UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de căutare și agenți adversariali

Inteligență Artificială

Autor: Sabău Oana-Maria
Grupa: 30232

Facultatea de Automatică
și Calculatoare

7 Decembrie 2023

Cuprins

1 Introducere

1.1 Prezentare

Inteligența artificială reprezintă simularea proceselor de inteligență umană de către mașini, în special de către sisteme informatice. Printre aplicațiile specifice ale inteligenței artificiale se numără sistemele experte, prelucrarea limbajului natural, recunoașterea vorbirii și vederea artificială.[6]

Tehnicile de căutare sunt metode universale de rezolvare a problemelor în inteligența artificială. Aceste strategii sau algoritmi de căutare au fost, în general, utilizate de agenții raționali sau de agenții de rezolvare a problemelor din AI pentru a rezolva o anumită problemă și a oferi cel mai bun rezultat. [10]

O problemă de căutare este compusă din: un spațiu de stări (setul tuturor stărilor posibile în care vă puteți afla), o stare de pornire (starea de la care începe căutarea), o stare de scop (o funcție care analizează starea curentă și returnează dacă aceasta este sau nu starea de scop). Soluția la o problemă de căutare este o secvență de acțiuni, numită plan, care transformă starea de pornire în starea obiectiv. Acest plan se realizează prin intermediul algoritmilor de căutare. [7]

Problemele de căutare reprezintă un domeniu de bază al AI care implică găsirea unei soluții pentru o anumită problemă prin explorarea unui spațiu de stări posibile. Agenții de căutare utilizează algoritmi precum căutarea în adâncime, căutarea în lățime, căutarea cu cost uniform, căutarea euristică etc., pentru a găsi soluții optime sau satisfăcătoare pentru anumite probleme. Aceste probleme pot fi exprimate ca grafuri sau spații de stări în care agenții caută un drum de la un nod de start către unul sau mai multe noduri țintă, utilizând diverse metode și euristică pentru a găsi cea mai bună soluție în funcție de criterii precum timpul, spațiul și optimizarea soluției.

Agenții adversariali sunt entități care acționează într-un mediu competitiv, având scopuri opuse sau în competiție directă unul cu celălalt. În agenții adversariali, algoritmi precum algoritmul minimax sunt folosiți pentru a lua decizii care să maximizeze beneficiile proprii și să minimizeze pierderile într-un mediu competitiv, cum ar fi jocurile, negocierea și securitatea cibernetică. Capacitatea de a gândi strategic și de a anticipa mișcările adversarului este crucială în aceste contexte.

1.2 Motivație

Problemele de căutare și agenții adversariali reprezintă două componente importante ale Inteligenței Artificiale, motivând la dezvoltarea de algoritmi și metode care permit agenților să exploreze spațiile de stări și să ia decizii în medii competitive sau complexe. Dezvoltarea și optimizarea algoritmilor în aceste domenii permit crearea de aplicații practice în diverse domenii, cum ar fi roboții autonomi, sistemele de planificare automată, jocurile video și sistemele de securitate cibernetică. Capacitatea de a gândi strategic și de a acționa în medii competitive sau complexe reprezintă unul dintre pilonii fundamentali în dezvoltarea IA și a aplicațiilor sale viitoare.

Scopul acestui proiect este de a implementa aceste tehnici descrise mai sus cu scopul de a avea un joc Pacman corect funcțional și de a dobândi cunoșințe noi în acest domeniu.

2 Căutare neinformată

Algoritmii de căutare neinformată nu au alte informații suplimentare despre nodul obiectiv decât cele furnizate în definiția problemei. Planurile de atingere a stării obiectivului din starea de pornire diferă doar prin ordinea și/sau lungimea acțiunilor. Căutarea neinformată se mai numește și "Blind search". Acești algoritmi pot genera doar succesorii și nu pot face diferența între goal state și non-goal state. [7]

2.1 Întrebarea 1 - Căutarea în adâncime

Algoritmul de căutare în adâncime (Depth-first Search - DFS) în inteligența artificială este ca un explorator. Este un algoritm de traversare a grafurilor care începe dintr-un punct de plecare, verifică mai întâi locurile din apropiere și continuă să meargă mai adânc înainte de a se deplasa în locuri noi. Acesta repetă acest model pentru a explora întregul graf. Atunci când DFS ajunge la un punct în care nu mai există căi neexplorate în iterația curentă, se întoarce la nodul anterior și încearcă o altă cale. În acest scop, utilizează o structură de date numită stivă (folosește principiul LIFO - Last In First Out) pentru a ține evidența nodului pe care trebuie să-l viziteze în următoarea etapă de căutare. [13]

Codul algoritmului de căutare în adâncime este prezentat mai jos și a fost scris în cadrul laboratorului de Inteligență Artificială, alături de doamna profesor. Făcând mici modificări acestui cod (schimbarea structurii de date pentru frontieră, folosirea condiției la scoatere din frontieră), obținem ceilalți algoritmi propuși.

```
def depthFirstSearch(problem: SearchProblem):
    frontier=util.Stack()
    explored=[]
    frontier.push((problem.getStartState(), []))
    while not frontier.isEmpty():
        current, path=frontier.pop()
        if problem.isGoalState(current):
            return path
        else:
            explored.append(current)
            for ns, a, _ in problem.getSuccessors(current):
                if ns not in explored:
                    frontier.push((ns, path+[a]))
```

Implementarea algoritmului dat implică utilizarea structurilor de date preexistente. Acesta returnează drumul până la goal state. Fiind un algoritm care se aplică pe grafuri neponderate, nu luăm în considerare atributul „cost".

Algoritmul nu este nici comlet, nici optimal și expandează un număr considerabil de noduri pentru a ajunge la soluția dorită, însă este unul dintre algoritmii clasici de căutare și traversare penytru grafuri.

2.2 Întrebarea 2 - Căutarea în lățime

Algoritmul de căutare în lățime (Breadth-first Search - BFS) pornește de la un anumit vârf sursă și explorează graful, examinând toate vârfurile de la același nivel înainte de a coborî mai departe. Acesta utilizează o structură de date de tip coadă (care folosește principiul FIFO - First In First Out) pentru a menține ordinea de explorare, asigurându-se că nodurile sunt vizitate înainte de a trece la copiii acestora. Există mai multe motive pentru care utilizarea

algoritmului BFS este esențială: găsirea celei mai scurte căi (BFS garantează cea mai scurtă cale între două vârfuri într-un graf neponderat, ceea ce îl face o alegere ideală pentru planificarea rutelor sau pentru sistemele de navigație), completitudine (BFS este complet pentru grafurile finite, garantând că explorează întregul graf și vizitează toate vârfurile accesibile), utilizare minimă a memoriei (utilizează o cantitate minimă de memorie în comparație cu alți algoritmi de traversare a grafurilor, cum ar fi DFS, deoarece trebuie doar să stocheze vârfurile în coadă), soluție optimă și acuratețe. [12]

Codul pentru implementarea algoritmului de căutare în lățime este descris mai jos:

```
    def breadthFirstSearch(problem: SearchProblem):
    "Modified the previous code: use Queue instead of Stack"

    frontier=util.Queue()
    explored=[]
    frontier.push((problem.getStartState(), []))
    while not frontier.isEmpty():
        current, path=frontier.pop()
        if problem.isGoalState(current):
            return path
        else:
            if current not in explored:
                explored.append(current)
                for ns, a, _ in problem.getSuccessors(current):
                    if ns not in explored:
                        frontier.push((ns, path + [a]))
```

Notă: algoritmul de mai sus și următorii doi s-au obținut prin varierea algoritmului de căutare în adâncime, anume modificarea structurii de date pentru frontieră.

Spre deosebire de algoritmul DFS, BFS este atât complet, cât și optimal, însă marele lui dezavantaj îl reprezintă spațiul de memorie.

2.3 Întrebarea 3 - Căutare cu cost uniform

Căutarea cu cost uniform (Uniform Search Cost - UCS) este un algoritm de căutare neinformată care utilizează cel mai mic cost cumulativ pentru a găsi o cale de la sursă la destinație. Nodurile sunt dezvoltate, începând de la rădăcină, în funcție de costul cumulativ minim. Căutarea cu costuri uniforme este implementată folosind o coadă de priorități (priority queue). [3] Este o variație a algoritmului lui Dijkstra. [15][14]

Codul pentru implementarea algoritmului de căutare cu cost uniform este descris mai jos:

```
def uniformCostSearch(problem: SearchProblem):

    frontier=util.PriorityQueue()
    explored=[]
    frontier.push((problem.getStartState(), []), 0)
    while not frontier.isEmpty():
        current, path=frontier.pop()
        if problem.isGoalState(current):
            return path
        else:
            if current not in explored:
                explored.append(current)
```

```
            for ns, a, c in problem.getSuccessors(current):
                if ns not in explored:
                    frontier.push((ns, path+[a]), problem.getCostOfActions(path+[a]))
```

Pentru implementarea acestui algoritm se ia în considerare și costul pentru fiecare muchie deoarece se va determina costul minim de la nodul de start la goal state. În loc de „clasicele" coadă și stivă, utilizăm coadă de priorități. Se poate remarca că acest algoritm este mult mai eficient decât primii doi deoarece expandează mai puține noduri. UCS este un algoritm complet și optimal dacă îndeplinește unele condiții, însă timpul de execuție este destul de crescut.

3 Căutare informată

În acest caz, algoritmii dispun de informații despre starea obiectivului, ceea ce ajută la o căutare mai eficientă. Aceste informații sunt obținute prin intermediul unui element numit euristică. [7]

3.1 Întrebarea 4 - Algoritmul de căutare A*

Algoritmul de căutare A* este un algoritm de căutare simplu și eficient care poate fi utilizat pentru a găsi calea optimă între două noduri dintr-un graf. Acesta va fi utilizat pentru găsirea celei mai scurte căi. A* utilizează, de asemenea, o funcție euristică care furnizează informații suplimentare cu privire la distanța la care ne aflăm față de nodul de destinație. [8]

Implementarea algoritmului de căutare A* în limbaj Python este descrisă mai jos:

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):

    frontier = util.PriorityQueue()
    explored = []
    frontier.push((problem.getStartState(), []), 0)
    while not frontier.isEmpty():
        current, path = frontier.pop()
        if problem.isGoalState(current):
            return path
        else:
            if current not in explored:
                explored.append(current)
                for ns, a, c in problem.getSuccessors(current):
                    if ns not in explored:
                        frontier.push((ns, path+[a]),
                            (problem.getCostOfActions(path+[a]) + heuristic(ns,
                            problem)))
```

O observație importantă asupra implementării codului este că este aproape identică cu cea a algoritmului de căutare cu cost uniform, doar că la cost se mai adaugă și euristica, iar calea până la goal state va avea suma minimă. Acest algoritm este cel mai eficient de până acum: în timp ce UCS expandează 620 de noduri, A* expandează doar 549, salvând timp și memorie.

Algoritmul A* oferă mai multe avantaje: garantează găsirea căii optime atunci când este utilizat cu euristica adecvată, este eficient și poate gestiona spații de căutare mari prin eliminarea eficientă a căilor nepromițătoare, poate fi adaptat cu ușurință pentru a se potrivi cu diferite domenii de probleme și euristici și este flexibil și adaptabil la costuri sau constrângeri variabile ale terenului. În plus, este un algoritm complet și optimal, folosit pe scară largă. [8]

3.2 Întrebarea 5 - Găsirea tuturor celor 4 colțuri

În fiecare colț al labirintului din codul Pacman există câte un punct (dot sau mai bine-zis mâncare), în total 4. Problema de căutare este reprezentată de găsirea celei mai scurte căi prin labirint care atinge toate cele patru colțuri (indiferent dacă labirintul are sau nu mâncare acolo). Se cere să fie implementat în prealabil algoritmul de căutare în lățime (lucru deja făcut).

Pentru întreaga funcționalitate, vom completa cod în metodele din clasa CornersProblem.

```
def getStartState(self):

        """ We will return the starting position of the Pacman and the location of the
            corners - I'll return it as a list (as specified in Hint 1) """
        return self.startingPosition, [self.corners[i] for i in range(4)]

    def isGoalState(self, state: Any):

        """ The function takes in a state parameter, which is expected to be a tuple
            consisting of Pacman's
         current position and a list representing the remaining unvisited corners.
         state[0] represents the current position and state[1] is the list of the
             unvisited corners
         if all the corners are visited, the length of the list is 0, that means we've
             got to the goal state, so it is True, False otherwise """

        return len(state[1]) == 0

    def getSuccessors(self, state: Any):

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
            Directions.WEST]:

            """ I've implemented based on the hint above and getSuccesor method """

            x,y = state[0] # the current position
            unvisited_corners = state[1] # the list of the unvisited corners
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            next_coordination = (nextx, nexty)
            hitWalls = self.walls[nextx][nexty]
            if not hitWalls: # check if the next position hits a wall
                # if the next position is an unvisited corner, update the unvisited
                    corners list
                if next_coordination in unvisited_corners:
                    unvisited_corners = [corner for corner in unvisited_corners if
                        corner != next_coordination]
                next_state = (next_coordination, unvisited_corners)
                successors.append((next_state, action, 1))

        self._expanded += 1 # DO NOT CHANGE
        return successors
```

Metoda *getStartState(self)* ne retrunează poziția de start a lui Pacman și o listă cu colțurile (eng. corners) labirintului. A doua metodă, *isGoalState(self, state: Any)*, returnează *True* dacă toate colțurile au fost vizitate, *False* contrar. Metoda *getSuccessors(self, state: Any)* urmărește să găsească stări succesoare pentru starea curentă în jocul Pacman, verifică validitatea acestora, actualizează starea pe baza acțiunii efectuate și menține o listă de colțuri nevizitate, generând în același timp stări succesoare legale.

3.3 Întrebarea 6 - Euristică pentru Corners Problem

O euristică (*eng. heuristic*) este o funcție care determină cât de aproape este o stare de starea dorită. Funcțiile euristice variază în funcție de problemă și trebuie să fie adaptate pentru a corespunde provocării respective. [11]

Pentru a fi admisibilă, valoarea euristică trebuie să fie o limită inferioară a costului real al drumului cel mai scurt până la cel mai apropiat obiectiv (și să nu fie negativă). Pentru a fi consecventă, trebuie să se aplice, în plus, că, dacă o acțiune are un cost c, atunci întreprinderea acestei acțiuni poate provoca o scădere a euristicii de cel mult c. Admisibilitatea nu este suficientă pentru a garanta corectitudinea în căutarea în grafuri - este nevoie de o condiție mai puternică de consistență. Cu toate acestea, euristicile admisibile sunt, de obicei, și consecvente, în special dacă sunt derivate din relaxări ale problemei. Prin urmare, de obicei, este mai ușor să începeți prin a face un brainstorming de euristici admisibile. Singura modalitate de a garanta coerența este o demonstrație. [2]

Date fiind acestea, euristica aleasă reprezintă maximul distanței Manhattan calculate de la poziția de start a lui Pacman până la unul dintre colțurile unde se află mâncare. Dacă nu se găsește, euristica este 0. Aceasta este admisibilă și consistentă. Implementarea metodei *cornersHeuristic(state: Any, problem: CornersProblem)* este descrisă mai jos:

```
def cornersHeuristic(state: Any, problem: CornersProblem):

    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

    # an admissible heuristic could and should be considered the manhattan distance

    position=state[0] # the position of Pacman
    corners=state[1] # the list of unvistied corners
    if len(corners) == 0: # if the goal state is reached
        heuristic = 0
        return heuristic


    manhattan_distances=[util.manhattanDistance(position, c) for c in corners] #
        compute the manhattan distances for each unvisited corner

    """ the heuristic is the biggest value computed above because it is the most
        appropiate value for all the nodes """


    heuristic = max(manhattan_distances)
    return heuristic
```

3.4 Întrebarea 7 - Mănâncă toată mâncarea (Eating All The Dots)

Scopul jocului Pacman este ca „personajul” eponim, Pacman, să mănânce toată mâncarea, reprezentată de puncte (dots), din labirint. Trebuie să găsim o euristică eficientă care să reducă semnificativ numărul de noduri extinse în timpul procesului de căutare, fiind în același timp suficient de consistentă și admisibilă pentru a ghida algoritmul de căutare către soluții optime sau aproape optime. Performanța euristicii este evaluată pe baza capacității sale de a extinde mai puține noduri și de a găsi totuși o soluție.

Implementarea se regăsește mai jos:

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):

position, foodGrid = state

"""I've adapted the code from the method cornersHeuristic(state: Any, problem:
    CornersProblem)"""

list_of_food = foodGrid.asList()
"""check if is any remaining food (dots) in the maze"""
if len(list_of_food)==0:
    heuristic=0
    return heuristic

"""It works computing the heuristic using the manhattan distance, but it passes 3
    of 4 tests"""

# manhattan_distances = [util.manhattanDistance(position, f) for f in
    list_of_food] # compute the manhattan distances for each unvisited corner
# heuristic = max(manhattan_distances)

""" I've used the mazeDistance which takes as arguments the position of the
    Pacman, the food location for each dot from the grid and the starting game
    position """

maze_distances = [mazeDistance(position, f, problem.startingGameState) for f in
    list_of_food]
"""the heuristic is the biggest value computed above because it is the most
    appropiate value for all the nodes"""

heuristic = max(maze_distances)
return heuristic
```

M-am folosit de implementarea metodei *cornersHeuristic(state: Any, problem: CornersProblem)*. Inițial am considerat euristica drept maximul distanței Manhattan, însă expanda un număr considerabil de noduri, deci am găsit o altă soluție: ne folosim de funcția *mazeDistance(point1: Tuple[int, int], point2: Tuple[int, int], gameState: pacman.GameState) -¿ int* care ne va computa o euristică mult mai precisă, iar numărul de noduri expandat este de cel mult 7000. Astfel, am găsit o euristică admisibilă și consistentă pentru problema noastră de căutare.

3.5 Întrebarea 8 - Căutare suboptimală

Această sarcină se concentrează în primul rând pe implementarea unui algoritm eficient de identificare a celei mai scurte căi către cel mai apropiat punct (mâncare).

Sarcina se rezolvă în puține linii de cod (mai exact 3!), dat fiind faptul că implementarile necesare sunt deja făcute în prealabil.

```
    def findPathToClosestDot(self, gameState: pacman.GameState):

        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        """ Our agent solves this maze in under a second with a path cost of 350,
            using any of the search problem below """
        """ I'll go with the A* search because it guarantees that the path found is
            optimal and it is complete and optimal """

        # return search.bfs(problem)
        # return search.ucs(problem)

        return search.astar(problem)


class AnyFoodSearchProblem(PositionSearchProblem):


    def isGoalState(self, state: Tuple[int, int]):

        x,y = state
        list_of_food=self.food.asList()
        return state in list_of_food # if the current state is a dot (food) in the maze
```

Putem considera problema de căutare ca fiind algoritmul A* deoarece găsește calea optimală și, în același timp, este complet și optimal. Verificăm dacă atingem starea de goal dacă starea curentă coincide cu poziția unui punct de mâncare, astfel știm că Pacman ajunge la ea.

4 Căutare adversarială

În domeniile de inteligență artificiale (printre care deep learning machine learning și computer vision) căutarea adversă este, în principiu, un tip de căutare în care se poate urmări mișcarea unui inamic sau a unui adversar. Astfel de căutări sunt importante în șah, în instrumentele de strategie de afaceri, în site-urile de tranzacționare și în jocurile bazate pe război în care există agenți AI.

În cazul unei căutări adversative, utilizatorul poate modifica starea actuală, dar nu are control asupra etapei următoare. În ceea ce privește căutările de acest tip în inteligența artificială, căutarea adversarială este unul dintre cele mai importante tipuri de căutare. Controlul stării următoare se află în mâinile adversarului și este imprevizibil. [1]

4.1 Întrebarea 9 - Agent Reflex

Inteligența artificială este definită ca fiind studiul agenților raționali. Un agent rațional poate fi orice lucru care ia decizii, cum ar fi o persoană, o firmă, o mașină sau un software. Acesta efectuează o acțiune cu cel mai bun rezultat după ce ia în considerare percepțiile trecute și actuale (intrările perceptuale ale agentului la un moment dat).

Agenții reflexi simpli ignoră restul istoricului perceptului și acționează doar pe baza percepției curente. Funcția agentului se bazează pe regula condiție-acțiune. O regulă de condiție-acțiune este o regulă care pune în corespondență o stare, adică o condiție, cu o acțiune. Această funcție a agentului reușește doar atunci când mediul este complet observabil.

Agentul bazat pe model poate gestiona medii parțial observabile prin utilizarea unui model despre lume. Agentul trebuie să țină evidența stării interne, care este ajustată de fiecare percept și care depinde de istoricul perceptului. Starea curentă este stocată în interiorul agentului, care păstrează un anumit tip de structură care descrie partea din lume care nu poate fi observată. [9]

Metoda *evaluationFunction(self, currentGameState: GameState, action)* urmărește să ghideze acțiunile lui Pacman prin evaluarea stărilor succesoare pe baza distanțelor față de mâncare și fantome, încurajându-l pe Pacman să acorde prioritate ajungerii la mâncare, evitând în același timp să se apropie de fantome. Ajustarea valorilor de recompensă și penalizare poate influența comportamentul lui Pacman în diferite situații. Astfel, am îmbunătățit această metodă prin adăugarea de noi funcționalități: calcularea scorului în funcție de fantome, calcularea distanței Manhattan pentru fiecare dot și pentru fantome în timp real, conducând la câștigul lui Pacman.

```
def evaluationFunction(self, currentGameState: GameState, action):

    successorGameState = currentGameState.generatePacmanSuccessor(action)
    newPos = successorGameState.getPacmanPosition()
    newFood = successorGameState.getFood()
    newGhostStates = successorGameState.getGhostStates()
    newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

    score = successorGameState.getScore()
    list_of_food = newFood.asList()
    manhattan_distances_for_food=[util.manhattanDistance(newPos, f) for f in
        list_of_food]
    manhattan_distances_for_ghosts=[util.manhattanDistance(newPos,
        g.getPosition()) for g in newGhostStates]
    min_ghost_distance = min(manhattan_distances_for_ghosts)
    if len(list_of_food) > 0:
        min_food_distance = min(manhattan_distances_for_food)
    else:
        min_food_distance = 0 # no more food

    #compute the score
    if action == "Stop":
        score -= 100
    score += 1/(min_food_distance+1)
    score -= 1/(min_ghost_distance+1)
    return score
```

4.2 Întrebarea 10 - Minimax

Algoritmul Minimax este un algoritm recursiv sau de backtracking care este utilizat în procesul decizional și în teoria jocurilor. Acesta oferă o mutare optimă pentru jucător, presupunând că adversarul joacă, de asemenea, în mod optim. Algoritmul utilizează recursivitatea pentru a căuta în arborele de joc. În acest algoritm, sunt doi jucători, unul se numește MAX, iar celălalt se numește MIN, iar ambii jucători se luptă, unde MAX va selecta valoarea maximizată, iar MIN va selecta valoarea minimizată. Minimax realizează un algoritm de căutare în adâncime pentru explorarea arborelui complet al jocului și merge până la nodul terminal al arborelui, apoi reface traseul arborelui sub formă de recursivitate. [5]

Algoritmul este implementat în clasa MinimaxAgent dupa pseudocodul din cursul de Inteligență Artificială și au fost adăugate funcționalități pentru a fi corect:

```
class MinimaxAgent(MultiAgentSearchAgent):

    """ The implementation of the functions is based on the pseudocode and further
        information is adder for the functionality """

    def max_value(self, gameState, depth):

        # check initial conditions
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState)
        v = float("-inf")
        for a in gameState.getLegalActions(0):
            s = gameState.generateSuccessor(0, a)
            v = max(v, self.min_value(s, 1, depth))
        return v

    def min_value(self, gameState, agentIndex, depth):

        # check initial conditions
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState)
        v = float("inf")
        for a in gameState.getLegalActions(agentIndex):
            s = gameState.generateSuccessor(agentIndex, a)
            if agentIndex == gameState.getNumAgents()-1:
                v = min(v, self.max_value(s, depth+1))
            else:
                v = min(v, self.min_value(s, agentIndex+1, depth))
        return v

    def getAction(self, gameState: GameState):

        # the algorithm is applied for the player - Pacman - represented by max

        legalActions = gameState.getLegalActions(0)
        bestScore = float("-inf")
        bestAction = None
        for a in legalActions:
            s = gameState.generateSuccessor(0, a)
            score = self.min_value(s, 1, 0)
```

```
            if score > bestScore:
                bestScore = score
                bestAction = a
        return bestAction
```

Funcția *max-value* reprezintă mutarea maximă a jucătorului în algoritmul Minimax. Aceasta caută valoarea maximă dintre acțiunile posibile pe care Pacman le poate întreprinde, având în vedere starea jocului și adâncimea dată. Funcția *min-value* reprezintă mutarea minimă a jucătorului în algoritmul Minimax. Aceasta simulează acțiunile agenților adversari (fantome) și încearcă să minimizeze valoarea potențială obținută de Pacman. Funcția *getAction* este responsabilă de selectarea celei mai bune acțiuni pentru Pacman. Ea inițiază căutarea prin apelarea *min-value* pentru următorul nivel al agentului (fantomă) și ține evidența celei mai bune acțiuni și a scorului acesteia. șadar, clasa *MinimaxAgent* implementează algoritmul Minimax pentru a lua decizii pentru Pacman în joc prin explorarea recursivă a arborelui jocului și alegerea acțiunii care maximizează șansele de câștig ale lui Pacman.

Algoritmul Minimax este complet și va găsi cu siguranță o soluție (dacă există), în arborele de căutare finit. Este optim dacă ambii adversari joacă în mod optim. Deoarece realizează DFS pentru arborele de joc, complexitatea în timp a algoritmului Minimax este O(b la puterea m), unde b este factorul de ramificare al arborelui de joc, iar m este adâncimea maximă a arborelui, iar complexitatea spațială a algoritmului este, de asemenea, similară cu DFS, care este O(bm). [5]

4.3 Întrebarea 11 - Alpha-Beta Pruning

Alpha-beta Pruning este o versiune modificată a algoritmului Minimax. Este o tehnică de optimizare a algoritmului Minimax. După cum am văzut în cazul algoritmului de căutare minimax, numărul de stări de joc pe care trebuie să le examineze este exponențial în funcție de adâncimea arborelui. Deoarece nu putem elimina exponentul, dar îl putem reduce la jumătate. Prin urmare, există o tehnică prin care, fără a verifica fiecare nod al arborelui de joc, putem calcula decizia minimax corectă, iar această tehnică se numește pruning (tăiere). Aceasta implică doi parametri de prag, alpha și beta, pentru o viitoare expansiune, de aceea se numește pruning alpha-beta. Se mai numește și Algoritmul Alpha-Beta. Tăierea alpha-beta poate fi aplicată la orice adâncime a unui arbore și, uneori, nu numai frunzele arborelui, ci și întregul subarbore. Alpha reprezintă cea mai bună alegere (cea mai mare valoare) pe care am găsit-o până acum în orice punct de pe traseul Maximizer. Valoarea inițială a lui alpha este -infinit, iar beta este cea mai bună alegere (cea mai mică valoare) pe care am găsit-o până acum în orice punct de pe traseul minimizatorului. Valoarea inițială a lui beta este +infinit. [4]

Acest algoritm este implementat în aceeași manieră ca și algoritmul Minimax, dar adăugăm funcționalitate adițională pentru alpha și beta:

```
class AlphaBetaAgent(MultiAgentSearchAgent):

    """ The code for this algorithm is obtained by modifying the Minimax algorithm, we
        need to add alpha and beta """

    def max_value(self, gameState, depth, alpha, beta):

        # check initial conditions
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
```

```
            return self.evaluationFunction(gameState)
        v = float("-inf")
        for a in gameState.getLegalActions(0):
            s = gameState.generateSuccessor(0, a)
            v = max(v, self.min_value(s, 1, depth, alpha, beta))
            if v > beta:
                return v
            alpha = max(alpha, v)
        return v

    def min_value(self, gameState, agentIndex, depth, alpha, beta):

        # check initial conditions
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState)
        v = float("inf")
        for a in gameState.getLegalActions(agentIndex):
            s = gameState.generateSuccessor(agentIndex, a)
            if agentIndex == gameState.getNumAgents()-1:
                v = min(v, self.max_value(s, depth+1, alpha, beta))
            else:
                v = min(v, self.min_value(s, agentIndex+1, depth, alpha, beta))
            if v < alpha:
                return v
            beta = min(beta, v)
        return v

    def getAction(self, gameState: GameState):

        legalActions = gameState.getLegalActions(0)
        bestScore = float("-inf")
        bestAction = None
        alpha = float("-inf")
        beta = float("inf")
        for a in legalActions:
            s = gameState.generateSuccessor(0, action)
            score = self.min_value(s, 1, 0, alpha, beta)
            if score > bestScore:
                bestScore = score
                bestAction = a
            alpha = max(alpha, bestScore)
        return bestAction
```

Alpha-beta Pruning, asemenea algoritmului standard de Minimax, returnează aceeași mutare ca și acesta, dar elimină toate nodurile care nu afectează cu adevărat decizia finală, ci fac algoritmul lent. Prin urmare, prin eliminarea acestor noduri, algoritmul devine rapid. Eficacitatea tăierii alfa-beta în algoritmul minimax depinde de ordinea în care sunt examinate nodurile. Există două tipuri de ordine: cea mai proastă ordine și ordinea ideală. Pentru prima ordine, algoritmul nu taie niciun nod și funcționează ca algoritmul Minimax. Aceasta poate duce la un timp de execuție mai mare și se numește "worst ordering". În această situație, opinie de lucru algoritmul este în partea dreaptă a arborelui. Ordinea ideală este atunci când se produce un număr mare de tăieri în arbore, iar cea mai bună mutare este în partea stângă a arborelui.

Acest lucru poate fi realizat utilizând o căutare DFS, care explorează mai întâi partea stângă a arborelui și se adâncește de două ori mai mult decât algoritmul minimax. [4]

Referințe

[1] Algoscale, *Why you need to know about Adversarial Search in AI?*, URL: https://algoscale.com/blog/why-you-need-to-know-about-adversarial-search-in-ai/.

[2] Berkeley, *Corners Problem: Heuristic*, URL: https://inst.eecs.berkeley.edu/~cs188/sp23/projects/proj1/#q5-3-pts-finding-all-the-corners.

[3] Fatima Hasan, *What is uniform-cost search?*, URL: https://www.educative.io/answers/what-is-uniform-cost-search.

[4] JavaTPoint, *Alpha-Beta Pruning*, URL: https://www.javatpoint.com/ai-alpha-beta-pruning.

[5] JavaTPoint, *Mini-Max Algorithm in Artificial Intelligence*, URL: https://www.javatpoint.com/mini-max-algorithm-in-ai.

[6] Nicole Laskowski, *artificial intelligence (AI)*, URL: https://www.techtarget.com/searchenterpriseai/definition/AI-Artificial-Intelligence.

[7] MdRafiAkhtar, *Search Algorithms in AI*, URL: https://www.geeksforgeeks.org/search-algorithms-in-ai/.

[8] Ravikiran A S, *A* Algorithm Concepts and Implementation*, URL: https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm.

[9] Sahil$_{B}ansall$, *Agents in Artificial Intelligence*, URL: https://www.geeksforgeeks.org/agents-artificial-intelligence/.

[10] *Search Algorithms in Artificial Intelligence*, URL: https://tutorialforbeginner.com/search-algorithms-in-ai.

[11] Simplilearn, *Introduction To The Heuristic Function In AI*, URL: https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/heuristic-function-in-ai.

[12] Pavan Vadapalli, *What is BFS Algorithm? Breath First Search Algorithm Explained*, URL: https://www.upgrad.com/blog/what-is-breath-first-search-algorithm/.

[13] *What is DFS in Artificial Intelligence?*, URL: https://intellipaat.com/blog/what-is-dfs-in-artificial-intelligence/.

[14] Wikipedia, *Dijkstra's algorithm*, URL: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[15] Wikipedia, *Edsger W. Dijkstra*, URL: https://en.wikipedia.org/wiki/Edsger_W._Dijkstra.