Pacman - Inteligenta Artificiala

Vlad Ursache

5 noiembrie 2022

Rezumat

Am implementat urmatoarele functionalitati:

- Cautare DFS, Cautare BFS, Cautare cu cost uniform, Caurare A* (cu diferite euristici),
- Problema colturilor (cu tot cu euristica), optimal sau suboptimal,
- Agentul reflex, Algoritmul MiniMax si Alpha-Beta Pruning.

Am testat tot ce am implementat si am notat niste observatii:

- BFS este un algoritm complet, optim. Din pacate, expandeaza multe noduri.
- DFS este complet, dar nu este optim. Din fericire, face economie la expandarea de noduri.
- Cautarea cu cost uniform este de fapt algoritmul lui Dijkstra.
- A* este un caz general de BFS, avand costuri si euristici.
- Agentul reflex in cauza tinde sa stea pe loc daca nu este urmarit de o fantoma. Acesta reactioneaza numai in proximitatea acesteia.
- Alpha-Beta Pruning este doar o optimizare a algorimului MiniMax. Acesta poate fi dus mai departe prin algoritmul ExpectiMiniMax (folosit de exemplu pentru jocul de table) pentru un adversar care nu ia decizii rationale.

1 Partea I

In aceasta sectiune voi prezenta algorimtii de cautare DFS, BFS, cu cost uniform si A*. Primii 3 algoritmi sunt foarte similari. Singura diferenta dintre ei este structura de date utilizata pentru frontiera. In cazul DFS, aceasta este o stiva. In cazul BFS, aceasta este o coada. In cazul cautarii cu cost uniform este utilizata o coada de prioritati. De execmplu:

```
def depthFirstSearch(problem):
    stack = util.Stack()  # frontier
    visited = []  # explored
    current = (problem.getStartState(), [])
    stack.push(current)

    while not stack.isEmpty():
        current = stack.pop()

        if problem.isGoalState(current[0]):
            return current[1]

        if current[0] not in visited:
            visited.append(current[0])
            for successor in problem.getSuccessors(current[0]):
                if successor[0] not in visited:
                    path = current[1] + [successor[1]]
                    stack.push([successor[0], path])
```

A* este un caz general al BFS. Diferenta costa in faptul ca avem de-a face cu costuri. De asemenea, ii puteam atasa o euristica. Am creat doua euristici (nerealiste pentru problema):

```
def chebyshevHeuristic(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    return max(abs(xy1[0] - xy2[0]), abs(xy1[1] - xy2[1]))


def reverseChebyshevHeuristic(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    return min(abs(xy1[0] - xy2[0]), abs(xy1[1] - xy2[1]))
```

Varianta mea personala este una cu ponderi. Costul propriu-zis valoreaza 30% din costul total, in vreme ce 70% reprezinta euristica aleasa (oricare ar fi ea).

2 Partea a II-a

In aceasta sectiune este vorba despre problema colturilor. Am reusit sa proiectez o euristica consistenta cu ajutorul distantei Manhattan pentru cautarea cu A*, numita cornersHeuristic.

```
def cornersHeuristic(state, problem):
    corners = problem.corners  # These are the corner coordinates
    walls = problem.walls  # These are the walls of the maze, as a Grid

    #  0 - position, 1 - corners
    heuristic = 0
    unvisited = state[1]
    position = state[0]
    distances = []

    if len(unvisited) == 0:  # is goal state
        return heuristic
    for i in unvisited:
        distance = abs(i[0] - position[0]) + abs(i[1] - position[1])
        distances.append((distance, i))
    distances = sorted(distances, reverse=True)

    heuristic += distances[0][0]
    return heuristic
```

O alta euristica, foodHeuristic functioneaza in mod similar, doar ca se foloseste de o functia deja definita, mazeDistance pentru problema cautarii mancarii.

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

    heuristic = 0
    unvisited = foodGrid.asList()
    position = state[0]
    distances = []

    if len(unvisited) == 0:
        return heuristic

    for i in unvisited:
        for j in unvisited:
            distance = mazeDistance(i, j, problem.startingGameState)
            distances.append((distance, i, j))
    distances = sorted(distances, reverse=True)
```

```
    heuristic += distances[0][0]
    distance2 = [mazeDistance(distances[0][1],
                    position, problem.startingGameState),
                 mazeDistance(distances[0][2], position,
                    problem.startingGameState)]
    distance2 = sorted(distance2)
    heuristic += distance2[0]
    return heuristic
```

Pentru cautarea suboptima, folosim BFS pentru a gasi distanta cea mai scurta dintre pozitia de start si primul colt. Apoi, mergem din colt in colt pana indeplinim misiunea.

3 Partea a III-a

Aceasta sectiune contine informatii despre agentul reflex si algoritmii MiniMax si Alpha-Beta Pruning.

Agentul reflex nu tine cont de starile precedente. Tot ce conteaza e prezentul si viitorul apropiat. Am proiectat o functie de evaluare pentru agentul reflex:

```
    def evaluationFunction(self, currentGameState, action):
        # Useful information you can extract from a GameState
        successorGameState = currentGameState.
                                        generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer
                                    for ghostState in newGhostStates]

        currentScore = scoreEvaluationFunction(currentGameState)
        newScore = successorGameState.getScore()
        newNumFood = successorGameState.getNumFood()
        ghostDistance = min([manhattanDistance(newPos,
                    ghost.getPosition()) for ghost in newGhostStates])
        foodList = newFood.asList()

        if foodList:
            foodDistance = min([manhattanDistance(newPos,
                                        food) for food in foodList])
        else:
            foodDistance = 0
        scoreDiff = newScore - currentScore

        return ((10 / (foodDistance + 1)) + (100 / (newNumFood + 1)))
                                    + (ghostDistance / 10) + scoreDiff
        # +1 to avoid ErrDiv0
```

Aceasta tine cont de numarul de unitati de mancare si distanta pana la cea mai apropiata unitate. De asemenea, conteaza distanta pana la cea mai apropiata fantoma si diferenta de scor de la o stare la alta. Coeficientii au fost alesi astfel incat sa dea o valoare mult mai mare decat feature-urile alese pentru functie.

In continuare, voi exemplifica algorimul MiniMax. Acesta modeleaza o lupta continua dintre 2 jucatori. Unul incearca sa-si maximizeze castigul, iar celalalt vrea exact opusul. Acest castig este modelat sub forma de arbore, pe baza mai multor decizii.

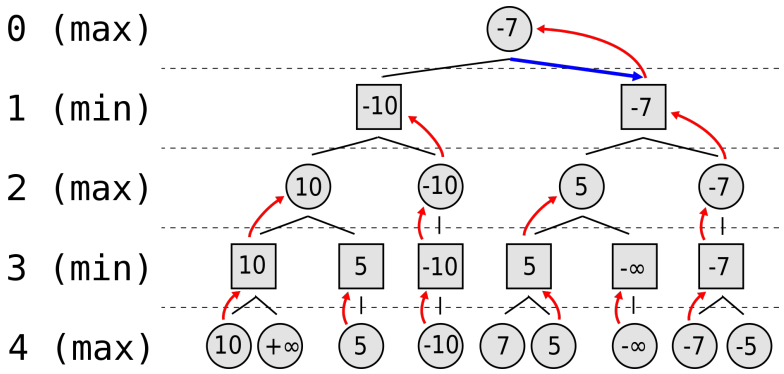

Figura 1: Logica algoritmului MiniMax.

Am incercat sa optimizez algoritmul astfel incat sa nu verifice fiecare nod in parte. Am reusit sa implementez partial corect Alpha-Beta Pruning. Problema este ca pacman este prins in ambuscada. Este posibil sa fie o problema la functia de evaluare.

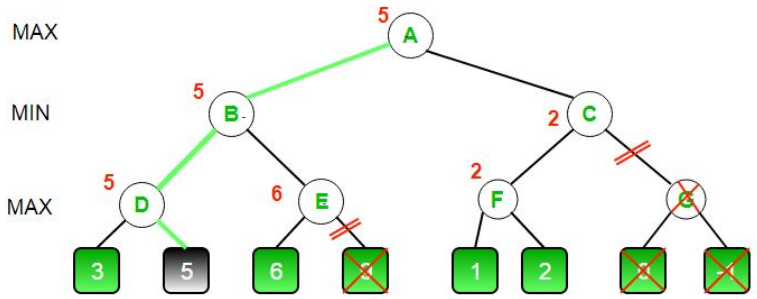

Figura 2: Logica algoritmului Alpha-Beta Pruning.