Pacman AI Implementation Documentation

Student Pântea Marius-Nicușor

1 decembrie 2024

Cuprins

1 Introduction

Acest document prezintă implementarea și analiza detaliată a unei serii de algoritmi de Inteligență Artificială aplicați în jocul Pacman. Proiectul a avut ca scop înțelegerea practică a conceptelor fundamentale din IA, de la algoritmi de căutare de bază până la strategii avansate de joc.

Implementarea include algoritmi clasici de căutare (DFS, BFS, UCS), algoritmi informați (A*), precum și tehnici de joc adversarial (Minimax, Alpha-Beta Pruning). Fiecare algoritm a fost adaptat specific pentru diferite provocări din jocul Pacman, cum ar fi găsirea celui mai scurt drum, colectarea tuturor punctelor de mâncare, și luarea deciziilor în prezența fantomelor.

2 Search Algorithms

2.1 Depth First Search (DFS)

```
def depthFirstSearch(problem: SearchProblem) -> List[
    Directions]:

    from util import Stack
    stack = Stack()
    visited = set()
    start_state = problem.getStartState()
    initial_path = []

    stack.push((start_state, initial_path))

    while not stack.isEmpty():
        current_state, current_path = stack.pop()
        if problem.isGoalState(current_state):
            return current_path

        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for successor, action, _ in successors:
                if successor not in visited:
                    new_path = current_path + [action]
                    stack.push((successor, new_path))
    return []
```

Implementarea DFS folosește o stivă pentru a explora mai întâi ramurile cele mai adânci ale arborelui de căutare înainte de a face backtracking. Algoritmul funcționează iterativ, folosind următorul proces:

- Menține o mulțime de stări vizitate pentru a evita ciclurile

- Inițializează o stivă cu starea de start și o cale goală

Cât timp stiva nu este goală:

- Extrage starea curentă și calea asociată
- Verifică dacă este starea țintă
- Dacă nu, marchează starea ca vizitată și explorează succesorii
- Pentru fiecare succesor nevizitat, creează o nouă cale și adaugă în stivă

2.2 Breadth First Search (BFS)

```
def breadthFirstSearch(problem: SearchProblem) -> List[
    Directions]:

 from util import Queue
    queue = Queue()
    visited = set()
    start_state = problem.getStartState()
    initial_path = []

    queue.push((start_state, initial_path))

    while not queue.isEmpty():
        current_state, current_path = queue.pop()
        if problem.isGoalState(current_state):
            return current_path

        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for successor, action, _ in successors:
                if successor not in visited:
                    new_path = current_path + [action]
                    queue.push((successor, new_path))
    return []
```

BFS folosește o coadă pentru a explora nodurile pe niveluri. Diferențele față de DFS:

- Utilizează o coadă în loc de stivă
- Garantează găsirea celui mai scurt drum în cazul costurilor egale
- Se explorează toate stările de la un nivel înainte de trecerea la următorul

2.3 Uniform Cost Search (UCS)

```
def uniformCostSearch(problem: SearchProblem) -> List[
    Directions]:

    from util import PriorityQueue
    queue = PriorityQueue()
    visited = set()
    start_state = problem.getStartState()
    initial_path = []
    initial_cost = 0

    queue.push((start_state, initial_path), initial_cost)

    while not queue.isEmpty():
        current_state, current_path = queue.pop()
        if problem.isGoalState(current_state):
            return current_path

        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for successor, action, step_cost in successors:
                if successor not in visited:
                    new_path = current_path + [action]
                    new_cost = problem.getCostOfActions(
    new_path)
                    queue.push((successor, new_path),
    new_cost)
    return []
```

Implementarea Uniform Cost Search pentru găsirea celui mai scurt drum într-un spațiu de stări folosește o coadă de priorități pentru a explora stările în ordinea costului cumulat.Returnează o lista de acțiuni care duc la starea țintă cu cost minim, sau lista vidă dacă nu există soluție.

Caracteristici principale:

- Completitudine: Garantează găsirea unui drum către starea țintă dacă acesta există
- Optimalitate: Garantează găsirea drumului cu cel mai mic cost total
- Explorare bazată pe cost: Vizitează întotdeauna starea nevizitată cu costul total minim

Funcționare:

- Menține o coadă de priorități unde prioritatea este costul total al drumului

- Pentru fiecare stare, păstrează atât starea cât și drumul care a dus la ea
- Folosește un set pentru a marca stările vizitate și a evita ciclurile
- La fiecare pas, extrage starea cu costul minim și explorează succesorii ei

3 A* Search

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic) -> List[Directions]:

    from util import PriorityQueue
    start_state = problem.getStartState()
    frontier = PriorityQueue()
    frontier.push((start_state, [], 0), 0)
    explored = {}

    while not frontier.isEmpty():

        state, actions, cost_so_far = frontier.pop()

        if state in explored and cost_so_far >= explored[state]:
            continue

        explored[state] = cost_so_far

        if problem.isGoalState(state):
            return actions

        for successor, action, step_cost in problem.getSuccessors(state):
            new_cost = cost_so_far + step_cost
            if successor not in explored or new_cost < explored[successor]:
                new_actions = actions + [action]
                priority = new_cost + heuristic(successor, problem)
                frontier.push((successor, new_actions, new_cost), priority)

    return []
```

Implementarea algoritmul A* Search combină costul drumului parcurs (g(n)) cu o euristică (h(n)) pentru a găsi drumul optim către țintă. Este o

îmbunătățire a algoritmului Uniform Cost Search prin adăugarea unei euristici pentru ghidarea căutării.

Caracteristici principale:

- Optimalitate: Garantează găsirea drumului optim când euristica este admisibilă
- Completitudine: Găsește întotdeauna o soluție dacă aceasta există
- Eficiență: Mai eficient decât UCS prin folosirea euristicii pentru ghidarea căutării

Functionare:

- Folosește f(n) = g(n) + h(n) ca prioritate, unde: * g(n) = costul real până la starea curentă * h(n) = estimarea costului rămas până la țintă
- Menține un dicționar 'explored' cu costurile minime cunoscute până la fiecare stare
- Permite re-explorarea stărilor dacă se găsește un drum mai bun

4 Finding All the Corners

4.1 Corners Problem Implementation

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

def isGoalState(self, state: Any):
    position, corners = state
    return len(corners) == 0

def getSuccessors(self, state: Any):
    successors = []
    currentPosition, remainingCorners = state
    for action in [Directions.NORTH, Directions.SOUTH,
Directions.EAST, Directions.WEST]:
        x, y = currentPosition
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            nextPosition = (nextx, nexty)
            newRemainingCorners = tuple(corner for corner in
remainingCorners
```

```
                                        if corner !=
    nextPosition)
                successors.append(((nextPosition,
    newRemainingCorners), action, 1))
     self._expanded += 1
     return successors
```

Implementarea Corners Problem reprezintă o problemă de căutare în care Pacman trebuie să viziteze toate colțurile unui labirint, utilizând o stare definită ca un tuplu format din poziția curentă și colțurile nevizitate. Pentru rezolvarea problemei, s-au implementat trei metode principale:

- getStartState(): care returnează poziția inițială și lista de colțuri
- isGoalState(): care verifică dacă toate colțurile au fost vizitate
- getSuccessors(): care generează stările următoare posibile verificând cele patru direcții de mișcare și actualizând lista de colțuri nevizitate

5 Corner Heuristic

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    current_pos, unvisited_corners = state
    if len(unvisited_corners) == 0:
        return 0

    closest_distance = float('inf')
    for corner in unvisited_corners:
        distance = util.manhattanDistance(current_pos, corner
    )
        if distance < closest_distance:
            closest_distance = distance
    if len(unvisited_corners) == 1:
        return closest_distance

    max_corner_distance = 0
    for i in range(len(unvisited_corners)):
        corner1 = unvisited_corners[i]
        for j in range(i + 1, len(unvisited_corners)):
            corner2 = unvisited_corners[j]
            distance = util.manhattanDistance(corner1,
    corner2)
            if distance > max_corner_distance:
                max_corner_distance = distance

    return closest_distance + max_corner_distance
```

Implementarea Corner Heuristic este o funcție euristică pentru problema colțurilor care estimează costul minim până la vizitarea tuturor colțurilor rămase, folosind două componente principale: distanța Manhattan până la cel mai apropiat colț nevizitat (closest_distance) și distanța maximă dintre oricare două colțuri nevizitate rămase (max_corner_distance) Euristica este admisibilă deoarece:

- Trebuie să parcurgem cel puțin distanța până la cel mai apropiat colț
- Trebuie să traversăm cel puțin distanța maximă între colțuri pentru a le vizita pe toate
- Distanța Manhattan este întotdeauna mai mică sau egală cu distanța reală

Euristica folosește cazuri speciale pentru eficiență:

- Returnează 0 când nu mai sunt colțuri de vizitat
- Returnează doar distanța până la ultimul colț când a rămas unul singur
- Combină ambele distanțe pentru cazul general cu multiple colțuri

6 Eating All The Dots

6.1 Food Search Implementation

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

     if not food_list:
         return 0

     distances = []
     for food in food_list:
         distance = mazeDistance(position, food, problem.
    startingGameState)
         distances.append(distance)
     return max(distances)
```

Euristică pentru problema Food Search în Pacman care estimează costul minim până la colectarea întregii mâncări folosind distanța maximă între poziția curentă și oricare punct de mâncare rămas.

Această euristică este:

- Admisibilă - nu supraestimează niciodată costul real deoarece Pacman trebuie să parcurgă cel puțin distanța până la cel mai îndepărtat punct de mâncare
- Consistentă - diferența între valorile euristice ale stărilor adiacente nu depășește costul tranziției
- Informativă - ghidează eficient căutarea folosind distanțele reale din labirint

Folosește mazeDistance pentru a calcula distanța exactă prin labirint între două puncte, luând în considerare zidurile. Acest lucru face euristica mult mai precisă decât folosirea distanței Manhattan sau Euclidiene, deoarece:

- Calculează calea reală pe care Pacman trebuie să o parcurgă
- Ia în considerare obstacolele și zidurile din labirint
- Oferă distanța minimă reală între două puncte în labirint

7 Suboptimal Search

7.1 Closest Dot Search Agent

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

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

Implementarea Suboptimal Search (Greedy Agent) este o strategie simplă dar eficientă pentru colectarea punctelor de mâncare în Pacman, folosind findPathToClosestDot care aplică BFS pentru a găsi mereu drumul până la cel mai apropiat punct de mâncare. Deși această abordare nu garantează găsirea celei mai scurte căi pentru colectarea tuturor punctelor (de aceea este 'suboptimală'), ea oferă o soluție practică și rapidă, folosind strategia greedy de a merge mereu către cel mai apropiat obiectiv. Strategia suboptimală:

- Găsește cel mai apropiat punct de mâncare

- Folosește BFS pentru găsirea drumului
- Simplifică problema prin focalizarea pe ținte apropiate

8 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]

    foodList = newFood.asList()
    if foodList:
        foodDistances = [manhattanDistance(newPos, food) for food in foodList]
        nearestFoodDist = min(foodDistances)
    else:
        nearestFoodDist = 0

    ghostDistances = [manhattanDistance(newPos, ghost.getPosition())
                      for ghost in newGhostStates]
    nearestGhostDist = min(ghostDistances) if ghostDistances else float('inf')
    scaredBonus = sum(ghostState.scaredTimer for ghostState in newGhostStates)

    foodScore = -nearestFoodDist
    ghostScore = 10 if nearestGhostDist < 2 else 0
    foodCountPenalty = -len(foodList) * 10
    scaredGhostBonus = scaredBonus * 5

    totalScore = (successorGameState.getScore() + foodScore +
                  foodCountPenalty + scaredGhostBonus - ghostScore)
    return totalScore
```

Agentul reflex evaluează stările folosind:

- Distanța până la cea mai apropiată mâncare
- Distanța până la fantome

- Numărul total de puncte de mâncare rămase
- Timpul în care fantomele sunt speriate

foodCountPenalty = -len(foodList) * 10

- Factorul 10 este suficient de mare pentru a face diferența semnificativă în scorul total
- Dacă penalizarea ar fi mai mică (de ex *2), Pacman ar putea prefera să evite fantome în loc să mănânc
- Dacă ar fi prea mare (de ex *50), Pacman ar risca prea mult încercând să mănânce, ignorând pericolele

scaredGhostBonus = scaredBonus * 5

- Factorul 5 este mai mic decât penalizarea : Mâncarea este obiectivul principal si urmărirea fantomelor speriate este secundară
- Este totuși suficient de mare pentru a face atractivă urmărirea fantomelor speriate

ghostScore = 10 if nearestGhostDist (mai mic decat) 2 else 0

- Valoarea 10 este o penalizare imediată și severă pentru apropierea de fantome
- Distanța de 2 unități este un prag bun pentru siguranță, oferind timp de reacție
- Este egală cu penalizarea pentru un punct de mâncare rămas

9 Agent Minimax

```
def getAction(self, gameState: GameState):
    bestValue = float('-inf')
    bestAction = None
    legalActions = gameState.getLegalActions(0)

    for action in legalActions:
```

```
7           successorState = gameState.generateSuccessor(0,
    action)
8           value = self.getMinValue(successorState, self.depth,
    1)
9           if value > bestValue:
10              bestValue = value
11              bestAction = action
12
13      return bestAction
14
15  def getMaxValue(self, gameState, depth, agentIndex):
16      if gameState.isWin() or gameState.isLose() or depth == 0:
17          return self.evaluationFunction(gameState)
18
19      value = float('-inf')
20      legalActions = gameState.getLegalActions(agentIndex)
21
22      for action in legalActions:
23          successorState = gameState.generateSuccessor(
    agentIndex, action)
24          value = max(value, self.getMinValue(successorState,
    depth, agentIndex + 1))
25
26      return value
27
28  def getMinValue(self, gameState, depth, agentIndex):
29      if gameState.isWin() or gameState.isLose() or depth == 0:
30          return self.evaluationFunction(gameState)
31
32      value = float('inf')
33      legalActions = gameState.getLegalActions(agentIndex)
34
35      if agentIndex == gameState.getNumAgents() - 1:
36          for action in legalActions:
37              successorState = gameState.generateSuccessor(
    agentIndex, action)
38              value = min(value, self.getMaxValue(
    successorState, depth - 1, 0))
39      else:
40          for action in legalActions:
41              successorState = gameState.generateSuccessor(
    agentIndex, action)
42              value = min(value, self.getMinValue(
    successorState, depth, agentIndex + 1))
43      return value
```

Implementarea agentului Minimax pentru Pacman determină cea mai bună acțiune evaluând toate posibilitățile până la o adâncime specificată. Consideră Pacman ca agent de maximizare și fantomele ca agenți de mini-

mizare. Structura algoritmului:

- getAction(): Punctul de intrare care evaluează toate acțiunile posibile ale lui Pacman. Returnează acțiunea cu cel mai bun scor maxim posibil
- getMaxValue(): Punctul de intrare care evaluează toate acțiunile posibile ale lui Pacma. Returnează acțiunea cu cel mai bun scor maxim posibil
- getMinValue():- Reprezintă nivelurile fantomelor. Caută acțiunile care minimizează scorul maxim posibil. Gestionează tranziția între fantome și înapoi la Pacman

Exploreaza complet până la adâncimea specificată , alternand corect între Pacman și multiple fantome.

10 Alpha-Beta Pruning

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

        alpha = float('-inf')
        beta = float('inf')
        bestValue = float('-inf')
        bestAction = None

        legalActions = gameState.getLegalActions(0)
        for action in legalActions:
            successorState = gameState.generateSuccessor(0,
action)
            value = self.getMinValue(successorState, self.
depth, 1, alpha, beta)
            if value > bestValue:
                bestValue = value
                bestAction = action
            alpha = max(alpha, bestValue)

        return bestAction

    def getMaxValue(self, gameState, depth, agentIndex, alpha
, beta):

        if gameState.isWin() or gameState.isLose() or depth
== 0:
            return self.evaluationFunction(gameState)

        value = float('-inf')
```

```
25          legalActions = gameState.getLegalActions(agentIndex)
26
27          for action in legalActions:
28              successorState = gameState.generateSuccessor(
    agentIndex, action)
29              value = max(value, self.getMinValue(
    successorState, depth, agentIndex + 1, alpha, beta))
30              if value > beta:
31                  return value
32              alpha = max(alpha, value)
33
34          return value
35
36      def getMinValue(self, gameState, depth, agentIndex, alpha
    , beta):
37
38          if gameState.isWin() or gameState.isLose() or depth
    == 0:
39              return self.evaluationFunction(gameState)
40
41          value = float('inf')
42          legalActions = gameState.getLegalActions(agentIndex)
43
44          if agentIndex == gameState.getNumAgents() - 1:
45              for action in legalActions:
46                  successorState = gameState.generateSuccessor(
    agentIndex, action)
47                  value = min(value, self.getMaxValue(
    successorState, depth - 1, 0, alpha, beta))
48                  if value < alpha:
49                      return value
50                  beta = min(beta, value)
51          else:
52              for action in legalActions:
53                  successorState = gameState.generateSuccessor(
    agentIndex, action)
54                  value = min(value, self.getMinValue(
    successorState, depth, agentIndex + 1, alpha, beta))
55                  if value < alpha:
56                      return value
57                  beta = min(beta, value)
58
59          return value
```

Implementarea algoritmul Minimax cu tăiere Alpha-Beta pentru Pacman, o optimizare a algoritmului Minimax care elimină explorarea ramurilor care nu pot influența decizia finală. Menține aceeași optimalitate ca Minimax dar cu eficiență mult îmbunătățită. Componente principale:

- Alpha : Cea mai bună valoare găsită pentru Pacman
- Beta : Cea mai bună valoare găsită Fantome
- Tăiere : Oprește explorarea când alfa mai mare decat beta (nu se poate găsi o valoare mai bună)

Structura algoritmului:

- getAction(): Nivelul rădăcină, inițializează alfa și beta
- getMaxValue(): Pentru Pacman
- getMinValue(): Pentru fantome . Gestionează tranzițiile între fantome și adâncime

Optimizări față de Minimax:

- Elimină explorarea ramurilor inutile
- Menține garanția de optimalitate
- Reduce complexitatea în cazul mediu

11 Conclusions

Implementarea acestui proiect a oferit o perspectivă practică asupra modului în care diferite concepte din Inteligența Artificială pot fi aplicate într-un scenariu de joc real. Prin dezvoltarea și testarea diferitelor componente, am observat:

- Importanța alegerii algoritmului potrivit pentru fiecare tip de problemă : de exemplu, BFS pentru găsirea drumului optim vs. strategii greedy pentru decizii rapide
- Rolul crucial al euristicilor în îmbunătățirea performanței - atât în căutarea informată (A*) cât și în evaluarea stărilor de joc
- Complexitatea implementării agenților inteligenți care trebuie să ia decizii în timp real, ținând cont de multiple obiective și constrângeri
- Beneficiile semnificative ale optimizărilor precum Alpha-Beta Pruning în reducerea spațiului de căutare