UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Vlad Durdeu și Moroz Lion
Grupa: 30232

Facultatea de Automatica
si Calculatoare

2 Decembrie 2024

Cuprins

1 Uninformed search

1.1 Question 1 - Depth First Search

În această întrebare, scopul este găsirea unui punct fix de mâncare folosind algoritmul **Depth First Search (DFS)**. DFS explorează nodurile unui graf mergând pe cea mai adâncă cale înainte de a reveni la nodurile anterioare.

Codul principal este implementat în funcția `depthFirstSearch` din `search.py` și este descris mai jos:

```
def depthFirstSearch(problem):
    stack = util.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 successor, action, cost in problem.getSuccessors(state):
                if successor not in visited:
                    stack.push((successor, path + [action]))
    return []
```

Acest algoritm utilizează un *stack* (stivă) pentru a explora nodurile și revine la cele anterioare dacă calea curentă nu este validă. Performanța sa depinde de adâncimea soluției și numărul total de noduri, având o complexitate de $\mathcal{O}(V+E)$, unde V este numărul de noduri, iar E este numărul de muchii.

1.2 Question 2 - Breadth First Search

În această întrebare, algoritmul **Breadth First Search (BFS)** este utilizat pentru a determina calea cea mai scurtă dintre punctul de start (`problem.getStartState()`) și punctul de oprire definit de condiția `problem.isGoalState(state)`.

Implementarea din `search.py` este:

```
def breadthFirstSearch(problem):
    queue = util.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 successor, action, cost in problem.getSuccessors(state):
                if successor not in visited:
```

```
                    queue.push((successor, path + [action]))
    return []
```

BFS explorează mai întâi nodurile la nivelul cel mai apropiat, utilizând un *queue* (coadă) și având o complexitate de $\mathcal{O}(V + E)$, unde V este numărul de noduri, iar E este numărul de muchii.

1.3 Question 3 - Varying the Cost Function

În această întrebare este prezentată funcției de cost, algoritmul fiind conceput să selecteze nodurile pe baza celui mai mic cost acumulat.

Implementarea metodei este prezentată mai jos:

```
def uniformCostSearch(problem):
    pqueue = util.PriorityQueue()
    pqueue.push((problem.getStartState(), [], 0), 0)
    visited = set()
    cost_so_far = {problem.getStartState(): 0}
    while not pqueue.isEmpty():
        state, path, stepCost = pqueue.pop()
        if problem.isGoalState(state):
            return path
        if state not in visited:
            visited.add(state)
            for successor, action, cost in problem.getSuccessors(state):
                new_cost = stepCost + cost
                if successor not in cost_so_far or new_cost < cost_so_far[successor]:
                    cost_so_far[successor] = new_cost
                    pqueue.push((successor, path + [action], new_cost), new_cost)
    return []
```

Algoritmul utilizat, **Uniform Cost Search (UCS)**, prioritizează explorarea nodurilor cu cel mai mic cost, garantând găsirea unei soluții optime în condițiile unor costuri variabile. Complexitatea algoritmului este de O(V $\log V + E \log V), unde V$ este numărul de noduri, iar E este numărul de muchii.

2 Informed search

2.1 Question 4 - A* Search

Algoritmul **A*** combină costul real $g(n)$, care reprezintă costul de la starea inițială până la nodul curent n, cu o estimare euristică $h(n)$, ce indică costul minim estimat de la nodul curent la nodul final. Funcția de evaluare folosită este:

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

unde:

- $g(n)$ este costul real acumulat până la nodul n,
- $h(n)$ este o estimare a costului rămas, bazată pe euristică.

Implementarea în cod este următoarea:

```
def aStarSearch(problem, heuristic=nullHeuristic):
    pqueue = util.PriorityQueue()
    pqueue.push((problem.getStartState(), [], 0),
                heuristic(problem.getStartState(), problem))
    cost_so_far = {problem.getStartState(): 0}
    while not pqueue.isEmpty():
        state, path, cost = pqueue.pop()
        if problem.isGoalState(state):
            return path
        for successor, action, stepCost in problem.getSuccessors(state):
            new_cost = cost + stepCost
            if successor not in cost_so_far or new_cost < cost_so_far[successor]:
                cost_so_far[successor] = new_cost
                priority = new_cost + heuristic(successor, problem)
                pqueue.push((successor, path + [action], new_cost), priority)
    return []
```

Algoritmul calculează valoarea $f(n)$ pentru fiecare nod n, utilizând o coadă de priorități pentru a explora mai întâi nodurile cu cel mai mic $f(n)$.

Performanța algoritmului depinde de euristica h. Dacă h este *admisibilă* (nu supraestimează costul real) și *consistentă* (respectă inegalitatea triunghiului), A* garantează găsirea unei soluții optime.

În întrebările ce urmează, euristica utilizată este **distanța Manhattan**, definită astfel:

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

unde (x_1, y_1) și (x_2, y_2) sunt coordonatele nodului curent, respectiv ale nodului țintă.

2.2 Question 5 - Finding All the Corners

Problema implică explorarea tuturor colțurilor dintr-un labirint. Aceasta este definită prin clasa `CornersProblem`, care include metode esențiale pentru configurarea stării inițiale, verificarea scopului și generarea succesorilor.

Metodele principale utilizate sunt următoarele:

- `getStartState`: Această metodă definește starea inițială a problemei, incluzând poziția de start și un tuplu de patru valori booleene pentru a urmări colțurile vizitate.
  Starea inițială este reprezentată ca o tuplă (`startingPosition`, `cornersVisited`), care reprezină dacă colțurile au fost vizitate.
- `isGoalState`: Această metodă determină dacă starea curentă este de tip scop, adică toate cele patru colțuri au fost vizitate:
  Dacă toate valorile din `state[1]` sunt `True`, înseamnă că Pacman a vizitat toate colțurile, iar problema este rezolvată.
- `getSuccessors`: Această metodă verifică fiecare direcție posibilă (nord, sud, est, vest) și calculează poziția următoare, evitând pereții. Dacă poziția următoare este un colț, flag-ul pentru acel colț este setat la `True`.

Implementarea metodelor este prezentată mai jos:

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

```
3
4  def isGoalState(self, state):
5      return all(state[1])
6
7  def getSuccessors(self, state):
8      successors = []
9      for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
10         x, y = state[0]
11         dx, dy = Actions.directionToVector(action)
12         nextx, nexty = int(x + dx), int(y + dy)
13         if not self.walls[nextx][nexty]:
14             nextPosition = (nextx, nexty)
15             cornersVisited = list(state[1])
16             if nextPosition in self.corners:
17                 cornersVisited[self.corners.index(nextPosition)] = True
18             successors.append(((nextPosition, tuple(cornersVisited)), action, 1))
19     self._expanded += 1
20     return successors
```

2.3 Question 6 - Corners Problem: Heuristic

Euristica dezvoltată pentru problema colțurilor calculează o estimare a costului minim necesar pentru a vizita toate colțurile rămase. Aceasta se bazează pe distanța Manhattan și adaugă distanțele minime necesare pentru a vizita colțurile neexplorate.

Implementarea metodei este următoarea:

```
1  def cornersHeuristic(state, problem):
2      corners = problem.corners
3      walls = problem.walls
4      heuristic = 0
5
6      cornersVisited = state[1]
7      unvisitedCorners = [corner for i, corner in enumerate(corners)
8                          if not cornersVisited[i]]
9
10     if not unvisitedCorners:
11         return 0
12
13     current = state[0]
14
15     while unvisitedCorners:
16         distances = [(util.manhattanDistance(current, corner), corner)
17                      for corner in unvisitedCorners]
18         minDistance, nextCorner = min(distances)
19         heuristic += minDistance
20         current = nextCorner
21         unvisitedCorners.remove(nextCorner)
22
23     return heuristic
```

Această metodă funcționează astfel:

- Se identifică toate colțurile care nu au fost vizitate (`unvisitedCorners`).
- Începând de la poziția curentă (`current`), se calculează distanța Manhattan de la nodul curent la fiecare colț neexplorat.
- Se selectează colțul cel mai apropiat, se adaugă distanța către acesta la euristică și se actualizează poziția curentă.
- Procesul continuă până când toate colțurile au fost vizitate.

Euristica returnează suma distanțelor minime necesare pentru a vizita toate colțurile rămase. Este:

- **Admisibilă**: Nu supraestimează costul real, deoarece se folosește distanța Manhattan.
- **Consistentă**: Respectă inegalitatea triunghiului. Pentru orice două stări consecutive n și n', diferența dintre euristicile acestora nu depășește costul deplasării directe între ele.

2.4 Question 7 - Eating All the Dots

Scopul acestei probleme este găsirea unei soluții eficiente pentru ca Pacman să consume toate punctele de mâncare de pe hartă. Problema este definită prin clasa `FoodSearchProblem`, iar euristica utilizată este implementată în metoda `foodHeuristic`.

Implementarea metodei este următoarea:

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    foodList = foodGrid.asList()

    if not foodList:
        return 0

    left = min(food[0] for food in foodList)
    right = max(food[0] for food in foodList)
    top = min(food[1] for food in foodList)
    bottom = max(food[1] for food in foodList)

    minDistance = min(util.manhattanDistance(position, food) for food in foodList)
    maxDistance = (right - left) + (bottom - top)
    heuristic = minDistance + maxDistance

    return heuristic
```

Această metodă funcționează astfel:

- Dacă nu există puncte de mâncare (`foodList` este goală), euristica returnează 0.
- Se calculează următoarele valori:
  - `minDistance`: Distanța Manhattan de la poziția curentă la cel mai apropiat punct de hrană.
  - `maxDistance`: Suma diferențelor maxime între coordonatele extreme (stânga-dreapta, sus-jos) ale punctelor de mâncare.
- Euristica este suma celor două valori:

$$heuristic = \texttt{minDistance} + \texttt{maxDistance}$$

Această euristică are următoarele proprietăți:

- Este **admisibilă**, deoarece nu supraestimează costul real. Suma distanțelor este întotdeauna mai mică sau egală cu costul minim necesar pentru a consuma toate punctele de mâncare.
- Este **consistentă**, deoarece respectă inegalitatea triunghiului. Pentru orice două stări consecutive n și n', diferența dintre euristicile acestora nu depășește costul deplasării directe între ele.

2.5 Question 8 - Suboptimal Search

În această problemă, se implementează o abordare suboptimă pentru consumarea tuturor punctelor de hrană. Metoda utilizată presupune identificarea celui mai apropiat punct de mâncare și navigarea către aceasta.

Metoda principală, `findPathToClosestDot`, este implementată astfel:

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

    return search.bfs(problem)
```

Descrierea metodei:

- Se obține poziția inițială a lui Pacman folosind `gameState.getPacmanPosition()`.
- Se obține grila punctelor de mâncare și grila pereților.
- Se creează o instanță a clasei `AnyFoodSearchProblem`, care definește problema căutării unui punct de mâncare.
- Se utilizează algoritmul **Breadth First Search (BFS)** pentru a găsi calea către cel mai apropiat punct de mâncare.

3 Adversarial search

3.1 Question 1 - Reflex Agent

Metoda `evaluationFunction` calculează un scor pentru fiecare acțiune posibilă, luând în considerare mai mulți factori ai stării jocului. Implementarea este prezentată mai jos:

```
def evaluationFunction(currentGameState, action):
    successorGameState = currentGameState.generatePacmanSuccessor(action)
    newPos = successorGameState.getPacmanPosition()
    newFood = successorGameState.getFood()
    newGhostStates = successorGameState.getGhostStates()
    newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
    newCapsules = successorGameState.getCapsules()

    foodList = newFood.asList()
    closestFoodDistance = min([manhattanDistance(newPos, food)
                           for food in foodList], default=0)
    closestGhostDistance = min([manhattanDistance(newPos, ghost.getPosition())
                            for ghost in newGhostStates], default=float('inf'))
```

```
    closestCapsuleDistance = min([manhattanDistance(newPos, capsule)
                                  for capsule in newCapsules], default=float('inf'))

    score = successorGameState.getScore()

    if closestFoodDistance > 0:
        score += 1.0 / closestFoodDistance

    if closestCapsuleDistance < float('inf'):
        score += 2.0 / closestCapsuleDistance

    for i, ghostState in enumerate(newGhostStates):
        if newScaredTimes[i] > 0:
            score += 200 / (manhattanDistance(newPos, ghostState.getPosition()) + 1)
        else:
            if closestGhostDistance > 0:
                score -= 10 / closestGhostDistance

    return score
```

Algoritmul calculează următoarele valori:

- **Distanța până la cea mai apropiată mâncare** (`closestFoodDistance`): Cu cât Pacman este mai aproape de mâncare, cu atât scorul va crește.
- **Distanța până la capsule** (`closestCapsuleDistance`): Capsulelele oferă un scor suplimentar, încurajându-l pe Pacman să le consume.
- **Distanța până la fantome**:
  - Dacă o fantomă este „scared" (timp rămas pozitiv în `newScaredTimes`), apropierea de ea crește scorul.
  - Dacă fantoma nu este „scared", apropierea excesivă reduce scorul.
- **Scorul stării succesive** (`successorGameState.getScore()`): Scorul general al stării este folosit ca bază pentru calcul.

3.2 Question 2 - Minimax

Algoritmul **Minimax** este utilizat pentru a determina cea mai bună strategie pentru Pacman într-un mediu adversarial. Pacman acționează ca **agentul max**, încercând să maximizeze scorul obținut, în timp ce fantomele, **agenții min** încearcă să reducă scorul prin minimizarea acestuia.

Implementarea metodei `getAction` este următoarea:

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

        numAgents = state.getNumAgents()
        nextAgent = (agentIndex + 1) % numAgents
        nextDepth = depth + 1 if nextAgent == 0 else depth

        if agentIndex == 0:
```

```
            return max(minimax(nextAgent, nextDepth,
                   state.generateSuccessor(agentIndex, action))
                   for action in state.getLegalActions(agentIndex))
        else:
            return min(minimax(nextAgent, nextDepth,
                   state.generateSuccessor(agentIndex, action))
                   for action in state.getLegalActions(agentIndex))

    legalMoves = gameState.getLegalActions(0)
    scores = [minimax(1, 0,
             gameState.generateSuccessor(0, action))
             for action in legalMoves]
    bestScore = max(scores)
    bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
    chosenIndex = random.choice(bestIndices)
    return legalMoves[chosenIndex]
```

Descrierea algoritmului:

- **Punct de oprire**: Dacă starea este terminală (`win` sau `lose`) sau dacă adâncimea maximă a fost atinsă, se returnează valoarea dată de `evaluationFunction`.
- **Pacman (Agentul Max)**:
  - Pacman caută să maximizeze scorul alegând succesorul cu cel mai mare scor minim returnat de fantome.
- **Fantomele (Agenții Min)**:
  - Fantomele încearcă să minimizeze scorul alegând succesorul cu cel mai mic scor returnat de Pacman.
- **Trecerea între agenți**:
  - Algoritmul iterează între agenți, avansând în adâncime la fiecare tur complet.

3.3 Question 3 - Alpha-Beta Pruning

Algoritmul **Alpha-Beta Pruning** optimizează procesul de căutare Minimax prin reducerea numărului de stări evaluate. Acesta utilizează două valori, `alpha` și `beta`, pentru a menține limitele scorurilor maxime și minime posibile în arborele de joc.

Implementarea metodei `getAction` este următoarea:

```
def getAction(gameState):
    def alphaBeta(agentIndex, depth, state, alpha, beta):
        if state.isWin() or state.isLose() or depth == self.depth:
            return self.evaluationFunction(state)

        numAgents = state.getNumAgents()
        nextAgent = (agentIndex + 1) % numAgents
        nextDepth = depth + 1 if nextAgent == 0 else depth

        if agentIndex == 0:
            value = float('-inf')
            for action in state.getLegalActions(agentIndex):
                value = max(value, alphaBeta(nextAgent, nextDepth,
```

```
                        state.generateSuccessor(agentIndex, action), alpha, beta))
                if value > beta:
                    return value
                alpha = max(alpha, value)
            return value
        else:
            value = float('inf')
            for action in state.getLegalActions(agentIndex):
                value = min(value, alphaBeta(nextAgent, nextDepth,
                        state.generateSuccessor(agentIndex, action), alpha, beta))
                if value < alpha:
                    return value
                beta = min(beta, value)
            return value

    alpha = float('-inf')
    beta = float('inf')
    bestAction = None
    bestValue = float('-inf')
    for action in gameState.getLegalActions(0):
        value = alphaBeta(1, 0,
                gameState.generateSuccessor(0, action), alpha, beta)
        if value > bestValue:
            bestValue = value
            bestAction = action
        alpha = max(alpha, bestValue)

    return bestAction
```

Descrierea algoritmului:

- **Punct de oprire**: Algoritmul se oprește când jocul este câștigat, pierdut sau adâncimea maximă este atinsă, utilizând `evaluationFunction` pentru evaluare.
- **Alpha (α)**: Păstrează cel mai bun scor găsit de agentul Max (Pacman).
- **Beta (β)**: Păstrează cel mai bun scor găsit de agenții Min (fantomele).
- **Tăierea (pruning)**: Dacă un nod are un scor mai bun pentru adversar decât limita α sau β, atunci sub-arborele acestuia este eliminat din căutare.