UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Laura Darolti, Vlad Scuturici
Grupa: 30235

Facultatea de Automatica
si Calculatoare

30 Decembrie 2023

Cuprins

1 Uninformed search

1.1 Question 1 - Depth-first search

DFS (Depth-First Search) este un algoritm utilizat pentru explorarea structurilor de tip arbore sau graf. Algoritmul începe de la un punct de plecare, care în cazul nostru este poziția de start a lui Pacman, și caută cât mai în adâncime în labirint pentru a găsi hrana, până când se întoarce prin backtracking. Deoarece structura este un arbore, algoritmul păstrează succesori într-o stivă, deoarece ultimul nod adăugat este primul care trebuie extras pentru a realiza o parcurgere cât mai adâncă. În cazul în care destinația nu este atinsă cu prima rută, algoritmul se întoarce de la cea mai apropiată bifurcație sau se continuă să caute un traseu alternativ.

```
def depthFirstSearch(problem: SearchProblem):
    visited = []
    path = []
    nextPath = []
    stack = util.Stack()
    stack.push((problem.getStartState(), path))
    while(not stack.isEmpty()):
        node, path = stack.pop()
        if problem.isGoalState(node):
            return path
        if node not in visited:
            visited.append(node)
            for successor, action, stepCost in problem.getSuccessors(node):
                if successor not in visited:
                    nextPath = path + [action]
                    stack.push((successor, nextPath))
```

1.2 Question 2 - Breadth-first search

BFS (Breadth-First Search) este un algoritm folosit pentru traversarea în lățime a unui arbore sau graf. Acesta începe de la un nod de start și explorează toți vecinii nodului curent înainte de a se deplasa la nivelul următor. Algoritmul utilizează o coadă pentru a memora nodurile și acțiunile asociate pe măsură ce sunt descoperite. Se inițializează o coadă goală, o listă de noduri vizitate și se obține nodul de start. Dacă nodul de start este și nodul țintă, se returnează o listă goală, deoarece nu este necesară nicio acțiune pentru a ajunge la țintă. Se adaugă nodul de start și o listă goală de acțiuni în coadă. În timp ce coada nu este goală: a. Se extrage un nod și acțiunile asociate din fața cozii. b. Dacă nodul nu a fost vizitat: Se adaugă nodul la lista de noduri vizitate. Dacă nodul este nod țintă, se returnează lista de acțiuni asociate. Altfel, se adaugă succesori și acțiunile corespunzătoare în coadă. Dacă coada devine goală și ținta nu a fost găsită, se returnează o listă goală. Algoritmul explorează succesiv nodurile de la nivelul curent înainte de a trece la nivelul următor, asigurându-se că se găsește cea mai scurtă cale către țintă.

```
def breadthFirstSearch(problem: SearchProblem):
    queue = util.Queue()
    visited = []
    start_node = problem.getStartState()

```

```
    if problem.isGoalState(start_node):
        return []

    queue.push((start_node, []))

    while not queue.isEmpty():

        n, actions = queue.pop()
        if n not in visited:
            visited.append(n)

            if problem.isGoalState(n):
                return actions

            for successor, action, cost in problem.getSuccessors(n):
                queue.push((successor, actions + [action]))
```

1.3 Question 3 - Uniform-Cost Search

Algoritmul UniformCostSearch implementează o căutare uniformă într-un spațiu de stări, prioriząnd explorarea nodurilor în ordinea costurilor crescătoare. În primul rând, se inițializează o coadă de priorități și o listă pentru nodurile vizitate. Se adaugă nodul de start în coadă cu costul total zero, iar apoi, într-o buclă, se extrag nodurile cu cel mai mic cost total și se adaugă succesori nevizitați în coadă cu noile costuri calculate. Procesul continuă până când un nod scop este găsit sau coada devine goală. Algoritmul utilizează o coadă de priorități pentru a asigura extragerea nodurilor în ordinea costurilor crescătoare, oferind o strategie eficientă de căutare.

```
def uniformCostSearch(problem: SearchProblem):
    queue = util.PriorityQueue()
    visited = []
    start_node = problem.getStartState()

    if problem.isGoalState(start_node):
        return []

    print("the start node is:", start_node)

    queue.push((start_node, [], 0), 0)
    # visited.append(start_node)

    while not queue.isEmpty():

        if n not in visited:
            visited.append(n)

            if problem.isGoalState(n):
                return actions

            for successor, action, cost in problem.getSuccessors(n):
```

```
                new_actions = actions + [action]
                if cost is not None:
                    new_cost = final_cost + cost
                    queue.push((successor, new_actions, new_cost), new_cost)
                else:
                    queue.push((successor, new_actions, final_cost), final_cost)
```

2 Informed search

2.1 Question 4 - A* search algorithm

Algoritmul aStarSearch implementează o căutare A* într-un spațiu de stări, combinând căutarea uniformă cu o euristică pentru a estima costurile rămase până la nodul scop. Se utilizează o coadă de priorități pentru a extrage nodurile în ordinea costurilor totale estimate, care includ costul curent și estimarea euristică. Inițial, nodul de start este adăugat în coadă cu costul zero, iar apoi se explorează succesiv succesorii nodurilor scoase din coadă. Dacă un nod scop este atins, se returnează acțiunile asociate, iar în caz contrar, se continuă căutarea până când coada devine goală. Euristică, specificată ca parametru opțional, contribuie la prioritizarea nodurilor în funcție de estimarea costului rămas până la scop. Algoritmul A* integrează astfel avantajele unei căutări informate bazate pe euristică cu strategia de explorare uniformă.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    queue = util.PriorityQueue()
    queue.push((problem.getStartState(), [], 0), 0)
    visited = []

    while not queue.isEmpty():
        state, actions, cost = queue.pop()
        if state not in visited:
            visited.append(state)
            if problem.isGoalState(state):
                return actions
            for successor, action, new_cost in problem.getSuccessors(state):
                queue.push((successor, actions + [action], cost + new_cost), heuristic(succe
```

2.2 Question 5 - Corner Problem

Algoritmul implementat în clasa CornersProblem reprezintă o problemă de căutare a unui traseu care trece prin cele patru colțuri ale unui labirint. Starea este definită de poziția actuală a lui Pacman și de colțurile deja vizitate. Scopul este atins atunci când Pacman ajunge la toate cele patru colțuri, iar succesorii reprezintă pozițiile accesibile din starea curentă, cu acțiunile corespunzătoare și costul incremental de 1. Algoritmul utilizează o listă de colțuri și verifică dacă Pacman a vizitat toate aceste colțuri pentru a determina dacă a atins scopul. Funcția getCostOfActions calculează costul total al unei secvențe de acțiuni, inclusiv o penalizare dacă acțiunile includ mișcări ilegale.

```
class CornersProblem(search.SearchProblem):
    def __init__(self, startingGameState: pacman.GameState):
        self.walls = startingGameState.getWalls()
```

```
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height - 2, self.walls.width - 2
        self.corners = ((1, 1), (1, top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner ' + str(corner))
        self._expanded = 0  # DO NOT CHANGE; Number of search nodes expanded

    def getStartState(self):
        return (self.startingPosition,[])

    def isGoalState(self, state: Any):
        position = state[0]
        visitedCorners = state[1]

        if (position in self.corners) and (len(visitedCorners) == 4):
            return True

        return False

    def getSuccessors(self, state: Any):
        x, y = state[0]
        visitedCorners = state[1]
        successors = []

        actions = [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
        vectors = [Actions.directionToVector(action) for action in actions]

        for vector in vectors:
            dx, dy = vector
            nextx, nexty = int(x + dx), int(y + dy)
            nextState = (nextx, nexty)

            if not self.walls[nextx][nexty]:
                cost = 1
                successorVisitedCorners = visitedCorners + [
                    nextState] if nextState in self.corners and nextState not in visitedCorr
                successors.append(((nextState, successorVisitedCorners), actions[vectors.ind
        self._expanded += 1
        return successors
```

2.3 Question 6 - Corners Heuristic

Algoritmul cornersHeuristic implementează o euristică utilizată în algoritmul A* pentru a estima costul optim de la starea curentă la starea scop în cadrul problemei CornersProblem. Funcția calculează distanța Manhattan între poziția curentă a lui Pacman și colțurile nevizitate. Dacă toate colțurile au fost deja vizitate, euristica returnează 0, indicând că obiectivul a fost atins. Altfel, calculează distanța până la cel mai apropiat colț nevizitat și distanța maximă dintre toate colțurile nevizitate. Rezultatul final este suma acestor două distanțe, sugerând că

Pacman ar trebui să se apropie de cel mai apropiat colț și, implicit, să se îndepărteze de cel mai îndepărtat. Această euristică ajută la ghidarea algoritmului A* în direcția optimă în cadrul căutării.

```
def cornersHeuristic(state, problem):
    xy, visitedCorners = state
    unvisited = [corner for corner in problem.corners if corner not in visitedCorners]

    if not unvisited:
        return 0

    def manhattan_distance_to_corner(corner):
        return util.manhattanDistance(xy, corner)

    closest = min(unvisited, key=manhattan_distance_to_corner)
    farthest = max(unvisited, key=manhattan_distance_to_corner)

    return util.manhattanDistance(xy, closest) + util.manhattanDistance(closest, farthest)
```

2.4 Question 7 - Food Search Problem

Clasa FoodSearchProblem reprezintă o problemă de căutare asociată cu găsirea unui traseu pentru ca Pacman să colecteze toată hrana într-un joc. Aceasta definește starea inițială, starea scop, succesori și costurile acțiunilor pentru a permite algoritmilor de căutare să găsească o soluție optimă. Metodele implementate în clasă permit interacțiunea cu algoritmii de căutare, determinând starea inițială, verificând dacă s-a atins starea scop, generând stările succesoare și calculând costul acțiunilor. Astfel, clasa oferă o interfață pentru a integra problema specifică în cadrul unor algoritmi generali de căutare.

```
def foodHeuristic(state, problem):
    position, grid = state
    foodList = []
    for x in range(grid.width):
        for y in range(grid.height):
            if grid[x][y]:
                foodList.append((x, y))
    maxDis = 0
    for food in foodList:
        distance = mazeDistance(position, food, problem.startingGameState)
        maxDis = max(distance, maxDis)
    return maxDis
```

2.5 Question 8 - Suboptimal Search

Căutarea suboptimă, aplicată în contextul jocului Pacman, implică utilizarea unor algoritmi care găsesc soluții "suficient de bune" în locul celor optime. Aceasta este relevantă în situații unde soluțiile optimale sunt prea costisitoare din punct de vedere al timpului sau resurselor computaționale. În această abordare, se poate ajusta un algoritm ca A* sau căutarea cost uniform, folosind euristici mai puțin precise, dar mai rapide, pentru a reduce complexitatea

calculului. Scopul este de a găsi un traseu eficient, dacă nu cel mai eficient, pentru Pacman, echilibrând astfel între calitatea soluției și resursele necesare pentru a o obține.

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

        return search.breadthFirstSearch(problem)
```

3 Adversarial search

3.1 Question 9 - Improve the ReflexAgent

Clasa ReflexAgent reprezintă un agent reflex care ia decizii la fiecare punct de decizie prin examinarea alternativelor disponibile printr-o funcție de evaluare a stării. Metoda getAction a agentului alege una dintre cele mai bune acțiuni posibile în funcție de funcția de evaluare. Această funcție colectează mișcările legale și stările succesoare, apoi atribuie scoruri fiecărei acțiuni pe baza funcției de evaluare. Mai apoi, alege una dintre cele mai bune acțiuni în mod aleatoriu dintre cele cu scor maxim. Funcția de evaluare (evaluationFunction) este proiectată pentru a lua în considerare informații precum poziția pacman-ului, harta alimentelor, stările fantomelor și starea de sperietoare a acestora. În implementarea specifică, se acordă un scor pe baza distanței la cea mai apropiată hrană și se penalizează dacă o fantoma este prea aproape. Funcția scoreEvaluationFunction returnează doar scorul stării curente și este destinată utilizării cu agenți de căutare adversarială, nu cu agenți reflex.

```
    def evaluationFunction(self, currentGameState: GameState, action):
        foodList = newFood.asList()
        score = successorGameState.getScore()
        if len(foodList):
            closestFoodDist = float('inf')
            for food in foodList:
                distance = manhattanDistance(newPos, food)
                if distance < closestFoodDist:
                    closestFoodDist = distance
            score += 1.0 / closestFoodDist
        ghostDistances = []
        for ghost in newGhostStates:
            ghostPos = ghost.getPosition()
            dist = manhattanDistance(newPos, ghostPos)
            ghostDistances.append(dist)
        if ghostDistances and min(ghostDistances) < 2:
            score -= 10

        return score
```

3.2 Question 10 - Minimax

Clasa MinimaxAgent reprezintă un agent care implementează algoritmul Minimax pentru luarea deciziilor în jocurile cu doi jucători. Metoda getAction a agentului returnează acțiunea optimă determinată de algoritmul Minimax, utilizând o anumită adâncime (self.depth) și o funcție de evaluare (self.evaluationFunction). Algoritmul Minimax explorează recursiv arborele de stări posibile ale jocului, alternând între maximizarea și minimizarea scorurilor, până când se atinge adâncimea maximă sau se ajunge la o stare finală (câștig sau pierdere). Funcția minimax este utilizată pentru a calcula scorurile asociate acțiunilor posibile, iar apoi se returnează acțiunea cu scorul optim în funcție de jucătorul curent.

```
def minimax(self, gameState, depth, agentIndex):
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState), None

        nextAgent = (agentIndex + 1) % gameState.getNumAgents()
        nextDepth = depth + 1 if nextAgent == 0 else depth
        actions = gameState.getLegalActions(agentIndex)

        if not actions:
            return self.evaluationFunction(gameState), None

        scores = []
        for action in actions:
            nextGameState = gameState.generateSuccessor(agentIndex, action)
            score = self.minimax(nextGameState, nextDepth, nextAgent)[0]
            scores.append((score, action))

        if agentIndex == 0:
            return max(scores)
        else:
            return min(scores)
```

3.3 Question 11- Alpha-Beta Pruning

Clasa AlphaBetaAgent reprezintă un agent care implementează algoritmul Minimax cu tăiere alfa-beta pentru luarea deciziilor în jocurile cu doi jucători. Metoda getAction a agentului returnează acțiunea optimă determinată de algoritmul Minimax cu tăiere alfa-beta, utilizând o anumită adâncime (self.depth) și o funcție de evaluare (self.evaluationFunction). Algoritmul Minimax cu tăiere alfa-beta optimizează explorarea arborelui de stări posibile ale jocului prin eliminarea explorării inutile a unor ramuri ale arborelui. Algoritmul utilizează doi parametri, alfa și beta, pentru a ține evidența celor mai bune scoruri ale jucătorilor, reducând astfel numărul de noduri evaluate. Funcția alphaBeta este recursivă și calculează scorurile asociate acțiunilor posibile, utilizând tăiere alfa-beta pentru a opri explorarea în anumite ramuri ale arborelui care nu vor influența decizia finală. Rezultatul returnat este scorul optim și acțiunea asociată acestuia.

```
def alphaBeta(self, gameState, depth, agentIndex, alpha, beta):
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState), None
```

```
4
5           numAgents = gameState.getNumAgents()
6           nextAgent = (agentIndex + 1) % numAgents
7           nextDepth = depth + 1 if nextAgent == 0 else depth
8           actions = gameState.getLegalActions(agentIndex)
9
10          if not len(actions):
11              return self.evaluationFunction(gameState), None
12
13          if agentIndex == 0:
14              value = float('-inf'), None
15              for action in actions:
16                  successorValue = self.alphaBeta(gameState.generateSuccessor(agentIndex, acti
17                  if successorValue > value[0]:
18                      value = successorValue, action
19                  if value[0] > beta:
20                      return value
21                  alpha = max(value[0], alpha)
22          else:
23              value = float('inf'), None
24              for action in actions:
25                  successorValue = self.alphaBeta(gameState.generateSuccessor(agentIndex, acti
26                  if successorValue < value[0]:
27                      value = successorValue, action
28                  if value[0] < alpha:
29                      return value
30                  beta = min(value[0], beta)
31          return value
```

3.4 Question 12 - Expectimax Agent

Expectimax este un algoritm de căutare folosit în jocuri cu mai mulți agenți, unde unii dintre acești agenți (de exemplu, fantomele în Pacman) iau decizii aleatorii, iar alții (ca Pacman) urmăresc să maximizeze un anumit scor sau beneficiu. Algoritmul Expectimax funcționează similar cu Minimax, dar, în loc să presupună că adversarul alege mișcări optime (minimizând scorul jucătorului), consideră că adversarul alege mișcările în mod aleatoriu. În implementarea sa, Expectimax alternează între nivelurile "max" (Pacman) și "min" (fantomele), evaluând starea jocului la fiecare nivel. În nivelurile "min", Expectimax calculează o valoare așteptată, bazată pe toate mișcările posibile ale adversarului și probabilitățile lor. Aceasta face Expectimax mai realist pentru situațiile în care adversarii nu iau neapărat decizii optime, dar adaugă complexitate în calculul deciziilor optime pentru agentul principal.

```
1  class ExpectimaxAgent(MultiAgentSearchAgent):
2
3      def getAction(self, gameState: GameState):
4          action = None
5          v = -float('inf')
6          for a in gameState.getLegalActions(0):
7              if v < self.minValue(gameState.generateSuccessor(0, a), 1, 0):
```

```
                v = self.minValue(gameState.generateSuccessor(0, a), 1, 0)
                action = a
        return action

    def maxValue(self, gameState: GameState, depth):
        if depth == self.depth or len(gameState.getLegalActions(0)) == 0:
            return self.evaluationFunction(gameState)
        else:
            v = -float('inf')
            for a in gameState.getLegalActions(0):
                v = max(v, self.minValue(gameState.generateSuccessor(0, a), 1, depth))
            return v

    def minValue(self, gameState: GameState, agentIndex, depth):
        if len(gameState.getLegalActions(agentIndex)) == 0:
            return self.evaluationFunction(gameState)
        nrPossibleActions = len(gameState.getLegalActions(agentIndex))
        if agentIndex < (gameState.getNumAgents() - 1):
            v = 0
            for a in gameState.getLegalActions(agentIndex):
                v += self.minValue(gameState.generateSuccessor(agentIndex, a), agentIndex +
            return (v / float(nrPossibleActions))
        else:
            v = 0
            for a in gameState.getLegalActions(agentIndex):
                v += self.maxValue(gameState.generateSuccessor(agentIndex, a), depth + 1)
            return (v / float(nrPossibleActions))
```

3.5 Question 13 - Evaluation Function

Funcția de evaluare betterEvaluationFunction pentru jocul Pacman este concepută pentru a calcula eficient valoarea unei stări a jocului, luând în considerare mai mulți factori. În primul rând, funcția calculează un scor bazat pe distanța Manhattan dintre Pacman și toate bucățile de mâncare și capsule rămase, acordând un scor mai mare pentru alimentele și capsulele mai apropiate. Acest lucru încurajează Pacman să se îndrepte către cea mai apropiată hrană și capsule, maximizând eficiența colectării. În plus, scorul global al jocului este amplificat și adăugat la evaluare, ceea ce reflectă progresul curent al jucătorului. Pentru a echilibra aceste scoruri, funcția deduce un anumit punctaj pe baza numărului total de bucăți de mâncare și capsule rămase, motivând Pacman să finalizeze nivelul cât mai repede posibil. Prin combinarea acestor elemente, betterEvaluationFunction oferă o evaluare complexă și dinamică a stării jocului, contribuind la optimizarea strategiilor lui Pacman pentru vânarea fantomelor, colectarea alimentelor și navigarea prin labirint.

```
def betterEvaluationFunction(currentGameState:GameState):

    pacmanPos = currentGameState.getPacmanPosition()
    foodList = currentGameState.getFood().asList()
    foodScore = 0
    if (len(foodList) != 0):
```

```
        for f in foodList:
            foodScore += 1.0 / manhattanDistance(f, pacmanPos)

    capsuleList = currentGameState.getCapsules()
    capScore = 0
    if (len(capsuleList) != 0):
        for c in capsuleList:
            capScore += 1.0 / manhattanDistance(c, pacmanPos)

    score = foodScore + capScore
    score += 8 * currentGameState.getScore()
    score -= 6 * (len(foodList) + len(capsuleList))

    return score
```