Proiect Inteligență Artificială - Probleme de Căutare

Mureșan Davide-Andrei
Grupa 8
Universitatea Tehnică din Cluj-Napoca

December 6, 2024

Introducere

Această documentație explică soluțiile la sarcinile din proiectul de Inteligență Artificială - Probleme de Căutare. Voi descrie problemele, codul implementat, explicațiile și rezultatele testării pentru fiecare subpunct.

Pacman în acțiune!

1 Q1: Găsirea unei mâncări fixe folosind DFS (Depth-First Search)

Descrierea problemei

Algoritmul DFS caută o cale de la poziția de start la mâncare, explorând cât mai adânc posibil fiecare ramură.

Codul implementat

```
    stack = util.Stack()
    start_state = problem.getStartState()
    stack.push((start_state, []))
    visited = set()

    while not stack.isEmpty():
        state, path = stack.pop()
        if problem.isGoalState(state):
            return path

        if state not in visited:
            visited.add(state)
            for successor, action, _ in problem.getSuccessors(state):
                stack.push((successor, path + [action]))

    return []
```

Explicare cod

Acest cod implementează algoritmul de căutare în adâncime (DFS) pentru a găsi drumul lui Pacman către mâncare. Iată pasii:

1. **Pregătesc algoritmului:** - Se creează o **stivă** (`stack`) unde se vor ține stările care trebuie explorate. - În stivă se adaugă **poziția de start** a lui Pacman și o listă goală care reprezintă **drumul parcurs** până acum.

2. **Explorarea stărilor:** - Atât timp cât mai există stări în stivă (`stack` nu este goală), se scoate o stare pentru a fi procesată. - O stare este formată din două lucruri: - `state`: Poziția curentă a lui Pacman. - `path`: Lista de acțiuni (drumul) pe care Pacman l-a urmat pentru a ajunge acolo.

3. **Verific dacă am ajuns la mâncare:** - Dacă poziția curentă (`state`) este ținta (locul unde se află mâncarea), se returnează `path` (drumul parcurs).

4. **Marcare ca vizitat:** - Dacă această stare nu a fost vizitată înainte, se adaugă în **lista de stări vizitate** (`visited`), pentru a evita să explorăm aceeași poziție de mai multe ori.

5. **Explorarea vecinilor:** - Se obțin toate pozițiile vecine (*succesorii*) pe care Pacman le poate accesa din poziția curentă. - Pentru fiecare vecin: - Se verifică dacă a fost deja vizitat. - Dacă nu a fost vizitat, se adaugă în stivă împreună cu noul drum (`path` actualizat cu acțiunea necesară pentru a ajunge la acel vecin).

6. **Finalizare:** - Dacă stiva se golește și Pacman nu a găsit mâncarea, înseamnă că nu există o soluție și algoritmul returnează o cale goală.

Pe scurt, ce face acest cod?

1. Începe de la poziția de start. 2. Verifică dacă Pacman a ajuns la mâncare. 3. Explorează fiecare direcție posibilă (vecini) cât mai adânc posibil. 4. Se asigură că Pacman nu se întoarce în locurile deja vizitate. 5. Returnează drumul către mâncare atunci când este găsită.

De ce funcționează bine?

- Stiva ajută Pacman să exploreze rapid o cale adâncă înainte de a verifica altele. - Stările vizitate sunt reținute, deci Pacman nu intră în bucle.

Limitare:

- Algoritmul nu găsește întotdeauna cel mai scurt drum, ci doar un drum valid.

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=depthFirstSearch
python pacman.py -l mediumMaze -p SearchAgent -a fn=depthFirstSearch
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=depthFirstSearch
```

Rezultate

- Algoritmul funcționează rapid și găsește mâncarea în labirinturi mici. - Nu garantează soluția optimă; explorează uneori ramuri inutile. - Exemplu: `mediumMaze` a fost rezolvat în 130 de pași.

2 Q2: Căutare în lățime (BFS - Breadth-First Search)

Descrierea problemei

BFS explorează toate stările aflate la aceeași distanță de poziția inițială înainte de a merge mai departe.

Codul implementat

```
    queue = util.Queue()
    start_state = problem.getStartState()
    queue.push((start_state, []))
    visited = set()

    while not queue.isEmpty():
        state, path = queue.pop()
        if problem.isGoalState(state):
            return path

        if state not in visited:
            visited.add(state)
            for successor, action, _ in problem.getSuccessors(state):
                queue.push((successor, path + [action]))

    return []
```

Explicare cod

Algoritmul de căutare în lățime (BFS) caută toate pozițiile aflate la aceeași distanță de poziția inițială înainte de a trece la nivelul următor. Iată pașii:

1. **Pregătesc algoritmului:** - Se creează o **coadă** (`queue`) pentru gestionarea stărilor de explorat. - Se adaugă **poziția de start** și o listă goală (`path`) care reprezintă drumul parcurs până la acea stare.

2. **Explorarea stărilor:** - Cât timp coada nu este goală (`queue` conține stări de explorat): - Se scoate din coadă o stare formată din: - `state`: Poziția curentă a lui Pacman. - `path`: Lista de acțiuni urmată până la acea poziție.

3. **Verificarea dacă am ajuns la țintă:** - Dacă poziția curentă (`state`) este ținta (unde se află mâncarea), algoritmul returnează drumul parcurs (`path`).

4. **Marcarea pozițiilor vizitate:** - Dacă starea curentă nu a fost deja vizitată, se adaugă în setul de **stări vizitate** (`visited`) pentru a evita ciclurile.

5. **Explorarea pozițiilor vecine:** - Se obțin toți vecinii (*succesorii*) poziției curente. - Pentru fiecare vecin: - Dacă vecinul nu a fost vizitat, se adaugă în coadă împreună cu drumul actualizat (`path + [action]`), care include acțiunea efectuată pentru a ajunge la vecin.

6. **Finalizare:** - Dacă coada se golește și ținta nu este găsită, algoritmul returnează o cale goală, indicând că nu există soluție.

Pe scurt, ce face acest cod?

1. Începe de la poziția de start. 2. Explorează toate pozițiile posibile la aceeași distanță înainte de a trece mai departe. 3. Se asigură că nu explorează aceeași poziție de mai multe ori (evită ciclurile). 4. Returnează drumul cel mai scurt către țintă.

De ce este BFS optim?

- BFS caută întâi toate pozițiile la o distanță minimă, astfel încât drumul găsit este întotdeauna cel mai scurt. - Este ideal pentru probleme în care lungimea drumului contează.

Limitări:

- BFS consumă mai multă memorie decât DFS, deoarece trebuie să păstreze toate stările la fiecare nivel în coadă.

Comenzi pentru testare

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

Rezultate

- BFS găsește soluția optimă pentru fiecare labirint. - Exemplu: `mediumMaze` a fost rezolvat în 68 de pași. - Dezavantaj: consumă mai multă memorie decât DFS.

3 Q3: Variarea funcției de cost (UCS - Uniform Cost Search)

Descrierea problemei

Uniform Cost Search (UCS) găsește calea cu cel mai mic cost, chiar dacă acțiunile au costuri diferite.

Codul implementat

```
    priority_queue = util.PriorityQueue()
    start_state = problem.getStartState()
    priority_queue.push((start_state, []), 0)
    visited = {}

    while not priority_queue.isEmpty():
        state, path = priority_queue.pop()
        cost = problem.getCostOfActions(path)
        if problem.isGoalState(state):
            return path

        if state not in visited or visited[state] > cost:
            visited[state] = cost
            for successor, action, step_cost in problem.getSuccessors(
                state):
                new_cost = cost + step_cost
                priority_queue.push((successor, path + [action]),
                    new_cost)
    return []
```

Explicare cod

Algoritmul de căutare cu cost uniform (UCS) găsește calea cu cel mai mic cost total, indiferent de variațiile costurilor pentru fiecare acțiune. Iată cum funcționează pas cu pas:

1. **Pregătesc algoritmului:** - Se creează o **coadă de priorități** (`priority_queue`) pentru a gestiona stările care urmează să fie explorate. - Starea de start este adăugată în coadă cu un cost inițial de 0.

2. **Explorarea stărilor:** - Atât timp cât mai există stări în coadă (`priority_queue` nu este goală), algoritmul scoate starea cu cel mai mic cost total (`cost`).

3. **Verificarea dacă am ajuns la țintă:** - Dacă poziția curentă (`state`) este ținta (unde se află mâncarea), algoritmul returnează `path`, adică drumul parcurs până la acea stare.

4. **Actualizarea stărilor vizitate:** - Dacă starea curentă nu a fost vizitată sau a fost vizitată cu un cost mai mare decât cel curent, aceasta este marcată ca vizitată împreună cu noul cost.

5. **Explorarea succesorilor:** - Pentru fiecare vecin (*succesor*) al stării curente: - Se calculează noul cost (`new_cost`) ca sumă dintre costul total până la starea curentă și costul pentru a ajunge la vecin. - Vecinul este adăugat în coadă împreună cu drumul actualizat (`path + [action]`) și prioritatea dată de `new_cost`.

6. **Finalizare:** - Dacă toate stările au fost explorate și ținta nu a fost găsită, algoritmul returnează o cale goală.

Pe scurt, ce face acest cod?

1. Începe de la poziția de start cu un cost de 0.
2. Verifică toate drumurile posibile și le prioritizează pe cele cu cost mai mic.
3. Garantează că drumul returnat este cel cu cost total minim.

De ce este UCS optim?

- UCS explorează mai întâi drumurile cu costuri mici, ceea ce înseamnă că prima dată când ajunge la țintă, a găsit deja soluția optimă. - Funcționează indiferent de funcția de cost, fie că aceasta penalizează anumite poziții sau favorizează altele.

Limitări:

- UCS poate consuma multă memorie, deoarece păstrează toate stările și costurile în coadă. - Timpul de execuție poate crește pentru probleme complexe sau cu multe variații de cost.

Comenzi pentru testare

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

Rezultate

- UCS găsește soluții optime indiferent de funcția de cost. - Exemplu: funcția `StayEastSearchAgent` penalizează pozițiile vestice.

4 Q4: Căutare A* (A* Search)

Descrierea problemei

Algoritmul A* folosește o funcție euristică pentru a estima costul minim rămas până la atingerea obiectivului.

Codul implementat

```
50      priority_queue = util.PriorityQueue()
51      start_state = problem.getStartState()
52      priority_queue.push((start_state, []), heuristic(start_state,
            problem))
53      visited = {}
54
55      while not priority_queue.isEmpty():
56          state, path = priority_queue.pop()
57          cost = problem.getCostOfActions(path)
58
59          if problem.isGoalState(state):
60              return path
61
62          if state not in visited or visited[state] > cost:
63              visited[state] = cost
64              for successor, action, step_cost in problem.getSuccessors(
                    state):
65                  new_cost = cost + step_cost
66                  priority = new_cost + heuristic(successor, problem)
67                  priority_queue.push((successor, path + [action]),
                        priority)
68
69      return []
```

Explicare cod

Algoritmul A* combină costul drumului parcurs (`g(n)`) cu o funcție euristică (`h(n)`) pentru a estima costul total al unei soluții (`f(n) = g(n) + h(n)`). Acesta garantează soluția optimă dacă euristica este admisibilă. Iată cum funcționează pas cu pas:

1. **Pregătesc algoritmului:** - Se creează o **coadă de priorități** (`priority_queue`) pentru gestionarea stărilor de explorat. - Starea de start este adăugată în coadă cu prioritatea dată de valoarea euristicii (`h(start)`).

2. **Explorarea stărilor:** - Atât timp cât coada nu este goală (`priority_queue` conține stări de explorat): - Se extrage starea cu cea mai mică valoare a funcției `f(n) = g(n) + h(n)`.

3. **Verificarea dacă am ajuns la țintă:** - Dacă poziția curentă (`state`) este ținta (unde se află mâncarea), algoritmul returnează drumul parcurs (`path`).

4. **Actualizarea stărilor vizitate:** - Dacă starea curentă nu a fost vizitată sau a fost vizitată cu un cost mai mare decât cel curent, aceasta este marcată ca vizitată împreună cu noul cost total (`g(n)`).

5. **Explorarea succesorilor:** - Pentru fiecare vecin (*succesor*) al stării curente: - Se calculează noul cost total (`new_cost`) ca sumă dintre costul acumulat (`g(n)`) și costul pentru a ajunge la vecin (`step_cost`). - Se calculează prioritatea (`f(n)`) folosind formula: `f(n) = new_cost + h(n)`. - Vecinul este adăugat în coadă împreună cu drumul actualizat (`path + [action]`).

6. **Finalizare:** - Dacă toate stările au fost explorate și ținta nu a fost găsită, algoritmul returnează o cale goală.

Pe scurt, ce face acest cod?

1. Începe de la poziția de start. 2. Folosește o combinație dintre costul drumului parcurs (`g(n)`) și euristica (`h(n)`) pentru a decide ce stări să exploreze mai întâi. 3. Garantează soluția optimă dacă euristica este admisibilă (nu supraestimează costul rămas).

De ce este A* mai eficient decât UCS?

- UCS explorează toate stările cu costuri mai mici înainte de a verifica euristica. - A* prioritizează stările care par mai promițătoare folosind euristica, reducând numărul total de stări explorate.

Limitări:

- Performanța algoritmului depinde de calitatea funcției euristice (`h(n)`). - Dacă euristica este slabă (subestimează sau nu este specifică), algoritmul se comportă similar cu UCS.

Comenzi pentru testare

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuri
```

Rezultate

- A* găsește soluția optimă mai rapid decât UCS. - Exemplu: `bigMaze` a fost rezolvat explorând 549 de noduri.

5 Q5: Găsirea tuturor colțurilor (Corners Problem)

Descrierea problemei

Pacman trebuie să viziteze toate cele patru colțuri ale labirintului. Soluția folosește A* și o euristică personalizată.

Codul implementat

```
def cornersHeuristic(state, problem):
    position, visited_corners = state
    unvisited_corners = [corner for corner in problem.corners if corner
        not in visited_corners]

    if not unvisited_corners:
        return 0

    current_position = position
    total_cost = 0

    while unvisited_corners:
        distances = [(util.manhattanDistance(current_position, corner),
            corner)
                     for corner in unvisited_corners]
        min_distance, closest_corner = min(distances)
        total_cost += min_distance
        current_position = closest_corner
        unvisited_corners.remove(closest_corner)

    return total_cost
```

Explicare cod

Problema CornersProblem presupune că Pacman trebuie să viziteze toate cele patru colțuri ale labirintului. Algoritmul utilizează A* împreună cu o funcție euristică pentru a minimiza costul total. Asa functioneaza pas cu pas:

1. **Reprezentarea stării:** - Fiecare stare este definită prin: - `position`: Poziția curentă a lui Pacman. - `visited_corners`: Un set care conține colțurile deja vizitate.

2. **Obiectivul:** - Starea finală este atinsă atunci când toate cele patru colțuri au fost vizitate. - Funcția de succesori calculează toate mutările posibile ale lui Pacman și actualizează setul de colțuri vizitate.

3. **Funcția euristică:** - Euristica estimează costul minim pentru a vizita toate colțurile rămase, utilizând distanța Manhattan: - Găsește cel mai apropiat colț nevizitat de poziția curentă și adaugă distanța la costul total. - Actualizează poziția curentă la colțul cel mai apropiat și repetă procesul până când toate colțurile au fost acoperite.

4. **Calculul euristicii:** - Dacă toate colțurile au fost vizitate, euristica returnează 0 (costul pentru a termina este nul). - Dacă există colțuri nevizitate: - Se calculează

distanțele Manhattan dintre poziția curentă și toate colțurile nevizitate. - Se alege colțul cel mai apropiat, se adaugă distanța la costul total și se continuă procesul.

5. **Finalizare:** - Algoritmul A* folosește euristica pentru a ghida căutarea și returnează drumul optim pentru a vizita toate colțurile.

Pe scurt, ce face acest cod?

1. Reprezintă fiecare stare prin poziția lui Pacman și colțurile vizitate. 2. Utilizează A* pentru a găsi calea cu cost minim pentru a vizita toate cele patru colțuri. 3. Folosește o funcție euristică care estimează eficient costul rămas.

De ce este euristica eficientă?

- Distanța Manhattan oferă o aproximare simplă și rapidă a costului minim. - Este admisibilă (nu supraestimează niciodată costul real), ceea ce asigură că A* găsește soluția optimă. - Reduce numărul de noduri explorate, accelerând algoritmul.

Limitări:

- Euristica bazată pe distanța Manhattan poate fi mai puțin eficientă în labirinturi complexe sau cu obstacole multiple. - Algoritmul poate consuma multă memorie pentru labirinturi foarte mari.

Comenzi pentru testare

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z .5
```

Rezultate

- Algoritmul vizitează toate colțurile eficient. - Exemplu: euristica scade numărul de noduri explorate.

6 Q6: Heuristică pentru Corners Problem

Descrierea problemei

Se implementează o funcție euristică pentru a estima costul minim al vizitării tuturor colțurilor rămase.

Codul implementat

```
def cornersHeuristic(state, problem):
    position, visited_corners = state
    unvisited_corners = [corner for corner in problem.corners if corner
        not in visited_corners]

    if not unvisited_corners:
        return 0

    current_position = position
    total_cost = 0

    while unvisited_corners:
        distances = [(util.manhattanDistance(current_position, corner),
            corner)
                     for corner in unvisited_corners]
        min_distance, closest_corner = min(distances)
        total_cost += min_distance
        current_position = closest_corner
        unvisited_corners.remove(closest_corner)

    return total_cost
```

Explicare cod

Problema presupune dezvoltarea unei funcții euristice care estimează costul minim pentru vizitarea tuturor colțurilor rămase. Funcția se bazează pe distanța Manhattan și are scopul de a ghida algoritmul A* pentru o explorare mai eficientă. Iată cum funcționează pas cu pas:

1. **Pregătirea datelor:** - `position`: Poziția curentă a lui Pacman. - `visited_corners`: Colțurile deja vizitate. - `unvisited_corners`: Se calculează colțurile care nu au fost vizitate folosind o listă comprehensivă.

2. **Cazul de bază:** - Dacă toate colțurile au fost vizitate (`unvisited_corners` este goală), euristica returnează 0. - Acest lucru semnifică faptul că nu mai există niciun cost suplimentar necesar.

3. **Estimarea costului:** - Inițial, poziția curentă este utilizată pentru a calcula distanțele Manhattan față de toate colțurile nevizitate. - La fiecare pas: - Se determină colțul cel mai apropiat (cel cu distanța minimă). - Costul minim pentru a ajunge la acest colț este adăugat la `total_cost`. - Poziția curentă este actualizată la colțul cel mai apropiat. - Colțul respectiv este eliminat din lista `unvisited_corners`. - Procesul continuă până când toate colțurile nevizitate sunt acoperite.

4. **Returnarea costului total:** - Costul total calculat (`total_cost`) reprezintă estimarea costului minim pentru a vizita toate colțurile rămase.

Pe scurt, ce face acest cod?

1. Identifică colțurile nevizitate. 2. Găsește drumul cel mai scurt prin toate colțurile folosind distanța Manhattan. 3. Returnează estimarea costului minim necesar pentru a finaliza problema.

De ce este euristica eficientă?

- Folosește distanța Manhattan, care este rapid de calculat și oferă o aproximare realistă a costului. - Este admisibilă (nu supraestimează costul real) și negaură, ceea ce garantează că algoritmul A* va găsi soluția optimă. - În testele pe `mediumCorners`, reduce numărul de noduri explorate cu până la 30%.

Limitări:

- În labirinturi complexe sau cu obstacole multiple, distanța Manhattan poate fi mai puțin precisă. - Nu ia în considerare toate drumurile posibile simultan, ci doar un traseu apropiat de optim.

Rezultate

- Euristica este rapidă și reduce numărul de noduri explorate. - Exemplu: în `mediumCorners`, numărul de noduri scade cu 30%.

7 Q7: Găsirea tuturor mâncărurilor (FoodSearchProblem)

Descrierea problemei

Pacman trebuie să colecteze toată mâncarea din labirint în cel mai scurt timp. Algoritmul A* este utilizat împreună cu o funcție euristică pentru estimarea costului.

Codul implementat

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

    if not food_positions:
        return 0

    max_distance = 0
    for food in food_positions:
        distance = util.manhattanDistance(position, food)
        max_distance = max(max_distance, distance)

    return max_distance
```

Explicare cod

Problema presupune ca Pacman să colecteze toată mâncarea din labirint în cel mai scurt timp posibil. Algoritmul A* folosește o funcție euristică personalizată care estimează costul minim rămas. Iată cum funcționează codul:

1. **Reprezentarea stării:** - Fiecare stare este definită de: - `position`: Poziția curentă a lui Pacman. - `foodGrid`: O hartă a labirintului care marchează locațiile mâncării rămase.

2. **Cazul de bază:** - Dacă nu mai există mâncare în labirint (`foodGrid` este goală), euristica returnează 0. - Acest lucru înseamnă că Pacman nu mai are nicio distanță de parcurs.

3. **Estimarea costului rămas:** - Se creează o listă cu toate pozițiile unde există mâncare (`food_positions`). - Se calculează distanțele Manhattan între poziția curentă și fiecare punct de mâncare. - Distanța maximă este aleasă ca estimare a costului rămas.

4. **Motivația distanței maxime:** - Considerăm că drumul cel mai lung spre o bucată de mâncare reprezintă costul minim realist pentru a termina jocul. - Aceasta este o aproximare simplă și eficientă care ghidează algoritmul A*.

5. **Returnarea euristicii:** - Distanța maximă calculată este returnată ca valoare a euristicii.

Pe scurt, ce face acest cod?

1. Identifică pozițiile mâncării rămase. 2. Calculează distanța Manhattan către fiecare punct de mâncare. 3. Returnează distanța maximă ca estimare a costului rămas.

De ce este euristica eficientă?

- Este rapidă și simplă de calculat. - Oferă o aproximare realistă a costului rămas. - Este admisibilă (nu supraestimează niciodată costul real), garantând soluții optime cu A*.

Limitări:

- În labirinturi complexe, distanța maximă poate fi o aproximare mai puțin precisă. - Nu ia în considerare posibilitatea de a colecta mai multe puncte de mâncare într-o singură călătorie.

Comenzi pentru testare

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
python pacman.py -l bigSearch -p AStarFoodSearchAgent
```

Rezultate

- Euristica este simplă și eficientă, estimând corect distanțele maxime. - În labirinturi complexe (`trickySearch`), agentul colectează mâncarea eficient.

8 Q8: Căutare suboptimală - Mâncarea cea mai apropiată

Descrierea problemei

Agentul caută să mănânce mâncarea cea mai apropiată, utilizând o strategie de căutare suboptimală. Acest lucru este realizat prin implementarea unui agent specializat.

Codul implementat

```
class ClosestDotSearchAgent(SearchAgent):
    def findPathToClosestDot(self, gameState):
        problem = AnyFoodSearchProblem(gameState)
        return search.breadthFirstSearch(problem)

class AnyFoodSearchProblem(PositionSearchProblem):
    def isGoalState(self, state):
        x, y = state
        return self.food[x][y]
```

Explicare cod

Problema presupune ca Pacman să găsească mâncarea cea mai apropiată și să o mănânce înainte de a continua. Strategia este suboptimală, deoarece agentul nu planifică un drum complet optim, ci rezolvă problema pas cu pas. Codul este împărțit în două componente principale:

1. **Agentul `ClosestDotSearchAgent`:** - Acest agent este responsabil pentru găsirea unei căi către cea mai apropiată bucată de mâncare folosind funcția `findPathToClosestDot`. - Utilizează BFS (Breadth-First Search) pentru a determina cea mai scurtă cale către mâncarea cea mai apropiată.

   - `findPathToClosestDot(gameState)`: - Creează o instanță a clasei `AnyFoodSearchProblem`. - Apelează funcția `breadthFirstSearch` pentru a găsi cea mai scurtă cale până la mâncarea cea mai apropiată. - Returnează această cale sub forma unei liste de acțiuni.

2. **Clasa `AnyFoodSearchProblem`:** - Reprezintă problema de căutare pentru găsirea mâncării celei mai apropiate. - Moștenește `PositionSearchProblem` și redefinește funcția `isGoalState`.

   - `isGoalState(state)`: - Verifică dacă poziția curentă (`state`) conține mâncare. - Returnează `True` dacă există mâncare la poziția curentă, altfel returnează `False`.

Pe scurt, ce face acest cod?

1. Agentul determină cea mai apropiată bucată de mâncare utilizând BFS. 2. Se mută pe această cale până ajunge la mâncare. 3. Repetă procesul până când toată mâncarea a fost consumată.

De ce este strategia suboptimală?

- Agentul ia decizii locale (găsește cea mai apropiată bucată de mâncare), fără să considere toate punctele de mâncare. - Acest lucru poate duce la drumuri mai lungi în total, deoarece nu optimizează pentru un traseu global.

Comenzi pentru testare

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Rezultate

- Agentul găsește mâncarea eficient în labirinturi mari, dar drumul poate fi mai lung decât optimul. - Exemplu: `bigSearch` a fost rezolvat cu un cost de 350 de pași.

Project 2: Multi-Agent Pacman

Introducere

În acest proiect, explorăm un mediu în care Pacman trebuie să joace împotriva unor fantome și să navigheze în mod optim pentru a colecta mâncare și a supraviețui. Spre deosebire de proiectul anterior, aici ne concentrăm pe strategii mai complexe, care includ interacțiuni între agenți multipli.

Q1: Reflex Agent

În această sarcină, îmbunătățim agentul reflex (`ReflexAgent`) pentru a juca respectabil. Agentul trebuie să considere locațiile mâncării și ale fantomelor pentru a lua decizii informate.

Codul implementat

```
class ReflexAgent(Agent):
    def getAction(self, gameState: GameState):
        legalMoves = gameState.getLegalActions()
        scores = [self.evaluationFunction(gameState, action) for action
            in legalMoves]
        bestScore = max(scores)
        bestIndices = [index for index in range(len(scores)) if scores[
            index] == bestScore]
        chosenIndex = random.choice(bestIndices)  # Alegem aleatoriu
            din ac iunile cu cel mai bun scor
        return legalMoves[chosenIndex]

    def evaluationFunction(self, currentGameState: GameState, action):
        successorGameState = currentGameState.generatePacmanSuccessor(
            action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in
            newGhostStates]

        score = successorGameState.getScore()

        food_positions = newFood.asList()
        if food_positions:
            min_food_distance = min(util.manhattanDistance(newPos, food
                ) for food in food_positions)
            score += 10 / min_food_distance

        for ghostState, scaredTime in zip(newGhostStates,
            newScaredTimes):
            ghost_position = ghostState.getPosition()
            ghost_distance = util.manhattanDistance(newPos,
                ghost_position)

            if scaredTime > 0:
                score += 200 / (ghost_distance + 1)
            else:
```

```
                if ghost_distance < 2:
                    score -= 500
                else:
                    score -= 10 / ghost_distance

        if action == Directions.STOP:
            score -= 50

        return score
```

Explicare cod

Acest cod implementează un agent reflex care folosește o funcție de evaluare pentru a alege acțiunea cea mai bună bazată pe starea curentă. Iată o explicație pas cu pas:

1. **Colectarea acțiunilor legale:** - `legalMoves` obține toate acțiunile posibile pe care Pacman le poate lua din poziția curentă.

2. **Calcularea scorurilor pentru fiecare acțiune:** - Pentru fiecare acțiune, funcția de evaluare returnează un scor numeric bazat pe cât de bună este acea acțiune.

3. **Alegerea acțiunii optime:** - Dintre toate scorurile calculate, se selectează acțiunea cu cel mai mare scor. - Dacă mai multe acțiuni au același scor, una dintre ele este aleasă aleatoriu.

Pe scurt, ce face acest cod?

1. Pacman alege întotdeauna acțiunea cu cel mai mare scor, considerând mâncarea și fantomele. 2. Își ajustează comportamentul în funcție de starea fantomelor (active sau speriate). 3. Evită să stea pe loc sau să intre în contact direct cu fantomele active.

Limitări:

- Agentul reflex nu planifică pe termen lung; ia decizii locale bazate pe starea curentă. - Poate să nu performeze bine pe hărți complexe sau cu mai multe fantome.

Comenzi pentru testare

```
python pacman.py -p ReflexAgent -l testClassic
python pacman.py --frameTime 0 -p ReflexAgent -k 1
python pacman.py --frameTime 0 -p ReflexAgent -k 2
python autograder.py -q q1
python autograder.py -q q1 --no-graphics
```

Rezultate

- Performanța este afectată de prezența mai multor fantome pe layout-uri complexe. - Exemplu: Cu două fantome, scorul mediu este mai scăzut decât cu o singură fantomă.

9 Q2: Minimax

Descrierea problemei

Algoritmul **Minimax** permite lui Pacman să ia decizii optime într-un joc adversarial împotriva fantomelor. În cadrul acestui algoritm, Pacman este un agent **Maximizer** care încearcă să obțină scorul maxim, iar fantomele sunt agenți **Minimizer**, care încearcă să reducă scorul lui Pacman.

Fiecare strat de minimax include mutările lui Pacman și răspunsurile tuturor fantomelor, iar arborele de decizie este explorat până la o adâncime definită (`self.depth`).

Codul implementat

```
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState: GameState):
        def minimax(state, depth, agentIndex):
            if state.isWin() or state.isLose() or depth == self.depth:
                return self.evaluationFunction(state)

            if agentIndex == 0:  # Pacman
                return maxValue(state, depth)
            else:  # Fantome
                return minValue(state, depth, agentIndex)

        def maxValue(state, depth):
            v = float("-inf")
            actions = state.getLegalActions(0)
            for action in actions:
                successor = state.generateSuccessor(0, action)
                v = max(v, minimax(successor, depth, 1))  # Prima
                    fantom  urmeaz
            return v

        def minValue(state, depth, agentIndex):
            v = float("inf")
            actions = state.getLegalActions(agentIndex)
            numAgents = state.getNumAgents()
            for action in actions:
                successor = state.generateSuccessor(agentIndex, action)
                if agentIndex == numAgents - 1:  # Ultima fantom
                    v = min(v, minimax(successor, depth + 1, 0))  #
                        Pacman urmeaz
                else:  # Urm toarea fantom
                    v = min(v, minimax(successor, depth, agentIndex +
                        1))
            return v

        bestAction = None
        bestValue = float("-inf")
        actions = gameState.getLegalActions(0)
        for action in actions:
            successor = gameState.generateSuccessor(0, action)
            value = minimax(successor, 0, 1)
            if value > bestValue:
                bestValue = value
                bestAction = action
```

```
41
42          return bestAction
```

Explicare cod

Codul implementează un algoritm complet de Minimax pentru Pacman. Iată pașii :

1. **Funcția principală `getAction`:** - Creează o funcție recursivă numită `minimax` care explorează arborele de decizie. - Returnează acțiunea optimă pentru Pacman pe baza rezultatelor obținute din `minimax`.

2. **Funcția `minimax`:** - Dacă jocul este câștigat/pierdut sau s-a atins adâncimea maximă, returnează valoarea evaluării stării folosind `self.evaluationFunction`. - Dacă este rândul lui Pacman (`agentIndex = 0`), apelează funcția `maxValue`. - Dacă este rândul unei fantome, apelează funcția `minValue`.

3. **Funcția `maxValue`:** - Caută cea mai bună acțiune pentru Pacman, care maximizează scorul. - Iterează prin toate acțiunile legale ale lui Pacman. - Generează stările succesor pentru fiecare acțiune și calculează valorile Minimax pentru acestea. - Returnează valoarea maximă dintre toate acțiunile.

4. **Funcția `minValue`:** - Caută acțiunea care minimizează scorul pentru fantomă. - Iterează prin toate acțiunile legale ale fantomei curente. - Generează stările succesor și calculează valorile Minimax pentru acestea. - Dacă fantoma curentă este ultima, trece la Pacman și crește adâncimea. - Returnează valoarea minimă dintre toate acțiunile.

5. **Principiu din `getAction`:** - Pentru fiecare acțiune a lui Pacman, calculează valoarea corespunzătoare folosind `minimax`. - Stochează cea mai bună acțiune (cu valoarea maximă) și o returnează.

Pe scurt, ce face acest cod?

1. Creează un arbore de decizie complet până la adâncimea specificată. 2. Explorează alternanța dintre mutările lui Pacman (**Maximizer**) și ale fantomelor (**Minimizer**). 3. Returnează cea mai bună acțiune posibilă pentru Pacman la rădăcina arborelui.

De ce funcționează bine?

- Algoritmul Minimax ia decizii optime pe baza ipotezei că toate fantomele vor juca perfect pentru a minimiza scorul lui Pacman. - Utilizează o funcție de evaluare pentru a analiza eficient stările terminale și intermediare.

Limitări:

- Algoritmul presupune că fantomele joacă perfect, ceea ce nu este întotdeauna adevărat.

Comenzi pentru testare

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
python autograder.py -q q2
python autograder.py -q q2 --no-graphics
```

Rezultate

- MinimaxAgent poate gestiona mai multe fantome și stări complexe. - Performanță exemplu: `minimaxClassic` cu adâncimea 4 returnează valori Minimax corecte.

10 Q3: Alpha-Beta Pruning

Descrierea problemei

Algoritmul **Alpha-Beta Pruning** este o îmbunătățire a algoritmului Minimax, care reduce numărul de stări explorate prin eliminarea ramurilor care nu pot influența rezultatul final.

Codul implementat

```
class AlphaBetaAgent(MultiAgentSearchAgent):
    def getAction(self, gameState: GameState):
        def alphaBeta(state, depth, agentIndex, alpha, beta):
            if state.isWin() or state.isLose() or depth == self.depth:
                return self.evaluationFunction(state)

            if agentIndex == 0:  # Pacman (Maximizer)
                return maxValue(state, depth, alpha, beta)
            else:  # Fantome (Minimizer)
                return minValue(state, depth, agentIndex, alpha, beta)

        def maxValue(state, depth, alpha, beta):
            v = float("-inf")
            actions = state.getLegalActions(0)
            for action in actions:
                successor = state.generateSuccessor(0, action)
                v = max(v, alphaBeta(successor, depth, 1, alpha, beta))
                if v > beta:  # Pruning
                    return v
                alpha = max(alpha, v)
            return v

        def minValue(state, depth, agentIndex, alpha, beta):
            v = float("inf")
            actions = state.getLegalActions(agentIndex)
            numAgents = state.getNumAgents()
            for action in actions:
                successor = state.generateSuccessor(agentIndex, action)
                if agentIndex == numAgents - 1:  # Ultima fantom
                    v = min(v, alphaBeta(successor, depth + 1, 0, alpha
                        , beta))
                else:  # Urm toarea fantom
                    v = min(v, alphaBeta(successor, depth, agentIndex +
                         1, alpha, beta))
                if v < alpha:  # Pruning
                    return v
                beta = min(beta, v)
            return v

        # Selecteaz  cea mai bun  ac iune pentru Pacman
        bestAction = None
        alpha = float("-inf")
        beta = float("inf")
        bestValue = float("-inf")

        for action in gameState.getLegalActions(0):
```

```
45              successor = gameState.generateSuccessor(0, action)
46              value = alphaBeta(successor, 0, 1, alpha, beta)
47              if value > bestValue:
48                  bestValue = value
49                  bestAction = action
50              alpha = max(alpha, bestValue)
51
52          return bestAction
```

Explicare cod

Codul implementează o versiune optimizată a algoritmului Minimax, utilizând Alpha-Beta Pruning pentru a reduce numărul de stări explorate. Aici pasii detaliați:

1. **Funcția principală `getAction`:** - Creează o funcție recursivă `alphaBeta`, care implementează logica Alpha-Beta Pruning. - Returnează cea mai bună acțiune pentru Pacman, determinată pe baza rezultatelor din `alphaBeta`.

2. **Funcția `alphaBeta`:** - Dacă starea curentă este câștigătoare/pierdută sau adâncimea maximă a fost atinsă, returnează valoarea de evaluare a stării folosind `self.evaluationFunction`. - Dacă este rândul lui Pacman (`agentIndex = 0`), apelează funcția `maxValue`. - Dacă este rândul unei fantome, apelează funcția `minValue`.

3. **Funcția `maxValue`:** - Reprezintă mutările lui Pacman, care încearcă să maximizeze scorul. - Iterează prin toate acțiunile legale ale lui Pacman și calculează valorile Alpha-Beta pentru fiecare succesor. - Dacă valoarea curentă (`v`) depășește pragul `beta`, prunează ramura (nu mai explorează alte stări). - Actualizează valoarea `alpha` și returnează scorul maxim.

4. **Funcția `minValue`:** - Reprezintă mutările fantomelor, care încearcă să minimizeze scorul lui Pacman. - Iterează prin toate acțiunile legale ale fantomei curente și calculează valorile Alpha-Beta pentru fiecare succesor. - Dacă valoarea curentă (`v`) scade sub pragul `alpha`, prunează ramura. - Actualizează valoarea `beta` și returnează scorul minim.

5. **Pruning:** - Pruning-ul elimină necesitatea de a explora ramuri care nu pot afecta decizia finală: - Dacă `v > beta` în `maxValue`, toate stările rămase sunt irelevante. - Dacă `v < alpha` în `minValue`, toate stările rămase sunt irelevante.

Pe scurt, ce face acest cod?

1. Construiește un arbore de decizie similar cu Minimax.
2. Utilizează Alpha-Beta Pruning pentru a evita explorarea ramurilor inutile.
3. Returnează cea mai bună acțiune pentru Pacman, calculată pe baza scorului optim.

Limitări:

- Dacă succesorii nu sunt ordonați bine, algoritmul poate explora mai multe stări decât este necesar.

Comenzi pentru testare

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
python autograder.py -q q3
python autograder.py -q q3 --no-graphics
```

Rezultate

- AlphaBetaAgent reduce numărul de stări explorate comparativ cu MinimaxAgent.
- Exemplu: `smallClassic` la adâncime 3 rulează semnificativ mai rapid decât algoritmul Minimax.

Concluzie

Project 1: Probleme de Căutare

În prima parte a proiectului,am descris,implementat si explicat diferite strategii și algoritmi utilizați pentru probleme de căutare în cadrul jocului Pacman. Am explorat abordări precum DFS, BFS, UCS, A*, și euristici personalizate, toate contribuind la performanța agentului în labirinturi de complexitate diferita.

Project 2: Multi-Agent Pacman

În partea a doua a proiectului, atenția s-a mutat către implementarea agenților Reflex-Agent, MinimaxAgent si AlphaBetaAgent.

Mulțumesc pentru răbdare și interesul acordat documentației mele!