UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Szekely Gergo
Grupa: 30237

Facultatea de Automatica
si Calculatoare

6 Ianuarie 2024

Cuprins

1 Uninformed search

1.1 Question 1 - Depth-first search

Cerinta era sa impementam Depth-first search (DFS) din proiectul Pacman-Search in functia *def depthFirstSearch(problem: SearchProblem):*

Algoritmul de parcurgere în adancime Depth-first search (DFS) exploreaza graful sau arborele incepand de la un nod și continua cat mai adanc în fiecare ramura înainte de a reveni și a explora alte ramuri. Este o metodă de cautare în structuri de date ierarhice.

```
def depthFirstSearch(problem: SearchProblem):
    frontier = util.Stack()
    frontier.push((problem.getStartState(), []))

    reached = set()

    while not frontier.isEmpty():

        state, moves = frontier.pop()

        if state not in reached:
            reached.add(state)

            if problem.isGoalState(state):
                return moves

            for successorState, direction, cost in problem.getSuccessors(state):
                frontier.push((successorState, moves + [direction]))

    return []]

```

Pe un bigMaze:

- cost total: 210
- noduri expandate: 390
- score: 300

1.2 Question 2 - Breadth-first search

Cerinta era sa impementam Breadth-first search (BFS) din proiectul Pacman-Search in functia *def breadthFirstSearch(problem: SearchProblem):*

Algoritmul de parcurgere in latime Breadth-first search (BFS) exploreaza graful sau arborele nivel cu nivel, incepand de la un nod sursa, expandandu-se la toti vecinii in ordine si continuand astfel pe nivelurile succesive. Este o metoda eficienta de cautare in structuri de date ierarhice.

```
def breadthFirstSearch(problem: SearchProblem):
    frontier = util.Queue()
    frontier.push((problem.getStartState(), []))

```

```
    reached = set()

    while not frontier.isEmpty():

        state, moves = frontier.pop()

        if state not in reached:
            reached.add(state)

            if problem.isGoalState(state):
                return moves

            for successorState, direction, cost in problem.getSuccessors(state):
                frontier.push((successorState, moves + [direction]))

    return []
```

Pe un bigMaze:

- cost total: 210
- noduri expandate: 620
- score: 300

1.3 Question 3 - Uniform-Const-Search

Cerinta era sa impementam Uniform-Const-Search (UCS) din proiectul Pacman-Serach in functia *def uniformCostSearch(problem: SearchProblem):*

Algoritmul de cautare uniforma constanta exploreaza spațiul de cautare cu costuri constante, selectand nodurile în ordinea crescatoare a costurilor pentru a gasi o soluție optima. Este utilizat în problemele de cautare a cailor cu cost minim în grafuri ponderat.

```
def uniformCostSearch(problem: SearchProblem):
     reached = set()

    while not frontier.isEmpty():

        state, moves, cost = frontier.pop()

        if state not in reached:
            reached.add(state)

            if problem.isGoalState(state):
                return moves

            for successorState, direction, successorCost in problem.getSuccessors(state):
                frontier.push((successorState, moves + [direction], cost + successorCost), c

    return []

```

Pe un mediumScaryMaze :

- cost total: 68719479864
- noduri expandate: 108
- score: 418

2 Informed search

2.1 Question 4 - A* search algorithm

Cerinta era sa impementam algoritmul A* search din proiectul Pacman-Serach in functia *def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):*

Algoritmul de cautare A* este o metoda care exploreaza spatiul de cautare, alegand in mod inteligent urmatoarea stare pe baza costului estimat total al unei cai de la inceput la respectiva stare si a costului real pana in acel moment. Este folosit in problemele de cautare a drumului optim in grafuri ponderate.

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

    while not frontier.isEmpty():

        state, moves, cost = frontier.pop()

        if state not in reached:
            reached.add(state)

            if problem.isGoalState(state):
                return moves

            for successorState, direction, successorCost in problem.getSuccessors(state):
                g = cost + successorCost
                h = heuristic(successorState, problem)
                f = g + h
                frontier.push((successorState, moves + [direction], g), f)

    return []


```

Pe un bigMaze:

- cost total: 210
- noduri expandate: 549
- score: 300

2.2 Question 5 - Finding All the Corners

Cerinta era sa impementam algoritmul pentru a gasim toate colturile din labirint din proiectul Pacman-SerachAgents in functia *class CornersProblem(search.SearchProblem):*

Aceasta problema foloseste un algoritm de cautare pentru a parcurge nodurile labirintului gasind astfel cel mai scurt drum catre toate colturile acestuia.

```
def getStartState(self):
        return self.startingPosition, self.areCornersVisited

    def isGoalState(self, state: Any):
        areCornersVisited = state[1]

        for isCornerVisited in areCornersVisited:
            if not isCornerVisited:
                return False

        return True

    def getSuccessors(self, state: Any):

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            x, y = state[0]
            areCornersVisited = list(state[1])
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]

            if not hitsWall:
                nextStateNode = (nextx, nexty)

                for i in range(4):
                    if nextStateNode == self.corners[i]:
                        areCornersVisited[i] = True

                successors.append(((nextStateNode, tuple(areCornersVisited)), action, 1))

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

```

Pe un mediumCorners :

- cost total: 106
- noduri expandate: 1966
- score: 434

2.3 Question 6 - Corners Problem: Heuristic

Cerinta era sa gasim o heuristica potrivita cu ajutorul caruia algoritmul de cautare va fi mai eficient din Pacman-SearchAgents in functia *def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem)::*

Acasta problema verifica toate colturile care nu au fost vizitate de pacman dupa ce calculeaza distanta manhattan dintre pozitia actuala si colturile parccurse si alege distanta cel mai mare asfel avand o consistenta mare.

```python
def cornersHeuristic(state: Any, problem: CornersProblem):
    currentPosition = state[0]
    areCornersVisited = list(state[1])
    h = 0

    for i in range(4):
        if not areCornersVisited[i]:
            distance = util.manhattanDistance(currentPosition, corners[i])
            h = max(h,distance)

    return h

```

Pe un mediumCorners :

- cost total: 106
- noduri expandate: 1136
- score: 434

2.4 Question 7 - Eating All The Dots

Cerinta era sa gasim o heuristica potrivita cu ajutorul caruia pacman sa gaseasca toate punctele si sa le manance care trebuia implementat in Pacman-SerachAgents in functia *def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):*

Aceasta problema cauta cel mai eficient drum pentru pacman pentru a manca toate punctele prin parcurerea nodurilor. Heuristica face o suma intre toate distantele dintre pozitia actuala si pozitia pe care se afla fiecare punctulet, dupa ce alege maximul dintre distantele calculate.

```python
def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    foodPositions = foodGrid.asList()
    h = 0

    for foodPosition in foodPositions:
        distance = mazeDistance(position, foodPosition, problem.startingGameState)
        h = max(h, distance)

    return h

```

Pe un trickySearch:

- cost total: 60
- noduri expandate: 4137
- score: 570

2.5 Question 8 - Suboptimal Search

Acest tip de cautare foloseste o metoda greedy pentru a gasi drumul optimal. Astfel, in comparatie cu algoritmul A*, acesta devine mult mai eficient in timp, insa nu prezinta mereu o solutie optima. Pentru a rezolva problema, doua functii trebuiau completate.

Prima functie se foloseste de clasa AnyFoodSearchProblem care gaseste un drum catre orice "mancare" pentru Pacmanul nostru, adica gaseste drumul dintr-o pozitie de inceput catre un alt punct din labirint.

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

A doua functie la care am lucrat pentru a rezolva aceasta problema este isGoalState din clasa precizata mai sus (AnyFoodSearchProblem). Acesta se uita daca am ajuns intr-un punct adecvat din labirint. Daca pozitia in care ne aflam are mancare pe el inseamna ca am ajuns in locul cautat.

```
def isGoalState(self, state: Tuple[int, int]):
        return self.food[x][y]

```

Pentru bigSearch:

- cost total: 350
- score: 2360

3 Adversarial search

3.1 Question 9 - Improve the ReflexAgent

Cerinta era sa gasim o metoda cu care putem bunatati reflexAgent-ul ca sa ocoleasca fantomele si sa manance toate mancarurile. Codul trebuia implementat in functia *def evaluationFunction(self, currentGameState: GameState, action):*

In algoritm trebuie sa parcurgem toate pozitiile noi ale tuturor fantomelor. Daca vreo fantoma este prea aproape, atunci pozitia noastra nu este une prea buna, deci functia de evaluare returneaza o valoare foarte mica. Daca nicio fantoma nu este aproape, cautam distanta intre pacman si cel mai apropiat punctulet nemancat, daca mai exista mancare pe bord. Rezultatul returnat de catre algoritm va fi o suma intre scorul pozitiei urmatoare a pacmanului si reciproca distantei calculate anterior. Astfel functia de evaluare va fi una mai imbunatatita.

```
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]

        inf = 99999999
        for newGhotState in newGhostStates:
            distance = manhattanDistance(newPos, newGhotState.getPosition())
            if newGhotState.scaredTimer == 0 and distance < 2:
```

```
                return - inf

        foodDistances = []

        for foodPosition in newFood.asList():
            foodDistances.append(manhattanDistance(newPos, foodPosition))

        minFoodDistance = 0

        if len(foodDistances):
            minFoodDistance = 1 / min(foodDistances)

        return successorGameState.getScore() + minFoodDistance

```

Dupa rularea comenzii de verificare a programului am obtinut scorul de 1392 de puncte.

3.2 Question 10 - Implement the MinMax

Cerinta era sa implementam algoritmul minimax completand functia *def getAction(self, gameState: GameState)* din clasa *MinimaxAgent*. Pentru asta am scris o noua functie recursiva care este vizibila mai jos.

Algoritmul minimax este folosit pentru a determina cea mai buna miscare posibila intr-o anumita stare a unui joc. In cazul nostru, ne folosim de minimax pentru a gasi cea mai buna ruta pentru pacman ca acesta sa poata manca cat mai multe puncte fara sa fie el mancat de catre fantome. In algoritm, evaluam toate posibilitatile de miscare pana la o anumita adancime. Cum functioneaza: prima data gasim cea mai buna miscare a pacmanului din pozitia lui actuala si trecem mai departe in arborele jocului si incercam sa gasim cea mai buna miscare a fantomelor - care in principiu nu este una buna pentru pacman. Dupa asta, urmeaza iar miscarea pacman-ului. Repetam aceste pasuri pana la o anumita adancime sau pana ce ajungem la finalul arborelui.

```
  def minimax(self, gameState: GameState, depth, agentIndex):
        if self.depth == depth or gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState), ''

        numAgents = gameState.getNumAgents()
        legalActions = gameState.getLegalActions(agentIndex)
        inf = 99999999

        nextDepth = depth
        nextAgentIndex = agentIndex + 1

        if nextAgentIndex == numAgents:
            nextAgentIndex = 0
            nextDepth += 1

        if agentIndex == 0:
            maxScore = - inf
```

```
            maxAction = ''

            for action in legalActions:
                successorGameState = gameState.generateSuccessor(agentIndex, action)

                score, action2 = self.minimax(successorGameState, nextDepth, nextAgentIndex)

                if score > maxScore:
                    maxScore, maxAction = score, action

            return maxScore, maxAction
        else:
            minScore = inf
            minAction = ''

            for action in legalActions:
                successorGameState = gameState.generateSuccessor(agentIndex, action)

                score, action2 = self.minimax(successorGameState, nextDepth, nextAgentIndex)

                if score < minScore:
                    minScore, minAction = score, action

            return minScore, minAction

```

Dupa rularea comenzilor de la problema rezultatul este ca pacman gaseste mereu cea mai buna ruta pentru a face cel mai mare scor posibil, iar daca isi da seama ca nu are nicio sansa de castig, se sinucide cat de repede posibil.

3.3 Question 11 - Implement the Alpha-Beta Pruning

Cerinta era sa implementam algoritmul minimax cu taierea alpha-beta prin completarea clasei *AlphaBetaAgent*. Pentru asta am creat o noua functie: *def minimaxAlphaBeta(self, gameState: GameState, depth, agentIndex, alpha, beta).*

Algoritmul minimax in sine este eficient, insa este destul de costisitor in timp. Din aceasta cauza, putem folosi diferite metode de a-l accelera. O solutie ar fi sa rulam algoritmul pana intr-o adancime mai mica, insa aceasta nu ar aduce un rezultat optimal jucatorului nostru. O alta solutie ar fi sa implementam algoritmul folosindu-ne si de taierea alpha-beta. Aceasta tehnica elimina evaluarea portiunilor inutile din arborele jocului. Astfel, pe cand minimax exploreaza tot arborele jocului, cu taierea alpha-beta, putem scapa de multe pasuri care nu aduc valoare jucatorului nostru.

```
 def minimaxAlphaBeta(self, gameState: GameState, depth, agentIndex, alpha, beta):
        if self.depth == depth or gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState), ''

        numAgents = gameState.getNumAgents()
        legalActions = gameState.getLegalActions(agentIndex)
```

```
        inf = 99999999

        nextDepth = depth
        nextAgentIndex = agentIndex + 1

        if nextAgentIndex == numAgents:
            nextAgentIndex = 0
            nextDepth += 1

        if agentIndex == 0:
            maxScore = - inf
            maxAction = ''

            for action in legalActions:
                successorGameState = gameState.generateSuccessor(agentIndex, action)

                score, action2 = self.minimaxAlphaBeta(successorGameState, nextDepth, nextAg

                if score > maxScore:
                    maxScore, maxAction = score, action

                if maxScore > beta:
                    break

                alpha = max(alpha, maxScore)

            return maxScore, maxAction
        else:
            minScore = inf
            minAction = ''

            for action in legalActions:
                successorGameState = gameState.generateSuccessor(agentIndex, action)

                score, action2 = self.minimaxAlphaBeta(successorGameState, nextDepth, nextAg

                if score < minScore:
                    minScore, minAction = score, action

                if minScore < alpha:
                    break

                beta = min(beta, minScore)

            return minScore, minAction
```

Putem observa ca prin taierea alpha-beta, algoritmul nostru va fi mult mai rapid, acesta expandand mi putine noduri din joc.