PACMAN - Search

Salajan Madalina, Negrut Ciprian

December 4, 2022

Abstract

In acest proiect, Pacman trebuie sa gaseasca drumul optim prin labirint pentru a ajunge la o anumita locatie si pentru a colecta mancarea in cat mai putini pasi. Pe parcursul acestui proiect ne-am imbunatatit gandirea prin compararea diferitelor euristici implementate.

1 Introducere

Pacman este unul dintre cele mai populare jocuri din lume. Goal-ul acestui joc este de a avea un scor cat mai mare mancand toate punctele din labirint.

Am implementat algoritmi de cautare neinformati care l-au ajutat pe Pacman sa isi indeplineasca visul. Insa acesti algoritmi nu l-au ajutat pe Pacman sa obtina un scor prea mare din cauza ca nu sunt prea eficienti. Prin urmre, am fost nevoiti sa implementam algoritmi de cautare informati care sa maximizeze scorul lui Pacman.

2 Cerinte

2.1 Cerinta 1: Depth First Search

Algoritmul DFS expandeaza cel mai adanc nod din arborele de cautare. Foloseste o structura de tip LIFO (Last In First Out) pentru a stoca ordinea nodurilor ce trebuie expandate, mai exact o stiva.

Am rulat urmatoarele comenzi pentru a testa implementarea:

- python pacman.py -l tinyMaze -p SearchAgent
- python pacman.py -l mediumMaze -p SearchAgent
- python pacman.py -l bigMaze -z .5 -p SearchAgent
- python autograder.py -q q1

Putem observa rezultatele in Table 1.

Maze form	Nodes expanded	Path cost	Score
Tiny	15	10	500
Medium	146	130	380
Big	390	210	300

Table 1: DFS results.

```
def depthFirstSearch(problem: SearchProblem):
    return_list = []
    stack = util.Stack()
    visited = []
    root_node = NodeT(problem.getStartState(), None, None, None)
    stack.push(root_node)
    while not stack.isEmpty():
        node = stack.pop()

        if node.getPosition() not in visited:
            visited.append(node.getPosition())

            if problem.isGoalState(node.getPosition()):
                while node.getParent() is not None:
                    return_list.append(node.getDirection())
                    node = node.getParent()
                return_list.reverse()
                return return_list

            for child in problem.getSuccessors(node.getPosition()):
                if child[0] not in visited:
                    stack.push(NodeT(child[0], child[1], None, node))
```

2.2 Cerinta 2: Breadth First Search

Algortimul BFS expandeaza nodul curent mai intai, apoi toti succesorii pe rand si asa mai departe. Foloseste o structura de tip FIFO (First In First Out) pentru a stoca ordinea nodurilor ce trebuie expandate, mai exact o coada.

Am rulat urmatoarele comenzi pentru a testa implementarea:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
- python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
- python autograder.py -q q2

Putem observa rezultatele in Table 2.

Maze form	Nodes expanded	Path cost	Score
Tiny	15	8	502
Medium	269	68	442
Big	620	210	300

Table 2: BFS results.

```
def breadthFirstSearch(problem: SearchProblem):
    my_list = []
    my_queue = util.Queue()
    root_node = NodeT(problem.getStartState(), None,
    problem.getSuccessors(problem.getStartState()), None)

    visited = [root_node.getPosition()]
    my_queue.push(root_node)

    while not my_queue.isEmpty():
        node = my_queue.pop()

        for child in node.getNeighbours():
            if problem.isGoalState(child[0]):
                my_list.append(child[1])
                while node.getParent() is not None:
                    my_list.append(node.getDirection())
                    node = node.getParent()
                my_list.reverse()
                return my_list

            if child[0] not in visited:
                new_node = NodeT(child[0], child[1], problem.getSuccessors(child[0]), node)
                visited.append(new_node.getPosition())
                my_queue.push(new_node)
```

2.3 Cerinta 3: Varying the Cost Function

Atunci cand fiecare step are acelasi cost, BFS-ul este optim pentru ca expandeaza cel mai adanc nod. UCS-ul expandeaza nodul ce are path-ul de cost minim. Acest lucru se poate realiza cu o coada de prioritati ce poate scoate nodul cu costul cel mai mic. Am observat diferente foarte mari intre costurile path-urilor pentru StayEastSearchAgent si StayWestSearchAgent datorate functiilor pentru cost. Pentru StayEastSearchAgent avem functia costfn = 0.5x, deci costul path-ului va fi foarte mic, iar pentru StayWestSearchAgent avem functia costfn = 2x care ne va da un cost al path-ului foarte mare.

Am rulat urmatoarele comenzi pentru a testa implementarea:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
- python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
- python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
- python autograder.py -q q3

Putem observa rezultatele in Table 3.

Maze form	Nodes expanded	Path cost	Score
Tiny	15	8	502
Medium	269	68	442
Big	620	210	300
mediumDotted EastAgent	186	1	646
mediumScarry WestAgent	108	68719479864	418

Table 3: UCS results.

```
def uniformCostSearch(problem: SearchProblem):
   my_list = []
   my_priority_queue = util.PriorityQueue()
   root_node = NodeT(problem.getStartState(), None, None, None, 0)
   solution = 0
   node_solution = root_node

   visited = {root_node.getPosition(): 0}
   my_priority_queue.push(root_node, 0)

   while not my_priority_queue.isEmpty():
       node = my_priority_queue.pop()

       if problem.isGoalState(node.getPosition()):
           break

       for child in problem.getSuccessors(node.getPosition()):
           if child[0] not in visited or (child[2] + node.getCost()) < visited[child[0]]:
               visited[child[0]] = child[2] + node.getCost()
               new_node = NodeT(child[0], child[1], None, node, child[2] + node.getCost())
               my_priority_queue.update(new_node, new_node.getCost())
```

```
            if problem.isGoalState(child[0]):
                if solution == 0:
                    solution = new_node.getCost()
                    node_solution = new_node
                else:
                    if new_node.getCost() < solution:
                        solution = new_node.getCost()
                        node_solution = new_node

    while node_solution.getParent() is not None:
        my_list.append(node_solution.getDirection())
        node_solution = node_solution.getParent()

    my_list.reverse()
    return my_list
```

2.4 Cerinta 4: A* Search

Aceasta functie A* evalueaza nodurile combinand 2 functii:

- g(n) : costul pentru a ajunge de la nodul start la nodul n
- h(n) : cel mai mic cost estimat pentru a ajunge de la nodul n la goal.
- f(n) = g(n) + h(n), f(n) : cel mai mic cost estimat al solutiei prin nodul n

Algoritmul este identic cu UCS, doar ca A* foloseste f(n) in loc de g(n). A* are rolul de a face cautarea mai rapida si de a micsora numarul de noduri expandate. Am folosit distanta Mnahattan deja implementata ca si functie de euristica.

Putem observa rezultatele in Table 4.

Algorithm	Nodes expanded	Path cost
DFS	576	298
BFS	682	54
UCS	682	54
A*	535	54

Table 4: Results of openMaze.

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    my_list = []
    my_priority_queue = util.PriorityQueue()
    root_node = NodeT(problem.getStartState(), None, None, None, 0)
    solution = 0
    node_solution = root_node

    visited = {root_node.getPosition(): 0}
    my_priority_queue.push(root_node, 0)

    while not my_priority_queue.isEmpty():
        node = my_priority_queue.pop()

        if problem.isGoalState(node.getPosition()):
            break

        for child in problem.getSuccessors(node.getPosition()):
            if child[0] not in visited or (child[2] + node.getCost()) < visited[child[0]]:
                heuristicCost = child[2] + node.getCost() + heuristic(child[0], problem)
                visited[child[0]] = child[2] + node.getCost()
                new_node = NodeT(child[0], child[1], None, node, child[2] + node.getCost())
                my_priority_queue.update(new_node, heuristicCost)

                if problem.isGoalState(child[0]):
                    if solution == 0:
                        solution = new_node.getCost()
                        node_solution = new_node
                    else:
                        if new_node.getCost() < solution:
                            solution = new_node.getCost()
                            node_solution = new_node

    while node_solution.getParent() is not None:
        my_list.append(node_solution.getDirection())
        node_solution = node_solution.getParent()

    my_list.reverse()
    return my_list
```

2.5 Cerinta 5: Finding All the Corners

In acest algoritm, exista 4 dots, cate unul in fiecare colt al gridului, pe care Pacman trebuie sa le manance. O problema care se pune este detectarea goal-ului. Am reprezentat state-ul ca o tupla de forma: ((x, y), (visited corners)) unde (x, y) reprezinta coordonatele Pacman-ului. De exemplu: ((1, 3), ((1, 1))) inseamna ca Pacman este in pozitia (1, 3) si a vizitat doar primul corner, adica cel din stanga jos.

Pentru a verifica implementarea, am rulat urmatoarele comenzi:

- python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
- python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
- python autograder.py -q q5

In functia getStartState(self) returnam pozitia de start a Pacmanului si o tupla cu cornerele vizitate.

In functia isGoalState(self, state: Any) verificam daca toate cornerele au fost vizitate si returnam True, in caz contrar returnam False.

In functia getSuccessors(self, state: Any) verificam daca nodul succesor nodului curent este corner, in acest caz il adaugam in lista de corneruri vizitate. Daca succesorul nu este zid, il adaugam la lista de succesori.

Putem observa rezultatele in Table 5.

Maze form	Nodes expanded	Path cost	Score
TinyCorners	252	28	512
MediumCorners	1966	106	434
BigCorners	7949	162	378

Table 5: Results of Corners Problem using BFS.

```
def getStartState(self):
    return self.startingPosition, tuple(self.visited_corners)

def isGoalState(self, state: Any):
    my_position, corners = state
    result = 0
```

```
        for corner in corners:
            if corner in self.corners:
                result += 1

        if result == 4:
            return True
        return False

    def getSuccessors(self, state: Any):
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            my_position, corners = state
            x, y = my_position

            dx, dy = Actions.directionToVector(action)
            nextx = int(x + dx)
            nexty = int(y + dy)

            corners_visited = list(corners)

            if (nextx, nexty) in self.corners:
                if (nextx, nexty) not in corners_visited:
                    corners_visited.append((nextx, nexty))

            if not self.walls[nextx][nexty]:
                next_state = (nextx, nexty)
                successors.append(((next_state, tuple(corners_visited)), action, 1))

        self._expanded += 1  # DO NOT CHANGE

        return successors
```

2.6 Cerinta 6: Corners Problem: Heuristic

Euristica propusa trebuie sa fie atat admisibila cat si consistenta. Noi am propus o euristica in care Pacman merge la cornerul nevizitat cel mai indepartat.

Am folosit functia mazeDistance pentru a ne genera distanta de la pacman la corners. Aceasta functie tine cont si de walls, spre deosebire de manhattanDistance si returneaza lungimea path-ului cel mai scurt (len(bfs)) de la o coordonata la alta.

Putem observa ca folosind aceasta euristica, valorile pentru nodurile expandate sunt mai mici decat in cazul in care am folosit BFS.

Pentru a verifica implementarea, am rulat urmatoarele comenzi:

- python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
- python autograder.py -q q6

Putem observa rezultatele in Table 6.

Maze form	Nodes expanded	Path cost	Score
TinyCorners	150	28	512
MediumCorners	980	106	434

Table 6: Results of Corners Problem using a Heuristic.

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    unvisitedCorners = []
    currentPosition = state[0]
    cornersPositions = state[1]

    for each in corners:
        if each not in cornersPositions:
            unvisitedCorners.append(each)

    distanceList = []
    for each in unvisitedCorners:
        distanceList.append(mazeDistance(currentPosition, each, problem.startingGameState))

    return max(distanceList, default=0)
```

2.7 Cerinta 7: Eating All the Dots

Aceasta cerinta aduce in plus faptul ca Pacman trebuie sa manance toate dots exitente in grid. Aici vom vedea ca A* search este cu mult mai eficient. Am folosit aceeasi euristica ca si in pasul anterior in care Pacman merge la cel mai indepartat dot, doar ca am inlocuit unvisitedCorners cu foodGrid.asList().

Pentru a verifica implementarea, am rulat urmatoarele comenzi:

- python pacman.py -l testSearch -p AStarFoodSearchAgent
- python autograder.py -q q7

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

    distanceList = []
    for each in foodGrid.asList():
        distanceList.append(mazeDistance(position, each, problem.startingGameState))

    return max(distanceList, default=0)
```

2.8 Cerinta 8: Suboptimal Search

Uneori pana si euristicile bune pot sa esueze in gasirea celui mai optim path in cel mai scurt timp. Data fiind problema, este mult mai rezonabil sa gasim un path bun, desi nu asa de bun ca si unul optim intr-un timp scurt.

Un agent care sa rezolve aceasta problema este unul care merge la cel mai apropiat dot. Acest lucru a fost realizat folosindu-ne de BFS in functia findPathToClosestDot. Am implementat functia isGoalState care verifica daca intr-o pozitie exista sau nu mancare.

Pentru a verifica implementarea, am rulat urmatoarele comenzi:

- python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
- python autograder.py -q q8

Putem observa rezultatele in Table 7.

Maze form	Path cost	Score
TinySearch	31	569
MediumSearch	171	1409
BigSearch	350	

Table 7: Results of suboptimal search.

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

    return search.breadthFirstSearch(problem)


def isGoalState(self, state: Tuple[int, int]):
    x, y = state

    if self.food[x][y]:
        return True
    else:
        return False
```

References

In realizarea implementarii acestor algoritmi, descrisi mai sus, ne-au fost de mare ajutor:

- Cartea: ”Artificial Intelligence, A Modern Approach - Fourth Edition” de Stuart Russell si Peter Norving (AIMA)
- Indrumatorul de laborator: ”Intoduction to Artificial Intelligence” de Anca Marginean, Adrian Groza si Radu Razvan Slavescu