UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Moșilă Luciana, Pinciuc Darius
Grupa: 30236

Facultatea de Automatica
si Calculatoare

03 Decembrie 2023

Cuprins

1 Introducere

În cadrul acestui proiect, agentul Pacman are rolul de a naviga prin labirintul său, având ca obiectiv atât ajungerea într-o anumită locație, cât și colectarea eficientă a alimentelor. Am dezvoltat algoritmi de căutare generali și i-am aplicat în diferite scenarii specifice jocului Pacman.

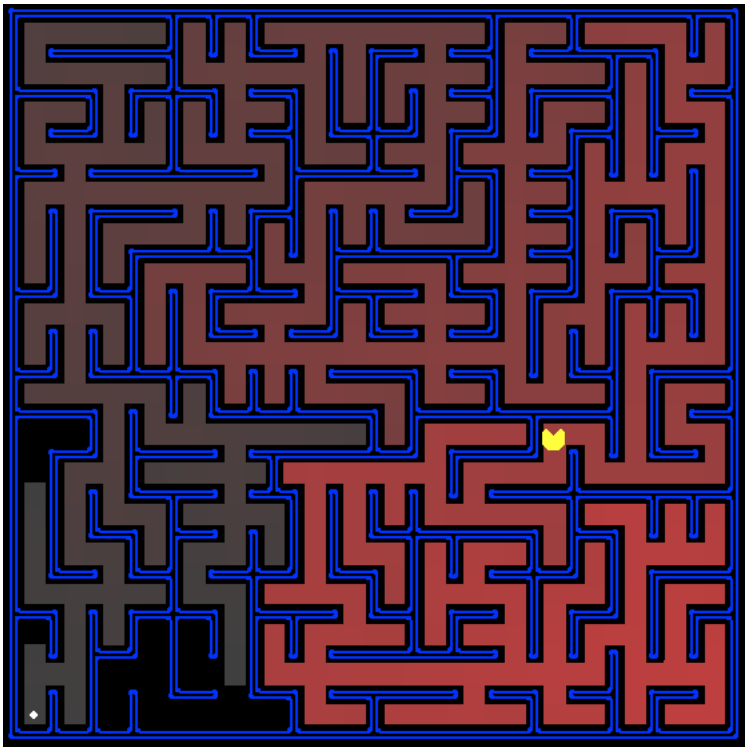

Figura 1: All those colored walls, Mazes give Pacman the blues, So teach him to search.

2 Uninformed search

2.1 Generic Search

Procesul de extindere a nodurilor continuă până când se găsește o soluție sau până când nu mai există stări de extins. Acești algoritimi impart aceeași structură, diferă doar prin alegerea următoarei stări să o extindă.

2.2 Question 1 - Depth-first search

Prezentare algoritm: Se pornește de la un vârf de start dat (v), care este vârful de la care începe căutarea în adâncime. Se creează un stack vid (S), care va fi utilizat pentru a gestiona vârfurile în timpul căutării. Se inițializează un set visited pentru a ține evidența vârfurilor care au fost vizitate sau nu. Inițial, toate vârfurile sunt marcate ca nevizitate. Se introduce vârful de start v în stack (S). Se intră într-un ciclu care continuă până când stack-ul devine gol. În interiorul buclei: Se extrage un vârf (u) din stack. Dacă vârful u nu a fost vizitat, este marcat ca vizitat. Se parcurg toți vecinii nevizitați ai vârfului u. Vecinii nevizitați sunt adăugați în stack pentru a fi explorați ulterior. Algoritmul continuă să extragă și să exploreze vârfuri până când nu mai există vârfuri nevizitate conectate în graf. La final, toate vârfurile accesibile din vârful de start au fost vizitate.

Concluzie: Căutarea în adâncime explorează întotdeauna cel mai indepartat nod din frontiera curentă a arborelui de căutare. Algoritmul de căutare în adâncime este o instanță a algoritmului de căutare în graf în jocul Pacman pentru a ajunge la hrană. Cu toate acestea, nu oferă o soluția optimă, având în vedere costul soluțiilor oferite.

Maze	Total Cost	Nodes Expanded	Score
tinyMaze	10	15	500
mediumMaze	130	146	380
bigMaze	210	390	300

Pseudocod:

```
DFS(G,v) ( v is the vertex where the search starts )
Stack S := {}; ( start with an empty stack )
for each vertex u, set visited[u] := false;
push S, v;
while (S is not empty) do
     u := pop S;
    if (not visited[u]) then
        visited[u] := true;
            for each unvisited neighbour w of u
                push S, w;
    end if
end while
END DFS()
```

2.3 Question 2 - Breadth-first search

Prezentare algoritm: BFS este o strategie simplă în care rădăcina este extinsă prima, apoi toți succesori săi, apoi succesorii succesorilor și tot așa. În general, toate nodurile sunt extinse la o adâncime dată în arborele de căutare înainte ca orice noduri la nivelul următor să fie extinse. În timp ce DFS folosește o coadă LIFO, BFS se realizează foarte simplu prin utilizarea unei cozi FIFO (numită și queue).

Concluzie: În jocul Pacman pentru a ajunge hrană, numărul de noduri expandate este relativ mare, ajungând la 269 și 620 pentru mediumMaze și bigMaze, ceea ce implică o mulțime de timp pentru a găsi soluția optimă.

Maze	Total Cost	Nodes Expanded	Score
tinyMaze	8	15	502
mediumMaze	68	269	442
bigMaze	210	620	300

Pseudocod:

```
function BFS(G, s)
    let Q be a queue
    Q.enqueue(s) // Inserting s into the queue until all its neighbor vertices are marked
    mark s as visited

    while (Q is not empty)
```

```
        v = Q.dequeue() // Removing the vertex from the queue whose neighbors will be visite

        // Processing all the neighbors of v
        for each neighbor w of v in Graph G
            if w is not visited
                Q.enqueue(w) // Stores w in Q to further visit its neighbors
                mark w as visited
```

2.4 Question 3: Varying the Cost Function

Prezentare algoritm: Când toate costurile de pas sunt egale, BFS este optimal deoarece întotdeauna extinde nodul cel mai puțin adânc, care nu a fost explorat încă. Cu toate acestea, în loc să extindă nodul cel mai puțin adânc, UCS extinde nodul cu cel mai mic cost al căii. Acest lucru se realizează prin stocarea frontierei ca o coadă de prioritate ordonată în funcție de cost. În scenariile specifice ale jocului Pac-man, funcțiile de cost ne permit să luăm în considerare pericolele de a fi prins de fantome, precum și șansa de a obține mai multă hrană.

Concluzie: Pentru funcția de cost a căii dată de stayEastAgent, costul corespunzător al căii ar fi destul de mic. Pe de altă parte, costul final pentru calea dată de stayWestAgent este 68719479864, ceea ce este foarte mare.

Maze	Total Cost	Nodes Expanded	Score
mediumDottedMaze	1	186	646
mediumMaze	68	269	442
bigMaze	68719479864	108	418

3 Informed search

3.1 Question 4: A* Search

Prezentare: Una dintre cele mai cunoscute căutări este denumită căutarea A*. Ea evaluează nodurile combinând g(n), costul pentru a ajunge la nod, și h(n), costul pentru a ajunge de la nod la scop: f(n) = g(n) + h(n). Deoarece g(n) reprezintă costul traseului de la nodul ales la nodul n, iar h(n) este costul estimat al celui mai scurt traseu de la n la scop, avem f(n) = costul estimat al celei mai scurte soluții prin nodul n. Astfel, dacă încercăm să găsim soluția cea mai scurtă, un lucru rezonabil de încercat este să începem cu nodul cu cea mai mică valoare a lui g(n) + h(n). Algoritmul este identic cu UCS, cu excepția faptului că A* utilizează g + h în loc de g.

Concluzie: Cu o euristică bună, A* expandează mai puține noduri decât UCS pentru mediumMaze și bigMaze. Pentru openMaze expandează cele mai puține noduri față de ceilalți algoritmi.

Pseudocod:

```
function aStarSearch(problem: SearchProblem, heuristic=nullHeuristic)
    priorityQueue = PriorityQueue()
    startState = problem.getStartState()
    startNode = Node(startState, [], 0)
    priorityQueue.push(startNode, 0 + heuristic(startState, problem))

    visited = set()
```

```
8
9       while not priorityQueue.isEmpty()
10          currentNode = priorityQueue.pop()
11
12          currentState = currentNode.state
13          actions = currentNode.actions
14          cost = currentNode.cost
15          if currentState is in visited
16              continue
17
18          visited.add(currentState)
19
20          if problem.isGoalState(currentState)
21              return actions
22
23          for nextState, nextAction, addedCost in problem.getSuccessors(currentState)
24              if nextState not in visited
25                  nextCost = cost + addedCost
26                  priority = nextCost + heuristic(nextState, problem)
27                  nextNode = Node(nextState, actions + [nextAction], nextCost)
28                  priorityQueue.push(nextNode, priority)
29
30      return []
```

4 Adversarial search

4.1 Question 9 - Improve the ReflexAgent

Funcția de Evaluare (evaluationFunction): Funcția de evaluare este personalizată pentru a reflecta obiectivele specifice ale agentului reflex. În implementarea curentă, se iau în considerare aspecte precum scorul stării succesoare, poziția lui Pacman după mutare, pozițiile mancării rămase și pozițiile fantomelor. Funcția este destinată să ofere un scor numeric, unde valori mai mari sunt considerate mai bune. Dacă o fantomă este la o distanță mai mică de 2 va fi atribuit un scor negativ mare pentru a fi evitată, altfel se calculează scorul pentru mâncare, bazat pe distanța până la aceasta.

Utilizare: Agenții reflex sunt utilizați pentru a modela strategii de luare a deciziilor în medii complexe, cum ar fi jocurile video. În contextul jocului Pacman, ReflexAgent își propune să aleagă acțiuni optime pentru a maximiza scorul și a evita obstacolele, cum ar fi fantomele.

4.2 Question 10 - Minimax

Algoritmul Minimax este o metodă de luare a deciziilor utilizată în jocurile cu doi jucători, în care un jucător încearcă să maximizeze beneficiile, iar celălalt să minimizeze pierderile. În cazul agentului Pacman, acest algoritm își propune să optimizeze strategia de mișcare pentru a obține cel mai bun rezultat în fața fantomelor.

Metodele cheie:

- getAction(gameState): Această metodă reprezintă punctul de intrare în algoritm. Ea returnează acțiunea optimă pentru starea de joc dată. Apelează metoda maxval pentru a începe explorarea arborelui de stări posibile.
- minimax(gameState, agentIndex, depth): Această funcție implementează algoritmul Minimax, evaluând și selectând acțiunile care duc la maximizarea sau minimizarea costului, în funcție de agentul curent și adâncimea curentă în arborele de căutare.
- maxval(gameState, agentIndex, depth): Această metodă reprezintă partea de maximizare a algoritmului, unde Pacman încearcă să aleagă acțiunile care maximizează utilitatea, adică costul, în starea de joc dată.
- minval(gameState, agentIndex, depth): Această metodă reprezintă partea de minimizare a algoritmului, în care ghost-urile încearcă să aleagă acțiunile care minimizează utilitatea, adică costul, pentru a împiedica Pacman să maximizeze scorul.

Condiții de Oprire: Algoritmul Minimax se oprește atunci când a atins adâncimea maximă specificată (self.depth) sau când starea de joc indică o stare de câștig sau pierdere. În aceste cazuri, se întoarce la nivelurile superioare ale arborelui de căutare.

Funcție de Evaluare: În implementarea curentă, se utilizează o funcție de evaluare (self.evaluationFunction) pentru a estima utilitatea unei stări de joc. Aceasta poate fi personalizată pentru a ajusta comportamentul agentului în funcție de obiectivele specifice ale jocului.

Utilizare: Agentul Minimax este integrat în jocul Pacman pentru a lua decizii strategice privind mișcarea în medii complexe și pentru a obține cel mai bun rezultat posibil în fața adversarilor.

4.3 Question 11 - Alpha-Beta Prunning

Algoritmul Alfa-Beta Pruning reprezintă o îmbunătățire a algoritmului Minimax, utilizat în problemele de luare a deciziilor în jocurile cu doi jucători, cum ar fi jocul de șah sau de dame. Scopul acestui algoritm este de a reduce numărul de stări ale jocului evaluate, menținând aceeași calitate a deciziilor luate de către algoritm.

Minimizarea Explorării Inutile: Algoritmul Minimax explorează toate stările posibile ale jocului, ceea ce poate duce la un număr mare de evaluări. Alfa-Beta Pruning optimizează acest proces prin eliminarea explorării inutile ale unor ramuri ale arborelui de joc care nu ar afecta decizia finală.

Alfa și Beta: Algoritmul utilizează două valori, alfa și beta, pentru a ține evidența a două intervale care definesc intervalul optim de valori pentru o anumită stare a jocului. Alfa reprezintă valoarea minimă pe care jucătorul MAX o poate obține, în timp ce beta reprezintă valoarea maximă pe care jucătorul MIN o poate obține.

Prelucrarea Pruning: În timp ce algoritmul explorează stările jocului, se actualizează valorile alfa și beta pentru a restrânge intervalul de căutare. Atunci când valoarea alfa devine mai mare sau egală cu beta într-un nod MAX sau valoarea beta devine mai mică sau egală cu alfa într-un nod MIN, explorarea ulterioară a acelei ramuri poate fi eliminată, deoarece nu va afecta decizia finală.

Pseudocod:

```
function minimax(position, depth, alpha, beta, maximizingPlayer)
        if depth == 0 or game over in position
                return static evaluation of position

        if maximizingPlayer
```

```
6                 maxEval = -infinity
7                 for each child of position
8                         eval = minimax(child, depth - 1, alpha, beta false)
9                         maxEval = max(maxEval, eval)
10                        alpha = max(alpha, eval)
11                        if beta <= alpha
12                                break
13                return maxEval
14
15        else
16                minEval = +infinity
17                for each child of position
18                        eval = minimax(child, depth - 1, alpha, beta true)
19                        minEval = min(minEval, eval)
20                        beta = min(beta, eval)
21                        if beta <= alpha
22                                break
23                return minEval
```