Project 2 - MultiAgents

Adela Morar

Noiembrie 2025

1 Introducere în Căutarea Multiagent

Acest document descrie implementarea algoritmilor de căutare **Minimax** și **Alpha-Beta Pruning**, fundamentali pentru luarea deciziilor în mediile multiagent, în special în jocuri cu sumă zero, precum Pacman. Agentul Pacman (jucătorul MAX) caută **maximizarea** utilității sale, în timp ce fantomele (jucătorii MIN) urmăresc **minimizarea** acestei utilități.

2 Funcția de Evaluare: `evaluationFunction`

2.1 Descriere și Scop

Funcția `evaluationFunction` servește drept **funcție euristică terminală** pentru stările non-terminale atinse la adâncimea limitată de căutare. Rolul său este de a atribui o valoare numerică fiecărei stări a jocului, reprezentând o **estimare a utilității** stării respective pentru agentul Pacman (MAX).

2.2 Structură și Componente Cheie

O funcție de evaluare eficientă ponderează următorii factori, unde S este starea curentă:

1. **Scorul Curent (Score(S)):** Baza evaluării, reflectând progresul jocului.
2. **Interacțiunea cu Fantomele (GhostInteraction(S)):** Pedeapsă mare dacă fantomele periculoase sunt aproape, recompensă dacă fantomele speriate pot fi consumate.
3. **Distanța până la mâncare (FoodDistance(S)):** O inversă a distanței față de cea mai apropiată mâncare, pentru a încuraja progresul.

3 Algoritmul Minimax

3.1 Principiul de Bază

Algoritmul **Minimax** explorează arborele de căutare pe o adâncime predefinită, făcând un joc perfect rațional din partea tuturor agenților (MAX și MIN).

- **Nodul MAX (Pacman):** Alege mișcarea care **maximizează** rezultatul așteptat.
- **Nodul MIN (Fantomele):** Alege mișcarea care **minimizează** rezultatul agentului MAX.

3.2 Reguli de Decizie

- MAX Value(s) = $\max_{a \in \text{Actions}(s)}$ Minimax(Successor(s, a))
- MIN Value(s) = $\min_{a \in \text{Actions}(s)}$ Minimax(Successor(s, a))

3.3 Pseudocod Minimax

```
Funcția getAction(gameState):
    // 1. Inițializare: Obținem numărul total de agenți (Pacman + Fantome)
    numAgents ← gameState.getNumAgents()

    // 2. Funcția recursivă Minimax
    Definește funcția recursivă minmax(state, agentIndex, depth):
        // Verificare Stări Terminale: Jocul s-a terminat SAU Adâncimea maximă a fost atinsă
        DACĂ state.isWin() SAU state.isLose() SAU depth = self.depth ATUNCI:
            // Returnează valoarea euristică (scorul) stării terminale
            RETURN self.evaluationFunction(state)

        // Obține acțiunile posibile pentru agentul curent
        legalActions ← state.getLegalActions(agentIndex)

        // Dacă nu există acțiuni legale (stare terminală neașteptată)
        DACĂ NU legalActions ATUNCI:
            RETURN self.evaluationFunction(state)

        // Funcție auxiliară pentru a determina următorul agent și adâncimea
        Definește funcția next_agent_depth(agentIndex):
            // Agentul următor
            nextAgent ← (agentIndex + 1) MODULO numAgents
            // Adâncimea crește cu 1 DOAR când se trece de la ultimul
            // agent (ultima fantomă) înapoi la Pacman (agentIndex = 0)
            nextDepth ← depth + 1 DACĂ nextAgent = 0 ALTFEL depth
            RETURN (nextAgent, nextDepth)

        // Nodul MAX: Agentul 0 (Pacman) maximizează scorul
        DACĂ agentIndex = 0 (Agentul MAX - Pacman):
            bestScore ← -INFINITE
            PENTRU FIECARE action ÎN legalActions:
                successor ← state.generateSuccessor(agentIndex, action)
                (nextAgent, nextDepth) ← next_agent_depth(agentIndex)
                // Recursivitate: Apelează minmax pentru starea succesoare
                score ← minmax(successor, nextAgent, nextDepth)
                bestScore ← MAX(bestScore, score)
            RETURN bestScore

        // Nodul MIN: Agenții > 0 (Fantomele) minimizează scorul
        ALTFEL (Agentul MIN - Fantome):
            bestScore ← +INFINITE
            PENTRU FIECARE action ÎN legalActions:
                successor ← state.generateSuccessor(agentIndex, action)
                (nextAgent, nextDepth) ← next_agent_depth(agentIndex)
                // Recursivitate: Apelează minmax pentru starea succesoare
                score ← minmax(successor, nextAgent, nextDepth)
                bestScore ← MIN(bestScore, score)
            RETURN bestScore

    // 3. Logica Principală: Determinarea celei mai bune
    // acțiuni inițiale pentru Pacman (Agentul 0)

    // Obține acțiunile legale pentru Pacman
    lecalAction ← gameState.getLegalActions(0)

    // Verifică dacă Pacman poate face vreo mișcare
    DACĂ NU lecalAction ATUNCI:
```

```
        RETURN Directions.STOP

    bestScore ← -INFINITE
    bestAction ← Directions.STOP

    // 4. Iterarea prin acțiunile lui Pacman
    PENTRU FIECARE action ÎN lecalAction:
        // Generează starea succesoare după mișcarea lui Pacman
        successor ← gameState.generateSuccessor(0, action)

        // Începe recursivitatea Minimax. Următorul agent este Fantoma 1
        // (index 1), la adâncimea 0 (primul nivel complet)
        score ← minmax(successor, 1, 0)

        // Actualizează cea mai bună acțiune (cea care maximizează scorul)
        DACĂ score > bestScore ATUNCI:
            bestScore ← score
            bestAction ← action

    // 5. Returnează acțiunea optimă
    RETURN bestAction
```

4 Algoritmul Alpha-Beta Pruning

4.1 Descriere Generală

Alpha-Beta Pruning este o optimizare esențială a Minimax, destinată creșterii eficienței. Permite algoritmului să obțină aceeași decizie ca Minimax, dar explorând un număr mult mai mic de noduri.

4.2 Mecanismul de Pruning

Pruning-ul utilizează două variabile pentru a menține cea mai bună valoare cunoscută până la nivelul curent:

- α **(Alpha):** Cea mai bună valoare (cea mai mare) găsită de oricare nod MAX pe drumul de la rădăcină la nodul curent. (α este o limită inferioară).
- β **(Beta):** Cea mai bună valoare (cea mai mică) găsită de oricare nod MIN pe drumul de la rădăcină la nodul curent. (β este o limită superioară).

4.3 Condiția de Tăiere (Cutoff Condition)

O ramură este tăiată și nu mai este explorată atunci când:

$$\alpha \geq \beta$$

Dacă un nod MIN găsește o valoare β mai mică decât α (cea mai bună opțiune a lui MAX de pe o ramură anterioară), înseamnă că MAX nu va alege niciodată calea care trece prin nodul curent, deoarece știe că MIN poate forța un rezultat mai mic decât cel deja asigurat.

4.4 Pseudocod Alpha-Beta Pruning

```
Funcția getAction(gameState):
    // Obține numărul total de agenți
    numAgents ← gameState.getNumAgents()

    // Funcția recursivă Alpha-Beta
    Definește funcția alphabeta(state, agentIndex, depth, alpha, beta):
        // alpha: Cea mai bună valoare (MAX) găsită până acum pe calea MAX
```

```
    // beta: Cea mai bună valoare (MIN) găsită până acum pe calea MIN

    // 1. Verificare Stări Terminale:
    DACĂ state.isWin() SAU state.isLose() SAU depth = self.depth ATUNCI:
        RETURN self.evaluationFunction(state)

    legalActions ← state.getLegalActions(agentIndex)
    DACĂ NU legalActions ATUNCI:
        RETURN self.evaluationFunction(state)

    // Funcție auxiliară pentru a calcula următorul agent și adâncimea
    Definește funcția next_agent_depth(agentIndex):
        nextAgent ← (agentIndex + 1) MODULO numAgents
        // Adâncimea crește doar când se trece înapoi la Pacman (agent 0)
        nextDepth ← depth + 1 DACĂ nextAgent = 0 ALTFEL depth
        RETURN (nextAgent, nextDepth)

    // 2. Nodul MAX (Pacman): încearcă să maximizeze
    DACĂ agentIndex = 0:
        bestScore ← -INFINITE
        PENTRU FIECARE action ÎN legalActions:
            successor ← state.generateSuccessor(agentIndex, action)
            (nextAgent, nextDepth) ← next_agent_depth(agentIndex)
            score ← alphabeta(successor, nextAgent, nextDepth, alpha, beta)
            bestScore ← MAX(bestScore, score)

            // TĂIERE BETA: Dacă scorul curent al lui MAX (bestScore)
            // este mai mare sau egal cu beta, ramura nu va fi aleasă
            // oricum de nodul MIN părinte, deci tăiem.
            DACĂ bestScore > beta ATUNCI:
                RETURN bestScore

            // Actualizează alpha
            alpha ← MAX(alpha, bestScore)
        RETURN bestScore

    // 3. Nodul MIN (Fantome): încearcă să minimizeze
    ALTFEL:
        bestScore ← +INFINITE
        PENTRU FIECARE action ÎN legalActions:
            successor ← state.generateSuccessor(agentIndex, action)
            (nextAgent, nextDepth) ← next_agent_depth(agentIndex)
            score ← alphabeta(successor, nextAgent, nextDepth, alpha, beta)
            bestScore ← MIN(bestScore, score)

            // TĂIERE ALPHA: Dacă scorul curent al lui MIN (bestScore)
            // este mai mic sau egal cu alpha, ramura nu va fi aleasă
            // oricum de nodul MAX părinte, deci tăiem.
            DACĂ bestScore < alpha ATUNCI:
                RETURN bestScore

            // Actualizează beta
            beta ← MIN(beta, bestScore)
        RETURN bestScore

// 4. Logica Principală: Determinarea celei mai bune acțiuni la rădăcină (Root Node)

lecalAction ← gameState.getLegalActions(0)
```

```
DACĂ NU lecalAction ATUNCI:
    RETURN Directions.STOP

bestScore ← -INFINITE
bestAction ← Directions.STOP
alpha ← -INFINITE // Inițializare Alpha
beta ← +INFINITE  // Inițializare Beta

// Iterăm pe acțiunile lui Pacman la rădăcină
PENTRU FIECARE action ÎN lecalAction:
    successor ← gameState.generateSuccessor(0, action)

    // Începe recursivitatea pentru prima fantomă (agent 1)
    score ← alphabeta(successor, 1, 0, alpha, beta)

    DACĂ score > bestScore ATUNCI:
        bestScore ← score
        bestAction ← action

    // Actualizăm alpha după fiecare acțiune la nivelul rădăcinii
    alpha ← MAX(alpha, score)

RETURN bestAction
```