UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autor: Simina Dan-Marius
Grupa: 30232

Facultatea de Automatica
si Calculatoare

3 Decembrie 2024

Cuprins

1 Introducere

Proiectul constă în aplicarea unei game de tehnici de inteligență artificială pentru jocul Pac-Man.
Cerințele Q1-Q8 reprezintă implementarea algoritmilor de căutare în adâncime (Depth-First Search), în lățime (Breadth-First Search), cu cost uniform (Uniform Cost Search) și A* (A-Star). Acești algoritmi sunt folosiți pentru a rezolva probleme de navigație și probleme ale comis-voiajorului în lumea Pacman.
Cerințele Q9 - Q11 reprezintă modelarea Pac-Man ca o problemă de căutare adversarială.

2 Uninformed search

2.1 Question 1 - Depth-first search

```
def depthFirstSearch(problem: SearchProblem) -> List[Directions]:
    stack = Stack()
    stack.push((problem.getStartState(), []))
    visited = set()

    while not stack.isEmpty():
        state, path = stack.pop()
        if problem.isGoalState(state):
            return path
        if state not in visited:
            visited.add(state)
            for next, action, _ in problem.getSuccessors(state):
                if next not in visited:
                    stack.push((next, path + [action]))
```

Cerință: Implementarea algoritmului de căutare în adâncime pentru găsirea unui punct fix de hrană.
Prezentare algoritm: Căutarea în adâncime (Depth-First Search - DFS) este o căutare în arbore/graf care utilizează o frontieră de tip LIFO (stivă), extinzând ultimul nod adăugat. Algoritmul începe de la nodul rădăcină (sau de la un nod ales arbitrar, în cazul unui graf) și expandează cel mai adânc nod neexpandat, iar nodurile expandate fara succesori in frontiera sunt eliminate din memorie.
Complexitate:

- Complexitate timp: $O(b^m)$
- Complexitate spațiu: $O(bm)$

Unde b este factorul de ramificare al arborelui (numărul mediu de copii per nod), m este lungimea celui mai lung drum din graf (adâncimea maximă a arborelui).

2.2 Question 2 - Breadth-first search

```
def breadthFirstSearch(problem: SearchProblem) -> List[Directions]:
    queue = Queue()
    queue.push((problem.getStartState(), []))
    visited = set()

```

```
    while not queue.isEmpty():
        state, path = queue.pop()
        if problem.isGoalState(state):
            return path
        if state not in visited:
            visited.add(state)
            for next, action, _ in problem.getSuccessors(state):
                if next not in visited:
                    queue.push((next, path + [action]))
```

Cerință: Implementarea algoritmului de căutare în lățime pentru găsirea unui punct fix de hrană.

Prezentare algoritm: Căutarea în lățime (Breadth-First Search - BFS) este o căutare în arbore/graf care folosește o frontieră de tip FIFO (coadă), extinzând nodurile în ordinea în care au fost adăugate. Algoritmul începe de la rădăcina arborelui și explorează toate nodurile de la nivelul curent de adâncime înainte de a trece la nodurile de la nivelul următor.

Complexitate:

- Complexitate timp: $O(b^{d+1})$
- Complexitate spațiu: $O(b^d)$

Unde b este factorul de ramificare al arborelui (numărul mediu de copii per nod), d este adâncimea celui mai apropiat nod țintă (adâncimea minimă a unei soluții).

2.3 Question 3 - Uniform-cost search

```
def uniformCostSearch(problem: SearchProblem) -> List[Directions]:
    start = problem.getStartState()
    priorityQueue = PriorityQueue()
    priorityQueue.push((start, []), 0)
    visited = set()

    currentCost = dict()
    currentCost[start] = 0

    while not priorityQueue.isEmpty():
        state, path = priorityQueue.pop()
        if problem.isGoalState(state):
            return path
        if state not in visited:
            visited.add(state)
            for next, action, cost in problem.getSuccessors(state):
                if next not in visited:
                    currentCost[next] = cost + currentCost[state]
                    priorityQueue.update((next, path + [action]), currentCost[state] + cost)
```

Cerință: Implementarea algoritmului de căutare cu cost uniform pentru găsirea unui punct fix de hrană.

Prezentare algoritm: Căutarea cu cost uniform (Uniform Cost Search - UCS) este o căutare care utilizează o frontieră sub forma unei cozi de priorități, ceea ce înseamnă că extinde nodul

cu cel mai mic cost al drumului g(n). Algoritmul funcționează pe un principiu simplu, dintre toate extensiile posibile, alege calea care are cel mai mic cost total de la nodul de start.

Complexitate:

- Complexitate timp: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- Complexitate spațiu: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

Unde b este factorul de ramificare (numărul mediu de copii per nod), C^* este costul total minim al drumului optim de la nodul de start la nodul țintă, ϵ este diferența minimă dintre costurile oricăror două noduri adiacente.

3 Informed search

Deși BFS, DFS și UCS sunt capabile să găsească soluții, ele nu o fac eficient. Acestea sunt denumite algoritmi neinformatizați (uninformed), deoarece utilizează doar definiția problemei și nu alte informații. Căutarea poate fi redusă în multe situații cu un ghidaj privind locul în care să căutăm soluțiile. Algoritmii informatizați folosesc informații suplimentare despre problemă pentru a reduce căutarea.

3.1 Question 4 - A* search algorithm

```python
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic) -> List[Directions]:
    start = problem.getStartState()
    priorityQueue = PriorityQueue()
    priorityQueue.push((start, []), 0)

    currentCost = dict()
    currentCost[start] = 0

    while not priorityQueue.isEmpty():
        state, path = priorityQueue.pop()
        if problem.isGoalState(state):
            return path
        for next, action, cost in problem.getSuccessors(state):
            if currentCost[state] + cost < currentCost.setdefault(next, 2**31 - 1):
                currentCost[next] = cost + currentCost[state]
                priorityQueue.update((next, path + [action]), currentCost[state] + cost + he
```

Cerință: Implementarea algoritmului de căutare A^* pentru găsirea unui punct fix de hrană.

Prezentare algoritm: Algoritmul A* (A-star) este un algoritm de căutare utilizat în inteligența artificială pentru a găsi calea optimă (cu costul minim) într-un graf, având un punct de plecare și un punct țintă. A* este o îmbunătățire a Căutării cu Cost Uniform (UCS) și folosește atât costul real de la start până la un nod, cât și o estimare a costului rămas până la țintă, pentru a ghida căutarea. Astfel, este un algoritm informatizat. Evaluarea unui nod combină costul pentru a ajunge la nodul respectiv din starea inițială cu costul estimat pentru a ajunge de la nod la scop.

$$f(n) = g(n) + h(n) \tag{1}$$

Unde g(n) este costul real parcurs până în acest punct, iar h(n) este estimarea costului rămas de la nodul n până la nodul țintă.

Complexitate:

- Complexitate timp: $O(b^d)$
- Complexitate spațiu: $O(b^d)$

Unde b este factorul de ramificare al arborelui (numărul mediu de copii per nod), d este adâncimea celui mai apropiat nod țintă (adâncimea minimă a unei soluții).

3.2 Question 5 - Finding All the Corners

```python
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

    def getStartState(self):
        return self.startingPosition, (False, False, False, False)

    def isGoalState(self, state: Any):
        _, corners = state

        if corners == (True, True, True, True):
            return True
        return False

    def getSuccessors(self, state: Any):
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
            pos, corners = state
            x, y = pos
            corners = list(corners)
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)

            if (nextx, nexty) in self.corners:
                corners[self.corners.index((nextx, nexty))] = True

            if not self.walls[nextx][nexty]:
                successors.append((((nextx, nexty), tuple(corners)), action, 1))

        self._expanded += 1
        return successors
```

Cerință: În labirinturile cu patru colțuri, există patru puncte, câte unul în fiecare colț. Noua problemă de căutare este să se găsească cel mai scurt drum prin labirint care atinge toate cele patru colțuri (indiferent dacă labirintul are sau nu mâncare acolo).

Prezentare "Corners Problem":

- Funcția **getStartState(...)**: Returnează starea inițială. O stare este o tupla alcătuită din alte două tuple, una care reprezintă poziția inițială și alta care conține 4 valori care indică pentru fiecare din cele 4 colțuti dacă a fost sau nu vizitat("False" daca colțul nu e vizitat, "True" dacă colțul a fost vizitat).
- Funcția **isGoalState(...)**: Returnează "True" dacă în starea curentă au fost vizitate toate colțurile, iar dacă nu returnează "False".
- Funcția **getSuccessors(...)**: Retunează stările în care se poate trece din starea curentă. Petru fiecare direcție de deplasare se verifică dacă poziția e un chiar un colț și dacă da se marchează cu "True", totodată se verifică daca e accesibilă(nu este perete), iar dacă e accesibilă se adaugă starea în lista de succesori.

3.3 Question 6 - Corners Problem: Heuristic

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    position, cornersState = state
    score = 0

    if not cornersState[0]:
        score = max(score, manhattanDistance(position,corners[0]))
    if not cornersState[1]:
        score = max(score, manhattanDistance(position,corners[1]))
    if not cornersState[2]:
        score = max(score, manhattanDistance(position,corners[2]))
    if not cornersState[3]:
        score = max(score, manhattanDistance(position,corners[3]))

    return score
```

Cerință: Implementarea unei euristici pentru **Corners Problem**.

Prezentare euristică: Se calculează distanțele Manhattan dintre poziția curentă și toate colțurile nevizitate, iar în final se returnează distanța maximă

3.4 Question 7 - Eating All The Dots

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

    if len(foodGrid.asList()) == 0:
        return 0

    nearestFood = min(mazeDistance(position, food, problem.startingGameState)
                      for food in foodGrid.asList())

    northernmost = min(foodGrid.asList(), key = lambda x: x[1])
    southernmost = max(foodGrid.asList(), key = lambda x: x[1])
```

```
    westernmost = min(foodGrid.asList(), key = lambda x: x[0])
    easternmost = max(foodGrid.asList(), key = lambda x: x[0])

    return nearestFood +
            mazeDistance(northernmost, southernmost, problem.startingGameState) * 0.5 +
            mazeDistance(westernmost, easternmost, problem.startingGameState) * 0.5
```

Cerință: Implementarea unei euristici pentru a mânca toată mâncarea din lumea Pacman în cât mai puțini pași posibil.
Prezentare euristică: Se calculează distanța până la cea mai arpopiată bucată de mâncare la care se adaugă media dintre distanța între cel mai nordic și cel mai sudic punct, și distanța dintre cel mai vestic și cel mai estic punct.

3.5 Question 8 - Suboptimal Search

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

        return search.bfs(AnyFoodSearchProblem(gameState))
```

Cerință: Realizareaa unui agent care mănâncă întotdeauna lacom punctul cel mai apropiat.
Prezentare soluție: Se folosește căutarea în lățime pentru a găsi cel mai apropiat punct.

4 Adversarial search

4.1 Question 9 - Improve the ReflexAgent

```
    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]

        currentPositon = currentGameState.getPacmanPosition()
        walls = currentGameState.getWalls()
        food = currentGameState.getFood()
        ghostState = currentGameState.getGhostStates()

        ghostDistance = 0
        isGhostThere = False
        penalty = 0

        x, y = currentPositon
        freeSpace = 0
        for i in range(max(0,  -2), min(x + 2, walls.width)):
```

```
            for j in range(max(0, -2), min(y + 2, walls.height)):
                if not walls[i][j]:
                    freeSpace += 1

        freeSpace = cntArea - cntWalls

        for ghost in newGhostStates:
            ghostPosition = ghost.getPosition()
            ghostDistance = manhattanDistance(newPos, ghostPosition)
            isGhostThere |= ghostPosition == newPos

        furthestDistance = manhattanDistance(currentPositon, (walls.height, walls.width))
        nearestFoodScore = furthestDistance - min([manhattanDistance(newPos, food)
                            for food in newFood.asList()]) if newFood.asList() else 0

        totalScaredTimes = sum(newScaredTimes)

        if currentPositon == newPos:
            penalty = 0.1 * successorGameState.getScore()

        bonusEatGhost = 0
        if totalScaredTimes > 10:
            bonusEatGhost += 10 * (furthestDistance - ghostDistance)
            if isGhostThere:
                bonusEatGhost += 100
            ghostDistance = 0
        elif isGhostThere:
            return -10

        ghostDistance = min(ghostDistance, 10)

        bonusEatFood = 10 if food[newPos[0]][newPos[1]] else 0

        return successorGameState.getScore() + ghostDistance + nearestFoodScore +
            bonusEatFood + bonusEatGhost - penalty + freeSpace
```

Cerință: Îmbunătățirea agentului reflex astfel încât să joace într-un mod respectabil.

Prezentare soluție: Scorul returnat de funcția de evaluare este suma dintre scorul curent, distanța de pe noua poziție până la cea mai apropiată fantomă, un bonus care se adaugă dacă pe poziția nouă se află o buctă de mâncare, un alt bonus pe care îl primește dacă poate să mănânce o fantomă, totalul de spațiu liber pe o rază de două poziții in toate direcțiile și din toate acestea se scade o penalizare dacă noua poziție e tot poziția actuală.

4.2 Question 10 - Minimax

```
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState: GameState):
        def minimaxDecision(agentIndex: int, state: GameState):
            v = -2 ** 31
```

```
5               bestAction = None
6               numOfAgents = state.getNumAgents()
7
8               for action in state.getLegalActions(agentIndex):
9                   result = minValue(0, (agentIndex + 1) % numOfAgents,
10                                  state.generateSuccessor(agentIndex, action))
11                  if v < result:
12                      v = result
13                      bestAction = action
14              return bestAction
15
16          def minValue(depth:int, agentIndex: int, state: GameState) -> int:
17              numOfAgents = state.getNumAgents()
18
19              if state.isWin() or state.isLose() or depth == self.depth:
20                  return self.evaluationFunction(state)
21
22              v = 2**31 - 1
23              for action in state.getLegalActions(agentIndex):
24                  if agentIndex == numOfAgents - 1:
25                      v = min(v, maxValue(depth + 1, 0,
26                              state.generateSuccessor(agentIndex, action)))
27                  else:
28                      v = min(v, minValue(depth, agentIndex + 1,
29                              state.generateSuccessor(agentIndex, action)))
30              return v
31
32          def maxValue(depth:int, agentIndex: int, state: GameState) -> int:
33              if state.isWin() or state.isLose() or depth == self.depth:
34                  return self.evaluationFunction(state)
35
36              v = -2**31
37              for action in state.getLegalActions(agentIndex):
38                  v = max(v, minValue(depth, agentIndex + 1,
39                          state.generateSuccessor(agentIndex, action)))
40
41              return v
42
43          return minimaxDecision(0, gameState)
```

Cerință: Implementarea algoritmului Minimax pentru un agent într-un mediu adversarial.

Prezentare algoritm: Valoarea Minimax asigură o strategie optimă pentru MAX. Aceasta reprezintă scorul maxim pe care MAX îl poate obține, presupunând că MIN joacă optim pentru a minimiza scorul lui MAX. Pe scurt, algoritmul funcționează astfel:

1. Dacă nodul curent este o stare terminală, returnează valoarea funcției de utilitate.
2. Dacă este rândul lui MAX să mute, returnează valoarea maximă dintre mutările posibile.
3. Dacă este rândul lui MIN să mute, returnează valoarea minimă dintre mutările posibile.

Complexitate:

- Complexitate timp: $O(b^m)$
- Complexitate spațiu: $O(bm)$

Unde b este factorul de ramificare al arborelui (numărul mediu de copii per nod), m este lungimea celui mai lung drum din graf (adâncimea maximă a arborelui).

4.3 Question 11 - Alpha-Beta Pruning

```
class AlphaBetaAgent(MultiAgentSearchAgent):
    def getAction(self, gameState: GameState):
        def alphaBetaSearch(agentIndex: int, state: GameState):
            v = -2 ** 31
            bestAction = None
            numOfAgents = state.getNumAgents()
            alpha = -2**31
            beta = 2**31 - 1

            for action in state.getLegalActions(agentIndex):
                result = minValue(0, (agentIndex + 1) % numOfAgents,
                            state.generateSuccessor(agentIndex, action), alpha, beta)
                if v < result:
                    v = result
                    bestAction = action
                alpha = max(alpha, v)
            return bestAction

        def minValue(depth:int, agentIndex: int, state: GameState, alpha, beta) -> int:
            numOfAgents = state.getNumAgents()

            if state.isWin() or state.isLose() or depth == self.depth:
                return self.evaluationFunction(state)

            v = 2**31 - 1
            for action in state.getLegalActions(agentIndex):
                if agentIndex == numOfAgents - 1:
                    v = min(v, maxValue(depth + 1, 0,
                            state.generateSuccessor(agentIndex, action), alpha, beta))
                else:
                    v = min(v, minValue(depth, agentIndex + 1,
                            state.generateSuccessor(agentIndex, action), alpha, beta))

                if v < alpha:
                    return v
                beta = min(beta, v)

            return v

        def maxValue(depth:int, agentIndex: int, state: GameState, alpha, beta) -> int:
```

```
41              if state.isWin() or state.isLose() or depth == self.depth:
42                  return self.evaluationFunction(state)
43
44              v = -2**31
45              for action in state.getLegalActions(agentIndex):
46                  v = max(v, minValue(depth, agentIndex + 1,
47                          state.generateSuccessor(agentIndex, action), alpha, beta))
48
49                  if v > beta:
50                      return v
51                  alpha = max(alpha, v)
52              return v
53
54          return alphaBetaSearch(0, gameState)
```

Cerință: Implemetarea unui nou agent care folosește Alpha-Beta Pruning pentru a explora eficient arborele minimax.

Prezentare algoritm: Pentru a limita numărul de stări de joc din arborele jocului, se poate aplica alpha-beta pruning, o tehnică de optimizare a algoritmului minimax. Aceasta reduce numărul de stări explorate în arborele jocului, păstrând totuși soluția optimă. Alfa(α) reprezintă cea mai bună alegere (valoarea maximă) pentru MAX găsită până în acel moment pe un anumit traseu din arbore, este inițial minus infinit și se actualizează atunci când MAX găsește o valoare mai mare. Beta(β) reprezintă cea mai bună alegere (valoarea minimă) pentru MIN găsită până în acel moment pe un anumit traseu din arbore, este inițial plus infinit și se actualizează atunci când MIN găsește o valoare mai mică. În timpul explorării arborelui, dacă un nod fiu oferă o valoare care face ca starea curentă să fie mai puțin favorabilă decât o stare anterioară (în funcție de alfa și beta) atunci tăiem acea ramură, deoarece nu este necesar să o explorăm. Oricare alte mutări posibile pe acea ramură nu vor influența decizia finală.

Complexitate:

- Complexitate timp: $O(b^m)$
- Complexitate spațiu: $O(bm)$

Unde b este factorul de ramificare al arborelui (numărul mediu de copii per nod), m este lungimea celui mai lung drum din graf (adâncimea maximă a arborelui).