Artificial intelligence - Project 1
- Search problems -

George Botis
Daria Francioli

7/11/2022

1 Introducere

Scopul acestui proiect este acela de a implementa alți algoritmi de căutare pentru a-l ajuta pe Mr. Pacman să găsească mâncarea în cel mai scurt timp. Vom face și o comparație între cei 2 algoritmi noi (Un algoritm bidirecțional și Weighted A*) și cei 4 deja reprezentați în laboratoarele anterioare (BFS, DFS, A* și UCS) .

1.1 Definirea problemei și soluționarea acesteia

Prin compararea mai multor algoritmi de căutare vrem să rezolvăm problema găsirii drumului spre mâncare în cel mai scurt timp.

Adițional am mai creat-o și pe Miss Pacman, jucându-ne cu grafica Pacmanului original, adăugând o fundiță, gene și ochi albaștrii ce se deplasează odată cu Miss Pacman.

1.2 Algoritmii folosiți

1. Bidirectional Algorithm
2. Weighted A*
3. BFS, DFS, UCS, A*

Algoritmii de căutare DFS, BFS, UCS și cel bidirecțional sunt algoritmi neinformați(constă în explorarea alternativelor, fără a utiliza informații specifice despre problemă).

Algoritmii de căutare A* și Weighted A* sunt algoritmi informați(euristică-încearcă alegerea „inteligentă” a nodurilor care trebuie expandate).

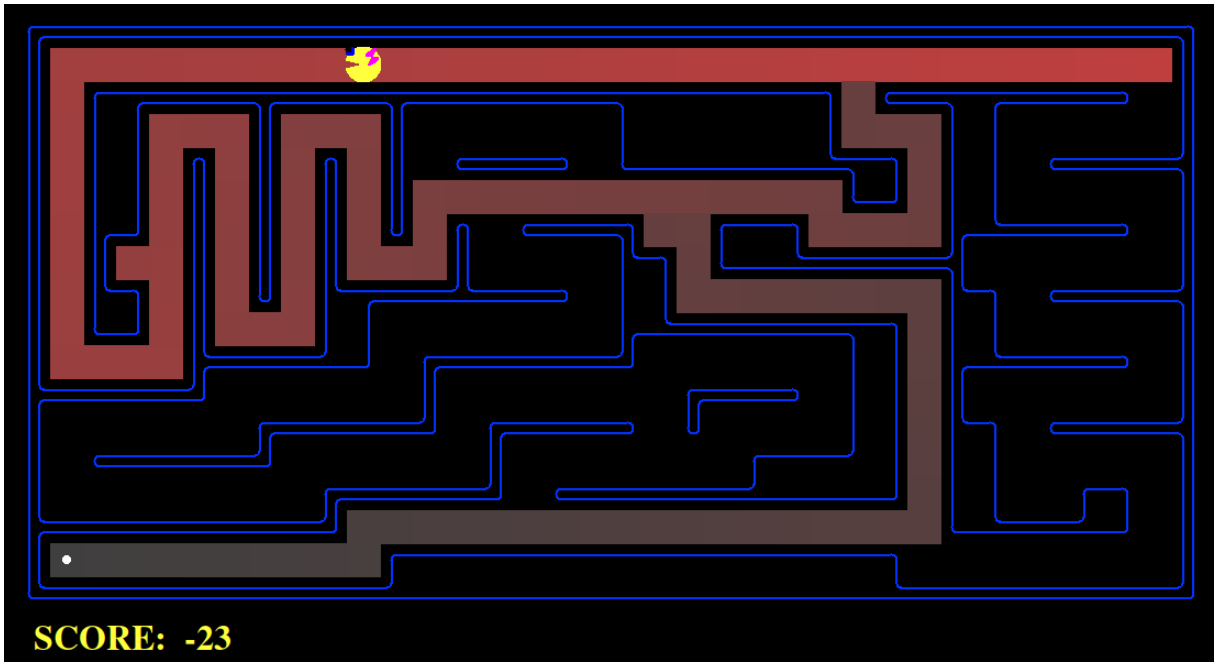

Figure 1: Medium Maze Miss Pacman

2 Implementare

În această secțiune vom prezenta implementarea fiecărui algoritm.

2.1 Breadth First Search (BFS)

Algoritmul BFS este un algoritm de căutare utilizat în grafuri și arbori, de asemenea poate fi folosit pentru Frontiera BFS este bazată pe FIFO (primul intrat, primul ieșit) și extinde succesorii în ordinea în care au fost adăugați. Se merge pe lățime prin nodurile vecine ale punctului în care ne aflăm.

Cod:

```
def breadthFirstSearch(problem):
    """
    Search the shallowest nodes in the search tree first.
    """
    start = problem.getStartState()
    exploredState = []
    exploredState.append(start)
    states = util.Queue()
    stateTuple = (start, [])
    states.push(stateTuple)
    while not states.isEmpty():
        state, action = states.pop()
        if problem.isGoalState(state):
            return action
        successor = problem.getSuccessors(state)
        for i in successor:
            coordinates = i[0]
            if not coordinates in exploredState:
                direction = i[1]
                exploredState.append(coordinates)
                states.push((coordinates, action + [direction]))
    return action
    util.raiseNotDefined()
```

Explicație:

- funcția problem va returna o solutie sau fail, se va stoca succesorii într-o coadă, verifică dacă fiecare nod a fost vizitat, îi dă push către visited pentru a evita verificarea de mai multe ori a unui nod deja vizitat, si se verifică daca am ajuns la goalState- dacă nu, de va da expand copiilor după algoritm.

Comandă:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

2.1.1 Întrebări

Q1: Soluția găsită este optimă? Explicați răspunsul.
A1: BFS este optim numai dacă costul tuturor arcelor din graficul spațiului de stare este același.

Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: Question q2: 3/3

2.2 Depth First Search (DFS)

DFS caută mai întâi cel mai adânc nod din arborele de căutare.

Cod:

```
def depthFirstSearch(problem):

start = problem.getStartState()
c = problem.getStartState()
exploredState = []
exploredState.append(start)
states = util.Stack()
stateTuple = (start, [])
states.push(stateTuple)
while not states.isEmpty() and not problem.isGoalState(c):
    state, actions = states.pop()
    exploredState.append(state)
    successor = problem.getSuccessors(state)
    for i in successor:
        coordinates = i[0]
        if not coordinates in exploredState:
            c = i[0]
            direction = i[1]
            states.push((coordinates, actions + [direction]))
return actions + [direction]

util.raiseNotDefined()
```

Explicație:

- Frontiera DFS este bazată pe LIFO (ultimul intrat, primul ieșit) și extinde succesorii în ordinea în care au fost adăugați. Se merge pe adâncime prin nodurile vecine ale vecinilo punctului în care ne aflăm până când ajungem la starea GOAL. Odată ce un nod este complet extins, acesta este popped din stivă.

Comandă:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs

2.2.1 Întrebări

Q1: Soluția găsită este optimă? Explicați răspunsul.
A1: DFS nu este optim. Găsește doar soluția cea mai din stânga în arborele de căutare, indiferent de adâncimea sau costul nodului.

Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: Question q2: 3/3

2.3 A* search algorithm

Costul total al unui nod nu trebuie să fie mai mic decât suma dintre cost(distanță start+distanță curentă) și euristica folosită.

Cod:

```
def aStarSearch(problem, heuristic=nullHeuristic):
    "Search the node that has the lowest combined cost and heuristic first."
    start = problem.getStartState()
    exploredState = []
    states = util.PriorityQueue()
    states.push((start, []), nullHeuristic(start, problem))
    nCost = 0
    while not states.isEmpty():
        state, actions = states.pop()
        if problem.isGoalState(state):
            return actions
        if state not in exploredState:
            successors = problem.getSuccessors(state)
            for succ in successors:
                coordinates = succ[0]
                if coordinates not in exploredState:
                    directions = succ[1]
                    nActions = actions + [directions]
                    nCost = problem.getCostOfActions(nActions) + heuristic(coordinates, problem)
                    states.push((coordinates, actions + [directions]), nCost)
        exploredState.append(state)
    return actions
    util.raiseNotDefined()
```

Explicație:

- Euristica ia două argumente: o stare în problema de căutare (argumentul principal) și problema în sine (pentru informații de referință). Folosim o coadă de priorități, care este dată de cost și euristică. Funcția folosită este f(n)= g(n)+h(n)(g=cost, h=euristica) și dacă următoarea stare în care ne-am afla este următorul nod, atunci expandăm nodul stării și actualizăm starea următoare cu locația porții destinație.

Comandă:

- -l tinyMaze -p SearchAgent -a fn=aStar

2.3.1 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: A* și UCS au aceeași soluție sau sunt diferite?
A1: Este asemănător cu UCS, dar alege starea următoare.

Q2: Run autograder *python autograder.py* and write the points for Question 4 (min 3 points).
A4: Question q2: 3/3

2.4 Uniform-Cost Search (UCS)

În loc să extindă cel mai puțin adânc nod, cum ar fi BFS, extinde nodul n cu calea cu cel mai mic cost g(n). Schimbând funcția de cost, putem încuraja Pacman să găsească căi diferite. De exemplu, costul pentru pașii periculoși în zonele pline de fantome este mai mare.

"In search.py, implement ***Uniform-cost graph search*** *algorithm in uniformCostSearchfunction"*
Cod:

```
def uniformCostSearch(problem):
    "Search the node of least total cost first. "

    start = problem.getStartState()
    exploredState = []
    states = util.PriorityQueue()
    states.push((start, []) ,0)
    while not states.isEmpty():
        state, actions = states.pop()
        if problem.isGoalState(state):
            return actions
        if state not in exploredState:
            successors = problem.getSuccessors(state)
            for succ in successors:
                coordinates = succ[0]
                if coordinates not in exploredState:
                    directions = succ[1]
                    newCost = actions + [directions]
                    states.push((coordinates, actions + [directions]), problem.getCostOfActions(newCost)
        exploredState.append(state)
    return actions
    util.raiseNotDefined()
```

Explicație:

- Aplică test Goal unui nod odată ce acesta este selectat pentru extindere,nu ca atunci când este generat prima dată (ca în cazul BFS și DFS). Acest lucru se datorează faptului că primul nod de goal care este generat poate fi pe o cale suboptimă. Include, de asemenea, un test pentru a verifica dacă a fost întâlnită o stare de goal mai bună.

Comenzi:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

2.4.1 Întrebări

Q1: Comparați rezultatele cu cele obținute cu DFS. Sunt soluțiile diferite? Explicați răspunsul.
A1: UCS este similar cu DFS, dar are o coadă de prioritate pentru stocarea succesorilor+ fiecare cost este atașat.

Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: Question q2: 3/3

2.5 Bidirectional Search

Algoritmul abordeaza problema pornind atat de la punctul de start cat si de la punctul target. Ideea generala este de a determina un path de la start la finish si un alt path de la finish la start pentru a determina locul in care aceste path-uri se intalnesc.

Cod:

```
def bidirectional(problem):
    # Path-ul de la start catre mijloc
    q1 = util.Queue()
    temp_q1 = []
    # Path-ul de la goal catre mijloc
    q2 = util.Queue()
    temp_q2 = []
    explorednode1 = set()
    explorednode2 = set()
    startnode = problem.getStartState()
    endnode = problem.goal
    q1.push((startnode,[]))
    q2.push((endnode, []))
    while not q1.isEmpty() and not q2.isEmpty():
        if not q1.isEmpty():
            currentnode, direction = q1.pop()
            if currentnode not in explorednode1:
                explorednode1.add(currentnode)
                if (currentnode in temp_q2) or problem.isGoalState(currentnode):
                    while not q2.isEmpty():
                        node, direc = q2.pop()
                        if node == currentnode:
                            solution = direction + direc.reverse()
                            return solution
                for(successor, action, stepCost) in problem.getSuccessors(currentnode):
                    q1.push((successor, direction + [action]))
                    temp_q1.append(successor)
        if not q2.isEmpty():
            currentnode, direction = q2.pop()
            if currentnode not in explorednode2:
                explorednode2.add(currentnode)
                if currentnode in temp_q1:
                    while not q1.isEmpty():
                        node, direc = q1.pop()
                        if node == currentnode:
                            direction.reverse()
                            solution = direc + ulta(direction)
                            return solution
                for(successor, action, stepCost) in problem.getSuccessors(currentnode):
                    q2.push((successor, direction + [action]))
                    temp_q2.append(successor)
```

Explicație:

- La prima instanta a unui punct de contact (daca exista) algoritmul va creea path-ul final care consta intr-o combinatie a celor 2 path-uri initiale: Prima parte de la start pana la punctul de contact si a doua parte de la finish la punctul de contact. In teorie, algoritmul este mai rapid decat un BFS sau un DFS clasic. Daca complexitatea unui DFS este O(b^d), $complexitatea bidirectionalului este O(b^{(}d/2) + b^{(}d/2))$ $care este mult mai mic decat O(b^d)$.

Comandă:

- python pacman.py -l mediumMaze -p SearchAgent -a fn=bi

2.5.1 Întrebări

Q1: Soluția găsită este optimă? Explicați răspunsul.
A1: BFS este optim numai dacă costul tuturor arcelor din graficul spațiului de stare este același.

Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: Question q2: 3/3

2.6 Weighted A*

După cum se poate observa, există o diferență între acest algoritm și cel clasic: Epsilon. Fiind un algoritm informat, are o euristică= Euristica Euclidiană.
Dacă epsilon < 1, epsilon merge catre 0, iar algoritmul va căuta calea cu cel mai redus cost. Dacă epsilon $= 0$, algoritmul se comportă la fel ca Uniform Cost Search. Dacă epsilon $= 1$, algoritmul se comportă la fel ca A Star. Dacă epsilon > 1, epsilon merge catre infinit, iar algoritmul va deveni unul de tip greedy.

Cod:

```
def euclideanDistanceHeuristic(state, problem=None):
    return math.sqrt((state[0] - problem.goal[0]) ** 2 + (state[1] - problem.goal[1]) ** 2)
    #sqrt((xA-xB)^2+(yA-yB)^2)

def weightedAStarSearch(problem, heuristic=euclideanDistanceHeuristic, epsilon = 0.5):

    solutie=[]
    exploredState = []
    start_node = problem.getStartState()
    frontier = util.PriorityQueue()
    start_heuristic = heuristic(start_node, problem)
    frontier.push((start_node, [], 0), start_heuristic)
    while not frontier.isEmpty():
        current_state, solutie, get_cost = frontier.pop()
        if problem.isGoalState(current_state):
            return solutie
        if current_state not in exploredState:
            exploredState.append(current_state)
            for coordinates, direction, successor_cost in problem.expand(current_state):
                if coordinates not in exploredState:
                    next_state = coordinates

                    actions_list = list(solutie)
                    actions_list += [direction]

                    cost_actions = problem.getCostOfActionSequence(actions_list)
                    frontier.push((next_state, actions_list, 1),
                    cost_actions + epsilon * heuristic(next_state, problem))
    return []
```

Explicație:

- În funcție de datele de intrare și soluția pe care vrem să îl atingem, alegem epsilon.

Comandă:

- python pacman.py -l bigMaze -p SearchAgent fn=wastar

2.6.1 Întrebări

Q1: Soluția găsită este optimă? Explicați răspunsul.
A1: Weighted A* este mai optim decât cel clasic datorită epsilonului cu care configurăm algoritmul.

3 Grafica pentru Miss Pacman

În această secțiune vom prezenta implementarea grafică a Miss Pacman.

3.1 Draw Miss Pacman

Pentru a o desena pe Miss Pacman am folosit modelul de grafică pentru fantome, dar în loc de două cercuri pentru doi ochi, va avea 2 cercuri pentru unul, si vor fi unul peste altul, de dimensiuni diferite. De asemenea Miss Pacman are o fundă mov care nu putea lipsi din designul nostru.

Cod din GraphicDesign:

```
def drawPacman(self, pacman, index):
        position = self.getPosition(pacman)
        screen_point = self.to_screen(position)
        print(screen_point)
        endpoints = self.getEndpoints(self.getDirection(pacman))
        (screen_x, screen_y) = (self.to_screen(position))
        dir = self.getDirection(pacman) #directia pacman

        width = PACMAN_OUTLINE_WIDTH
        outlineColor = PACMAN_COLOR
        fillColor = PACMAN_COLOR

        BLUE = formatColor(0.0, 0.0, 1.0) #culoarea ochilor
        PURPLE = formatColor(1.0, 0.0, 1.0) #culoarea funditei
        BLACK = formatColor(0.0, 0.0, 0.0) #culoarea pupilei si a genelor
        WHITE = formatColor(1.0, 1.0, 1.0)

        dx = 0
        dy = 0
        if dir == 'North':
            dy = -0.2
        if dir == 'South':
            dy = 0.2
        if dir == 'East':
            dx = 0.2
        if dir == 'West':
            dx = -0.2

        bow_point = (screen_point[0] - self.gridSize / 4, screen_point[1] - self.gridSize / 4) #punctul
        bpts = [bow_point]
        X1 = 10 #cat de in stanga e funda
        X2 = 20 #cat de in dreapta e funda ////X=latime
        Y1 = 20
        Y2 = 0 #cat de groasa e //// Y=inaltime
        bpts.append((bow_point[0] - X1, bow_point[1] + Y1))
        bpts.append((bow_point[0] + X2, bow_point[1] - Y2))
        bpts.append((bow_point[0] - X2, bow_point[1] + Y2))
        bpts.append((bow_point[0] + X1, bow_point[1] - Y1))

        return [circle(screen_point, PACMAN_SCALE * self.gridSize,
                       fillColor=fillColor, outlineColor=outlineColor,
                       endpoints=endpoints,
```

```
43                 width=width),
44             circle((screen_x + self.gridSize * PACMAN_SCALE * (-0.5 + dx / 1.0),
45                     screen_y - self.gridSize * PACMAN_SCALE * (0.5 - dy / 1.0)), self.gridSize * PA
46                 BLUE, BLUE),
47             circle((screen_x + self.gridSize * PACMAN_SCALE * (-0.5 + dx),
48                     screen_y - self.gridSize * PACMAN_SCALE * (0.5 - dy)),
49                 self.gridSize * PACMAN_SCALE * 0.10, BLACK, WHITE),
50             polygon(bpts, WHITE, [PURPLE])] ##Exterior, Interior funda
```

Figure 2: Miss Pacman

4 Testare și Comparare

DFS:
• Nu este optimal, deoarece se poate întâmpla să aleagă un drum mai lung către goal, deși existau alte drumuri mai scurte.
• Complexitatea este O(V + E), unde V este numărul de vârfuri ale grafului, iar E este numărul de muchii.

BFS:
• Este optimal și complet.
• Complexitatea este O(V + E).

UCS:
• Este optimal și complet.
• Complexitatea este O(bC/e), unde C reprezintă costul soluției optimale, b este factorul de branching și e este cel mai ieftin cost.

A Star:
• Este complet, chiar dacă graful este infinit, A Star va găsi o soluție mereu.
• Dacă euristica este consistentă, atunci algoritmul este optimal.

4.1 Comparare

Algoritm	Scor	Timp	Cost	Noduri Expandate
BFS	502	0.0	8	15
DFS	500	0.0	10	15
UCS	502	0.0	8	15
A*	502	0.0	8	15
Weighted A*	502	0.0	8	15
Bidirectional	502	0.0	8	12

Figure 3: Rezultatul simularilor pentru tinyMaze

Algoritm	Scor	Timp	Cost	Noduri Expandate
BFS	442	0.0	68	269
DFS	380	0.0	130	146
UCS	442	0.0	68	289
A*	442	0.0	68	269
Weighted A*	442	0.0	68	269
Bidirectional	442	0.0	68	269

Figure 4: Rezultatul simularilor pentru mediumMaze

Algoritm	Scor	Timp	Cost	Noduri Expandate
BFS	300	0.0	210	620
DFS	300	0.0	210	390
UCS	300	0.0	210	620
A*	502	0.0	8	620
Weighted A*	502	0.0	8	620
Bidirectional	502	0.0	8	620

Figure 5: Rezultatul simularilor pentru bigMaze

5 Conluzii

Prin intermediul acestui proiect, noi am reușit să descoperim bazele și sintaxele limbajului python: definirea variabilelor, apelul unor funcții sau metode, instanțierea de obiecte și transmiterea parametriilor de la o funcție sau metodă la alta.

De asemenea ne-am amintit algoritmii de căutare și utilozarea acestora în grafuri, arbori, printr-un mod mai practic și creativ. Am reușit să le testam astfel optimitatea și să-l alegem pe cel câștigător pentru Pacman. Pacmanul nostru și-a atins scopul: a ajuns la mâncare.