Proiect: Search-Agent for Pacman

Pal Roland
Grupa 30228
Facultatea Automatica si Calculatoare
Specializarea Calculatoare
Seria B

November, 2025

Cuprins

1 Introducere

In acest proiect am implementat algoritmi fundamentali de cautare pentru a-l ajuta pe Pacman sa navigheze prin labirint. Scopul a fost sa gasesc cai de la o pozitie de start pana la un punct de mancare sau chiar mai multe, trecand de la cautari neinformate (DFS, BFS) la cautari informate (A* cu euristici).

Spre deosebire de simpla miscare aleatorie, acesti algoritmi garanteaza gasirea unei solutii (daca exista), iar unii dintre ei garanteaza chiar drumul optim, adica cel mai scurt drum.

2 Q1: Depth First Search (DFS)

2.1 Descriere

DFS este un algoritm care exploreaza cat mai adanc posibil fiecare ramura a grafului inainte de a reveni la un nod anterior. Nu garanteaza cel mai scurt drum, ci doar gasirea unui drum.

2.2 Implementare

Pentru DFS am folosit o structura de date de tip *Stiva* (LIFO - Last In, First Out). Am implementat un algoritm general de cautare pe grafuri care tine minte starile vizitate pentru a evita buclele infinite.

- Pun starea initiala in stiva.
- Cat timp stiva nu e goala, scot ultimul nod adaugat.
- Daca nodul nu a fost vizitat, il marchez si ii adaug succesorii in stiva.

Rezultatul este ca Pacman gaseste mancare, dar drumul este de obicei foarte lung, pentru ca DFS merge pe o directie pana se loveste de perete.

```
def depthFirstSearch(problem: SearchProblem):
    stack = util.Stack()
    visited = set()
    start_state = problem.getStartState()

    stack.push((start_state, []))

    while not stack.isEmpty():
        state, path = stack.pop()

        if state in visited:
            continue

        if problem.isGoalState(state):
           return path

        visited.add(state)

        for successor,action,cost in problem.getSuccessors(state):
```

```
            if successor not in visited:
                # constuim calea noua pt succesor
                new_path = path + [action]
                stack.push((successor, new_path))

    return []
```

Listing 1: Implementare DFS

3 Q2: Breadth First Search (BFS)

3.1 Descriere

Spre deosebire de DFS, BFS exploreaza nodurile strat cu strat, viziteaza toate nodurile invecinate cu nodul curent. Avantajul este ca BFS garanteaza gasirea celui mai scurt drum intr-un graf neponderat.

3.2 Implementare

Implementarea este foarte similara cu DFS, singura diferenta majora fiind structura de date: am inlocuit Stiva cu o Coada (FIFO - First In, First Out).

Aceasta modificare face ca Pacman sa gaseasca drumul optim catre mancare, fara ocolisuri inutile.

```
def breadthFirstSearch(problem: SearchProblem):
    que = util.Queue()
    visited = set()

    start_state = problem.getStartState()

    que.push((start_state, []))
    visited.add(start_state)

    while not que.isEmpty():
        state, path = que.pop()

        if problem.isGoalState(state):
            return path

        for successor,action,cost in problem.getSuccessors(state):
            if successor not in visited:
                visited.add(successor)
                new_path = path + [action]
                que.push((successor, new_path))

    return []
```

Listing 2: Implementare BFS

4 BFS vs DFS

4.1 Descriere

Breadth-First Search (BFS) si Depth-First Search (DFS) sunt algoritmi pentru parcurgerea grafurilor. Parcurgerea este procesul de accesare a fiecarui varf (nod) al unei structuri de date intr-o ordine definita. Alegerea algoritmului depinde de tipul de date cu care lucram.

Exista doua tipuri de parcurgeri, iar principala diferenta dintre ele consta in ordinea in care acceseaza nodurile:

- Cautarea in latime (Breadth-first search)
- Cautarea in adancime (Depth-first search)

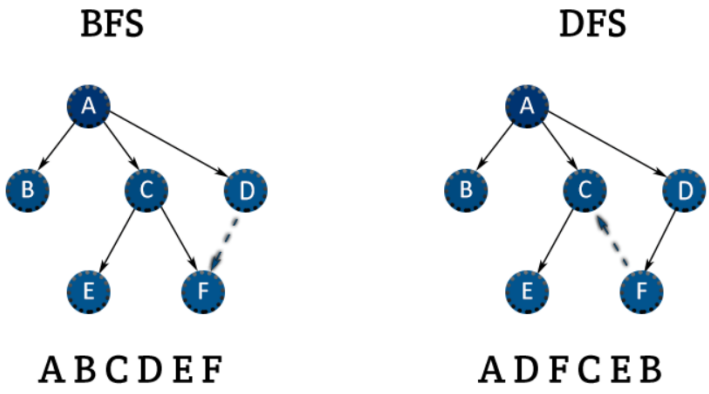

Figura 1: BFS vs DFS

5 Q3: Varying the Cost Function (UCS)

5.1 Descriere

Uniform Cost Search este util atunci cand actiunile au costuri diferite (nu toate au costul egal). Algoritmul expandeaza nodul cu cel mai mic cost total acumulat de la start ($g(n)$).

5.2 Implementare

Am folosit un `PriorityQueue`. In coada de prioritati am stocat tuple care contin starea, calea pana acolo si costul curent. Prioritatea in coada este data chiar de acest cost. Astfel, algoritmul alege mereu sa exploreze drumul "cel mai ieftin" descoperit pana in acel moment.

```
while not priority_queue.isEmpty():
        state, path, cost = priority_queue.pop()

        # daca am vazut stare cu un cost mai mic/egal, atunci ignoram calea aceasta
        if state in visited and visited[state] <= cost:
            continue
        visited[state] = cost

        if problem.isGoalState(state):
            return path

        for successor, action, stepCost in problem.getSuccessors(state):
            new_cost = cost + stepCost
            # intram doar daca se merita, daca avem cost mai mic sau nu am fost inca
            if successor not in visited or new_cost < visited.get(successor, float('inf')):
                new_path = path + [action]
                priority_queue.push((successor, new_path, new_cost), new_cost)
```

Listing 3: Logica principala a implementarii

6 Q4: A* Search

6.1 Descriere

A* este o implementare importanta a acestui proiect. Combina costul uniform (UCS) cu o estimare euristica a distantei pana la tinta ($h(n)$). Functia de cost total devine:

$$f(n) = g(n) + h(n)$$

unde $g(n)$ este costul real pana la nodul curent, iar $h(n)$ este costul estimat pana la final.

6.2 Implementare

Am folosit tot un `PriorityQueue`, dar prioritatea este data de $f(n)$. Daca euristica folosita este admisibila (nu supraestimeaza niciodata costul real), A* gaseste drumul optim mult mai repede decat UCS, pentru ca cautarea merge in directia tintei, nu exploreaza in toate directiile si asltfel castiga timp.

```
    while not priority_queue.isEmpty():
        state, path, cost = priority_queue.pop()

        if state in visited and visited[state] <= cost:
            continue
        visited[state] = cost

        if problem.isGoalState(state):
            return path
```

```
10
11          for successor, action, stepCost in problem.getSuccessors(
    state):
12              new_cost = cost + stepCost
13              h = heuristic(successor, problem)  # h = estimarea
    costului de aici pana la goal
14              f = new_cost + h  # alege nodul dupa F, in UCS alegeam
    doar dupa new_cost
15              # f = g + h, unde g e costul
16
17              if successor not in visited or new_cost < visited.get(
    successor, float('inf')):
18                  new_path = path + [action]
19                  priority_queue.push((successor, new_path, new_cost)
    , f)
```

Listing 4: Logica implementarii functiei A*

7 Q5: Finding All the Corners

7.1 Descriere

In aceasta problema, scopul lui Pacman este sa atinga cele patru colturi ale hartii in cel mai scurt timp. Problema principala a fost definirea spatiului starilor.

Daca starea ar fi fost doar pozitia (x, y), algoritmul nu ar fi stiut istoricul (ce colturi au fost deja atinse). De aceea, am redefinit starea ca fiind o pereche compusa din pozitia curenta si un tuplu de booleeni care tine evidenta colturilor:

$$Stare = ((x, y), (c_1, c_2, c_3, c_4))$$

unde c_i este $True$ daca coltul i a fost vizitat.

7.2 Implementare

Am implementat metodele clasei `CornersProblem` astfel:

- `getStartState`: Initializam starea cu pozitia de start a lui Pacman si un tuplu `(False, False, False, False)`.
- `isGoalState`: Consideram ca am ajuns la final doar daca nu mai exista nicio valoare de `False` in tuplul de colturi vizitate.
- `getSuccessors`: Generam succesorii pentru nord, sud, est, vest. Daca noua pozitie este un zid, o ignoram. Daca este un spatiu liber, verificam daca acel spatiu este unul dintre colturi. Daca da, actualizam lista de vizitate setand indexul corespunzator pe `True`.

```
def getStartState(self):
    startPos = self.startingPosition
    visited = (False, False, False, False)
    return (startPos, visited)
```

```
5
6  def isGoalState(self, state: Any):
7      # Daca mai exista un False in lista, inseamna ca nu am terminat
8      if False in state[1]:
9          return False
10     return True
11
12 def getSuccessors(self, state: Any):
13     successors = []
14     (x, y), visited = state
15
16     for action in [Directions.NORTH, Directions.SOUTH, Directions.
       EAST, Directions.WEST]:
17         dx, dy = Actions.directionToVector(action)
18         nextx, nexty = int(x + dx), int(y + dy)
19
20         if not self.walls[nextx][nexty]:
21             newVisited = list(visited)
22
23             # Daca noua pozitie e un colt, il marcam ca vizitat
24             if (nextx, nexty) in self.corners:
25                 idx = self.corners.index((nextx, nexty))
26                 newVisited[idx] = True
27
28             newState = ((nextx, nexty), tuple(newVisited))
29             successors.append((newState, action, 1))
30
31     self._expanded += 1
32     return successors
```

Listing 5: Implementarea logicii pentru CornersProblem

8 Q6: Corners Problem: Heuristic

8.1 Descriere

Ideea pe care am mers este intuitiva: pentru a vizita toate colturile, Pacman trebuie sa ajunga, cel putin pana la cel mai indepartat colt nevizitat fata de pozitia sa curenta.

8.2 Implementare

In functia `cornersHeuristic`, identific intai colturile care au ramas nevizitate. Apoi, calculez distanta Manhattan de la pozitia curenta a lui Pacman pana la fiecare dintre aceste colturi.

Am ales o euristica bazata pe distanta maxima. Iterez prin colturile ramase nevizitate si calculez distanta Manhattan de la pozitia curenta pana la ele. De asemenea, iau in calcul si distantele dintre colturi pentru a avea o estimare mai buna.

La final, returnez valoarea maxima dintre aceste distante `max(dists)`. Desi nu este cea mai complexa euristica, reduce numarul de noduri expandate si garanteaza gasirea solutiei optime.

```python
def cornersHeuristic(state: Any, problem: CornersProblem):
    pos, visited = state
    corners = problem.corners

    unvisited = []
    for i in range(4):
        if not visited[i]:
            unvisited.append(corners[i])

    if not unvisited:
        return 0

    # cel mai apropiat colt nevizitat
    dists = [abs(pos[0] - cx) + abs(pos[1] - cy) for (cx, cy) in
    unvisited]
    return max(dists)
```

Listing 6: Implementarea gasirii tuturor colcurilor

9 Q7: Eating All The Dots

9.1 Descriere

Aceasta a fost cea mai dificila parte a proiectului. Pacman trebuie sa manance toata mancarea. Spatiul starilor este enorm ($2^{numar\_mancare}$), asa ca A* are nevoie de o euristica foarte puternica pentru a nu expanda zeci de mii de noduri.

Euristica pe care am implementat-o se bazeaza pe doua observatii logice:

- Pacman trebuie neaparat sa ajunga la cea mai indepartata bucata de mancare fata de pozitia curenta.
- Indiferent unde e Pacman, el trebuie sa parcurga distanta dintre cele doua bucati de mancare cele mai departate una de alta (un fel de "diametru" al setului de mancare).

Returnand maximul dintre aceste doua valori, obtinem o estimare foarte buna.

9.2 Implementare

Deoarece functia `mazeDistance` (care face un BFS intern pentru a afla distanta reala prin labirint) este costisitoare, am luat in considerare sa folosesc un dictionar `problem.heuristicInfo`. Astfel, daca am calculat odata distanta dintre doua puncte, o refolosesc direct din memorie, adica din dictionar.

Codul parcurge lista de mancare in doua etape:

- Calculeaza `maxToFood`: distanta maxima de la Pacman la orice bucata de mancare.
- Calculeaza `maxFoodPair`: distanta maxima dintre oricare doua bucati de mancare ramase pe harta.

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

    # daca nu avem mancare -> euristica e 0
    if not foodList:
        return 0

    # initializam un dictionar pentru distante daca nu exista
    if "distantaDintre2Puncte" not in problem.heuristicInfo:
        problem.heuristicInfo["distantaDintre2Puncte"] = {}
    cache = problem.heuristicInfo["distantaDintre2Puncte"]

    # returneaza distanta dintre Pacman si pozitia Food-ului (tine cont de pereti)
    def getMazeDist(a, b):
        key = (a, b)
        if key not in cache:  # daca aceasta distanta nu a fost inca calculata, o calculam si o salvam (mult mai eficient decat sa tot recalculam)
            cache[key] = mazeDistance(a, b, problem.startingGameState)
        return cache[key]

    # 1. gasim cel mai indepartat food fata de Pacman
    maxToFood = 0
    farFood = None
    for food in foodList:
        d = getMazeDist(position, food)
        if d > maxToFood:
            maxToFood = d
            farFood = food

    # 2. "diametrul" food-ului: distanta cea mai mare intre doua bucati de mancare
    maxFoodPair = 0
    for i in range(len(foodList)):
        for j in range(i + 1, len(foodList)):
            a, b = foodList[i], foodList[j]
            d = getMazeDist(a, b)
            if d > maxFoodPair:
                maxFoodPair = d

    # heuristic-ul final
    return max(maxToFood, maxFoodPair)
```

Listing 7: Implementarea euristicii care mananca toata mancarea eficient

10 Q8: Suboptimal Search

10.1 Descriere

Uneori, chiar si cu A* si o euristica buna, gasirea drumului optim prin toate punctele de mancare este prea costisitoare (dureaza prea mult) pe harti mari.

In aceste cazuri, preferam o solutie suficient de buna (suboptima), dar care se calculeaza instant.

Agentul `ClosestDotSearchAgent` foloseste urmatoarea strategie: in loc sa planifice tot traseul de la inceput, cauta mereu doar cea mai apropiata bucata de mancare, se duce la ea, si repeta procesul. Nu garanteaza cel mai scurt drum total, dar curata harta cat de cat repede.

10.2 Implementare

Pentru a implementa functia `findPathToClosestDot`, am folosit direct algoritmul BFS implementat anterior.

Deoarece BFS exploreaza uniform (nivel cu nivel), el garanteaza gasirea celui mai scurt drum catre prima stare intalnita. In cazul problemei `AnyFoodSearchProblem`, tinta este orice patrat care contine mancare. Astfel, ruland BFS pe aceasta problema, obtinem automat calea catre cea mai apropiata mancare.

```
def findPathToClosestDot(self, gameState: pacman.GameState):
        problem = AnyFoodSearchProblem(gameState)
        return search.bfs(problem)
```

Listing 8: Implementarea cautarii celei mai apropiate mancari cu BFS

11 Rezultate si Concluzii

Proiectul a fost finalizat cu succes, obtinand punctajul maxim la testele autograderului. Trecand prin fiecare etapa, am observat cateva lucruri la fiecare algoritm:

- **Q1 (DFS):** Desi gaseste mereu mancare, solutia este adesea foarte lunga deoarece algoritmul merge pe o ramura pana la capat fara sa tine cont de nimic altceva.
- **Q2 (BFS):** Garanteaza cel mai scurt drum (optim) ca numar de pasi, dar consuma multa memorie pentru ca retine toate nodurile de pe un nivel.
- **Q3 (UCS):** Este esential cand costurile de miscare difera, alegand mereu calea cu costul total cel mai mic pana in acel moment.
- **Q4 (A* Search):** Este cel mai eficient algoritm implementat; reuseste sa gaseasca drumul optim mult mai rapid decat UCS, fiind ghidat de euristica spre tinta.
- **Q5 (Corners Problem):** Am inteles importanta definirii corecte a spatiului starilor (pozitie + colturi vizitate) pentru a putea rezolva probleme de cautare mai complexe.
- **Q6 (Corners Heuristic):** O euristica admisibila bine aleasa (distanta pana la cel mai indepartat colt) a redus numarul de noduri expandate de A*.

- **Q7 (Eating All Dots):** A fost partea cea mai grea din proiect. Am vazut ca fara sa tinem minte distantele si fara o euristica buna, algoritmul ruleaza prea incet si da time-out pe hartile mari.
- **Q8 (Suboptimal Search):** Pe hartile uriase, nu mai putem cauta drumul perfect. Am observat ca strategia Greedy (mergi la cea mai apropiata mancare) e un compromis bun: nu scoate cel mai scurt drum, dar macar agentul se misca instant.

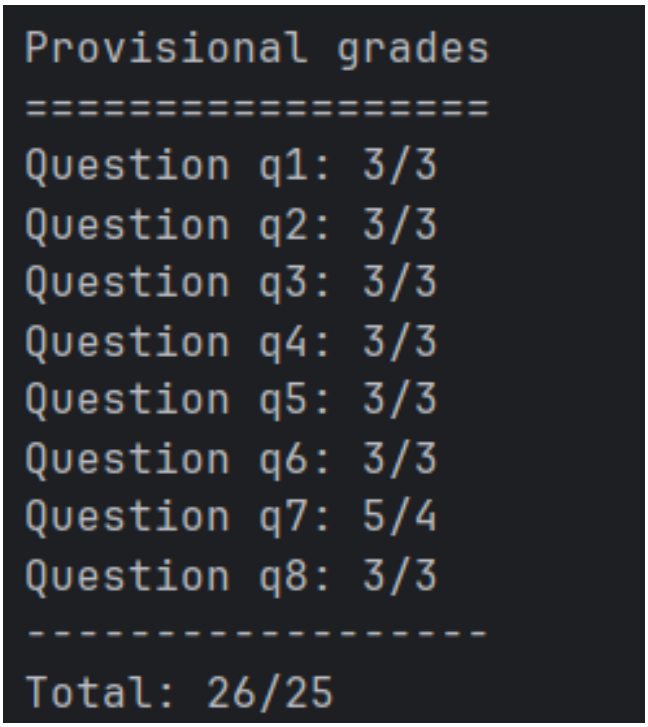

Figura 2: Punctajul final obtinut la rularea autograder.py

In concluzie, proiectul m-a ajutat sa inteleg practic cum o simpla schimbare a structurii de date (Stiva vs Coada vs PriorityQueue) schimba complet comportamentul agentului inteligent.