
# Berkeley Pac-Man Project

*Inteligenta Artificiala*

Autor: Ciobanu Radu-Rares

Grupa: 30232

Facultatea de Automatica
si Calculatoare

3 Decembrie 2024

# Contents

# 1 Uninformed search

## 1.1 Question 1 - Depth First Search

Acest algoritm implementează căutarea în adâncime pentru a găsi o secvență de acțiuni care duc de la starea inițială la starea țintă. Folosește o stivă pentru a explora stările, adăugând noduri și acțiuni la fiecare pas, iar dacă se ajunge la starea țintă, se returnează drumul de acțiuni.

Algoritmul evită vizitarea acelorași noduri de mai multe ori, iar dacă nu se găsește un drum, returnează o listă goală.

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

    while not stack.isEmpty():
        curr_node, actions = stack.pop()

        if problem.isGoalState(curr_node):
            return actions

        if curr_node not in visited:
            visited.add(curr_node)

        for successor, action, costUnit in
            problem.getSuccessors(curr_node):
            if successor not in visited:
                newActions = actions + [action]
                stack.push((successor, newActions))
    return []
```

## 1.2 Question 2 - Breadth First Search

Acest algoritm implementează căutarea în lățime pentru a găsi o secvență de acțiuni care duc de la starea inițială la starea țintă. Folosește o coadă pentru a explora stările pe măsură ce le întâlnește, adăugând succesori în coadă pe măsură ce sunt descoperiți.

Algoritmul garantează că se va găsi drumul cel mai scurt către obiectiv, dacă există unul. Dacă nu se găsește niciun drum, va returna o listă goală.

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

    while not queue.isEmpty():
        curr_node, actions = queue.pop()
        if problem.isGoalState(curr_node):
            return actions
```

```
        if curr_node not in visited:
            visited.add(curr_node)
            for successor, action, costUnit in
                problem.getSuccessors(curr_node):
                if successor not in visited:
                    newActions = actions + [action]
                    queue.push((successor, newActions))
    return []
```

### 1.3 Question 3 - Varying the Cost Function

Acest algoritm implementează căutarea cu cost uniform pentru a găsi drumul cu cel mai mic cost între starea de început și obiectiv. Folosește o coadă de prioritate pentru a explora stările, selectând mereu nodul cu cel mai mic cost total.

Algoritmul asigură că se va găsi drumul cel mai ieftin, dacă există unul, și îl va returna împreună cu secvența de acțiuni necesare pentru a-l parcurge. Dacă nu se găsește un drum, returnează o listă goală.

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

    cos_t = {start_state: 0}

    while not priorityQueue.isEmpty():
        curr_node, actions, cost = priorityQueue.pop()

        if problem.isGoalState(curr_node):
            return actions

        if curr_node not in visited:
            visited.add(curr_node)
            for successor, action, step_cost in
                problem.getSuccessors(curr_node):
                new_cost = cost + step_cost
                if successor not in cos_t or new_cost < cos_t[successor]:
                    cos_t[successor] = new_cost
                    new_actions = actions + [action]
                    priorityQueue.push((successor, new_actions,
                        new_cost), new_cost)
    return []
```

## 2 Informed search

### 2.1 Question 4 - A* Search

Algoritmul A* combină căutarea cu cost uniform și estimarea heuristică pentru a găsi drumul cel mai eficient către obiectiv. Folosește o coadă de prioritate pentru a explora

stările, alegând mereu nodul cu costul total cel mai mic (costul actual + estimarea costului rămas).

Funcția de heuristică este folosită pentru a ghida căutarea, astfel încât algoritmul să poată identifica mai rapid soluția optimă. Dacă nu există niciun drum posibil, se returnează o listă goală.

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic) ->
    List[Directions]:
    start_state = problem.getStartState()
    frontier = util.PriorityQueue()
    frontier.push((start_state, [], 0), heuristic(start_state, problem))
    cos_t = {start_state: 0}

    while not frontier.isEmpty():
        curr_node, actions, cost = frontier.pop()

        if problem.isGoalState(curr_node):
            return actions

        for successor, action, step_cost in
            problem.getSuccessors(curr_node):
            new_cost = cost + step_cost
            if successor not in cos_t or new_cost < cos_t[successor]:
                cos_t[successor] = new_cost
                new_actions = actions + [action]
                priority = new_cost + heuristic(successor, problem)
                frontier.push((successor, new_actions, new_cost),
                    priority)
    return []
```

## 2.2 Question 5 - Finding All the Corners

Aceste funcții sunt utilizate într-un joc în care agentul trebuie să navigheze printr-un labirint și să viziteze toate colțurile. Funcția `getStartState` inițializează poziția de început și colțurile nevizitate. Funcția `isGoalState` verifică dacă toate colțurile au fost vizitate.

Funcția `getSuccessors` generează mișcările posibile ale agentului, actualizând lista de colțuri vizitate și adăugând fiecare succesor în lista de stări posibile.

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

    def isGoalState(self, state: Any):
        visited_corners = state[1]
        for corner in visited_corners:
            if corner:
                continue
            else:
                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]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                next_position = (nextx, nexty)
                visited_corners = list(state[1])
                if next_position in self.corners:
                    visited_corners[self.corners.index(next_position)] =
                        True
                successors.append(((next_position,
                    tuple(visited_corners)), action, 1))
        self._expanded += 1
        return successors
```

## 2.3 Question 6 - Corners Problem: Heuristic

Această funcție calculează o estimare a costului pentru a vizita toate colțurile rămase, folosind distanța Manhattan între poziția curentă și colțurile neatinse. Algoritmul presupune că agentul vizitează colțurile unul câte unul, alegând mereu cel mai apropiat colț neating.

Această euristică este adesea folosită pentru a ghida algoritmi de căutare (cum ar fi `A*`) într-un mod eficient, oferind o aproximare rezonabilă a costului real.

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    corners = problem.corners
    walls = problem.walls

    visited_corners = state[1]
    left_corners = [corner for i, corner in enumerate(corners) if not
        visited_corners[i]]

    current_point = state[0]
    total_cost = 0

    while left_corners:
        distance, nearest_corner =
            min([(util.manhattanDistance(current_point, corner), corner)
            for corner in left_corners])
        total_cost += distance
        current_point = nearest_corner
        left_corners.remove(nearest_corner)

    return total_cost
```

## 2.4 Question 7 - Eating All the Dots

Heuristica selectează distanța până la piesa de mâncare cea mai îndepărtată ca estimare a costului rămas. Aceasta este o strategie admisibilă (nu supraestimează costul real), deoarece agentul va trebui cel puțin să ajungă la piesa de mâncare cea mai îndepărtată pentru a termina sarcina.

Funcția `mazeDistance` calculează distanța exactă luând în considerare pereții labirintului, oferind o estimare precisă a costului minim necesar pentru a colecta mâncarea.

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

    position, foodGrid = state
    remain_food = foodGrid.asList()
    total_cost = 0
    if not remain_food:
        return 0

    for food in remain_food:
        foodDistance = mazeDistance(position, food,
            problem.startingGameState)
        if foodDistance > total_cost:
            total_cost = foodDistance
    return total_cost
```

## 2.5 Question 8 - Suboptimal Search

Această funcționalitate îi permite lui Pac-Man să identifice și să se deplaseze către cea mai apropiată piesă de mâncare din labirint. Problema este definită în clasa `AnyFoodSearchProblem`, care stabilește pozițiile pieselor de mâncare ca stări țintă (goal states).

Algoritmul `BFS` asigură că drumul găsit este cel mai scurt posibil, deoarece explorează toate pozițiile la o anumită distanță înainte de a trece la distanțe mai mari. Abordarea este eficientă și garantează găsirea celei mai apropiate ținte în termen de pași minimali.

```
    def findPathToClosestDot(self, gameState: pacman.GameState):

        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        return search.bfs(problem)

class AnyFoodSearchProblem(PositionSearchProblem):

    def __init__(self, gameState):
        self.food = gameState.getFood()
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        self._visited, self._visitedlist, self._expanded = {}, [], 0

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

# 3 Adversial search

## 3.1 Question 1 - Reflex Agent

Funcția `evaluationFunction` analizează fiecare acțiune posibilă pentru Pac-Man, luând în considerare următorii factori: proximitatea mâncării, a capsulelor și a fantomelor. Scorul este ajustat pentru a încuraja acțiuni care duc Pac-Man mai aproape de mâncare sau capsule, evitând în același timp pericolul fantomelor. Dacă fantomele sunt speriate, Pac-Man este încurajat să le captureze.

Agentul reflex folosește această funcție pentru a alege acțiunea cu cel mai mare scor în fiecare pas, optimizând strategia sa în mod local, pe baza stării curente de joc.

```
    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]

        score = successorGameState.getScore()

        if newFood:
            next_food = min(util.manhattanDistance(newPos, food) for food in newFood)
            score += 1 / (1 + next_food)

        capsules = currentGameState.getCapsules()
        if capsules:
            nearest_capsule = min(util.manhattanDistance(newPos, capsule) for capsule in capsules)
            score += 50 / (1 + nearest_capsule)

        for ghost, scare_time in zip(newGhostStates, newScaredTimes):
            ghost_distance = util.manhattanDistance(newPos, ghost.getPosition())
            if scare_time > 0:
                score += 500 / (1 + ghost_distance)
            else:
                if ghost_distance < 2:
                    score -= 2000
                else:
                    score += 50 / (1 + ghost_distance)
        return score
```

## 3.2 Question 2 - MiniMax Agent

Funcția implementează un agent care utilizează algoritmul `Minimax` pentru a alege cea mai bună mutare pentru Pac-Man, ținând cont de mutările optime ale fantomelor. Pac-Man încearcă să maximizeze scorul, iar fantomele încearcă să minimizeze progresul lui Pac-Man.

Algoritmul analizează fiecare combinație posibilă de mutări până la o adâncime specificată, iar rezultatul final este determinat pe baza unei funcții de evaluare. Această abordare garantează o strategie bine calculată pentru fiecare mutare, optimizând șansele de succes.

```
    def getAction(self, gameState: GameState):

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

            if agentIndex == 0:
                return max_value(agentIndex, depth, gameState)
            else:
                return min_value(agentIndex, depth, gameState)

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

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

            return min(
                minimax(nextAgent, nextDepth,
                   gameState.generateSuccessor(agentIndex, action))
                for action in legalMoves
            )
        legalMoves = gameState.getLegalActions(0)
        scores = [
            minimax(1, 0, gameState.generateSuccessor(0, action)) for
               action in legalMoves
        ]
        bestScore = max(scores)
        bestIndices = [index for index, score in enumerate(scores) if
           score == bestScore]

        chosenIndex = random.choice(bestIndices)
        return legalMoves[chosenIndex]
```

## 3.3 Question 3 - Alpha-Beta Pruning Agent

Acest agent folosește algoritmul `Alpha-Beta Pruning` pentru a reduce numărul de stări analizate în procesul de luare a deciziilor, păstrând rezultatul optim al algoritmului `Minimax`. Pac-Man joacă rolul agentului max, încercând să maximizeze scorul, iar fantomele sunt agenți min, care încearcă să-l minimizeze.

Algoritmul explorează mutările legale și folosește două limite, alpha (valoarea maximă garantată pentru max) și beta (valoarea minimă garantată pentru min), pentru a elimina ramurile irelevante din arborele de căutare. În funcția `getAction`, agentul evaluează toate mutările posibile pentru Pac-Man și alege cea care duce la cel mai bun scor conform funcției de evaluare. Această funcție ia în considerare poziția mâncării, a capsulelor și distanța până la fantome, atât în stările normale, cât și în cele speriate.

Agentul este mai eficient decât un simplu `Minimax`, dar performanța depinde de calitatea funcției de evaluare.

```
class AlphaBetaAgent(MultiAgentSearchAgent):

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

            if agentIndex == 0:
                return max_value(agentIndex, depth, gameState, alpha,
                    beta)
            else:
                return min_value(agentIndex, depth, gameState, alpha,
                    beta)

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

            v = float('-inf')
            for action in legalMoves:
                successor = gameState.generateSuccessor(agentIndex,
                    action)
                v = max(v, alphabeta(1, depth, successor, alpha, beta))
                if v > beta:
                    return v
                alpha = max(alpha, v)
            return v

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

            v = float('inf')
            nextAgent = (agentIndex + 1) % gameState.getNumAgents()
            nextDepth = depth + 1 if nextAgent == 0 else depth
```

```
35
36              for action in legalMoves:
37                  successor = gameState.generateSuccessor(agentIndex,
                        action)
38                  v = min(v, alphabeta(nextAgent, nextDepth, successor,
                        alpha, beta))
39                  if v < alpha:
40                      return v
41                  beta = min(beta, v)
42              return v
43
44          alpha, beta = float('-inf'), float('inf')
45          legalMoves = gameState.getLegalActions(0)
46          bestAction = None
47          bestScore = float('-inf')
48
49          for action in legalMoves:
50              successor = gameState.generateSuccessor(0, action)
51              score = alphabeta(1, 0, successor, alpha, beta)
52              if score > bestScore:
53                  bestScore = score
54                  bestAction = action
55              alpha = max(alpha, bestScore)
56
57          return bestAction
```