UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autor: Laura-Luisa Voicu
Grupa: 30236

Facultatea de Automatica
si Calculatoare

4 Decembrie 2023

Cuprins

1 Introducere

1.1 Despre Proiect

Proiectul consta in implementarea unor algoritmi de cautare pentru asigurarea castigarii jocului PAC-MAN. Jocul consta intr-un personaj ce se deplaseaza intr-un labirint cu scopul a manca "puncte". De mentionat faptul ca labirinturile nu sunt aceleasi, configuratia acestora fiind diversificata prin prezenta unor ziduri. Dificultatea jocului este data atat de gasirea drumului optim printre ziduri, cat si de ocolirea agentilor antagonisti.

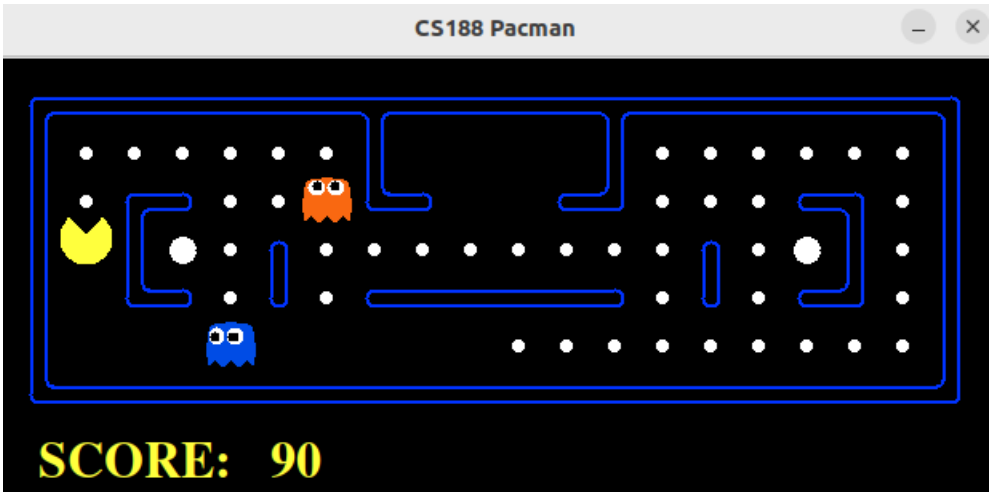


Figura 1: Exemplu PAC-MAN

1.1.1 Structura proiectului

Proiectul este alcatuit din doua parti:

- Partea I: Uninformed and Informed Search
- Partea a II a : Multi-Agent Search

Fiecare dintre aceste parti sunt alcatuite dintr-un numar variabil de exercitii, numite Questions. Astfel ca:

- Pentru prima parte s-au implementat **Questions 1 - 8**
- Pentru cea de-a doua parte s-au implementat **Questions 1 - 3**

Cele doua parti sunt implementate in 2 proiecte diferite, ce se pot gasi in arhivele **search.zip** si **multiagent.zip**.

1.1.2 Agenti - Stari - Frontiere - Labirinte

Agenti : entitati ce au capacitatea de a lua decizii si a realiza o actiune in functie acestea. Un agent poate fi controlat prin intermediul unor algoritmi ce vor fi prezentati ulerior.

Stari : Un agent se poate afla in 5 stari (ca rezultat al decizilor luate):

- North
- South
- West
- East
- Stop

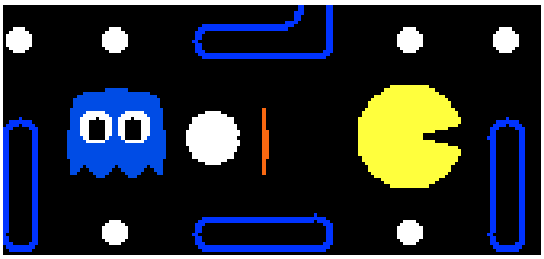


Figura 2: Agents

Frontiere : Structuri de date in care se vor stoca nodurile explorate.

- Stack - DFS
- Queue - DFS
- Priority Queue - UCS and A*

Observatie: Acestea sunt deja implementate in **util.py**

Tipuri de labirint, how to run it:

$ python3 pacman.py -l tinyMaze -p SearchAgent

$ python3 pacman.py -l mediumMaze -p SearchAgent

$ python3 pacman.py -l bigMaze -z .5 -p SearchAgent

1.2 Instructiuni de rulare

In arhivele originale, cele download-ate direct de pe site-ul `https://inst.eecs.berkeley.edu/~cs188/`, proiectul este insotit de un fisier text in care sunt prezentate toate instructiunile de rulare. Acestea sunt atasate si in documentatie: Search Run Instructions.

In cadrul proiectului se pot gasi teste ce pot fi rulate manual, dar si un **autograde** ce calculeaza numarul de teste trecute.

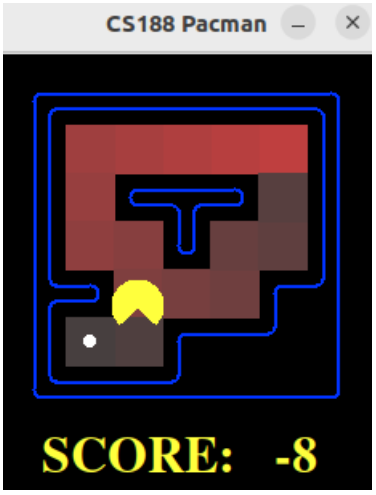
* Pentru fiecare Question se vor prezenta individual instructiunile necesare pentru rulare si verificare a cerintei respective, atat prin varianta grafica, cat si prin intermediul autograde-ului;

1.3 Uninformed Search I: 1.Depth-first search: Descoperirea unui Food Dot fixa

1.3.1 Cerinta. Prezentare algoritm

Depth First Search este o **strategie de cautare neinformata**, prin faptul ca nu exista informatii aditionale despre starile altor agenti (Ghost,etc.), decat cei definiti in problema (Pac-Man).

Acesta se bazeaza explorarea in adancime a fiecarei ramuri inainte de a se intoarce la pasii anteriori. Ca structura de date pentru stocarea nodurilor ce urmeaza sa fie exlorate se foloseste **stiva**. Odata ce nodul este explorat in totalitate, acesta este scos din stiva, urmand sa inceapa explorarea urmatorului nod, conform principiului LIFO. Se repeta procedeul pana la **descoperirea nodului ce reprezinta Goal State**(Food Dot in acest caz).



1.3.2 Cod

1.3.3 Instructiuni de rulare

python3 pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
python3 autograder.py -q q1

1.3.4 Complexitate

DFS complexity: O(V + E)
unde: V = numar de varfuri; E = numar de muchii

1.3.5 Completeness

DFS e complet doar daca arborele general e complet (daca spatiul de stari nu are cicluri)

1.3.6 Optimal

DFS nu e optim.

1.3.7 Probleme intampinate

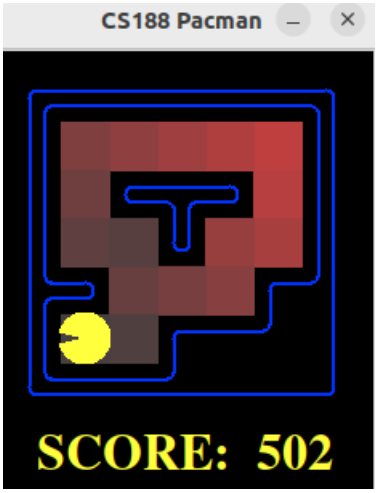
Una din problemele intalnite la acest Exercitiu consta in adaptarea algoritmului de DFS la template-ul dat. De asemenea, optarea pentru folosirea efectiva a unei stive, si nu cea generata de apelurile recursive (intrucat la final era necesara returnarea unei actiuni, iar folosirea unui algoritm DFS recursiv ar fi putut complica acest pas), poate fi considerat un mic challange.

1.4 Uninformed Search I: 2.Breadth-First Search: Descoperirea unui Food Dot fix

1.4.1 Cerinta. Prezentare algoritm

Breadth First Search se realizeaza prin explorarea tuturor nodurilor de pe un nivel inainte de trecerea la un alt nivel al grafului.

Ca si structura de date ce stocheaza nodurile explorate se va folosi **o coada**.



1.4.2 Cod

1.4.3 Instructiuni de rulare

python3 pacman.py -l tinyMaze -p SearchAgent -a fn=bfs
python3 autograder.py -q q2

1.4.4 Complexitate

BFS complexity: O(V + E)
unde: V = numar de varfuri; E = numar de muchii

1.4.5 Completeness

BFS e complet daca arboerele e finit (si e finit in aceste circumstante).

1.4.6 Optimal

BFS e optim doar cand actiunile nu au cost!

1.4.7 Probleme intampinate

Daca se doreste imbunatatirea algoritmului de BFS (asta consta in faptul ca atunci cand pentru starea X, unul dintre succesorii sai represinta GoalState, ne vom opri si vom returna actiunea ce ne va duce in acea stare, fara explorarea efectiva a acesteia),nu se va trece de testele generate de autograder, intrucat acesta compara si "expanded-states" la care nu am acces direct; Exemplu mai jos:

Imbunatatire:

if problem.isGoalState(successor):
return actions-copy

Eroare:

*** student solution: ['1:A-¿C', '0:C-¿D', '1:D-¿F', '0:F-¿G']
*** student expanded-states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F']
*** correct solution: ['1:A-¿C', '0:C-¿D', '1:D-¿F', '0:F-¿G']
*** correct expanded-states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

1.5 Uninformed Search I: 3.Uniform Cost Search - Determinarea drumului ponderat catre un Food Dot fix

1.5.1 Cerinta. Prezentare algoritm

Uniform cost search este o adaptare a algoritmului de cautare Dijkstra, imbunatatit prin faptul sa nu se va realiza initializarea initiala a ponderilor cu care se poate ajunge la un nod la o valoare INF.

Acesta consta in explorarea nodurilor pentru care cost path-ul este minim. Din acest motiv, ca structura de date pentru stocarea nodurilor explorate se va folosi in Priority Queue, ce va pastra ca prim element, nodul ce are asociata cea mai mica pondere.

1.5.2 Cod

1.5.3 Instructiuni de rulare

python3 pacman.py -l tinyMaze -p SearchAgent -a fn=ucs
python3 autograder.py -q q3

1.5.4 Complexitate

UCF complexity: O(b$\hat{}$(1+C/e))

unde: b = branch factor (1 nod la nivelul 0, b noduri la nivelul 1, b$\hat{2}$ noduri la nivelul 2 s.a.); C = costul solutei optimale; e – fiecare pas te aduce de e ori mai aproape; Explicatie detaliata aici: `https://stackoverflow.com/questions/19204682/time-complexity-of-uniform-cost-search`

1.5.5 Completeness

UCF e complet doar daca solutia optima are un cost finit.

1.5.6 Optimal

UCS e optim.

1.5.7 Probleme intampinate

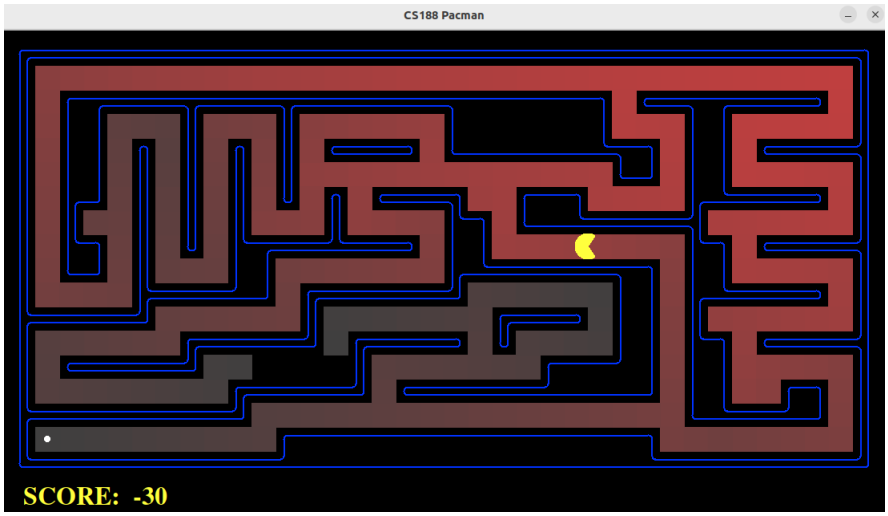
Spre deosebire de BFS si DFS, unde explored-nodes era dat de o lista, la UCS nodurile explorate/visitate trebuie tinute intr-un set, unde key = nod, value = cost (acesta asigura si unicitatea in functie de cheie), intrucat e nevoie de cunoasterea costului pentru viitoarele iteratii, el putand fi imbunatatit.

1.6 Informed Search I: 4.A Star - Determinarea drumului ponderat catre un Food Dot fix

1.6.1 Cerinta. Prezentare algoritm

Algoritmul de cautare A* reprezinta o optimizare a algoritmului UCS. Astfel, acesta este tot o adaptare a algoritmului Dijkstra, aducand in plus o euristica h ce estimeaza distanta intre doua puncte.

Formula dupa care se bazeaza calculul costului este **f(n) = g(n) + h(n)**, unde g(n) reprezinta costul pentru a ajunge la nodul n, si h(n) costul necesar de a ajunge la nodul n, la GoalState.



1.6.2 Cod

1.6.3 Instructiuni de rulare

python3 pacman.py -l tinyMaze -p SearchAgent -a fn=astar
python3 autograder.py -q q4

1.6.4 Complexitate

UCF complexity: O($b\hat{d}$) (worst case), dar in principiu depinde de euristica aleasa

unde: b = branch factor (1 nod la nivelul 0, b noduri la nivelul 1, $b\hat{2}$ noduri la nivelul 2 s.a.); d = depth of goal state

1.6.5 Completeness

AStar e complet.

1.6.6 Optimal

AStar e optim.

1.6.7 Probleme intampinate

Implementarea e aproape identica cu cea pentru UCS, singurele diferente fiind date de calculul costului tinand cont de euristica, si verificand pentru toate nodurile explorate daca prin intermediul state-ul curent pot obtine un cost mai bun.

1.7 Informed Search I: 5.Finding All the Corners aka define new problem

1.7.1 Cerinta. Prezentare abordarii

Problema ce trebuie implementata consta determinarea drumului optim catre cele 4 colturi ale labirintului, daca acestea sunt disponibile (adica la acele pozitii nu se afla un zid), indiferent daca acolo exista sau nu mancare.

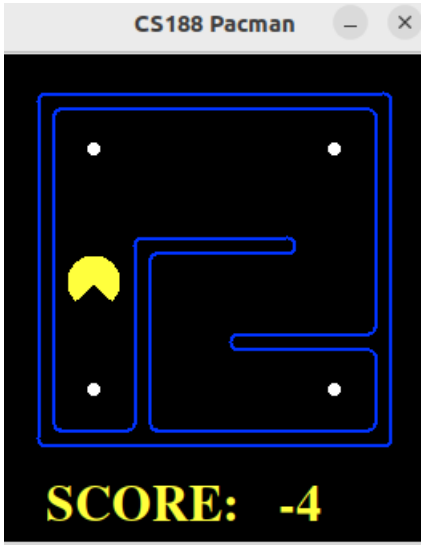
In codul initial se vor da informatii despre pozitile zidurilor, pozitia initiala a agentului PAC-MAN si definirea colturilor.

Informatii aditionale aduse de mine:

- available-corners: salveaza doar acele colturi la care se poate ajunge
- visited-corners: lista in care se vor salva toate colturile vizitate; observatie - initial se vor verifica inclusiv colturile, pac-man poate avea ca start-state chiar unul dintre ele
- start-state: o tupla formata din starea curenta si colturile vizitate pana la starea curenta

De ce sa retinem visited-corners ca si stare?

Problema va fi folosita in algoritmii de cautare implementati anterior in functie de algoritmul ales, putem sa ajungem in diverse stari la care s-a ajuns anterior (de exemplu A star re-verifica nodurile deja explorate) prin urmare nu putem avea control asupra colturile vizitate ale nodurilor deja explorate, doar daca retinem efectiv aceasta informatie ca stare



1.7.2 Cod

1.7.3 Instructiuni de rulare

python3 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python3 autograder.py -q q5

1.7.4 Probleme intampinate

Identificarea starilor. Trebuia tinut cont si de faptul ca pe problema respetiva va fi aplicata algoritmi de cautare, deci prin urmare era necesara retinerea colturilor vizitate intr-o anumita stare.

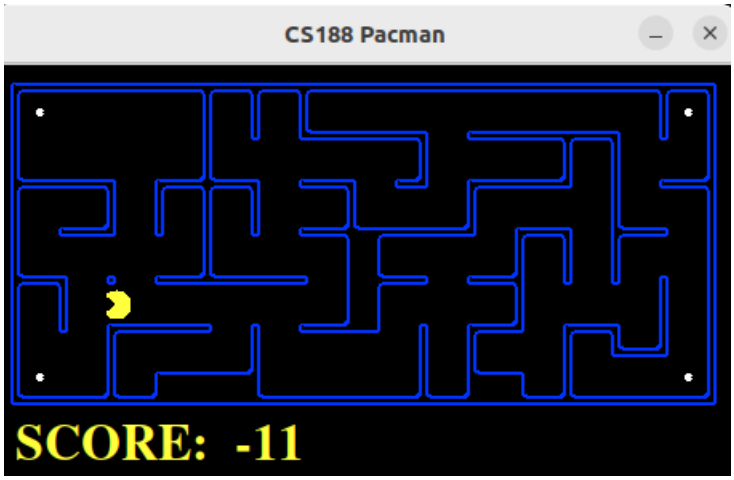
O varianta gresita la care m-am gandit a fost sa retin la ce colturi a ajuns agentul intr-o variabila globala, ceea ce ar fi facut ca pentru fiecare stare sa se fi retinut aceleasi valori.

1.8 Informed Search I: 6.Corners Problem: Heuristic

1.8.1 Cerinta. Prezentare abordarii

Good to know:

- Heuristic: functie ce ia ca input o stare de cautare si returneaza costul estimativ catre goal.
- Admissible Heuristic: valorile euristicii trebuie sa fie Lower Bound pentru adevaratul cost al path-ului catre Goal.
  Observatie: admisibilitatea nu e suficienta pentru a garanta corectitudine, DAR euristicile asigura in general **consistenta**
- Trivial Heuristics: returneaza 0 mereu, calculeaza adevaratul cost
- Non-Trivial Heuristics: euristici ce reduc timpul total de calcul (sau care reduc numarul de noduri expandate)



1.8.2 Cod

1.8.3 Instructiuni de rulare

python3 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python3 autograder.py -q q6

1.8.4 Probleme intampinate

1.9 Project I: 7.Eating All The Dots

1.9.1 Cerinta. Prezentare abordarii

Problema consta in acumularea tuturor Food dots-urilor in cat mai putini pasi posibili.

Problema se rezuma in a gasi cel mai scurt drum catre in Food Dot. Acest lucru se poate realiza prin algoritmii de cautare implementati anterior, cu mentiunile:

- A* va gasi mereu drumul cel mai scurt DACA are o euristica corespunzatoare
- UCS gaseste drumul cel mai scurt dar poate fi ineficient
  DFS,BFS nu cauta drumul cel mai scurt,doar extind noduri pana la gasirea unuia (sau tuturor) – nu se bazeaza pe euristici

1.9.2 Cod

1.9.3 Instructiuni de rulare

python3 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python3 autograder.py -q q7

1.9.4 Probleme intampinate

Alegerea unui algortim de cautare ce indeplineste conditia.

1.10 Project I: 8. Suboptimal Search

1.10.1 Cerinta. Prezentare abordarii

Scrierea unui agent ce mananca Food Dot-ul ca urmare a tehnicii Greedy

1.10.2 Cod

1.10.3 Instructiuni de rulare

python3 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
python3 autograder.py -q q8

1.10.4 Probleme intampinate

Alegerea unui algortim de cautare ce indeplineste conditia.

2 Project II - MultiAgent

2.1 Multiagent - ReflexAgent

2.1.1 Cerinta. Prezentare abordarii

Abordarea unei solutii astfel incat Food Dots sa fie mancate fara a fi omorat de Ghost Agent.

Caut distanta cea mai mica catre un Food Dot. Daca dupa mutarea lui PacMan in acea pozitie, pozitia unei fantome va coincide cu aceasta, jocul e pierdut.

2.1.2 Cod

2.1.3 Instructiuni de rulare

python3 pacman.py –frameTime 0.1 -p ReflexAgent -k 3
python3 autograder.py -q q1 –no-graphics

2.1.4 Probleme intampinate

Alegerea unui abordari optime

2.2 Multiagent - Minimax

2.2.1 Cerinta. Prezentare abordarii

Implementarea algoritmului minimax - consta in alegerea celei mai bune optiuni pentru (MAX) pentru PAC-MAN si celei mai gresite optiuni pentru (MIN) pentru Ghost.

Un nivel de adancime este format dintr-o decizie luata pentru PAC-MAN si o decizie luata pentru fiecare Ghost Agent.

Arborele de dezicie se compune TOP-DOWN si se evalueaza BOTTOM-UP.

2.2.2 Cod

2.2.3 Instructiuni de rulare

python3 pacman.py –frameTime 0.1 -p ReflexAgent -k 3
python3 autograder.py -q q1 –no-graphics

2.2.4 Probleme intampinate

Intelegerea agoritmului si adaptarea lui pentru depth=n

2.3 Multiagent - Alpha-Beta Pruning

2.3.1 Cerinta. Prezentare abordarii

Optimizare a algoritmului de Minimax. Prin intermediul acestui algoritm nu mai e necesara parcurgerea tuturor ramurilor din arborele de decizie.

Daca se obtine pentru min-value o valoare mai mare decat valoarea curenta a unui ghost-agent (ce determina alt min-value) , cel din urma nu va mai fi expandat, intrucat se urmareste obtinerea celui mai mare dintre min-values, ce va fi comparat cu max-value pentru pac-man, respectiv a celei mai mici valori pentru max-values (max-pacman)

2.3.2 Cod

2.3.3 Probleme intampinate

Intelegerea agoritmului si adaptarea lui pentru depth=n

2.3.4 Instructiuni de rulare

python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
python3 autograder.py -q q3 –no-graphics

3 Project I : Search Instructions

python pacman.py
python pacman.py –layout testMaze –pacman GoWestAgent
python pacman.py –layout tinyMaze –pacman GoWestAgent
python pacman.py -h
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
python eightpuzzle.py
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
python pacman.py -l testSearch -p AStarFoodSearchAgent
python pacman.py -l trickySearch -p AStarFoodSearchAgent
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5