return board[0][0];

    if (board[0][3] != EMPTY &&
        board[0][3] == board[1][2] &&
        board[1][2] == board[2][1] &&
        board[2][1] == board[3][0])
        return board[0][3];

    return ' ';
}
```

```cpp
int heuristic() {
    int score = 0;

    for (int i = 0; i < SIZE; i++) {
        int aiCount = 0, playerCount = 0;
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] == AI) aiCount++;
            if (board[i][j] == PLAYER) playerCount++;
        }
        if (playerCount == 0) score += aiCount;
        if (aiCount == 0) score -= playerCount;
    }
    return score;
}

int minimax(int depth, bool isMax, int alpha, int beta) {
    char winner = checkWinner();
    if (winner == AI) return 100 - depth;
    if (winner == PLAYER) return depth - 100;
    if (depth == MAX_DEPTH) return heuristic();

    bool movesLeft = false;
    for (auto &r : board)
        for (char c : r)
            if (c == EMPTY) movesLeft = true;

    if (!movesLeft) return 0;

    if (isMax) {
        int best = numeric_limits<int>::min();
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                if (board[i][j] == EMPTY) {
                    board[i][j] = AI;
                    int value = minimax(depth + 1, false, alpha, beta);
                    board[i][j] = EMPTY;
                    best = max(best, value);
                    alpha = max(alpha, best);
                    if (beta <= alpha) return best;
                }
            }
        }
        return best;
    } else {
        int best = numeric_limits<int>::max();
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                if (board[i][j] == EMPTY) {
                    board[i][j] = PLAYER;
                    int value = minimax(depth + 1, true, alpha, beta);
                    board[i][j] = EMPTY;
                    best = min(best, value);
                    beta = min(beta, best);
                    if (beta <= alpha) return best;
                }
            }
        }
        return best;
    }
}
```

```
    }
}

pair<int, int> bestMove() {
    int bestScore = numeric_limits<int>::min();
    pair<int, int> move = {-1, -1};

    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] == EMPTY) {
                board[i][j] = AI;
                int score = minimax(0, false,
                                    numeric_limits<int>::min(),
                                    numeric_limits<int>::max());
                board[i][j] = EMPTY;

                if (score > bestScore) {
                    bestScore = score;
                    move = {i, j};
                }
            }
        }
    }
    return move;
}

int main() {
    while (true) {
        printBoard();

        int r, c;
        cout << "Ingresa fila y columna (0-3): ";
        cin >> r >> c;

        if (r < 0 || r >= SIZE || c < 0 || c >= SIZE || board[r][c] != EMPTY) {
            cout << "Movimiento invalido\n";
            continue;
        }

        board[r][c] = PLAYER;

        if (checkWinner() != ' ') break;

        auto ai = bestMove();
        board[ai.first][ai.second] = AI;

        if (checkWinner() != ' ') break;
    }

    printBoard();
    char winner = checkWinner();
    if (winner == PLAYER) cout << "Ganaste\n";
    else if (winner == AI) cout << "Gana la computadora\n";
    else cout << "Empate\n";

    return 0;
}
```