UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

# Pacman

*Inteligență Artificială*

Autori: Rotariu Laura-Alexandra și Nistor Dalia-Emilia
Grupa: 30234

Facultatea de Automatica
si Calculatoare

2022-2023

# Cuprins

# 1 Introducere

## 1.1 Context

Pac-Man este unul dintre cele mai populare jocuri din lume. Jucătorul controlează Pac-Man, care trebuie să mănânce toate punctele dintr-un labirint închis, evitând fantomele colorate. Consumând puncte mari de mâncare, numite power pellets, face ca fantomele să devină albe, permițându-i lui Pac-Man să le mănânce pentru puncte bonus. Aici, scopul principal este să acumulezi puncte colectând mâncare din labirint și să te ferești de fantomele care umblă în jurul labirintului. Dacă fantoma capturează Pac-Man, jocul se termină.

## 1.2 Motivație

Vom proiecta un agent Pac-Man inteligent care va găsi căi optime prin labirint pentru a ajunge la starea scopului, mâncând toate punctele și evitând fantomele într-un număr minim de pași. Pentru a proiecta acest agent inteligent, am implementat mai mulți algoritmi de căutare. Algoritmii de căutare sunt: algoritmi de căutare neinformați (DFS, BFS, UCS), algoritmi de căutare informați (căutarea A*). Diferența principală între căutarea neinformată și cea informată este că algoritmii de căutare neinformați nu primesc nicio informație despre problema dată, în timp ce algoritmii de căutare informați sunt furnizați cu informații despre problemă. De asemenea, am implementat algoritmi pentru mai mulți agenți, precum ReflexAgent, Minimax și Alpha-Beta. Prin utilizarea acestor algoritmi, agentul Pac-Man va încerca să scape de agenții fantomă și să mănânce toată hrana din labirint pentru a câștiga jocul.

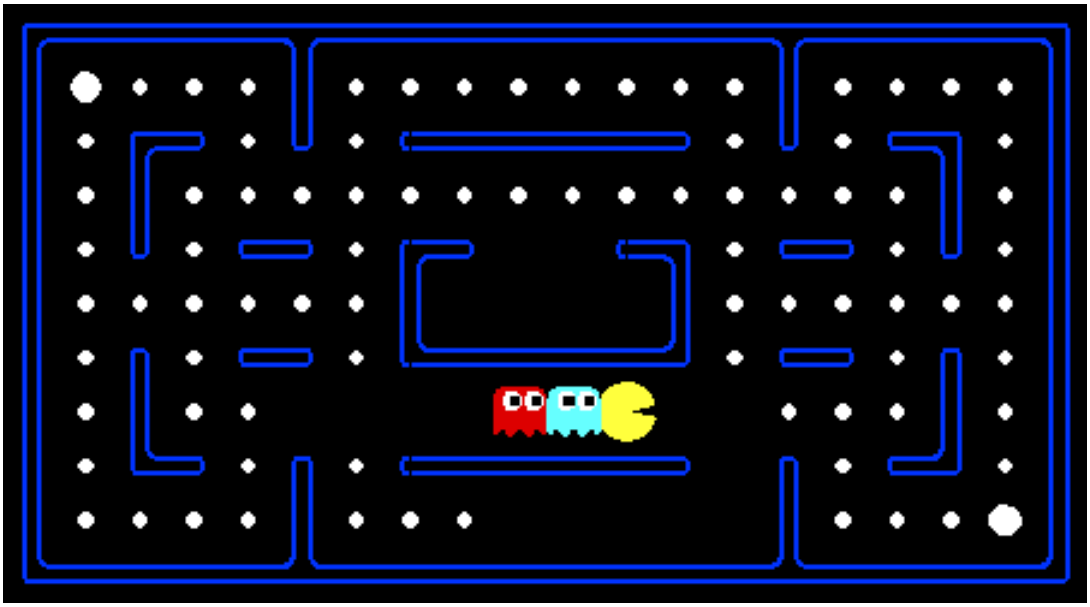

Figura 1: Pacman

Figura 1 reprezintă interfața jocului Pacman.

# 2 Uninformed search

## 2.1 Depth-first search

Căutarea în adâncime încearcă întotdeauna să extindă nodul cel mai profund din stiva arborelui de căutare. Acest algoritm este bazat pe o structură de date tip stivă (LIFO - ultimul intrat, primul ieșit). Prin intermediul stivei, este ales nodul generat cel mai recent pentru expansiune.

```
def depthFirstSearch(problem: SearchProblem):
    state_stack = util.Stack()
    start = problem.getStartState()
    state_stack.push((start, []))
    visited = set()

    while not state_stack.isEmpty():
        popped_element = state_stack.pop()
        current_state = popped_element[0]
        path = popped_element[1]
        if problem.isGoalState(current_state):
            return path
        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state, direction, _ in successors:
                if next_state not in visited:
                    state_stack.push((next_state, path + [direction]))
```

Această funcție depthFirstSearch implementează algoritmul de căutare în adâncime pentru a găsi soluția unei probleme date. Pașii:

1. Se creează o stivă (state stack) pentru a gestiona stările.
2. Se adaugă starea de start în stivă împreună cu o listă goală de acțiuni (path) asociate cu acea stare.
3. Se inițializează un set (visited) pentru a urmări stările vizitate.
4. În timp ce stiva nu este goală:
   - Se extrage ultimul element (popped element) din stivă.
   - Se obține starea curentă și calea asociată cu acea stare.
   - Dacă starea curentă este starea scop, se returnează calea găsită până la acel punct.
   - Dacă starea curentă nu a fost vizitată, se marchează starea curentă ca vizitată și se obțin succesorii pentru starea curentă și se iterează prin ei. Dacă un succesor nu a fost vizitat, acesta este adăugat în stivă cu calea actualizată (path + [direction]).

## 2.2 Breadth-first search

Algoritmul de căutare în lățime (BFS) începe prin extinderea nodului rădăcină și apoi extinde toți copiii nodului rădăcină, apoi succesiv pe urmașii lor, și așa mai departe. Aici, toate nodurile sunt extinse nivel cu nivel, ceea ce înseamnă că toate nodurile de la un anumit nivel vor fi extinse înainte de a trece la nivelul următor. BFS folosește o coadă (structură de date FIFO). Căutarea în lățime returnează o soluție cu cel mai mic cost în ceea ce privește efortul depus de către Pacman pentru a ajunge la punctul de hrană.

```python
def breadthFirstSearch(problem: SearchProblem):
    state_queue = util.Queue()
    start = problem.getStartState()
    state_queue.push((start, []))
    visited = set()

    while not state_queue.isEmpty():
        popped_element = state_queue.pop()
        current_state = popped_element[0]
        path = popped_element[1]
        if problem.isGoalState(current_state):
            return path
        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state, direction, _ in successors:
                if next_state not in visited:
                    state_queue.push((next_state, path + [direction]))
```

Acest cod implementează algoritmul de căutare în lățime (BFS) pentru găsirea soluției unei probleme date. Pașii:

1. Se creează o coadă (state queue) pentru a gestiona stările.
2. Se adaugă starea de start în coadă împreună cu o listă goală de acțiuni (path) asociate cu acea stare.
3. Se inițializează un set (visited) pentru a urmări stările vizitate.
4. În timp ce coada nu este goală:
   - Se extrage primul element (popped element) din coadă.
   - Se obține starea curentă și calea asociată cu acea stare.
   - Dacă starea curentă este starea scop, se returnează calea găsită până la acel punct.
   - Dacă starea curentă nu a fost vizitată, se marchează starea curentă ca vizitată și se obțin succesorii pentru starea curentă și se iterează prin ei. Dacă un succesor nu a fost vizitat, acesta este adăugat în coadă cu calea actualizată (path + [direction]).

### 2.3 Uniform Cost Search

Principalul aspect al algoritmului de căutare cu cost uniform este că, în loc să extindă nodul cel mai profund, încearcă să extindă nodul cu cel mai mic cost al căii.

Acest lucru este realizat prin utilizarea unei structuri de date de tip coadă de priorități în care elementele sunt ordonate în funcție de cost.

```python
def uniformCostSearch(problem: SearchProblem):
    state_queue = util.PriorityQueue()
    start = problem.getStartState()
    state_queue.push((start, [], 0), 0)
    visited = set()

    while not state_queue.isEmpty():
        popped_element = state_queue.pop()
        current_state = popped_element[0]
```

```
        path = popped_element[1]
        cost = popped_element[2]
        if problem.isGoalState(current_state):
            return path
        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state, direction, state_cost in successors:
                if next_state not in visited:
                    total_cost = cost + state_cost
                    state_queue.push((next_state, path + [direction], total_cost),
                    total_cost)
```

Pașii:

1. Funcția utilizează o coadă de priorități (state queue) pentru a prioritiza nodurile în funcție de costul total. Prioritatea este determinată de costul cumulativ al atingerii stării curente de la starea de start.
2. Starea inițială (starea de start) este adăugată în coada de priorități cu un șir vid și cost zero.
3. Funcția menține un set de stări vizitate (visited) pentru a evita reluarea aceleiași stări.
4. Bucla principală continuă până când coada de priorități este goală, indicând că au fost explorate toate căile posibile.
5. În fiecare iterație a buclei, starea cu cel mai mic cost total este extrasă din coada de priorități.
6. Dacă starea extrasă este o stare scop, funcția returnează calea corespunzătoare.
7. În caz contrar, sunt obținuți succesori pentru starea curentă, și pentru fiecare succesor, dacă nu a fost vizitat, o nouă stare este adăugată în coada de priorități cu o cale actualizată și un cost total.
8. util.PriorityQueue se asigură că stările cu costuri totale mai mici sunt explorate în primul rând.

# 3 Informed search

## 3.1 A* search algorithm

Algoritmul de căutare A* este un tip de algoritm de căutare informat care dispune de informații despre problema înainte de a începe căutarea. Acesta evaluează nodurile prin adăugarea a x(n), costul de a ajunge la nod, și y(n), costul de a ajunge de la nod la scop: f(n) = x(n) + y(n). Deoarece x(n) reprezintă costul de la nodul de start la nodul n, iar y(n) este costul căii optime de la n la scop, avem f(n) = costul total al celei mai ieftine soluții prin n.

Prin urmare, dacă dorim să găsim cea mai ieftină soluție, ar trebui să luăm în considerare nodul cu cea mai mică valoare a lui f(n).

Singura diferență între căutarea A* și UCS este că UCS ia în considerare costul de a ajunge la nod, în timp ce A* Search consideră suma costului de a ajunge la nod și costul de a ajunge la nodul țintă din acel nod (x + y) în loc de x. Aici, funcția euristică folosită este euristica distanței Manhattan. Această euristică este folosită pentru a determina care nod sau stare este mai aproape de starea scop.

```python
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    state_queue = util.PriorityQueue()
    start = problem.getStartState()
    state_queue.push((start, [], 0), 0)
    visited = set()

    while not state_queue.isEmpty():
        popped_element = state_queue.pop()
        current_state = popped_element[0]
        path = popped_element[1]
        cost = popped_element[2]
        if problem.isGoalState(current_state):
            return path
        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state, direction, state_cost in successors:
                if next_state not in visited:
                    new_cost = cost + state_cost
                    total_cost = new_cost + heuristic(next_state, problem)
                    state_queue.push((next_state, path + [direction], new_cost),
                    total_cost)
```

Pașii:

1. Funcția utilizează o coadă de priorități (state queue) pentru a prioritiza nodurile în funcție de costul total și o euristică. Prioritatea este determinată de suma dintre costul acumulat până la starea curentă și valoarea euristicii pentru starea următoare.
2. Starea inițială (starea de start) este adăugată în coada de priorități cu un șir vid și cost zero. Valoarea euristicii pentru starea inițială poate fi specificată prin argumentul heuristic.
3. Funcția menține un set de stări vizitate (visited) pentru a evita reluarea aceleiași stări.
4. Bucla principală continuă până când coada de priorități este goală, indicând că au fost explorate toate căile posibile.
5. În fiecare iterație a buclei, starea cu cea mai mică valoare de prioritizare este extrasă din coada de priorități.
6. Dacă starea extrasă este o stare scop, funcția returnează calea corespunzătoare.
7. În caz contrar, sunt obținuți succesori pentru starea curentă, și pentru fiecare succesor, dacă nu a fost vizitat, un nou cost este calculat (costul acumulat până la succesor) și se adaugă în coada de priorități cu o valoare de prioritizare actualizată.
8. util.PriorityQueue se asigură că stările cu costuri totale și euristice mai mici sunt explorate în primul rând.

### 3.2 Finding All the Corners

Funcția getSuccessors din clasa CornersProblem este responsabilă pentru generarea stărilor succesoare pentru o stare dată în cadrul problemei de căutare a colțurilor.

```
  def getSuccessors(self, state: Any):
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
        Directions.WEST]:
            x, y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                next_state = (nextx, nexty)
                cost = 1
                set_of_corners = set(state[1])
                if next_state in self.corners and next_state in set_of_corners:
                    set_of_corners.remove(next_state)
                successors.append(((next_state, tuple(set_of_corners)), action, cost))
        self._expanded += 1
        return successors
```

Pașii:

1. Se iterează prin direcțiile posibile (NORTH, SOUTH, EAST, WEST).
2. Pentru fiecare direcție, se calculează noul x și y pe baza direcției.
3. Se verifică dacă noul x și y nu intersectează un zid (not self.walls[nextx][nexty]).
4. Dacă nu există zid, se adaugă o stare succesoare în lista successors. Aceasta conține noul x și y, acțiunea necesară pentru a ajunge acolo și un cost de 1.
5. Se actualizează self. expanded pentru a ține evidența nodurilor extinse.

### 3.3 Corners Problem: Heuristic

Funcția cornersHeuristic este o euristică folosită în problema căutării colțurilor (CornersProblem). Această euristică furnizează o evaluare estimată a distanței dintre starea curentă și cel mai îndepărtat colț nevizitat. Scopul principal al acestei euristici este să ofere o valoare inferioară (admisibilă) pentru cea mai scurtă cale de la starea curentă la un scop al problemei (toate colțurile vizitate).

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    corners = problem.corners
    walls = problem.walls
    current_position = state[0]
    unvisited_corners = state[1]
    if len(unvisited_corners) == 0:
        return 0
    farthest_distance = 0
    for corner in unvisited_corners:
        distance = util.manhattanDistance(current_position, corner)
        if distance > farthest_distance:
            farthest_distance = distance
    return farthest_distance
```

Pașii:

1. Se verifică dacă nu există colțuri nevizitate (if len(unvisited corners) == 0). În acest caz, distanța estimată este 0, deoarece s-a atins deja obiectivul.
2. Se inițializează farthest distance la 0. Aceasta va fi valoarea pe care funcția o va returna la final.
3. Pentru fiecare colț nevizitat, se calculează distanța Manhattan de la poziția curentă la acel colț.
4. Dacă distanța calculată este mai mare decât farthest distance, atunci se actualizează farthest distance.
5. La final, funcția returnează farthest distance, care este considerată o estimare inferioară a distanței la cel mai îndepărtat colț nevizitat.

### 3.4 Eating All the Dots

Funcția foodHeuristic este o euristică folosită în problema căutării hranei (FoodSearchProblem). Această euristică furnizează o evaluare estimată a distanței dintre starea curentă și cel mai apropiat punct de hrană neconsumat. Scopul principal al acestei euristici este să ofere o valoare inferioară (admisibilă) pentru cea mai scurtă cale de la starea curentă la un scop al problemei (consumarea întregii hrane).

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    uneaten = foodGrid.asList()
    if len(uneaten) == 0:
        return 0
    farthest_distance = 0
    for food in uneaten:
        distance = util.manhattanDistance(position, food)
        if distance > farthest_distance:
            farthest_distance = distance
    return farthest_distance
```

Pașii:

1. Se obține lista de coordonate ale punctelor de hrană neconsumate folosind metoda asList() a obiectului foodGrid.
2. Se verifică dacă nu există puncte de hrană neconsumate (if len(uneaten) == 0). În acest caz, distanța estimată este 0, deoarece s-a atins deja obiectivul.
3. Se inițializează farthest distance la 0. Aceasta va fi valoarea pe care funcția o va returna la final.
4. Pentru fiecare punct de hrană neconsumat, se calculează distanța Manhattan de la poziția curentă la acel punct.
5. Dacă distanța calculată este mai mică decât farthest distance, atunci se actualizează farthest distance.
6. La final, funcția returnează farthest distance, care este considerată o estimare inferioară a distanței către cel mai apropiat punct de hrană neconsumat.

### 3.5 Suboptimal Search

Funcția findPathToClosestDot este o metodă a clasei ClosestDotSearchAgent și are rolul de a returna un traseu (o listă de acțiuni) către cel mai apropiat punct de hrană neconsumată, începând din starea curentă a jocului Pacman (gameState).

Această metodă este utilizată în cadrul metodei registerInitialState, care construiește o listă de acțiuni pentru a colecta toată hrana din labirint.

```
def findPathToClosestDot(self, gameState: pacman.GameState):
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)
        return search.bfs(problem)
```

Pașii:

1. gameState: Starea curentă a jocului Pacman, care conține informații despre poziția lui Pacman, harta hranei și altele.
2. findPathToClosestDot(gameState: pacman.GameState) - List[str]:
   - Metoda începe prin a obține poziția curentă a lui Pacman utilizând gameState.getPacmanPosition().
   - Apoi, se obține harta hranei folosind gameState.getFood().
   - Se inițializează o instanță a clasei AnyFoodSearchProblem cu ajutorul stării curente a jocului. Această problemă de căutare are scopul de a găsi un traseu către oricare dintre punctele de hrană rămase.
   - Se utilizează algoritmul BFS (breadth-first search) pentru a găsi un traseu către cel mai apropiat punct de hrană. Astfel, se apelează search.bfs(problem), unde problem este instanța problemei de căutare.
   - Metoda returnează traseul găsit.

# 4 Adversarial search

## 4.1 Improve the ReflexAgent

Aici creăm un agent reflex care, la fiecare pas, alege o acțiune aleatoare din cele legale disponibile. Acest lucru este diferit de un agent de căutare aleatorie, deoarece un agent reflex nu construiește o secvență de acțiuni, ci alege o singură acțiune și o execută. Un agent reflex capabil va trebui să ia în considerare atât locațiile alimentelor, cât și locațiile fantomă pentru a funcționa bine. Am implemetat metoda evaluationFunction din clasa ReflexAgent.

```
 def evaluationFunction(self, currentGameState: GameState, action):
        # information from a GameState (pacman.py)
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
        evaluation = successorGameState.getScore()
        capsules = successorGameState.getCapsules()
        closestFoodDistance = float('inf')
        closestGhostDistance = float('inf')
        for food in newFood.asList():
            distance = util.manhattanDistance(newPos, food)
            if distance < closestFoodDistance:
                closestFoodDistance = distance
```

```
        for ghostState in newGhostStates:
            ghostPos = ghostState.getPosition()
            distance = util.manhattanDistance(newPos, ghostPos)
            if distance < closestGhostDistance:
                closestGhostDistance = distance
        evaluation = successorGameState.getScore()
        if closestGhostDistance <= 1:
            evaluation -= 500
        else:
            evaluation += 1.0 / (closestFoodDistance + 1)
        if newPos in capsules:
            evaluation += 500
        return evaluation
```

Funcția evaluationFunction este folosită pentru a evalua o anumită stare a jocului și o acțiune propusă în jocul Pac-Man. Pașii:

**1. Inițializare variabile și extragerea informațiilor din starea curentă:**

- successorGameState: Se generează starea jocului după o anumită acțiune propusă.

- newPos: Obține poziția nouă a lui Pac-Man din starea succesoare.

- newFood: Obține informații despre alimentele rămase în starea succesoare.

- newGhostStates: Obține informații despre stările fantomelor din starea succesoare.

**2. Evaluarea stării succesoare:**

- closestFoodDistance: Se inițializează cu o valoare infinită pentru a calcula distanța la cea mai apropiată hrană.

- closestGhostDistance: Se inițializează cu o valoare infinită pentru a calcula distanța la cea mai apropiată fantomă.

- Iterare prin hrană: Pentru fiecare hrană rămasă în newFood, se calculează distanța până la aceasta și se actualizează closestFoodDistance cu cea mai mică distanță găsită.

- Iterare prin stările fantomelor: Pentru fiecare fantomă din newGhostStates, se calculează distanța până la aceasta și se actualizează closestGhostDistance cu cea mai mică distanță găsită.

- Evaluare generală: Se inițializează evaluation cu scorul stării succesoare folosit inițial. Se scade un cost semnificativ (-500) dacă Pac-Man se află la o distanță de 1 unitate de cea mai apropiată fantomă, pentru a evita coliziunea. Se adaugă o valoare în funcție de inversul distanței până la cea mai apropiată hrană. Cu cât este mai aproape, cu atât evaluarea va crește. Se adaugă un bonus (+500) dacă Pac-Man se află pe o pastilă specială.

**3. Returnarea evaluării:** Funcția returnează evaluarea rezultată a stării succesoare în funcție de factorii evaluați mai sus. Această evaluare va fi utilizată pentru a alege cea mai bună acțiune pentru Pac-Man în acel moment al jocului.

## 4.2 Minimax

Algoritmul Minimax este o metodă de luare a deciziilor utilizată în jocurile cu doi jucători în care un jucător încearcă să maximizeze câștigul său, iar celălalt jucător încearcă să minimizeze câștigul primului. Am implementat funcția getAction din clasa MinimaxAgent.

```
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):

```

```
        def minimax(state, depth, agentIndex):
            if depth == 0 or state.isWin() or state.isLose():
                return self.evaluationFunction(state), None
            if agentIndex == 0:  #Pacman
                bestValue = float("-inf")
            else:  #Ghost
                bestValue = float("inf")
            bestAction = None
            legalActions = state.getLegalActions(agentIndex)
            for action in legalActions:
                successor = state.generateSuccessor(agentIndex, action)
                value, _ = minimax(successor, depth - 1, (agentIndex + 1) %
                    state.getNumAgents())
                if (agentIndex == 0 and value > bestValue) or (agentIndex != 0
                    and value < bestValue):
                        bestValue = value
                        bestAction = action
            return bestValue, bestAction

        _, bestAction = minimax(gameState, self.depth * gameState.getNumAgents(), 0)
        return bestAction
```

Funcția **getAction** este funcția principală a agenților Minimax. Ea calculează cea mai bună acțiune folosind algoritmul Minimax, bazându-se pe adâncimea dată (self.depth) și funcția de evaluare (self.evaluationFunction). Funcția **minimax** este funcția recursivă care implementează algoritmul Minimax.

1. Dacă adâncimea este 0 sau starea este o victorie sau o înfrângere, se întoarce valoarea evaluată și nicio acțiune (self.evaluationFunction(state), None).

2. Maximizare și minimizare: Dacă este rândul lui Pac-Man (jucătorul de maximizare), se inițializează bestValue la -infinit pentru a găsi cea mai mare valoare posibilă. Dacă este rândul unui ghost (jucătorul de minimizare), se inițializează bestValue la infinit pentru a găsi cea mai mică valoare posibilă.

3. Parcurgerea acțiunilor legale: Pentru fiecare acțiune legală a jucătorului curent, se generează starea succesoare și se calculează recursiv valoarea acesteia folosind funcția minimax. Se actualizează bestValue și bestAction în funcție de valoarea calculată pentru starea succesoare.

4. La final, funcția minimax întoarce cea mai bună valoare și acțiunea asociată acesteia.

5. Apelarea funcției minimax: În getAction, se apelează funcția minimax cu starea jocului curent (gameState), adâncimea și indicele agentului (0 pentru Pac-Man). Rezultatul final al algoritmului Minimax este acțiunea optimă pentru starea dată, care este întoarsă și utilizată ca acțiune recomandată pentru agentul Minimax.

### 4.3 Alpha-Beta Pruning

Algoritmul Alpha-Beta Pruning este o îmbunătățire a algoritmului Minimax, menită să reducă numărul de noduri explorate într-un arbore al jocului, fără a afecta decizia finală luată de algoritm prin actualizarea valorilor alpha și beta pentru a exclude unele ramuri. Astfel, algoritmul Alpha-Beta Pruning poate fi mult mai eficient decât algoritmul Minimax în explorarea spațiului de stări.

```python
class AlphaBetaAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        def alphaBeta(state, depth, alpha, beta, agentIndex):
            if depth == 0 or state.isWin() or state.isLose():
                return self.evaluationFunction(state), None
            if agentIndex == 0:  #Pacman
                bestValue = float("-inf")
                for action in state.getLegalActions(agentIndex):
                    successor = state.generateSuccessor(agentIndex, action)
                    value, _ = alphaBeta(successor, depth - 1, alpha, beta, (agentIndex + 1)
                    if value > bestValue:
                        bestValue = value
                        bestAction = action
                    if bestValue > beta:
                        return bestValue, bestAction
                    alpha = max(alpha, bestValue)
                return bestValue, bestAction
            else:  #Ghost
                bestValue = float("inf")
                for action in state.getLegalActions(agentIndex):
                    successor = state.generateSuccessor(agentIndex, action)
                    value, _ = alphaBeta(successor, depth - 1, alpha, beta, (agentIndex + 1)
                    if value < bestValue:
                        bestValue = value
                        bestAction = action
                    if bestValue < alpha:
                        return bestValue, bestAction
                    beta = min(beta, bestValue)
                return bestValue, bestAction

        _, bestAction = alphaBeta(gameState, self.depth * gameState.getNumAgents(), float("-
        return bestAction

```

Funcția **getAction** este funcția principală a agenților Alpha-Beta. Calculează cea mai bună acțiune folosind algoritmul Alpha-Beta Pruning, bazându-se pe adâncimea dată (self.depth) și funcția de evaluare (self.evaluationFunction). Funcția **alphaBeta** este funcția recursivă care implementează algoritmul Alpha-Beta.

1. Dacă adâncimea este 0 sau starea este o victorie sau o înfrângere, se întoarce valoarea evaluată și nicio acțiune (self.evaluationFunction(state), None).

2. Maximizarea și minimizarea cu tăieri Alpha-Beta: Pentru Pac-Man (jucătorul de maximizare), se încearcă maximizarea valorii. Se parcurg acțiunile legale și se calculează valorile asociate, actualizând alpha și beta. Pentru fiecare acțiune, dacă valoarea este mai mare decât bestValue (cea mai bună valoare până în acel moment), se actualizează bestValue și bestAction. Se efectuează tăierea Beta dacă valoarea depășește beta, și se întoarce valoarea și acțiunea. Similar pentru ghost (jucătorul de minimizare), dar acum se efectuează tăierea Alpha.

3. La final, funcția alphaBeta întoarce cea mai bună valoare și acțiune asociată acesteia, cu tăieri Alpha-Beta aplicate pentru a reduce explorarea inutilă a nodurilor.

4. Apelarea funcției alphaBeta: În getAction, se apelează funcția alphaBeta cu starea jocului curent (gameState), adâncimea și valorile alpha și beta inițiale. Rezultatul final al algoritmului Alpha-Beta este acțiunea optimă pentru starea dată, care este întoarsă și utilizată ca acțiune recomandată pentru agentul Alpha-Beta.

### 4.4 Expectimax

Algoritmul de căutare Expectimax este un algoritm de teorie a jocurilor folosit pentru a maximiza utilitatea așteptată. Este o variantă a algoritmului Minimax. În timp ce Minimax presupune că adversarul (minimizer-ul) joacă optim, Expectimax nu face această presupunere. Acest lucru este util pentru modelarea mediilor în care agenții adversi nu sunt optimi sau acțiunile lor se bazează pe șansă.

```
def expectimax(state, depth, agentIndex):
    if depth == 0 or state.isWin() or state.isLose():
        return self.evaluationFunction(state), None
    if agentIndex == 0:  # Pacman's turn
        bestValue = float("-inf")
        bestAction = None
        for action in state.getLegalActions(agentIndex):
            successor = state.generateSuccessor(agentIndex, action)
            value, _ = expectimax(successor, depth - 1, (agentIndex + 1) %
            state.getNumAgents())
            if value > bestValue:
                bestValue = value
                bestAction = action
        return bestValue, bestAction
    else:  # Ghosts' turn
        sumValues = 0.0
        legalActions = state.getLegalActions(agentIndex)
        probability = 1.0 / len(legalActions)
        for action in legalActions:
            successor = state.generateSuccessor(agentIndex, action)
            value, _ = expectimax(successor, depth - 1, (agentIndex + 1) %
            state.getNumAgents())
            sumValues += value * probability
        return sumValues, None
```

Pașii:

1. Funcția expectimax este o funcție recursivă care explorează arborele de joc pentru jocul Pacman.
2. Primește starea curentă a jocului (state), adâncimea rămasă de explorat (depth) și indexul agentului curent (agentIndex).
3. Dacă adâncimea este 0 sau jocul este într-o stare terminală (victorie sau înfrângere), returnează evaluarea stării folosind funcția de evaluare și None ca cea mai bună acțiune.
4. Dacă este rândul lui Pacman (agentIndex este 0), găsește acțiunea care maximizează valoarea apelând recursiv expectimax pentru fiecare acțiune legală. Cea mai bună valoare și acțiunea corespunzătoare sunt returnate.

5. Dacă este rândul unui ghost, calculează valoarea așteptată prin medierea valorilor tuturor stărilor succesoare posibile ponderate cu probabilitatea fiecărei acțiuni. Apoi returnează valoarea așteptată și None ca cea mai bună acțiune.

### 4.5 Evaluation Function

Funcția betterEvaluationFunction este o funcție de evaluare pentru starea curentă a jocului Pacman. Această funcție își propune să ofere o evaluare mai "inteligentă" a stării curente decât funcția de evaluare implicită.

```
def betterEvaluationFunction(currentGameState: GameState):
    pacmanPosition = currentGameState.getPacmanPosition()
    foodGrid = currentGameState.getFood()
    ghostStates = currentGameState.getGhostStates()
    capsules = currentGameState.getCapsules()
    evaluation = currentGameState.getScore()
    closestFoodDistance = float('inf')
    for food in foodGrid.asList():
        distance = util.manhattanDistance(pacmanPosition, food)
        if distance < closestFoodDistance:
            closestFoodDistance = distance
    for ghostState in ghostStates:
        ghostPos = ghostState.getPosition()
        distance = util.manhattanDistance(pacmanPosition, ghostPos)
        if distance <= 1:
            evaluation -= 500
        else :
            evaluation += 1.0 / (closestFoodDistance + 1)
    if pacmanPosition in capsules:
        evaluation += 500
    return evaluation
```

Pașii:

1. Se calculează distanța cea mai mică la un punct de mâncare pentru a evidenția importanța apropierii de sursa de hrană.
2. Se parcurg toate ghost-urile, și dacă un ghost este în proximitatea imediată (la o distanță de 1), se scade un scor semnificativ (-500) pentru a evita întâlnirea cu acestea.
3. În caz contrar, se adaugă un scor în funcție de inversul distanței la cel mai apropiat punct de mâncare. Cu cât Pacman este mai aproape de mâncare, cu atât scorul va fi mai mare.
4. Se adaugă un scor semnificativ (+500) dacă poziția curentă a lui Pacman se află într-o capsulă, indicând un avantaj strategic atunci când Pacman mănâncă o capsulă și devine capabil să înfrunte ghost-urile.