Connect 4 - 3

Echipă

Ciucanu Eric și Carp Daniel-Cristian, grupa 1408A

Descriere

Realizarea jocului Connect 4 pentru 3 jucători utilizând algoritmul MiniMax cu eliminare Alfa-Beta.

Jocul permite participarea unui singur jucător uman, ceilalți doi fiind gestionați de algoritmi de inteligență artificială.
Algoritmul MiniMax implementează o metodă de optimizare prin tăiere Alfa-Beta pentru a reduce numărul de stări evaluate și pentru a crește eficiența deciziilor luate de AI.
Jocul se desfășoară pe o tablă standard de Connect 4, cu dimensiuni de 6 x 7 și se încheie atunci când unul dintre jucători pune 4 piese la rând (vertical, orizontal sau pe diagonale) sau când nu mai sunt spații disponibile.

Roluri

Carp Daniel-Cristian:

Implementarea interfeței grafice a jocului folosind Windows Forms.
Dezvoltarea logicii jocului care permite plasarea pieselor pe tablă, gestionarea tururilor, verificarea remizei și afișarea grafică a mutărilor.
Configurarea meniurilor și a opțiunilor de dificultate pentru algoritm.

Ciucanu Eric:

Crearea algoritmului MiniMax cu eliminare Alfa-Beta.
Implementarea funcțiilor de evaluare a stării tablei, utilizate de AI pentru a lua decizii strategice.
Dezvoltarea algoritmului de verificare a câștigătorului și integrarea acestuia cu logica generală a jocului.
Optimizarea performanței algoritmului prin reducerea adâncimii arborelui de decizie în funcție de dificultate.

Aspecte teoretice privind algoritmul

Algoritmul folosește Minimax pentru a analiza mutările posibile, alternând între maximizarea avantajului pentru AI și minimizarea avantajului adversarului, iar prin tăierea Alfa-Beta se optimizează algoritmul prin tăierea ramurilor nefolositoare.

Algoritmul suportă patru nivele de dificultate, care determină adâncimea maximă a căutării în arborele de decizie, facând AI-ul mai simplu sau mai avansat.

Algoritmul verifică ce coloane sunt încă disponibile și simulează plasarea pieselor în grilă pentru fiecare posibilitate, creând o listă de mutări valide.

Scorul unui tabel este calculat folosind EvaluateBoard, care analizează pozițiile pieselor pe tablă:

Funcția evaluează liniile orizontale, coloanele verticale și diagonalele pentru a identifica șiruri favorabile (ale AI-ului) sau nefavorabile (ale adversarului).

Se acordă scoruri pentru pattern-uri precum patru piese consecutive, trei piese și un spațiu liber, sau două piese și două spații libere.

Amenințările adversarului sunt penalizate, iar penalizarea crește odată cu dificultatea.

Dacă mai multe mutări au același scor, se alege un element aleatoriu pentru a evita ca AI-ul să fie previzibil.

Modalitatea de rezolvare

S-a implementat interfața cu Windows Forms.
Coloanele interactive sunt realizate cu ajutorul butoanelor definite în Windows Forms.
Piesele sunt realizate cu PictureBox iar poziția lor este dată de butonul reprezentativ coloanei, dar și de restul pieselor ce deja au fost puse.

```
PictureBox pictureBox1 = new PictureBox();
        pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
        Bitmap MyImage = images[board.playerTurn];
        pictureBox1.ClientSize = new Size(imgSize, imgSize);
```

```
        pictureBox1.Image = (Image)MyImage;
        pictureBox1.BackColor = Color.Transparent;
        this.Controls.Add(pictureBox1);
        Point btnPos = senderBtn.Location;
        int posX = btnPos.X + (senderBtn.Width - imgSize) / 2;
        int posY = btnPos.Y + senderBtn.Height - imgSize * (board.heights[columnNr]) - 5;
        pictureBox1.BringToFront();
        pictureBox1.Location = new Point(posX, posY);

        pictures[xJustAdded, yJustAdded] = pictureBox1;
```

Sunt stocate piesele într-o matrice cu următoarele valori:

-1: spatiu gol
0: piesă jucător
1: piesă AI 1
2: piesa AI 2

```
/// <summary>
      ///  un joc normal are 6 randuri si 7 coloane
      /// </summary>
      public int[,] grid = new int[6, 7];

      /// <summary>
      /// vector ce contine inaltimile pentru fiecare coloana
      /// </summary>
      public int[] heights = new int[7];
```

Pentru a nu itera prin toată coloana pentru a descoperi un spațiu gol, s-a utilizat un vector ce reține înălțimile coloanelor.

Ulterior, fiecare AI mută în urma calculelor realizate de algoritmul MiniMax cu tăiere Alfa-Beta. Pentru fiecare jucător este aplicată o verificare dacă a câștigat, iar dacă da, se va evidenția pe ecran ce piese compun victoria (prin backgroundul imaginii).

```
//Parte din cod ce verifică victoria pe linia orizontală și returnează pozitiile în caz de victorie
List<(int, int)> winningPiecesPositions = new List<(int, int)>();

        for (int j = 0; j < 4; j++)
        {
```

```
            for (int i = 0; i < 6; i++)
            {

                if (grid[i, j] == playerTurn && grid[i, j + 1] == playerTurn && grid[i, j + 2] ==
playerTurn && grid[i, j + 3] == playerTurn)
                {
                    winningPiecesPositions.Add((i, j));
                    winningPiecesPositions.Add((i, j + 1));
                    winningPiecesPositions.Add((i, j + 2));
                    winningPiecesPositions.Add((i, j + 3));
                    return winningPiecesPositions;
                }
            }
        }
```

```
 // Pentru evaluarea pozitiilor orizontale in algoritmul MiniMax
        for (int i = 0; i < 6; i++)
        {
            List<int> row_array = new List<int>();
            for (int j = 0; j < 7; j++)
            {
                row_array.Add(board.grid[i, j]);
            }
            for (int j = 0; j < 4; j++)
            {
                List<int> window = new List<int>();
                for (int k = j; k < j + 4; k++)
                {
                    window.Add(row_array[k]);
                }
                board.F += EvaluateLine(window, ai);
            }
        }
```

```
//Părți relevante din algoritmule MiniMax cu tăiere AlfaBeta
public static Board Minimax2L(Board table, int depth, int alpha, int beta, int ai)
    {

        if (depth == MAX_DEPTH)
        {
```

```
            EvaluateBoard(table, ai);
            return table;
        }
        else
        {
            List<Board> validMove = new List<Board>();

            for (int i = 0; i < table.heights.Length; i++) //creaza toate mutarile posibile pe care le
poate face AI actual
            {
                if (table.heights[i] != 6)
                {
                    Board newManager = new Board(table);
                    var cursorX = 5 - table.heights[i];
                    newManager.grid[cursorX, i] = table.playerTurn;
                    newManager.newPiecePos.Item1 = cursorX;
                    newManager.newPiecePos.Item2 = i;
                    newManager.heights[i]++;
                    validMove.Add(newManager);
                }
            }
            Board nextBoard = new Board();
            if (depth % 3 == 0) //Ramura este nivelul maximizant
            {
                nextBoard.F = -999;
                foreach (Board move in validMove)
                {
                    move.NextTurn();
                    var tryNextBoard = Minimax2L(move, depth + 1, alpha, beta, ai);
                    if (tryNextBoard.F > nextBoard.F) //compara scor board potential cu cel actual
                    {
                        if (depth == 0) //initializa board-ul care trebuie sa iasa din return
                            nextBoard = move;
                        nextBoard.F = tryNextBoard.F;
                    }
                    else if (tryNextBoard.F == nextBoard.F && _rand.Next(100) < 50) //alegere
aleatoarie daca exista scoruri egale
                    {
                        if (depth == 0)
                            nextBoard = move;
                        nextBoard.F = tryNextBoard.F;
                    }
                    alpha = Math.Max(alpha, nextBoard.F);
                    if (alpha >= beta)
                        break;
                }
            }
            else //Ramura este nivelul minimizant
            {
                nextBoard.F = 999;
```

Complexitatea algoritmului

Caz favorabil:

În cel mai bun caz, tăierea Alfa-Beta reduce semnificativ numărul de stări evaluate.

Complexitatea aproximativă este $O\left(b^{\frac{d}{2}}\right)$ unde:

b este factorul de ramificare (max 7)

d este adâncimea arborelui (setată la 2 + DIFFICULTY).

Caz defavorabil:

Complexitatea este $O\left(b^{d}\right)$ (complexitate brută)

Caz mediu:

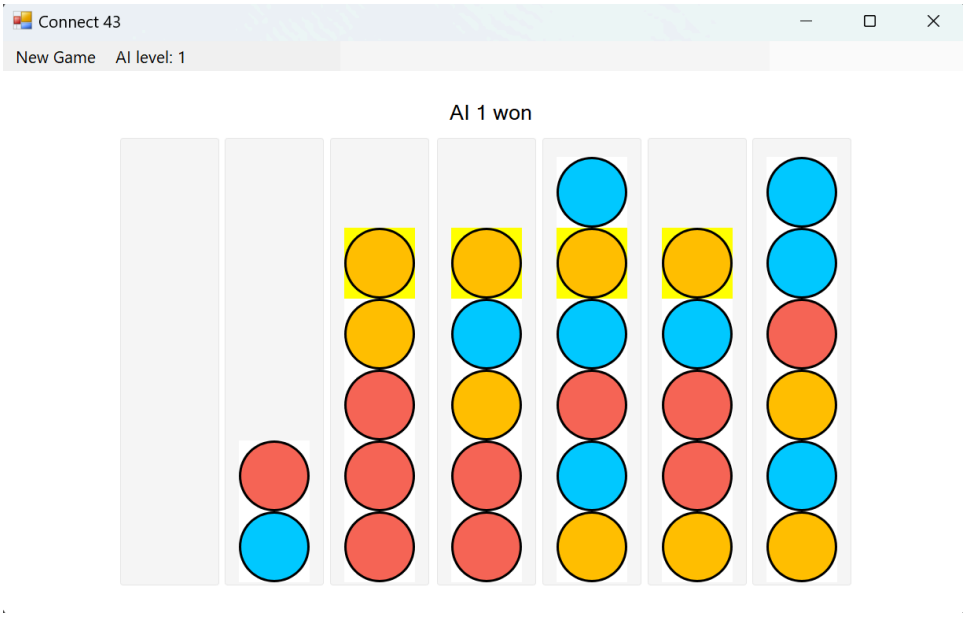
Între $O\left(b^{\frac{d}{2}}\right)$ și $O\left(b^{d}\right)$. (în mod uzual tăierea nu realizează cazul ideal)

Complexitate spațială

Stocarea stărilor: $O\left(d\right)$ (d este adâncimea maximă).

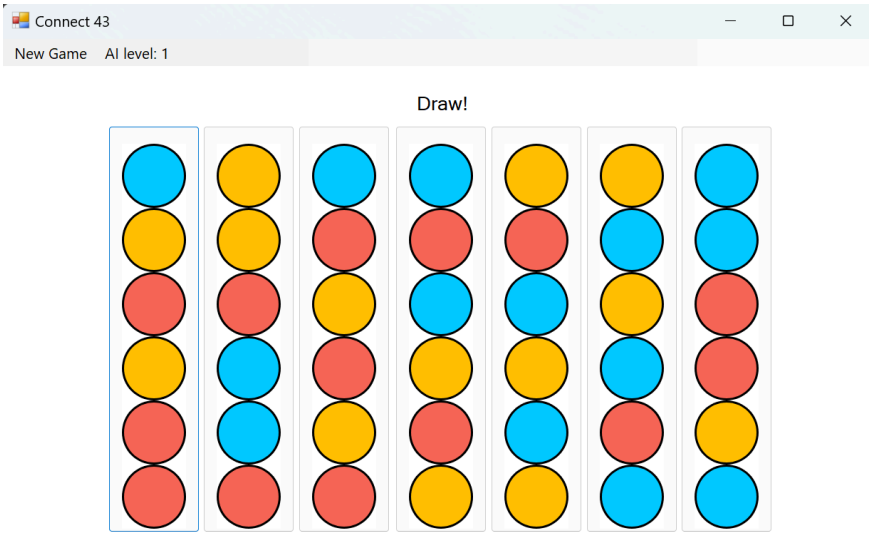
Memoria pentru tablă: $O\left(1\right)$ (tablă de 6 x 7 + vector de înălțimi de 7)

Capturi de ecran



Situație în care AI 1 câștigă

Situație în care AI 2 câștigă

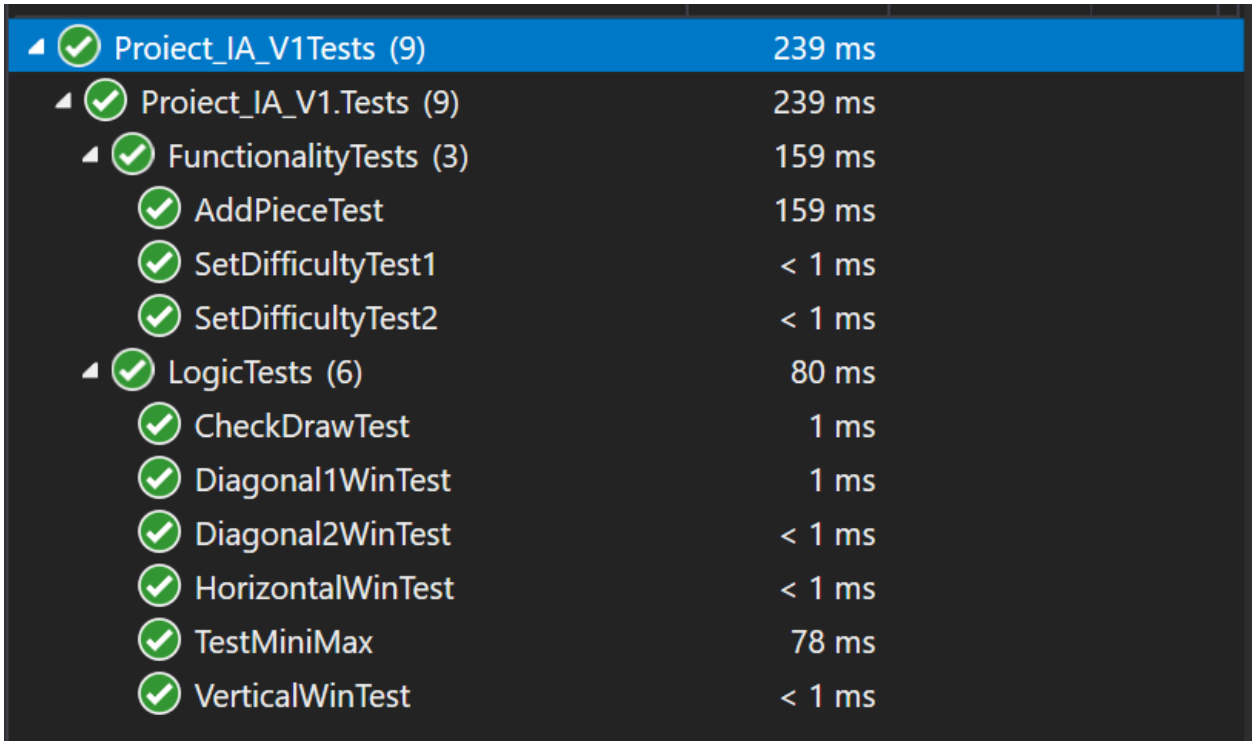


Remiză

```
. . 1 . 1 . .
. . 2 . 2 . .
. . 1 . 1 . .
. . 2 0 0 . .
. . 0 2 2 . 2
. 0 0 0 1 . 1
- - - - - - -
0 1 6 3 6 0 2 <- Heights
```

Reprezentare joc la nivel de debug

Teste



Au fost realizate o multitudine de teste pentru a verifica buna funcționare a programului.

- Funcționale, ce verifică corectitudinea metodelor de adăugare piese în zona dorită și a setării dificultăților a oponenților artificiali.
- Logice, ce verifică situațiile de câștig și corectitudinea algoritmului MiniMax.

Concluzie

Proiectul a respectat cerințele și funcționarea corectă în toate scenariile testate. Interfața grafică este simplă și intuitivă, iar algoritmul MiniMax cu eliminare Alfa-Beta asigură decizii inteligente din partea AI-urilor.

Bibliografie

Inspirație pentru algoritmul de testare și verificare a câștigătorului
https://roboticsproject.readthedocs.io/en/latest/ConnectFourAlgorithm.html