Algoritmul Minimax - Sah 5x5

Descrierea problemei considerate

Proiectul realizat abordează tema utilizării algoritmilor de inteligență artificială pentru a juca un joc cu calculatorul. Algoritmul folosit în acest caz este algoritmul minimax cu retezare alfa-beta. Jocul în cadrul căruia a fost aplicat este șah pe o tablă de dimensiune 5x5, o variantă alternativă a șahului original. Șahul 5x5 cuprinde tipurile principale de piese din șahul original, excepție făcând regina. Tabla mai mică de joc și lipsa reginei contribuie la reducerea complexității jocului, însă este suficient pentru a demonstra performanțele algoritmului.

Punctele cheie vizate de acest proiect sunt capacitatea de a evalua configurația curentă a tablei de joc și identificarea celei mai avantajoase mutări pentru calculator. Prin utilizarea retezării alfa-beta, cazurile nefezabile din arborele de decizie sunt eliminate, reducând semnificativ timpul de calcul. Algoritmul optimizează decizia calculatorului analizând mai întâi mutările cele mai promițătoare, în timp ce adversarul uman încearcă să minimizeze avantajul calculatorului.

Aspecte teoretice privind algoritmul

Retezarea alfa-beta este un algoritm de căutare utilizat pentru a optimiza procesul de evaluare a nodurilor în arborele de decizie al algoritmului minimax, specific jocurilor cu doi jucători. Scopul principal al acestei tehnici este de a reduce numărul de noduri evaluate, eliminând acele ramuri ale arborelui care nu influențează decizia finală.

Înainte de a prezenta particularitățile algoritmului optimizat se va expune conceptul algoritmului minimax pentru o înțelegere completă a funcționării. Algoritmul Minimax este un proces decizional utilizat în inteligența artificială pentru jocurile cu doi jucători. Algoritmul Minimax este folosit în special în teoria jocurilor și în jocurile pe calculator, fiind conceput pentru a minimiza pierderile posibile în cel mai rău scenariu.

În cadrul unui joc cu doi jucători, maximizatorul are obiectivul de a obține un scor cât mai mare, în timp ce minimizatorul caută să îi reducă avantajul. Evaluarea corespunzătoare se realizează prin utilizarea unei funcții de evaluare, care poate fi pozitivă sau negativă. Structura algoritmului se bazează pe un arbore în care fiecare nivel (strat) aparține alternativ părții minimizante și maximizante. În cazul de față, calculatorului îi aparține stratul maximizant, adică i se atribuie scorul cel mai favorabil, iar omul joacă pe stratul minimizant, cel nefavorabil. Algoritmul funcționează prin analizarea tuturor mutărilor posibile pentru ambii jucători, anticipând răspunsurile adversarului și selectând cea mai bună mutare pentru a asigura cel mai favorabil rezultat. Funcțional, se construiește arborele de adâncime prestabilită, și se pornește evaluarea de la frunze în sus, folosind funcția de evaluare și selectând pe fiecare nivel minimul sau maximul, după caz.

Într-un arbore de joc, fiecare nod reprezintă o posibilă stare a jocului, iar nodurile terminale sunt asociate cu un scor numeric ce indică valoarea rezultatului pentru jucătorul aflat la mutare. Algoritmul menține două valori: alfa și beta. Alfa reprezintă scorul minim garantat pentru jucătorul care maximizează (Max), iar beta reprezintă scorul maxim garantat pentru jucătorul care minimizează (Min). Inițial, alfa este setat la minus infinit, iar beta la plus infinit.

Pe măsură ce arborele este explorat, aceste valori sunt actualizate. Dacă, la un moment dat, beta devine mai mic decât alfa, ramura respectivă este retezată, deoarece nu poate influența decizia finală.

Principalul avantaj al retezării alfa-beta constă în eliminarea ramurilor inutile din arborele de căutare, permițând focalizarea pe subarborii mai promițători și explorarea unor adâncimi mai mari în același interval de timp. În condiții ideale, cu o ordonare optimă a mutărilor, complexitatea algoritmului poate fi redusă de la O(b^d) la O(b^(d/2)), unde b este factorul de ramificare, iar d este adâncimea arborelui. Aceasta înseamnă că, în loc să evalueze toate nodurile, algoritmul poate evalua doar o parte semnificativ mai mică, menținând totodată acuratețea deciziei.

Modalitatea de rezolvare

Pentru implementarea algoritmului s-a folosit ca mediu de dezvoltare Visual Studio, împreună cu limbajul C#. Interfața grafică a fost construită cu ajutorul framework-ului .NET Windows Forms.

Proiectul este structurat în mai multe componente principale:

Clasa Form1: Reprezintă interfața grafică a utilizatorului (GUI), gestionând afișarea tablei de joc și interacțiunile utilizatorului.

Clasa Board: Modelează starea jocului, incluzând dimensiunea tablei și funcționalitățile asociate cu realizarea mișcărilor.

Clasa Piece: Reprezintă fiecare piesă de pe tablă, incluzând poziția, tipul (Rege, Tura etc.) și jucătorul căruia îi aparține.

Clasa Move: Modelează o mișcare posibilă pe tablă.

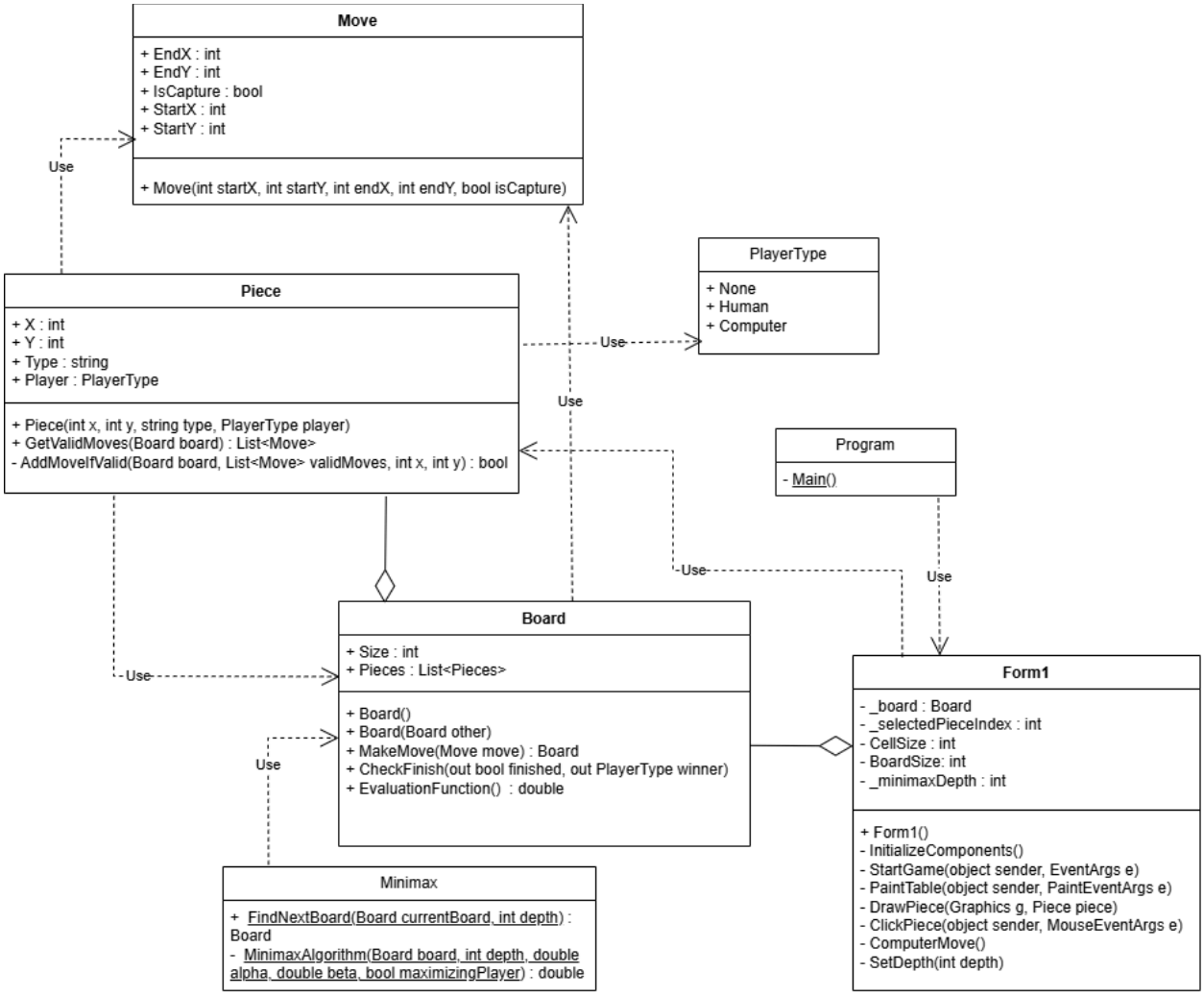
Clasa Minimax: Implementarea algoritmului Minimax cu retezare alfa-beta, folosit pentru a determina mișcarea optimă a computerului.

Enum-ul PlayerType: Definește tipurile de jucători (Om și Calculator).

Fișierul Program.cs: Punctul de intrare al aplicației.

Diagrama UML a proiectului

Mai jos se regăsește diagrama UML a proiectului.



Interacțiunea utilizatorului cu jocul se realizează prin evenimentele din Form. Evenimentul principal este cel de tip MouseClick, care permite selectarea și mutarea pieselor. Se utilizează metode de redare grafică pentru a desena piesele și tabla, cu suport pentru imagini PNG în loc de simple forme geometrice.

Meniul jocului conține opțiuni pentru începerea unui joc nou, ieșirea din aplicație și setarea dificultății. La fiecare mutare, interfața este actualizată automat, iar dacă unul dintre jucători câștigă, este afișat un mesaj corespunzător, prin MessageBox.

Regulile pentru fiecare piesă au fost implementate în clasa Piece.cs, folosind metoda GetValidMoves(). Aceasta verifică mutările valide pe baza poziției curente și a regulilor specifice fiecărui tip de piesă.

Pentru a evita mutările nevalide, jocul verifică:

- Dacă o piesă încearcă să se deplaseze în afara tablei.

- Dacă o piesă încearcă să se mute pe un loc deja ocupat de o piesă proprie.
- Dacă un pion poate captura o piesă adversă în diagonală.

Algoritmul Minimax a fost utilizat pentru a permite calculatorului să analizeze mai multe mutări în avans și să aleagă cea mai bună opțiune. Numărul de niveluri acceptate este între 1 și 3, fiind la latitudinea jucătorului să aleagă nivelul în funcție de dificultatea dorită. Deoarece evaluarea tuturor mutărilor posibile ar fi ineficientă, s-a aplicat retezarea alfa-beta, care reduce numărul de stări analizate eliminând ramurile irelevante.

Algoritmul este implementat în cadrul clasei Minimax.cs, în cadrul funcției FindNextBoard().

Pașii algoritmului sunt:

1. Se evaluează configurația curentă a tablei folosind o funcție de evaluare statică.
2. Se generează toate mutările posibile pentru jucătorul curent.
3. Se aplică fiecare mutare și se apelează recursiv algoritmul pentru adversar.
4. Se actualizează valorile alfa și beta pentru a evita explorarea inutilă a unor ramuri din arborele de decizie.
5. Se selectează mutarea cu cel mai bun rezultat posibil pentru calculator.

Funcția de evaluare atribuie scoruri diferite pieselor în funcție de importanța lor (Rege = 100, Turn = 5, Nebun/Cal = 3, Pion = 1) și ține cont de poziția acestora pe tablă în calcularea scorului.

Părțile semnificative din codul sursă

Clasa Minimax - funcția MinimaxAlgorithm

```
 private static double MinimaxAlgorithm(Board board, int depth, double alpha, double beta, bool maximizingPlayer)
 {
    bool finished;
    PlayerType winner;
    board.CheckFinish(out finished, out winner);

    if (depth == 0 || finished)
    {
       if (finished)
       {
          return winner == PlayerType.Computer ? double.MaxValue : double.MinValue;
       }

       return board.EvaluationFunction();
    }
```

```
    if (maximizingPlayer)
    {
        double maxEval = double.MinValue;

        foreach (Piece piece in board.Pieces.Where(p => p.Player == PlayerType.Computer))
        {
            foreach (Move move in piece.GetValidMoves(board))
            {
                Board nextBoard = board.MakeMove(move);
                double eval = MinimaxAlgorithm(nextBoard, depth - 1, alpha, beta, false);
                maxEval = Math.Max(maxEval, eval);
                alpha = Math.Max(alpha, eval);
                if (alpha >= beta)
                    break;
            }
        }

        return maxEval;
    }
    else
    {
        double minEval = double.MaxValue;

        foreach (Piece piece in board.Pieces.Where(p => p.Player == PlayerType.Human))
        {
            foreach (Move move in piece.GetValidMoves(board))
            {
                Board nextBoard = board.MakeMove(move);
                double eval = MinimaxAlgorithm(nextBoard, depth - 1, alpha, beta, true);
                minEval = Math.Min(minEval, eval);
                beta = Math.Min(beta, eval);
                if (beta <= alpha)
                    break;
            }
        }

        return minEval;
    }
}
```

Pentru nodurile de pe ultimul nivel/nodurile terminale se calculează scorul. Pentru celelalte noduri, se iterează prin toate piesele configurației curente a tablei, și pentru fiecare se găsește mulțimea de mutări valide care se pot face, determinând nodurile fiu. Se aplică algoritmul recursiv pe nodurile fiu. Se actualizează alpha/beta. Valorile alpha (max) și beta (min) se inițializează cu minim și, respectiv, maxim. Dacă alpha depășește beta, se retezește ramura.

Clasa Board - funcția EvaluationFunction

```
public double EvaluationFunction()
{
   var pieceValues = new Dictionary<string, double>
         {
                  { "Pawn", 1.0 },
                  { "Knight", 3.0 },
                  { "Bishop", 3.0 },
                  { "Rook", 5.0 },
                  { "King", 100.0 }
         };

   double score = 0.0;

   foreach (Piece piece in Pieces)
   {
      double pieceValue = pieceValues.ContainsKey(piece.Type) ? pieceValues[piece.Type] : 0.0;

      if (piece.Player == PlayerType.Computer)
      {
         score += pieceValue;

         // Encourage central positions
         score += 0.1 * (Math.Abs(Size / 2 - piece.X) + Math.Abs(Size / 2 - piece.Y));
      }
      else if (piece.Player == PlayerType.Human)
      {
         score -= pieceValue;

         // Penalize central positions for opponent
         score -= 0.1 * (Math.Abs(Size / 2 - piece.X) + Math.Abs(Size / 2 - piece.Y));
      }
   }

   return score;
}
```

În această funcție se calculează scorul pentru o tablă de joc dată. Regele are cel mai mare scor, întrucât capturarea sa e fatală.

Clasa Piece - funcțiile GetValidMoves și AddMoveIfValid

```
public List<Move> GetValidMoves(Board board)
{
```

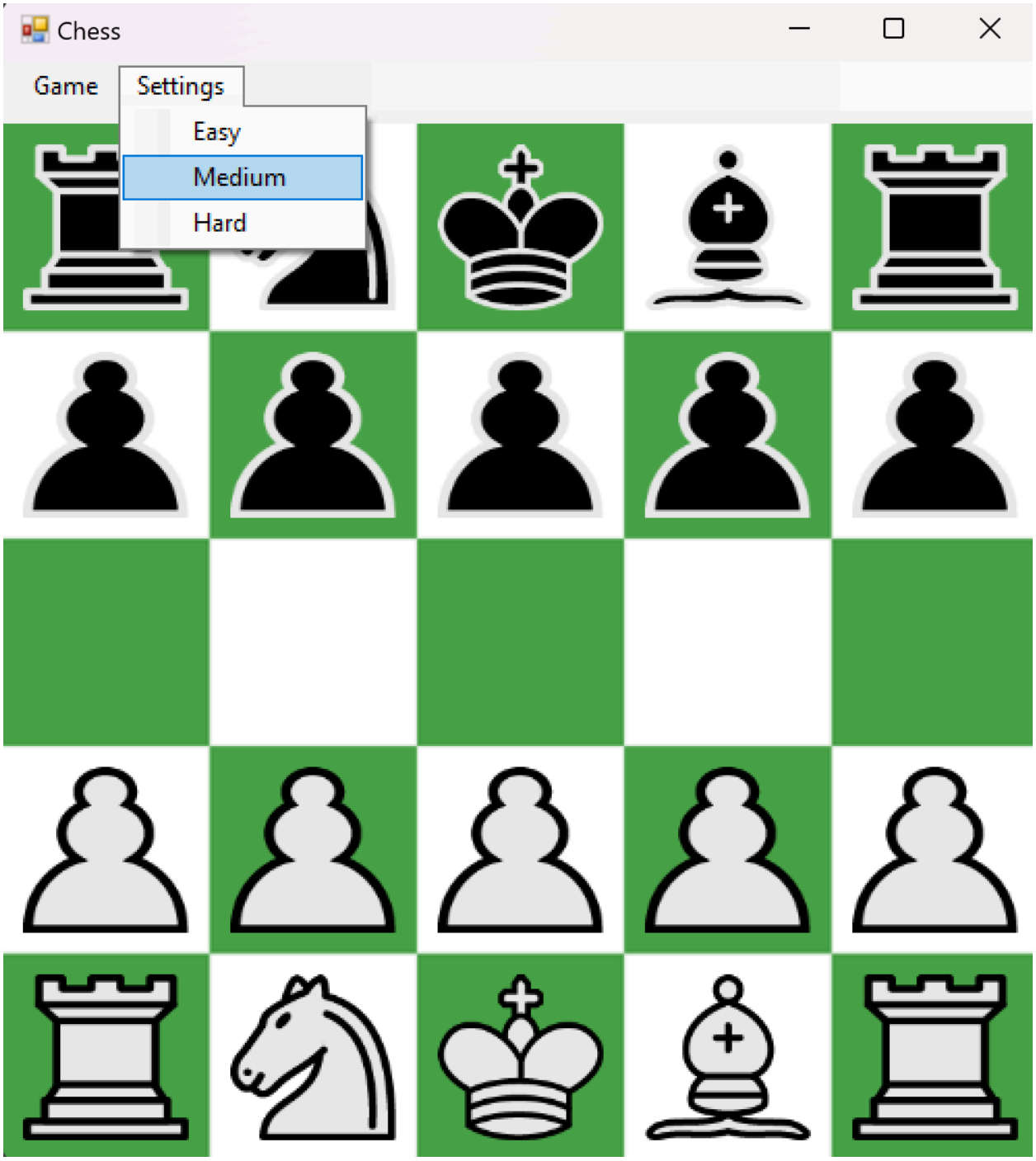
```
        List<Move> validMoves = new List<Move>();
        bool[] directions = {true, true, true, true};

        switch (Type)
        {
                case "Pawn":
                        int direction = Player == PlayerType.Human ? -1 : 1;
                        int newY = Y + direction;
                        if (newY >= 0 && newY < board.Size)
                        {
                                if (!board.Pieces.Any(p => p.X == X && p.Y == newY))
                                        validMoves.Add(new Move(X, Y, X, newY, false));

                                if (X - 1 >= 0 && board.Pieces.Any(p => p.X == X - 1 && p.Y
== newY && p.Player != Player))
                                        validMoves.Add(new Move(X, Y, X - 1, newY, true));

                                if (X + 1 < board.Size && board.Pieces.Any(p => p.X == X + 1
&& p.Y == newY && p.Player != Player))
                                        validMoves.Add(new Move(X, Y, X + 1, newY, true));
                        }
                        break;

                case "Rook":
                        for (int i = 1; i < board.Size; i++)
                        {
                                if (directions[0])
                                        directions[0] = AddMoveIfValid(board, validMoves, X + i,
Y);
           if (directions[1])
              directions[1] = AddMoveIfValid(board, validMoves, X - i, Y);
           if (directions[2])
              directions[2] = AddMoveIfValid(board, validMoves, X, Y + i);
           if (directions[3])
              directions[3] = AddMoveIfValid(board, validMoves, X, Y - i);
                        }
                        break;

                case "Bishop":

                        for (int i = 1; i < board.Size; i++)
                        {
           if (directions[0])
              directions[0] = AddMoveIfValid(board, validMoves, X + i, Y + i);
           if (directions[1])
              directions[1] = AddMoveIfValid(board, validMoves, X - i, Y + i);
```

```
            if (directions[2])
               directions[2] = AddMoveIfValid(board, validMoves, X + i, Y - i);
            if (directions[3])
               directions[3] = AddMoveIfValid(board, validMoves, X - i, Y - i);
                        }
                        break;

                case "King":
                        for (int dx = -1; dx <= 1; dx++)
                                for (int dy = -1; dy <= 1; dy++)
                                {
                                        if (dx == 0 && dy == 0) continue;
                                        AddMoveIfValid(board, validMoves, X + dx, Y + dy);
                                }
                        break;

                case "Knight":
                        var knightMoves = new List<(int dx, int dy)>
                        {
                                (2, 1), (1, 2), (-1, 2), (-2, 1), (-2, -1), (-1, -2), (1, -2), (2, -1)
                        };

                        foreach (var move in knightMoves)
                        {
                                AddMoveIfValid(board, validMoves, X + move.dx, Y + move.dy);
                        }
                        break;
        }

        return validMoves;
}

private bool AddMoveIfValid(Board board, List<Move> validMoves, int x, int y)
{
        if (x >= 0 && x < board.Size && y >= 0 && y < board.Size)
        {
      var pieceAtPosition = board.Pieces.FirstOrDefault(p => p.X == x && p.Y == y);
                if (pieceAtPosition == null)
                {
                        validMoves.Add(new Move(X, Y, x, y, false));
                        return true;
                }
                else if (pieceAtPosition.Player != Player)
                {
                        validMoves.Add(new Move(X, Y, x, y, true));
                        return false;
```

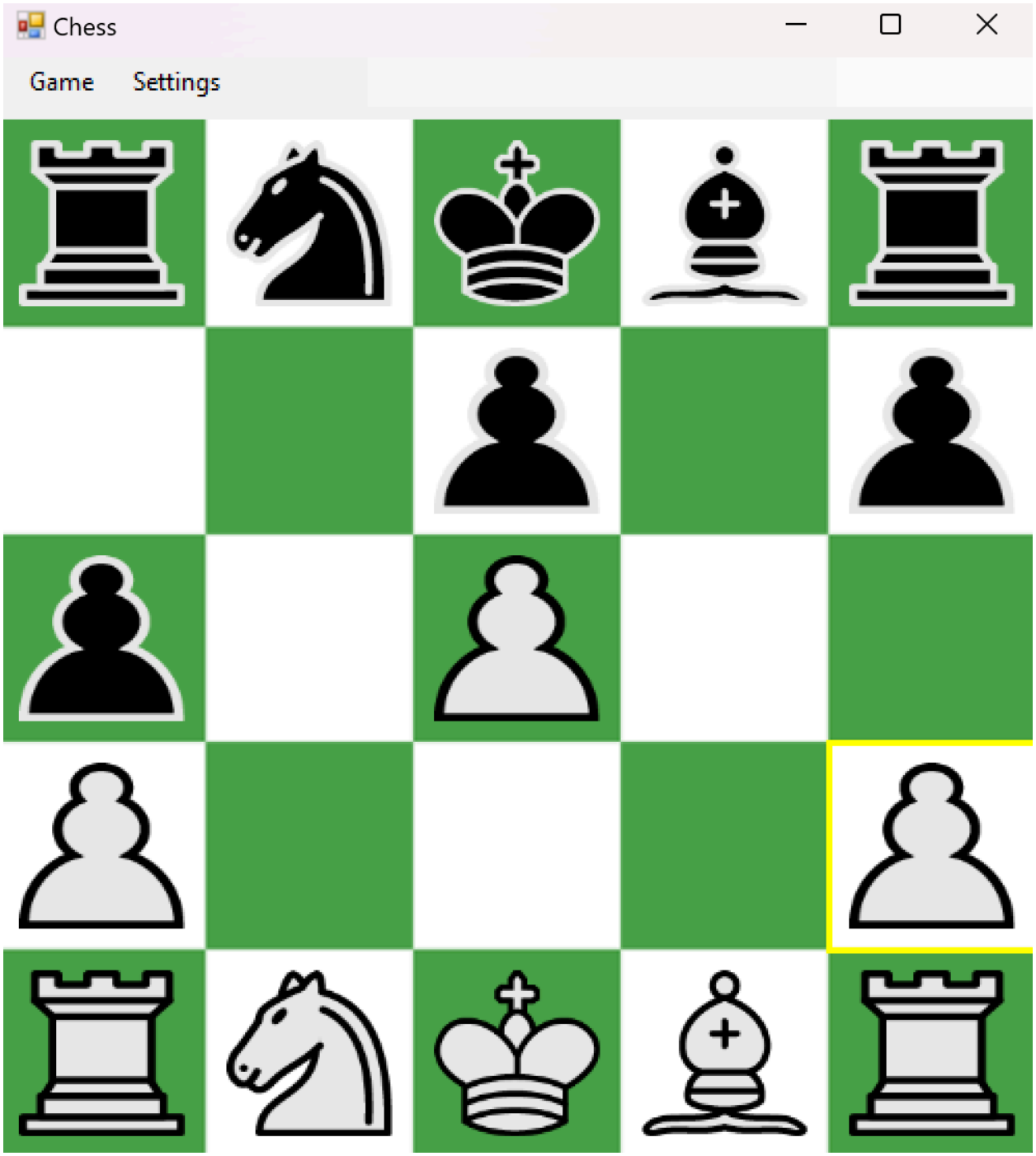
```
                }
    }
        return false;
}
```

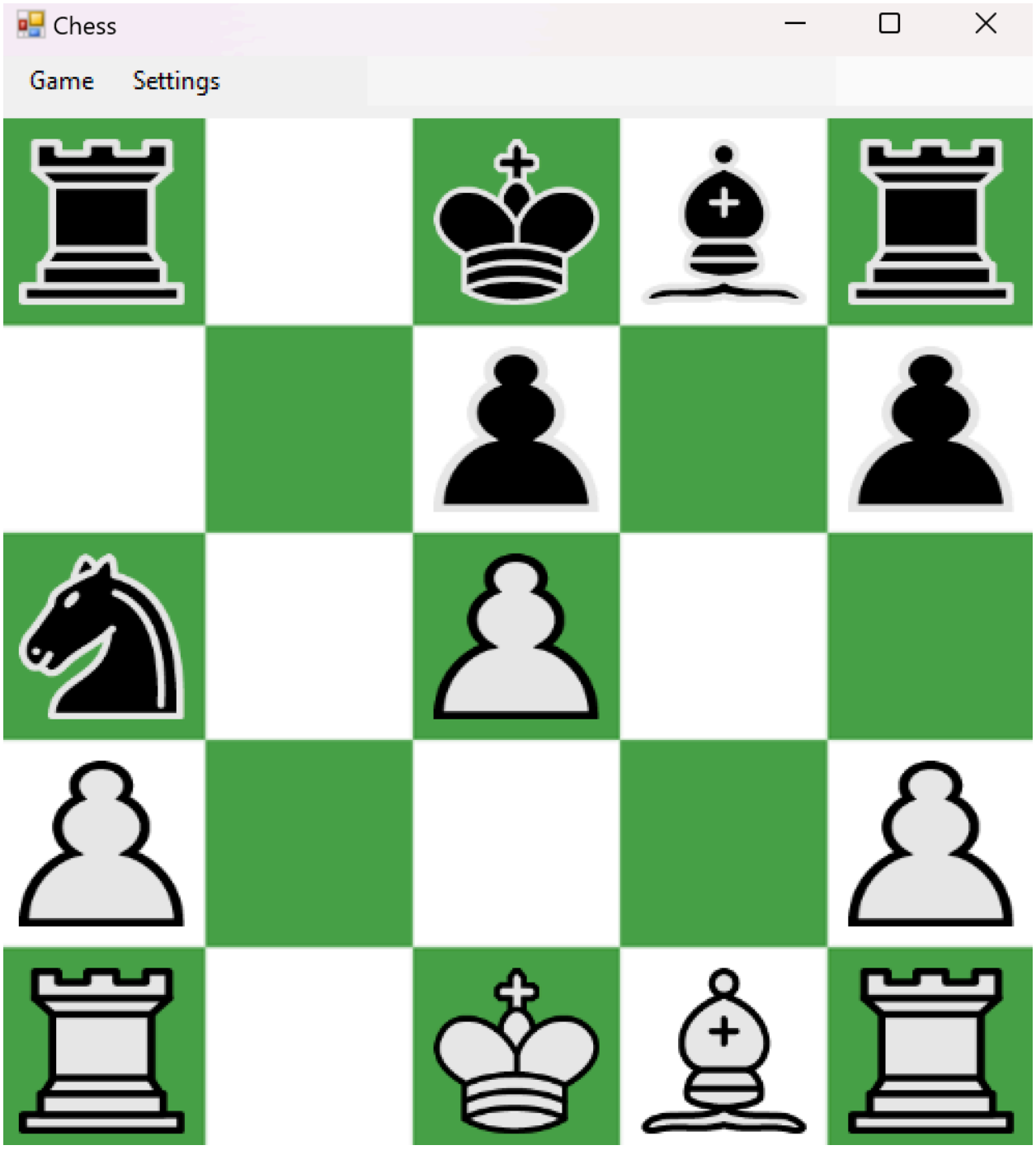
Aceste funcții se ocupă cu verificarea mișcării pieselor. În cadrul switch-ului se iterează prin mulțimea de deplasări posibile în funcție de tipul piesei și se aleg cele valide. Noțiunea de valid implică faptul că poziția se află în cadrul tablei și că nu există o piesă aliată pe acea poziție. În cazul în care poziția e validată, aceasta se adaugă la lista de mișcări valide care se returnează.

Rezultatele obținute

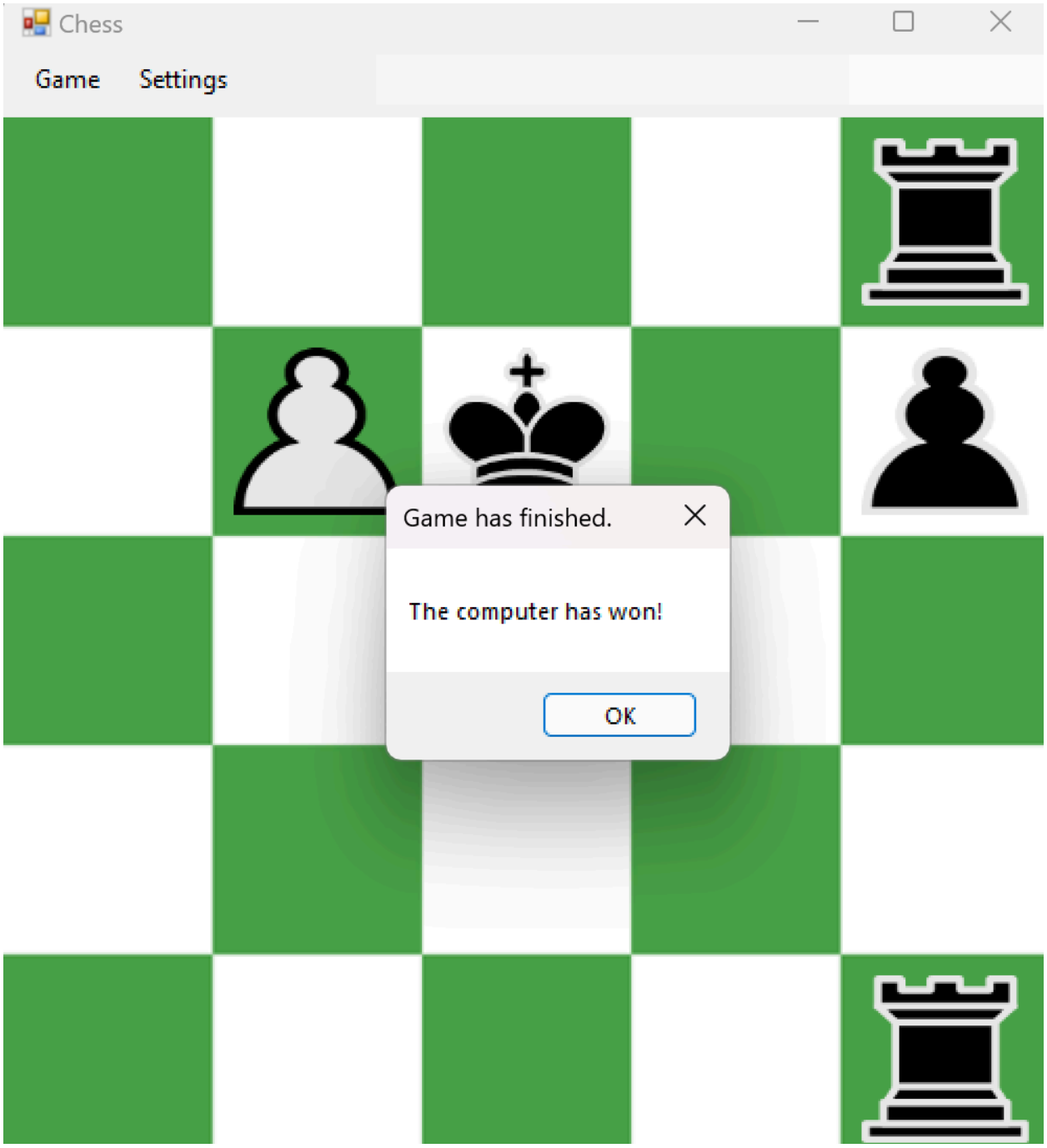


Meniul de setare a dificultății


Chess
Game
Settings



Un joc în desfășurare - mutarea omului și a calculatorului



Joc terminat - a câștigat calculatorul

Concluzii

În concluzie, proiectul realizat demonstrează aplicabilitatea algoritmului Minimax cu retezare alfa-beta într-un joc de șah 5x5. Implementarea acestui algoritm a permis calculatorului să ia decizii strategice optimizate, reducând timpul de calcul prin eliminarea mutărilor nefezabile din arborele de decizie.

Bibliografie

1. Inteligență artificială, Laborator 7, profesor Florin Leon
2. Alpha-beta pruning, https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
3. Minimax algorithm and alpha-beta pruning, https://medium.com/@aaronbrennan.brennan/minimax-algorithm-and-alpha-beta-pruning-646beb01566c

Atribuțiile membrilor

Baroncea Andrei-Florin
- Implementare clase pentru algoritmul Minimax și logica de captură a pieselor
- Capitolele rezultatele obținute prin rularea programului în diverse situații, capturi ecran și comentarii asupra rezultatelor obținute, descrierea problemei considerate, concluzii din documentație

Kerestely Alexandra-Nicola
- Implementare clase pentru tabla, piese, miscarea pieselor
- Capitolele modalitatea de rezolvare; listarea părților semnificative din codul sursă însoțite de explicații și comentarii, aspecte teoretice privind algoritmul, bibliografie din documentație