UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Medvichi Flavian
Grupa: 30236

Facultatea de Automatica
si Calculatoare

4 Decembrie 2023

Cuprins

1 Search

1.1 Introducere

Proiectul Pac-Man de la Universitatea Berkeley reprezintă o inițiativă captivantă și educațională în domeniul inteligenței artificiale. Scopul fundamental al acestui proiect este să ofere studenților oportunitatea de a aplica cunoștințele dobândite în cadrul laboratorului de inteligență artificială pentru a dezvolta strategii avansate de control pentru Pac-Man și pentru a îmbunătăți comportamentul agenților de joc, inclusiv al fantomelor.

Prin această experiență practică, vom explora diverse tehnici și algoritmi din domeniul inteligenței artificiale, cum ar fi căutarea informațională și neinformațională. Proiectul oferă oportunități unice de a testa și de a implementa soluții creative pentru provocările întâlnite în cerințele jocului Pac-Man.

2 Uninformed search

2.1 Question 1 - Depth-first search

Implementează algoritmul de căutare pe adâncime (DFS) în funcția 'depthFirstSearch' din fișierul 'search.py'. Algoritmul trebuie să planifice rute în lumea lui Pacman, iar versiunea de căutare pe graf a DFS trebuie să evite expandarea stărilor deja vizitate. Rezultatul trebuie să fie o listă de acțiuni legale care vor ghida agentul de la start la destinație în labirint, fără a trece prin pereți. Testează inițial funcția cu comanda python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch pentru a asigura corectitudinea. Concentrează-te pe DFS și implementarea celorlalte algoritme va fi relativ simplă.

Depth-First Search (DFS) explorează pe adâncime, pornind de la poziția inițială. Algoritmul preferă să exploreze cât mai profund pe o singură ramură înainte de a reveni. Eficient pentru strategii în care se urmărește descoperirea extensivă a unui singur traseu, dar poate întâmpina dificultăți în spații de căutare cu adâncimi mari.

```python
def depthFirstSearch(problem: SearchProblem):
    problem.getSuccessors(problem.getStartState())
    startingNode = problem.getStartState()
    if problem.isGoalState(startingNode):
        return []
    stack = util.Stack()
    visited = []
    stack.push((startingNode, []))
    while stack.isEmpty() == False:
        currentNode, actions = stack.pop()
        if currentNode not in visited:
            visited.append(currentNode)
            if problem.isGoalState(currentNode):
                return actions
            for nextNode, action, cost in problem.getSuccessors(currentNode):
                nextAction = actions + [action]
                stack.push((nextNode, nextAction))
```

2.2 Question 2 - Breadth-first search

Implementează algoritmul de căutare pe lățime (BFS) în funcția 'breadthFirstSearch' din fișierul 'search.py'. La fel ca și în cazul DFS, scrie o versiune de căutare pe graf care evită să extindă stări deja vizitate. Testează codul în același mod ca și în cazul căutării pe adâncime.

Breadth-First Search (BFS) explorează nivel cu nivel, începând de la poziția inițială. Prioritizează explorarea completă a vecinilor înainte de a avansa. Eficient pentru găsirea rutelor scurte și evitarea blocajelor, asigurând o acoperire extinsă a spațiului de căutare în joc.

```
def breadthFirstSearch(problem: SearchProblem):
    startingNode = problem.getStartState()
    if problem.isGoalState(startingNode):
        return []
    que = util.Queue()
    visited = []
    que.push((startingNode, []))

    while que.isEmpty() == False:
        currentNode, actions = que.pop()
        if currentNode not in visited:
            visited.append(currentNode)
            if problem.isGoalState(currentNode):
                return actions
            for nextNode, action, cost in problem.getSuccessors(currentNode):
                nextAction = actions + [action]
                que.push((nextNode, nextAction))
```

2.2.1 UCS

Implementează algoritmul de căutare pe graf cu cost uniform (UCS) în funcția 'uniformCostSearch' din fișierul 'search.py'. Schimbând funcția de cost, putem încuraja Pacman să găsească căi "optime" în alte sensuri. De exemplu, putem taxa mai mult pentru pașii periculoși în zonele cu fantome sau mai puțin pentru pași în zone bogate în hrană. Implementează și testează apoi comportamentul cu succes în toate cele trei aranjamente de mai jos, unde agenții sunt toți agenți UCS care diferă doar în funcția de cost pe care o utilizează (agenții și funcțiile de cost sunt scrise pentru tine):

Uniform Cost Search (UCS) este un algoritm de căutare care prioritizează costurile minime. Explorează succesiv stările cu cele mai mici costuri, optimizând pentru găsirea celor mai eficiente rute. În contextul Pac-Man, UCS poate fi folosit pentru a determina trasee cu costuri minime, fiind util în situații în care optimizarea costului este crucială pentru succesul jocului.

```
def uniformCostSearch(problem: SearchProblem):
    startingNode = problem.getStartState()
    if problem.isGoalState(startingNode):
        return []
    priorityQueue = util.PriorityQueue()
    visited = []
    priorityQueue.push((startingNode, [], 0), 0)

    while priorityQueue.isEmpty() == False:
```

```
        curentNode, actions, cost = priorityQueue.pop()
        if curentNode not in visited:
            visited.append(curentNode)
            if problem.isGoalState(curentNode):
                return actions
            for nextNode, action, currentCost in problem.getSuccessors(curentNode):
                nextAction = actions + [action]
                newCost = cost + currentCost
                priorityQueue.push((nextNode, nextAction, newCost), newCost)
```

3 Informed search

3.1 Questions Project 1

3.1.1 Q5

Implementează problema de căutare CornersProblem în 'searchAgents.py'. Va trebui să alegi o reprezentare a stării care să încorporeze toate informațiile necesare pentru a detecta dacă toate cele patru colțuri au fost atinse.

În cadrul acestei etape a proiectului, am implementat funcțiile "getStartState", "isGoalState" și "getSuccessors" pentru "CornersProblem". Pe scurt, funcția "getStartState" returnează o tuplă formată din poziția de start (coordonate x și y) și o listă goală în care se vor adăuga succesori. Funcția "isGoalState" returnează "True" doar dacă Pac-Man a trecut prin toate cele patru colțuri. Metoda "getSuccessors" generează stări succesoare posibile pentru o stare dată, verificând mișcările posibile în cele patru direcții și actualizând lista de colțuri vizitate, generând astfel stări corespunzătoare..

3.1.2 Q6

Implementează o euristică non-trivială și consistentă pentru problema 'CornersProblem' în funcția 'cornersHeuristic'.

În rezolvarea problemei "CornersHeuristic", am creat o listă de colțuri nevizitate. Pentru fiecare colț, am calculat o euristică Manhattan și am adăugat rezultatul la lista de distanțe până la colțuri. Din această listă, am selectat distanța minimă, deoarece ar reprezenta cel mai optimist scenariu în care Pac-Man ajunge la un colț în acea estimare.

3.1.3 Q7

Implementează problema de căutare a hranei (FoodSearchProblem) în 'searchAgents.py'.

În cadrul întrebării 7, nu am înțeles cum ar trebui să implementez, așa că am returnat 'foodGrid.asList()', iar această abordare a trecut 2 din cele 4 cazuri de testare.

3.1.4 Q8

Implementează funcția 'findPathToClosestDot' în 'searchAgents.py' pentru 'ClosestDotSearchAgent'. Agentul tău trebuie să găsească întotdeauna un traseu care să mănânce cel mai apropiat punct

În rezolvarea întrebării 8, am avut de implementat metodele "isGoalState" și "findPathToClosestDot". Metoda "isGoalState" are o implementare simplă, verificând doar dacă starea

curentă se află sau nu în ‘foodGrid‘. Pentru cealaltă metodă, am apelat fie ‘bfs‘, fie ‘ucs‘ (ambii algoritmi trec testele).

3.2 Question 4 - A* search algorithm

Implementează căutarea pe graf cu A* în funcția ’aStarSearch’ din fișierul ’search.py’. A* primește o funcție heuristică ca argument. Funcțiile heuristice primesc două argumente: un stadiu în problema de căutare (argumentul principal) și problema în sine (pentru informații de referință). Funcția heuristică ’nullHeuristic’ în search.py este un exemplu trivial.

A* Search (A-Star) este un algoritm de căutare care combină eficiența căutării informate cu găsirea celui mai scurt drum. Folosește o funcție de cost și o euristică pentru a evalua nodurile, prioritizând astfel expandarea către soluții promițătoare. În jocul Pac-Man, A* poate fi implementat pentru a găsi rutele optime, luând în considerare atât costurile cât și estimările euristice pentru a ajunge la destinație cu eficiență.

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    priorityQueue = util.PriorityQueue()
    visited = []
    startNode = problem.getStartState()
    priorityQueue.push((startNode, [], 0, 0), 0) #it contains (the node, the actions, the co

    while priorityQueue.isEmpty() == False:
        currentNode, actions, cost, heuristicCost = priorityQueue.pop()
        if currentNode not in visited:
            visited.append(currentNode)
            if problem.isGoalState(currentNode):
                return actions
            for nextNode, action, currentCost in problem.getSuccessors(currentNode):
                nextAction = actions + [action]
                newCost = cost + currentCost
                priorityQueue.push((nextNode, nextAction, newCost, heuristic(nextNode, probl
```

4 Adversarial search

4.1 Question 9 - Improve the ReflexAgent

Implementați funcția de evaluare a agentului reflex în multiAgents.py pentru a juca în mod rezonabil. Agentul ar trebui să ia în considerare atât locațiile alimentelor, cât și locațiile fantomelor pentru a obține performanțe bune.

În timpul implementării algoritmului pentru agentul reflex, am întâmpinat dificultăți în determinarea modului de returnare a scorului optim. În loc să adunăm direct distanța minimă până la hrană, am constatat că este mai eficient să o împărțim la 1. Explicația este că, pe măsură ce Pac-Man se apropie de hrană, valoarea 1/nearestFood crește, reflectând o strategie mai agresivă. În plus, dacă distanța dintre Pac-Man și o fantomă este egală cu 1, funcția va returna 0, semnificând că Pac-Man va muri.