Universitatea Tehnică “Gheorghe Asachi” din Iași
Facultatea de Automatică și Calculatoare
Domeniul Calculatoare și Tehnologia Informației
Specializarea Tehnologia Informației

INTELIGENȚĂ ARTIFICIALĂ

"Connect 4"

Utilizarea algoritmului minimax cu retezarea alfa-beta în dezvoltarea unui joc om vs calculator

Coordonator,
Asist. drd. ing.
Codruţ-Georgian Artene

Studenți,

Cîrjă Ioan, 1409B
Cazamir Andrei, 1409B

An universitar 2024-2025

Cuprins

1. Descrierea problemei considerate

Connect Four este un joc strategic pentru doi jucători, în care fiecare participant selectează o culoare și plasează alternativ discuri într-o grilă verticală de o dimensiune specificată. Obiectivul este de a forma o linie de patru discuri de aceeași culoare, dispuse pe verticală, orizontală sau diagonală, înaintea adversarului.

În cadrul acestui proiect, dorim să implementăm o versiune om vs calculator, în care inteligența deciziilor calculatorului va fi bazată pe algoritmul minimax optimizat prin **retezarea alfa-beta.**

2. Aspecte teoretice privind algoritmul

2.1. Algoritmul minimax

Algoritmului minimax este un algoritm recursiv pentru a găsi cea mai bună mișcare într-o situație dată. Algoritmul minimax constă dintr-o funcție de evaluare poziționlă care măsoară bunătatea unei poziții (sau a stării de joc) și indică cât de dorit este ca jucătorul dat să atingă acea poziție; jucătorul face apoi mișcarea care minimizează valoarea celei mai bune poziții atinse de celălalt jucător.

Cei doi jucători sunt numiți maximizant și minimizant. Maximizantul încearcă să obțină cel mai mare scor posibil, în timp ce minimizantul încearcă să obțină cel mai mic scor posibil. Dacă asociem fiecărei table de joc un scor de evaluare, atunci unul din jucători încearcă să aleagă o mutare care să îi maximizeze scorul, iar celălalt alege o mutare care are un scor minim, încercând să contra-atace.

Vom considera jocul ca fiind de sumă nulă, ceea ce înseamnă că aceeași funcție de evaluare poate fi aplicată ambilor jucători.

- Dacă $f(n)>0$, poziția n este avantajoasă pentru calculator și nefavorabilă pentru om.
- Dacă $f(n)<0$, poziția n este dezavantajoasă pentru calculator și favorabilă pentru om.

Astfel, se calculează funcția de evaluare pentru frunze și se propagă evaluarea în sus, selectând minimele pe nivelul minimizant (decizia omului) și maximele pe nivelul maximizant (decizia calculatorului).

2.2. Retezarea alfa-beta

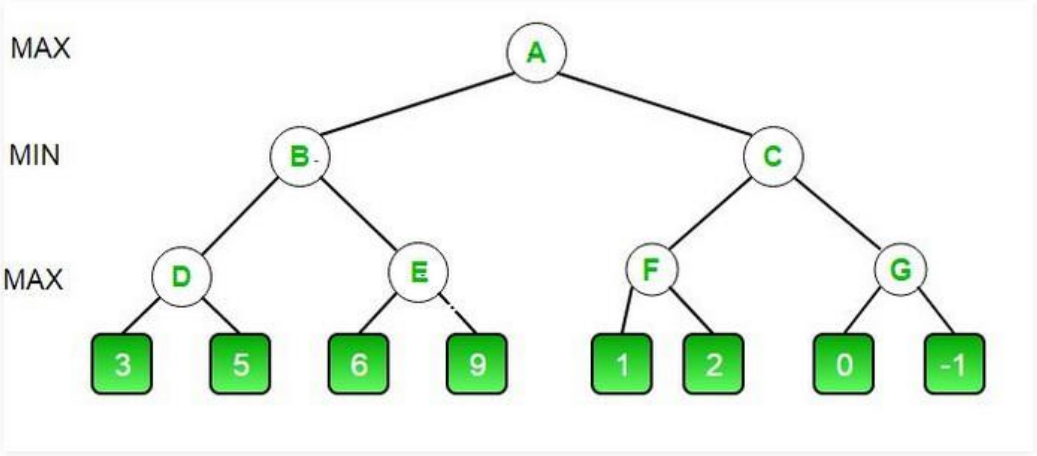
Algoritmul de tăiere alfa-beta îmbunătățește Minimax eliminând din analiză subarborii inutili. Dacă o mutare m este mai slabă decât cea mai bună mutare curentă, restul variantelor sale nu mai sunt evaluate. Parametrul alfa reprezintă cea mai bună valoare garantată pentru maximizator, iar beta pentru minimizator. Pe măsură ce arborele este parcurs, alfa și beta se actualizează, iar ramurile care nu pot îmbunătăți rezultatul sunt ignorate, economisind timp și resurse

3. Modalitatea de rezolvare

Acest proiect implementează algoritmul minimax pentru un joc între un jucător uman și un computer. Jucătorul uman maximizează scorul, iar computerul îl minimizează. Scorurile fiecărei poziții sunt ajustate în funcție de starea curentă a jocului pentru ca algoritmul să funcționeze eficient. Se va prezenta o porțiune din logica algoritmului în cele ce urmează.

Definim următoarele variabile:

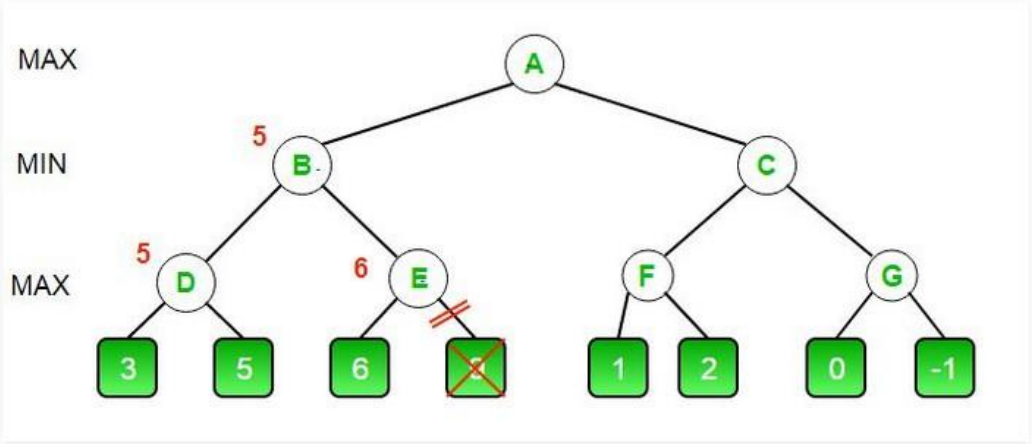
- alfa: este cea mai bună valoare pe care maximizantul o poate garanta la nivelul curent sau la nivelul următor;
- beta: este cea mai bună valoare pe care minimizantul o poate garanta la nivelul curent sau la nivelul următor.



Arborele inițial

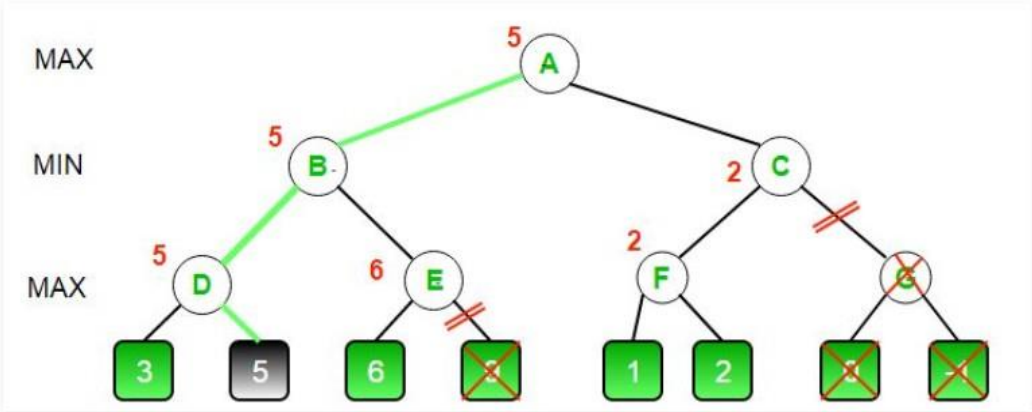
În continuare vor fi prezentați o serie de pași pe care algoritmul îi va efectua în calcularea valorilor nodurilor E, D, B:

1. **Apelul începe**: La A, alfa = -INF și beta = +INF.
2. **Maximizatorul alege B**: Maximizatorul alege nodul B.
3. **B alege D**: B explorează nodul D.
4. **Evaluarea lui D**: D verifică copilul drept (valoare 3). Alfa devine 3. Se continuă căutarea (beta > alfa).
5. **Evaluarea altui copil al lui D**: D verifică copilul drept (valoare 5). Alfa devine 5 și returnează 5 la B.
6. **Actualizarea lui beta la B**: Beta devine 5 la B.
7. **B explorează E**: B verifică nodul E (alfa = -INF, beta = 5).
8. **Evaluarea lui E**: E verifică copilul stâng (valoare 6). Alfa devine 6. Condiția beta <= alfa se îndeplinește, căutarea în E este oprită.
9. **Returnarea valorii de la E**: E returnează 6 la B.
10. **Actualizarea valorii lui beta la B**: Beta rămâne 5. Valoarea finală a lui B este 5.



Arborele după calcularea valorilor pentru nodurile E, D și B

În continuare se respectă aceeași pași pentru calcularea valorilor pentru fiecare nod până se ajunge la următoarea structură arborescentă.



Arborele final

4. Exemple semnificative de implementare

4.1. Funcția de evaluare

```python
def evaluate(tabla, piece):
    score = 0
    opponent_piece = 3 - piece
    center_col = COLOANE // 2

    # Evaluăm centrul tablei, acordând un scor mai mare pentru piesele din centrul coloanei.
    center_count = sum([1 for r in range(RANDURI) if tabla[r][center_col] == piece])
    score += center_count * 3

    # Evaluăm fiecare fereastră orizontală de 4 coloane.
    for r in range(RANDURI):
        for c in range(COLOANE - 3):
            window = [tabla[r][c + i] for i in range(4)]
            score += scoreWindow(window, piece)

    # Evaluăm fiecare fereastră verticală de 4 rânduri.
    for c in range(COLOANE):
        for r in range(RANDURI - 3):
            window = [tabla[r + i][c] for i in range(4)]
            score += scoreWindow(window, piece)

    # Evaluăm feronstrele diagonale (de sus în jos și de jos în sus).
    for r in range(RANDURI - 3):
        for c in range(COLOANE - 3):
            window = [tabla[r + i][c + i] for i in range(4)]
            score += scoreWindow(window, piece)

        for c in range(COLOANE - 3):
            window = [tabla[r + 3 - i][c + i] for i in range(4)]
            score += scoreWindow(window, piece)

    return score


def scoreWindow(window, piece):
    opponent_piece = 3 - piece
    score = 0

    # Dacă fereastra are 4 piese ale jucătorului, acordăm cel mai mare scor.
    if window.count(piece) == 4:
        score += 100
    # Dacă fereastra are 3 piese ale jucătorului și o poziție goală, acordăm un scor mediu.
    elif window.count(piece) == 3 and window.count(0) == 1:
        score += 10
    # Dacă fereastra are 2 piese ale jucătorului și 2 goluri, acordăm un scor mic.
    elif window.count(piece) == 2 and window.count(0) == 2:
        score += 5

    # Penalizăm feronstrele în care adversarul are 3 piese și o poziție goală.
    if window.count(opponent_piece) == 3 and window.count(0) == 1:
        score -= 8

    return score
```

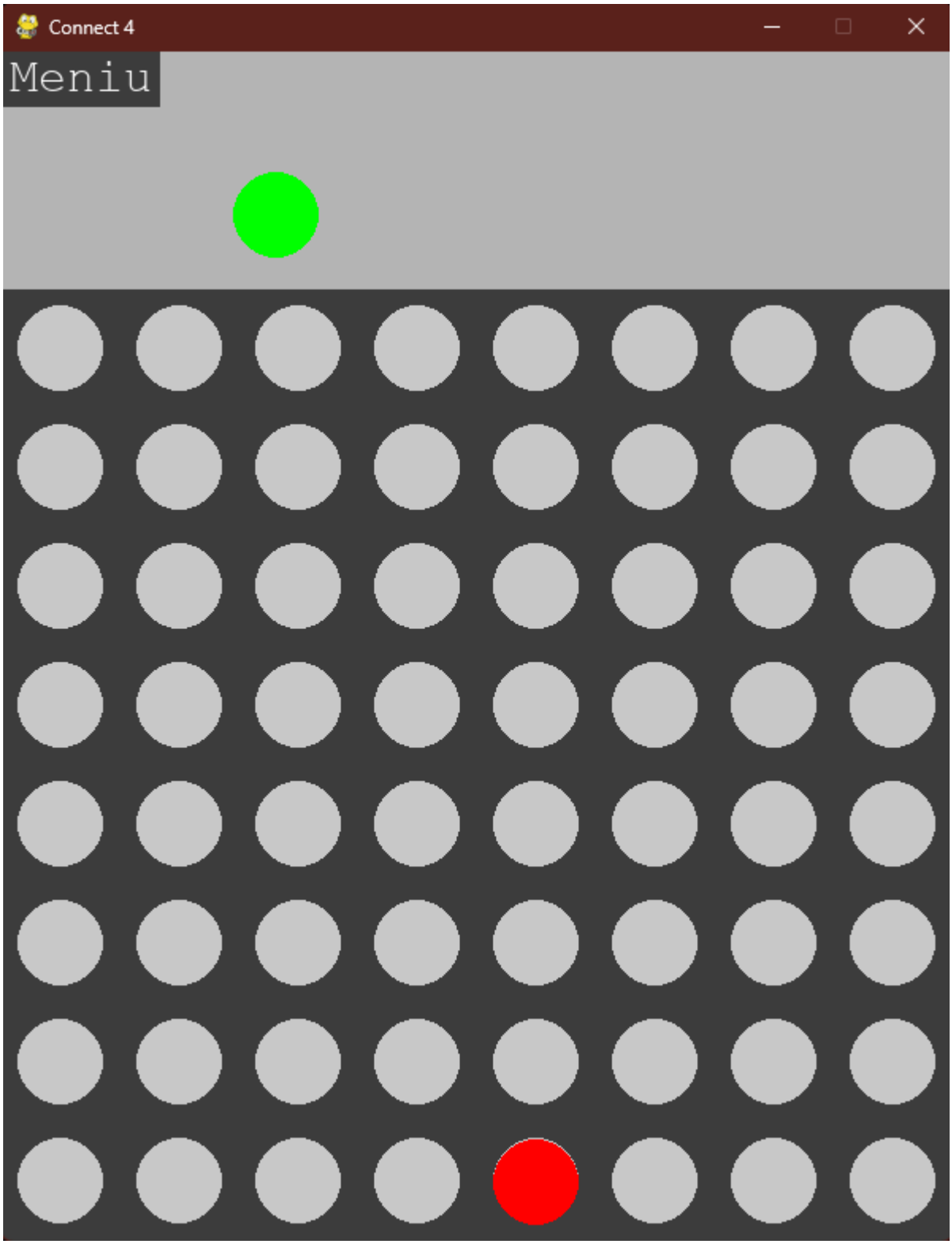
4.2. Calcularea următoarei poziții a calculatorului

```python
def aiMove(tabla, depth=4):
    # Inițializăm scorul cel mai bun ca fiind cel mai mic posibil și o listă pentru mutările cele mai bune
    best_score = -math.inf
    best_moves = []

    # Obținem toate mutările valide pe tabla de joc
    valid_moves = getValidMoves(tabla)

    # Verificăm dacă AI-ul poate câștiga imediat cu o mutare
    for move in valid_moves:
        row = urmRandLiber(tabla, move)  # Determinăm rândul unde se face mutarea
        makeMove(tabla, row, move, 2)  # AI-ul face mutarea
        if checkWin(tabla, 2):  # Verificăm dacă AI-ul câștigă
            undoMove(tabla, row, move)  # Anulăm mutarea
            return move  # Întoarcem mutarea câștigătoare
        undoMove(tabla, row, move)  # Anulăm mutarea dacă nu este câștigătoare

    # Dacă nu există o mutare câștigătoare, căutăm cea mai bună mutare utilizând minimax
    for move in valid_moves:
        row = urmRandLiber(tabla, move)  # Determinăm rândul unde se face mutarea
        makeMove(tabla, row, move, 2)  # AI-ul face mutarea
        # Calculăm scorul pentru această mutare folosind algoritmul minimax
        score = minimax(tabla, depth, -math.inf, math.inf, False, 2)
        undoMove(tabla, row, move)  # Anulăm mutarea după evaluare

        # Dacă scorul obținut este mai bun decât cel anterior, actualizăm lista mutărilor cele mai bune
        if score > best_score:
            best_score = score
            best_moves = [move]  # Începem o nouă listă cu mutarea curentă
        # Dacă scorul este același, adăugăm mutarea în lista celor mai bune mutări
        elif score == best_score:
            best_moves.append(move)

    # Alege aleatoriu una dintre cele mai bune mutări
    return random.choice(best_moves)
```

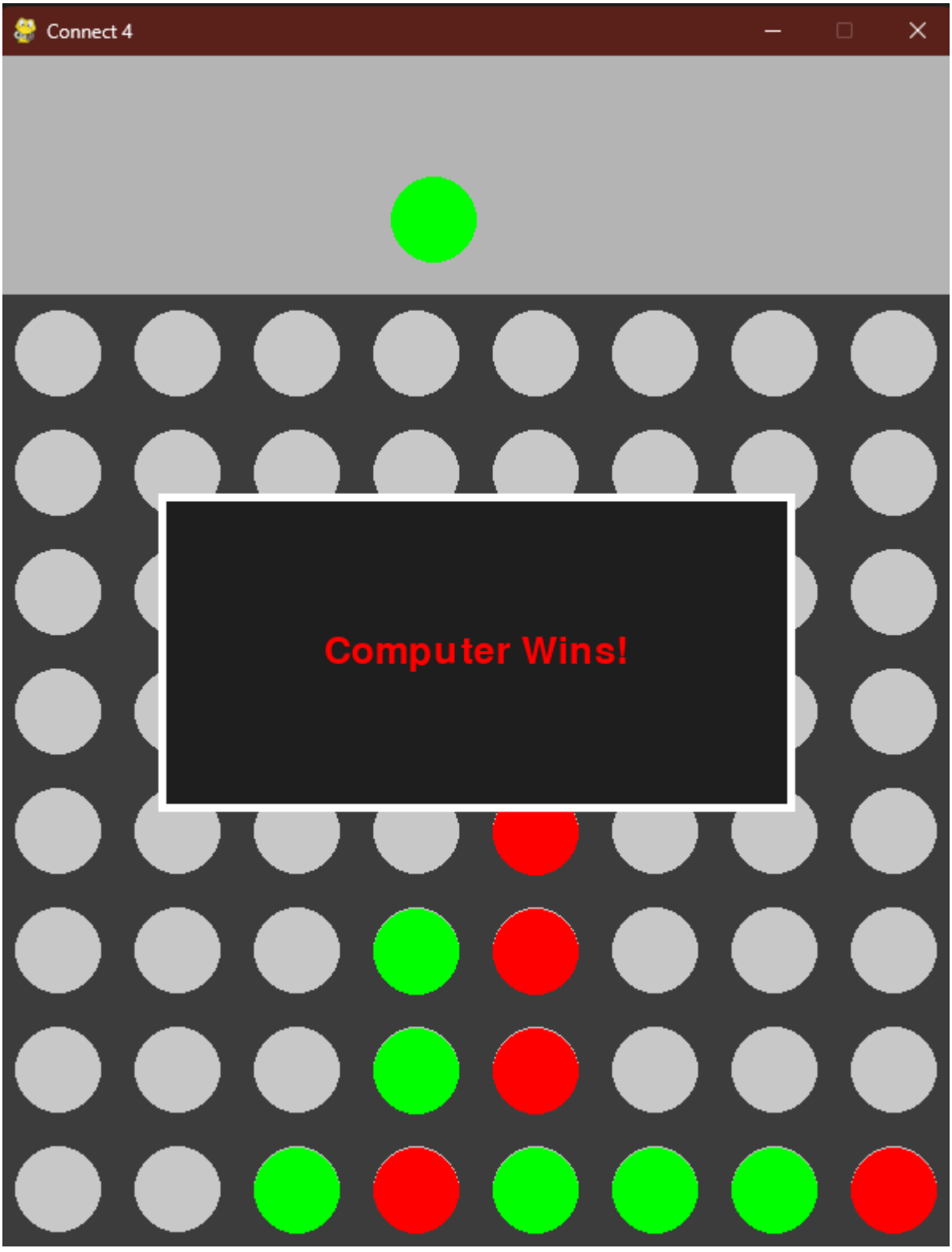
4.3. Algoritmul minimax cu retezare alfa-beta

```python
def minimax(tabla, depth, alpha, beta, is_maximizing, piece):
    # Determinăm piesa adversarului (dacă piesa jucătorului este 1, piesa adversarului va fi 2 și invers).
    opponent_piece = 3 - piece
    # Obținem toate mutările valide pe tabla de joc.
    valid_moves = getValidMoves(tabla)
    # Verificăm dacă jucătorul curent a câștigat. Dacă da, returnăm o valoare mare (pentru maximizator) sau mică (pentru minimizator).
    if checkWin(tabla, piece):
        return float('inf') if is_maximizing else float('-inf')
    # Verificăm dacă adversarul a câștigat. Dacă da, returnăm o valoare mică (pentru maximizator) sau mare (pentru minimizator).
    if checkWin(tabla, opponent_piece):
        return float('-inf') if is_maximizing else float('inf')
    # Dacă am ajuns la adâncimea maximă de căutare sau nu mai sunt mutări valide, evaluăm tabla de joc.
    if depth == 0 or not valid_moves:
        return evaluate(tabla, piece)
    # Dacă este rândul jucătorului care vrea să maximizeze scorul (de obicei jucătorul curent), căutăm mutarea optimă.
    if is_maximizing:
        max_eval = -math.inf  # Începem cu o valoare foarte mică pentru a găsi maximul
        for move in valid_moves:
            # Calculăm rândul pe care se va face mutarea.
            row = urmRandLiber(tabla, move)
            # Facem mutarea pe tabla de joc.
            makeMove(tabla, row, move, piece)
            # Apelăm recursiv minimax pentru a obține evaluarea mutării.
            eval = minimax(tabla, depth - 1, alpha, beta, False, piece)
            # Undo mutarea pentru a încerca următoarea mutare posibilă.
            undoMove(tabla, row, move)
            # Actualizăm valoarea maximă de evaluare.
            max_eval = max(max_eval, eval)
            # Actualizăm valoarea alpha (valoarea maximă garantată pentru maximizator).
            alpha = max(alpha, eval)
            # Dacă beta este mai mic sau egal cu alpha, înseamnă că nu mai are rost să explorăm mai multe mutări.
            if beta <= alpha:
                break
        return max_eval
    # Dacă este rândul adversarului care vrea să minimizeze scorul, căutăm mutarea optimă pentru el.
    else:
        min_eval = math.inf  # Începem cu o valoare foarte mare pentru a găsi minimul
        for move in valid_moves:
            # Calculăm rândul pe care se va face mutarea.
            row = urmRandLiber(tabla, move)
            # Facem mutarea pe tabla de joc pentru adversar.
            makeMove(tabla, row, move, opponent_piece)
            # Apelăm recursiv minimax pentru a obține evaluarea mutării.
            eval = minimax(tabla, depth - 1, alpha, beta, True, piece)
            # Undo mutarea pentru a încerca următoarea mutare posibilă.
            undoMove(tabla, row, move)
            # Actualizăm valoarea minimă de evaluare.
            min_eval = min(min_eval, eval)
            # Actualizăm valoarea beta (valoarea minimă garantată pentru minimizator).
            beta = min(beta, eval)
            # Dacă beta este mai mic sau egal cu alpha, înseamnă că nu mai are rost să explorăm mai multe mutări.
            if beta <= alpha:
                break
        return min_eval
```

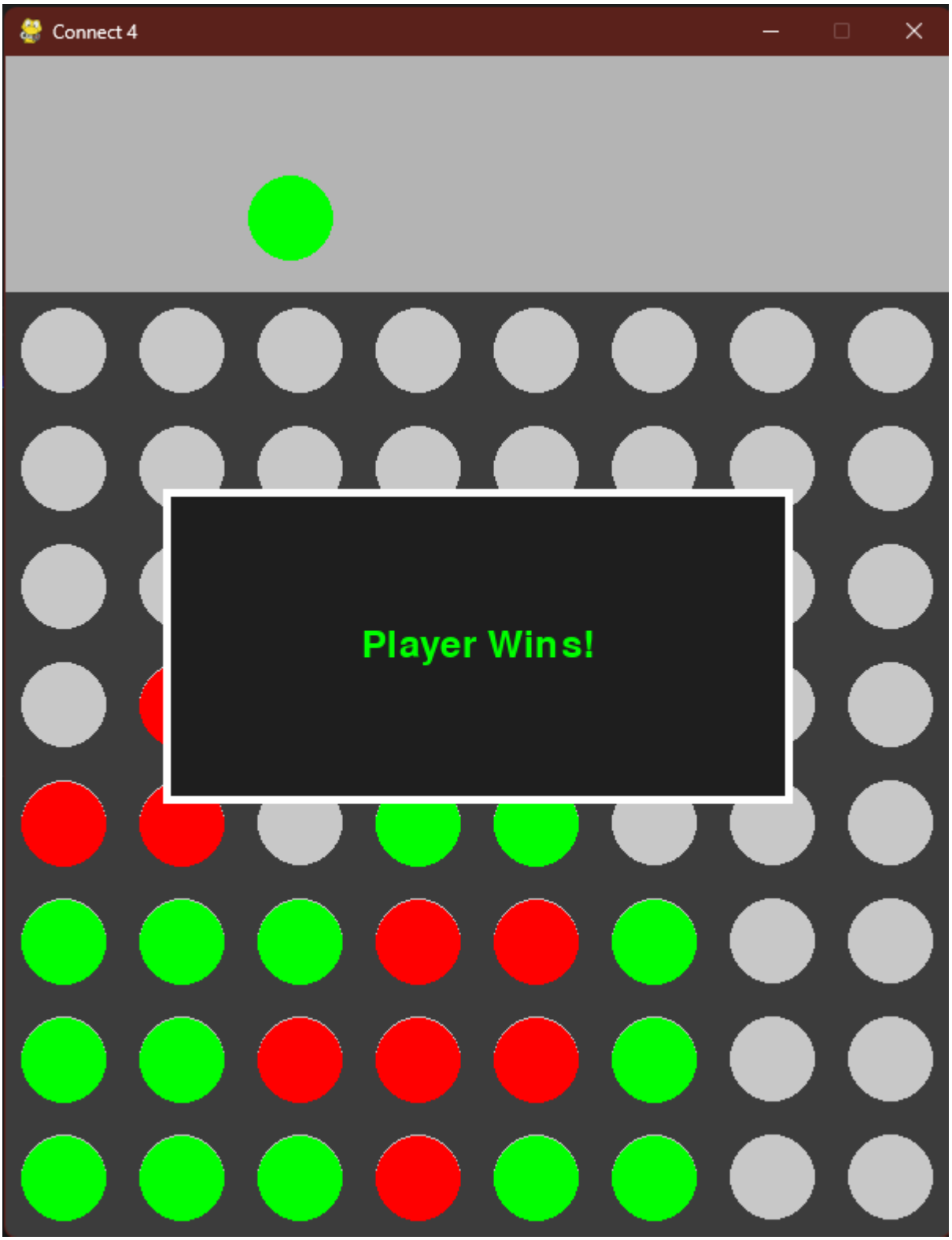
5. Rezultatele obținute prin rularea programului



Începerea unui joc nou (calculatorul – roșu a făcut prima mutare)



Calculatorul a câștigat



Player-ul a câștigat

Meniu principal

6. Concluzii

Algoritmul minimax este folosit în mod frecvent în jocurile de tip sumă zero pentru a determina strategia optimă, explorând toate posibilele soluții ale problemei. Cu toate acestea, această abordare este ineficientă în ceea ce privește timpul de procesare și utilizarea memoriei, deoarece adesea sunt examinate stări de joc care nu aduc niciun beneficiu.

Pentru a îmbunătăți acest proces, a fost introdus algoritmul de retezare alfa-beta, care adaugă o optimizare semnificativă. În scenariul ideal, atunci când mutările cele mai bune sunt analizate întâi, algoritmul poate elimina rapid opțiunile care nu sunt relevante. Totuși, în cel mai nefavorabil caz, atunci când evaluarea celor mai bune mutări are loc la sfârșit, retezarea alfa-beta nu aduce nicio îmbunătățire față de complexitatea algoritmului minimax standard.

7. Bibliografie

[1] https://www.studocu.com/row/document/gift-university/artificial-intellegence/assignment-3-implementing-the-minimax-algorithm-with-alpha-beta-pruning-tree-visualization/95890524
[2] https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/
[3] https://www.youtube.com/watch?v=xBXHtz4Gbdo
[4] Inteligență artificială – Laborator 7, Curs 3
[5] https://www.pygame.org/news
[6] https://www.youtube.com/watch?v=y9VG3Pztok8

Github Repository : https://github.com/IoanCirja/MinMax-Algorithm

8. Rolul fiecărui membru al echipei

Cazamir Andrei

- creare prototip interfață grafică
- adăugare meniu principal interfață
- perfecționare algoritm minimax (bug fix)
- optimizare game loop
- selectare număr de nivele algoritm

Cîrjă Ioan

- creare prototip efectuare mutări de către player/calculator
- evaluarea tablei pentru identificarea câștigătorului
- implementare minimax recursiv (variantă inițială)
- optimizare funcție de evaluare
- documentație