UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Szekely Gergo
Grupa: 30237

Facultatea de Automatica
si Calculatoare

17 Noiembrie 2022

Cuprins

1 Uninformed search

1.1 Question 1 - Depth-first search

Cerinta era sa impementam Depth-first search (DFS) din proiectul Pacman-Search in functia *def depthFirstSearch(problem: SearchProblem):*

Algoritmul de parcurgere în adancime Depth-first search (DFS) exploreaza graful sau arborele incepand de la un nod și continua cat mai adanc în fiecare ramura înainte de a reveni și a explora alte ramuri. Este o metodă de cautare în structuri de date ierarhice.

```
def depthFirstSearch(problem: SearchProblem):

    frontier = util.Stack()
    frontier.push((problem.getStartState(), []))

    reached = set()
    reached.add(problem.getStartState())

    while not frontier.isEmpty():

        state, moves = frontier.pop()

        for successorState, direction, cost in problem.getSuccessors(state):
            if problem.isGoalState(successorState):
                return moves + [direction]

            if successorState not in reached:
                reached.add(successorState)
                frontier.push((successorState, moves + [direction]))

    return []

```

Pe un bigMaze:

- cost total: 210
- noduri expandate: 390
- score: 300

1.2 Question 2 - Breadth-first search

Cerinta era sa impementam Breadth-first search (BFS) din proiectul Pacman-Search in functia *def breadthFirstSearch(problem: SearchProblem):*

Algoritmul de parcurgere in latime Breadth-first search (BFS) exploreaza graful sau arborele nivel cu nivel, incepand de la un nod sursa, expandandu-se la toti vecinii in ordine si continuand astfel pe nivelurile succesive. Este o metoda eficienta de cautare in structuri de date ierarhice.

```
def breadthFirstSearch(problem: SearchProblem):

    frontier = util.Queue()
```

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

    reached = set()
    reached.add(problem.getStartState())

    while not frontier.isEmpty():

        state, moves = frontier.pop()

        for successorState, direction, cost in problem.getSuccessors(state):
            if problem.isGoalState(successorState):
                return moves + [direction]

            if successorState not in reached:
                reached.add(successorState)
                frontier.push((successorState, moves + [direction]))

    return []

```

Pe un bigMaze:

- cost total: 210
- noduri expandate: 617
- score: 300

1.3 Question 3 - Uniform-Const-Search

Cerinta era sa impementam Uniform-Const-Search (UCS) din proiectul Pacman-Serach in functia *def uniformCostSearch(problem: SearchProblem):*

Algoritmul de cautare uniforma constanta exploreaza spațiul de cautare cu costuri constante, selectand nodurile în ordinea crescatoare a costurilor pentru a gasi o soluție optima. Este utilizat în problemele de cautare a cailor cu cost minim în grafuri ponderat.

```
def uniformCostSearch(problem: SearchProblem):

    frontier = util.PriorityQueue()
    frontier.push((problem.getStartState(), [], 0), 0)

    reached = set()
    reached.add(problem.getStartState())

    while not frontier.isEmpty():

        state, moves, cost = frontier.pop()

        for successorState, direction, successorCost in problem.getSuccessors(state):
            if problem.isGoalState(successorState):
                return moves + [direction]

```

```
            if successorState not in reached:
                reached.add(successorState)
                frontier.push((successorState, moves + [direction], cost + successorCost), c

    return []

```

Pe un mediumScaryMaze :

- cost total: 68719479864
- noduri expandate: 108
- score: 418

2 Informed search

2.1 Question 4 - A* search algorithm

Cerinta era sa impementam algoritmul A* search din proiectul Pacman-Serach in functia *def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):*

Algoritmul de cautare A* este o metoda care exploreaza spatiul de cautare, alegand in mod inteligent urmatoarea stare pe baza costului estimat total al unei cai de la inceput la respectiva stare si a costului real pana in acel moment. Este folosit in problemele de cautare a drumului optim in grafuri ponderate.

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):

    frontier = util.PriorityQueue()
    frontier.push((problem.getStartState(), [], 0), 0)

    reached = set()
    reached.add(problem.getStartState())

    while not frontier.isEmpty():

        state, moves, cost = frontier.pop()

        for successorState, direction, successorCost in problem.getSuccessors(state):
            if problem.isGoalState(successorState):
                return moves + [direction]

            if successorState not in reached:
                reached.add(successorState)
                g = cost + successorCost
                h = heuristic(successorState, problem)
                f = g + h
                frontier.push((successorState, moves + [direction], f), f)

    return []


```

Pe un bigMaze:

- cost total: 210
- noduri expandate: 585
- score: 300

2.2 Question 5 - Finding All the Corners

Cerinta era sa impementam algoritmul pentru a gasim toate colturile din labirint din proiectul Pacman-SerachAgents in functia *class CornersProblem(search.SearchProblem):*

Aceasta problema foloseste un algoritm de cautare pentru a parcurge nodurile labirintului gasind astfel cel mai scurt drum catre toate colturile acestuia.

```
def getStartState(self):

        return self.startingPosition, self.areCornersVisited


def isGoalState(self, state: Any):
        areCornersVisited = state[1]

        for isCornerVisited in areCornersVisited:
            if not isCornerVisited:
                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]
            areCornersVisited = list(state[1])
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]

            if not hitsWall:
                nextStateNode = (nextx, nexty)

                for i in range(4):
                    if nextStateNode == self.corners[i]:
                        areCornersVisited[i] = True

                successors.append(((nextStateNode, tuple(areCornersVisited)), action, 1))

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

```

Pe un mediumCorners :

- cost total: 106
- noduri expandate: 1921
- score: 434

2.3 Question 6 - Corners Problem: Heuristic

Cerinta era sa gasim o heuristica potrivita cu ajutorul caruia algoritmul de cautare va fi mai eficient din Pacman-SearchAgents in functia *def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem)::*

Acasta problema verifica toate colturile care nu au fost vizitate de pacman dupa ce calculeaza distanta manhattan dintre pozitia actuala si colturile parccurse,astfel calculand o distanta totala.

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    currentPosition = state[0]
    areCornersVisited = list(state[1])
    h = 0

    for i in range(4):
        if not areCornersVisited[i]:
            distance = util.manhattanDistance(currentPosition, corners[i])
            h += distance

    return h


```

Pe un mediumCorners :

- cost total: 106
- noduri expandate: 904
- score: 434

2.4 Question 7 - Eating All The Dots

Cerinta era sa gasim o heuristica potrivita cu ajutorul caruia pacman sa gaseasca toate punctele si sa le manance care trebuia implementat in Pacman-SerachAgents in functia *def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):*

Aceasta problema cauta cel mai eficient drum pentru pacman pentru a manca toate punctele prin parcurerea nodurilor. Heuristica face o suma intre toate distantele dintre pozitia actuala si pozitia pe care se afla fiecare punctulet.

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    foodPositions = foodGrid.asList()
    h = 0

    for foodPosition in foodPositions:
        distance = mazeDistance(position, foodPosition, problem.startingGameState)
        h += distance

```

```
    return h

```

Pe un trickySearch:

- cost total: 60
- noduri expandate: 8527
- score: 570

2.5 Question 8 - Suboptimal Search

Acest tip de cautare foloseste o metoda greedy pentru a gasi drumul optimal. Astfel, in comparatie cu algoritmul A*, acesta devine mult mai eficient in timp, insa nu prezinta mereu o solutie optima. Pentru a rezolva problema, doua functii trebuiau completate.

Prima functie se foloseste de clasa AnyFoodSearchProblem care gaseste un drum catre orice "mancare" pentru Pacmanul nostru, adica gaseste drumul dintr-o pozitie de inceput catre un alt punct din labirint.

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

A doua functie la care am lucrat pentru a rezolva aceasta problema este isGoalState din clasa precizata mai sus (AnyFoodSearchProblem). Acesta se uita daca am ajuns intr-un punct adecvat din labirint. Daca pozitia in care ne aflam are mancare pe el inseamna ca am ajuns in locul cautat.

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

```

Pentru bigSearch:

- cost total: 350
- score: 2360