UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autor: Tamasoi Stefania-Maria
Grupa: 30239

Facultatea de Automatica si Calculatoare

6 Noiembrie 2024

Cuprins

1 Pacman Project

2 Question 1 - Depth First Search

2.1 Descrierea algoritmului

DFS este un algoritm de cautare utilizat in grafuri, implementat utilizand un *stack* (LIFO).

2.2 Prezentarea codului

```
def depthFirstSearch(problem):
    from util import Stack

    myStack = Stack()
    visited = set()
    myStack.push((problem.getStartState(), []))
    while not myStack.isEmpty():
        current_state, path = myStack.pop()
        if problem.isGoalState(current_state):
            return path
        if current_state not in visited:
            visited.add(current_state)
            for successor, action, step_cost in problem.getSuccessors(current_state):
                if successor not in visited:
                    myStack.push((successor, path + [action]))
    return []
```

2.3 Performanta si complexitate

In cel mai rau caz, complexitatea este $O(b^d)$, unde b numarul de succesori ai fiecarui nod si d este adancimea maxima a arborelui de cautare.

3 Question 2 - Breadth First Search

3.1 Descrierea algoritmului

BFS este un algoritm de cautare in grafuri care exploreaza toate nodurile de la un anumit nivel inainte de a trece la nivelul urmator. Acest algoritm garanteaza gasirea celei mai scurte solutii intr-un graf neponderat BFS este util mai ales in problemele in care vrem sa gasim drumul cel mai scurt de la un nod la altul.

Algoritmul BFS foloseste o structura de date de tip *queue*(FIFO).

3.2 Prezentarea codului

```
def breadthFirstSearch(problem):

    from util import Queue
    myQueue = Queue()
    visited = set()
```

```
    myQueue.push((problem.getStartState(), []))

    while not myQueue.isEmpty():
        current_state, path = myQueue.pop()
        if problem.isGoalState(current_state):
            return path
        if current_state not in visited:
            visited.add(current_state)
            for successor, action, step_cost in problem.getSuccessors(current_state):
                if successor not in visited:
                    myQueue.push((successor, path + [action]))

    return []
```

3.3 Performanta si complexitate

In cel mai rau caz, complexitatea temporala este $O(b^d)$, unde b numarul de succesori ai fiecarui nod si d este adancimea maxima a arborelui de cautare

4 Question 3 - Uniform Cost Search

4.1 Descrierea algoritmului

UCS este un algoritm de cautare care exploreaza nodurile în ordinea costului total acumulat, selectand nodul cu costul minim de extins. Acesta este o metoda eficientă pentru a gasi drumul cu cel mai mic cost intr-un graf.

UCS utilizeaza o coada de prioritati.

4.2 Prezentarea codului

```
def uniformCostSearch(problem):
    from util import PriorityQueue
    myPriorityQueue = PriorityQueue()
    visited = {}

    start = problem.getStartState()
    myPriorityQueue.push((start, [], 0), 0)

    while not myPriorityQueue.isEmpty():
        current_state, path, cost = myPriorityQueue.pop()
        if problem.isGoalState(current_state):
            return path
        if current_state not in visited or cost < visited[current_state]:
            visited[current_state] = cost
            for successor, action, stepCost in problem.getSuccessors(current_state):
                new_cost = cost + stepCost
                myPriorityQueue.push((successor, path + [action], new_cost), new_cost)

    return []
```

4.3 Performanta si complexitate

UCS exploreaza fiecare nod și fiecare succesor al acestuia. Daca exista b ramuri per nod si adancimea maximăa a arborelui este d, iar numarul total de noduri este n, complexitatea este aproximativ $O(n \log n)$

5 Question 4 - A* Search Algorithm

5.1 Descrierea algoritmului A*

A* este un algoritm de cautare care este utilizat pentru a gasi cel mai scurt drum de la un nod de start la un nod final. Acesta este o combinatie intre BFS si Dijkstra.

Functia de evaluare este:

$$f(n) = g(n) + h(n)$$

5.2 Explicatia algoritmului A*

A* incepe prin initializarea unei cozi de prioritate si a unui dictionar pentru noduri vizitate, apoi exploreaza nodurile pe baza valorii $f(n)$ continuand pana gaseste calea optima catre nodul tinta sau constata ca nu exista solutie.

5.3 Prezentarea codului

```
def aStarSearch(problem, heuristic=nullHeuristic):
    from util import PriorityQueue

    myPriorityQueue = PriorityQueue()
    visited = {}

    start = problem.getStartState()
    myPriorityQueue.push((start, [], 0), 0)
    while not myPriorityQueue.isEmpty():
        current_state, path, cost = myPriorityQueue.pop()
        if problem.isGoalState(current_state):
            return path
        if current_state not in visited or cost < visited[current_state]:
            visited[current_state] = cost
            for successor, action, stepCost in problem.getSuccessors(current_state):
                new_cost = cost + stepCost
                priority = new_cost + heuristic(successor, problem)
                myPriorityQueue.push((successor, path + [action], new_cost), priority)

    return []
```

6 Question 5 - Corners Problem

6.1 Descrierea algoritmului

Algoritmul cauta un drum care sa treaca prin toate cele patru colturi ale unui labirint, pornind de la pozitia initiala a lui Pacman.

6.2 Explicatia algoritmului

Algoritmul exploreaza nodurile pe baza pozitiei curente a lui Pacman si a colturilor deja vizitate, extinzand starea pana cand toate colturile sunt vizitate.

6.3 Prezentarea codului

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor function
    """

    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        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
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"


    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        return self.startingPosition, ()


    def isGoalState(self, state):
        """
```

```
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        return len(state[1]) == len(self.corners)

    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.

         As noted in search.py:
            For a given state, this should return a list of triples, (successor,
            action, stepCost), where 'successor' is a successor to the current
            state, 'action' is the action required to get there, and 'stepCost'
            is the incremental cost of expanding to that successor
        """

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]
            # Add a successor state to the successor list if the action is legal
            # Here's a code snippet for figuring out whether a new position hits a wall:
            #   x,y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   hitsWall = self.walls[nextx][nexty]

            "*** YOUR CODE HERE ***"
            x,y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                next_pos = (nextx, nexty)
                new_state = (next_pos, state[1])
                if next_pos in self.corners and next_pos not in state[1]:
                    new_state = (next_pos, (state[1] + (next_pos, )))
                successors.append((new_state, action,1))

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

    def getCostOfActions(self, actions):
        """
        Returns the cost of a particular sequence of actions.  If those actions
        include an illegal move, return 999999.  This is implemented for you.
        """
        if actions == None: return 999999
        x,y= self.startingPosition
        for action in actions:
```

```
                dx, dy = Actions.directionToVector(action)
                x, y = int(x + dx), int(y + dy)
                if self.walls[x][y]: return 999999
            return len(actions)
```

6.4 Performanta si complexitate

Complexitatea in timp este $O(b^d)$, iar complexitatea in spatiu este similara. Performanta depinde de numarul de colturi ramase de vizitat.

7 Question 6 - Corners Heuristic

7.1 Descrierea algoritmului

Euristica calculeaza distanta minima pe care Pacman trebuie sa o parcurga pentru a vizita toate colturile ramase, folosind distanta Manhattan.

7.2 Explicatia algoritmului

Euristica selecteaza coltul cel mai apropiat de pozitia curenta si aduna distanta pana la toate colturile ramase, asigurand o estimare optima a costului ramas.

7.3 Prezentarea codului

```
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

    "*** YOUR CODE HERE ***"

    total_heuristic = 0
    pos = state[0]
    reached_corners = state[1]
    corners_left = list(set(corners) - set(reached_corners))

    while len(corners_left):
        best_corner = corners_left[0]
```

```
        best_distance = util.manhattanDistance(pos, best_corner)

        for corner in corners_left[1:]:
            distance = util.manhattanDistance(pos, corner)
            if distance < best_distance:
                best_distance = distance
                best_corner = corner

        total_heuristic += best_distance
        pos = best_corner
        corners_left.remove(best_corner)

    return total_heuristic
```

7.4 Performanta si complexitate

Complexitatea in timp este $O(c^2)$, unde c este numarul de colțuri ramase de vizitat, iar complexitatea in spatiu este $O(c)$, datorita stocarii colturilor ramase.

8 Question 7 - Food Search Problem

8.1 Descrierea algoritmului

Algoritmul cauta o cale prin labirint pentru a colecta toate punctele de mancare, avand ca stare curenta pozitia lui Pacman si grila de mancare.

8.2 Prezentarea codului

```
class FoodSearchProblem:
    """
    A search problem associated with finding the a path that collects all of the
    food (dots) in a Pacman game.

    A search state in this problem is a tuple ( pacmanPosition, foodGrid ) where
      pacmanPosition: a tuple (x,y) of integers specifying Pacman's position
      foodGrid:       a Grid (see game.py) of either True or False, specifying remaining fo
    """
    def __init__(self, startingGameState):
        self.start = (startingGameState.getPacmanPosition(), startingGameState.getFood())
        self.walls = startingGameState.getWalls()
        self.startingGameState = startingGameState
        self._expanded = 0 # DO NOT CHANGE
        self.heuristicInfo = {} # A dictionary for the heuristic to store information

    def getStartState(self):
        return self.start

    def isGoalState(self, state):
        return state[1].count() == 0
```

```
    def getSuccessors(self, state):
        "Returns successor states, the actions they require, and a cost of 1."
        successors = []
        self._expanded += 1 # DO NOT CHANGE
        for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WE
            x,y = state[0]
            dx, dy = Actions.directionToVector(direction)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                nextFood = state[1].copy()
                nextFood[nextx][nexty] = False
                successors.append( ( ((nextx, nexty), nextFood), direction, 1) )
        return successors

    def getCostOfActions(self, actions):
        """Returns the cost of a particular sequence of actions.  If those actions
        include an illegal move, return 999999"""
        x,y= self.getStartState()[0]
        cost = 0
        for action in actions:
            # figure out the next state and see whether it's legal
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]:
                return 999999
            cost += 1
        return cost
```

8.3 Performanta si complexitate

Complexitatea in timp este $O(b^d)$, unde b este numarul de noduri de explorat, iar d este adancimea solutiei.

9 Question 8 - Find Path To Closest Dot

9.1 Descrierea algoritmului

Algoritmul cauta calea cea mai scurta catre cel mai apropiat punct de mancare folosind UCS.

9.2 Prezentarea codului

```
    def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
```

```
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        "*** YOUR CODE HERE ***"
        return search.uniformCostSearch(problem)
```

9.3 Performanta si complexitate

Complexitatea in timp este $O(b^d)$. Performanta depinde de numarul de puncte de mancare ramase de colectat.

10 Question 1 - Reflex Agent

10.1 Descrierea algoritmului

Reflex Agent ia decizii pe baza unei functii de evaluare care masoara calitatea unei stari a jocului. Agentul selecteaza actiunea care maximizeaza aceasta functie, dintre actiunile legale disponibile.

10.2 Explicatia implementarii

Agentul analizeaza fiecare actiune legala si calculeaza un scor folosind functia de evaluare. Functia considera factori precum distanta pana la mancare, pozitia fantomelor, capsulele disponibile si alte elemente care influenteaza scorul jocului. Daca exista mai multe actiuni cu acelasi scor maxim, alegerea finala este facuta aleatoriu.

10.3 Complexitatea algoritmului

Complexitatea algoritmului este $O(m \cdot n)$, unde m este numarul de actiuni legale si n este numarul de elemente relevante din evaluare (de exemplu, mancare si fantome).

10.4 Prezentarea codului

```
class ReflexAgent(Agent):

    def getAction(self, gameState: GameState):

        # Collect legal moves and successor states
        legalMoves = gameState.getLegalActions()

        # Choose one of the best actions
        scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
        bestScore = max(scores)
        bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
        chosenIndex = random.choice(bestIndices) # Pick randomly among the best

        "Add more of your code here if you want to"

```

```
        return legalMoves[chosenIndex]

    def evaluationFunction(self, currentGameState: GameState, action):
        """
        Design a better evaluation function here.

        The evaluation function takes in the current and proposed successor
        GameStates (pacman.py) and returns a number, where higher numbers are better.

        The code below extracts some useful information from the state, like the
        remaining food (newFood) and Pacman position after moving (newPos).
        newScaredTimes holds the number of moves that each ghost will remain
        scared because of Pacman having eaten a power pellet.

        Print out these variables to see what you're getting, then combine them
        to create a masterful evaluation function.
        """
        # Useful information you can extract from a GameState (pacman.py)
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

        "*** YOUR CODE HERE ***"

        score = successorGameState.getScore()
        foodList = newFood.asList()
        if foodList:
            closestFoodDist = min(manhattanDistance(newPos, food) for food in foodList)
            score += 10 / (1 + closestFoodDist)

        for ghostState, scaredTime in zip(newGhostStates, newScaredTimes):
            ghostPos = ghostState.getPosition()
            distToGhost = manhattanDistance(newPos, ghostPos)

            if scaredTime > 0:
                score += 200 / (1 + distToGhost)
            elif distToGhost <= 1:
                score -= 500

        if action == Directions.STOP:
            score -= 10

        capsules = currentGameState.getCapsules()
        if capsules:
            closestCapsuleDist = min(manhattanDistance(newPos, capsule) for capsule in capsu
            score += 100 / (1 + closestCapsuleDist)
```

```

        score -= len(foodList) * 2

        return score
```

11 Question 2 - Minimax

11.1 Descrierea algoritmului

Minimax este un algoritm pentru luarea deciziilor intr-un joc cu mai multi agenti. Pacman urmareste sa maximizeze scorul, in timp ce fantomele incearca sa il minimizeze. Algoritmul exploreaza arborele de decizie pana la o adancime specificata si evalueaza starile folosind o functie de evaluare.

11.2 Explicatia implementarii

1. **Maximizare (Pacman):** Functia maxValue alege actiunea cu scorul maxim pentru Pacman. 2. **Minimizare (Fantome):** Functia minValue alege actiunea cu scorul minim pentru fantome, tinand cont de strategia lor de reducere a scorului Pacman. 3. Recursiv, algoritmul parcurge toate starile posibile pana la adancimea maxima sau pana la o stare finala, alegand actiunea optima pentru Pacman.

11.3 Complexitatea algoritmului

Complexitatea este $O(b^{d \cdot n})$, unde b este factorul de ramificare, d este adancimea maxima si n este numarul de agenti.

11.4 Prezentarea codului

```
class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState: GameState):
        """
        Returns the minimax action from the current gameState using self.depth
        and self.evaluationFunction.

        Here are some method calls that might be useful when implementing minimax.

        gameState.getLegalActions(agentIndex):
        Returns a list of legal actions for an agent
        agentIndex=0 means Pacman, ghosts are >= 1

        gameState.generateSuccessor(agentIndex, action):
        Returns the successor game state after an agent takes an action

        gameState.getNumAgents():
        Returns the total number of agents in the game

        gameState.isWin():
```

```
        Returns whether or not the game state is a winning state

        gameState.isLose():
        Returns whether or not the game state is a losing state
        """
        "*** YOUR CODE HERE ***"

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

            if agentIndex == 0:
                return maxValue(agentIndex, depth, gameState)

            else:
                return minValue(agentIndex, depth, gameState)

        def maxValue(agentIndex, depth, gameState):
            legalMoves = gameState.getLegalActions(agentIndex)
            if not legalMoves:
                return self.evaluationFunction(gameState)

            return max(
                minimax(1, depth, gameState.generateSuccessor(agentIndex, action))
                for action in legalMoves
            )

        def minValue(agentIndex, depth, gameState):
            legalMoves = gameState.getLegalActions(agentIndex)
            if not legalMoves:
                return self.evaluationFunction(gameState)

            nextAgent = agentIndex + 1
            nextDepth = depth
            if nextAgent == gameState.getNumAgents():
                nextAgent = 0
                nextDepth += 1

            return min(
                minimax(nextAgent, nextDepth, gameState.generateSuccessor(agentIndex, actio
                for action in legalMoves
            )
        legalMoves = gameState.getLegalActions(0)
        bestAction = None
        bestScore = float('-inf')

        for action in legalMoves:
                successor = gameState.generateSuccessor(0, action)
```

```python
                score = minimax(1, 0, successor)
                if score > bestScore:
                    bestScore = score
                    bestAction = action

        return bestAction
```

12 Question 3 - Alpha-Beta Pruning

12.1 Descrierea algoritmului

Alpha-Beta pruning optimizeaza algoritmul Minimax reducand numarul de stari evaluate. Se folosesc doua variabile: - α: cel mai bun scor pentru agentul maximizator (Pacman) pana acum. - β: cel mai bun scor pentru agentii minimizatori (fantomele) pana acum.

12.2 Explicatia implementarii

1. **maxValue:** Pacman, agentul maximizator, calculeaza scorul maxim pentru fiecare succesor. Daca $maxValue \geq \beta$, calculul pentru ramura curenta este intrerupt.
2. **minValue:** Fantomele, agentii minimizatori, calculeaza scorul minim pentru fiecare succesor. Daca $minValue \leq \alpha$, calculul pentru ramura curenta este intrerupt.
3. **Selectia actiunii:** Functia `getAction` initiaza evaluarea starilor, iterand prin toate actiunile legale si selectand actiunea care maximizeaza scorul pentru Pacman, folosind pruning pentru a reduce numarul de stari explorate.

12.3 Complexitatea algoritmului

In cel mai bun caz, complexitatea este $O(b^{d/2})$, daca actiunile sunt evaluate intr-o ordine favorabila.
In cel mai rau caz, complexitatea ramane $O(b^d)$, similar algoritmului Minimax.

12.4 Prezentarea codului

```python
class AlphaBetaAgent(MultiAgentSearchAgent):
    """
    Your minimax agent with alpha-beta pruning (question 3)
    """

    def getAction(self, gameState: GameState):
        """
        Returns the minimax action using self.depth and self.evaluationFunction
        """
        "*** YOUR CODE HERE ***"

        def max_value(gameState, depth, alpha, beta):
            curr_depth = depth + 1
            if gameState.isWin() or gameState.isLose() or curr_depth == self.depth:
                return self.evaluationFunction(gameState)
            maxvalue = -999999
```

```
            actions = gameState.getLegalActions(0)
            alpha1 = alpha
            for action in actions:
                successor = gameState.generateSuccessor(0, action)
                maxvalue = max(maxvalue, minLevel(successor, curr_depth, 1, alpha1, beta))
                if maxvalue > beta:
                    return maxvalue
                alpha1 = max(alpha1, maxvalue)
            return maxvalue

        # for all ghosts
        def minLevel(gameState, depth, agentIndex, alpha, beta):
            minvalue = 999999
            if gameState.isWin() or gameState.isLose():
                return self.evaluationFunction(gameState)
            actions = gameState.getLegalActions(agentIndex)
            beta1 = beta
            for action in actions:
                successor = gameState.generateSuccessor(agentIndex, action)
                if agentIndex == (gameState.getNumAgents() - 1):
                    minvalue = min(minvalue, max_value(successor, depth, alpha, beta1))
                    if minvalue < alpha:
                        return minvalue
                    beta1 = min(beta1, minvalue)
                else:
                    minvalue = min(minvalue, minLevel(successor, depth, agentIndex + 1, alph
                    if minvalue < alpha:
                        return minvalue
                    beta1 = min(beta1, minvalue)
            return minvalue

        #alpha beta pruning
        actions = gameState.getLegalActions(0)
        currentScore = -999999
        returnAction = ''
        alpha = -999999
        beta = 999999
        for action in actions:
            nextState = gameState.generateSuccessor(0, action)
            score = minLevel(nextState, 0, 1, alpha, beta)
            if score > currentScore:
                returnAction = action
                currentScore = score
            if score > beta:
                return returnAction
            alpha = max(alpha, score)
        return returnAction
```