**Echipa:** Albu Antonia, Crisan Paul

**Inteligenta artificiala**

Proiect PACMAN

1.Obiectivul lucrarii

In acest proiect, se intentioneaza optimizarea jocului de Pacman, in asa fel incat acesta sa ajunga la punctele de mancare intr-un mod eficient. Se mentioneaza faptul ca in labirintul ales drept mediu pentru exploatare a Pacman-ului, nu exista fantome.

Ca si algoritmi de cautare, atentia se va pune asupra:

* Depth First Search (DFS)
* Breadth First Search (BFS)
* Uniform Cost Search (UCS)
* A\* (aStar Search)

2. Descrierea algoritmilor

Pentru inceput, vom incepe cu prezentarea primului algoritm, si anume: DFS

**DFS (cauatre in adancime):**

Descriere generala:

* Ideea de baza a acestui algoritm consta in alegerea arbitarar a unui nod de start(pe care il numim si radacina), nod de la care vom porni apoi parcurgerea arboreului/grafului, mergand cat mai departe posibil pe aceeasi ramura, inainte de a face backtracking.

**BFS (cautare in latime):**

Descriere generala:

* La fel ca si la DFS, vom avea nevoie de un nod radacina, de la care sa incepem cautarea intr-un arbore/graf, cautare ce se va desfasura astfel: se incepe de la nodul de start(radacina) si apoi se exploateaza nodurile vecine acestuia, si doar apoi se trece la vecinii de pe nivelul urmator. De obicei, pentru o parcurgere in latime, vom avea nevoie de o structura suplimentara de tip coada (queue), pentru a memora nodurile copil care au fost intampinate pe parcurs, dar nu si explorate.

**UCS (uniform cost search)**

Descriere generala:

* UCS este o imbunatatire a algoritmului lui Dijkstra, doar ca, in loc de a insera toate nodurile intr-o coada de prioritati, vom insera doar nodul sursa, iar apoi, vom insera nodurile de care vom avea nevoie, unul cate unul. La fiecare pas, vom verifica daca valoarea exista deja in priority queue, folosing un array in care sa stocam nodurile vizitate. Daca nodul se afla in array, atunci vom scadea cheia, altfel, il vom insera.
* Aceasta varianta algoritmului lui Dijkstra este folositoare pentru grafurile inifnite, sau a grafurilor de dimensiuni prea mari pentru a fi reprezentate in memorie.

**A\* (aStar Search)**

Descriere generala:

* La fel ca si UCS, A\* este tot o imbunatatire a algorimtmului lui Dijkstra, insa pentru a putea aplica algoritmul, avem nevoie de o euristica.
* Pentru a intelege cel mai bine modul in care functioneaza acest algoritm, este sa ne imaginam ca avem o tabla dreptunghiulara (sau particular, un patrat), iar scopul nostru este de a ajunge intr-un mod eficient si rapid dintr- casuta de inceput, intr-o casuta destinatie. De mentionat este ca, in acest spatiu in care avem nevoie sa ne deplasam, pot exista obstacole(precum pereti).

Cum functioneaza A\*?

* La fiecare pas, se va alege cate un nod, in functie de valoare lui **f** (parametru egal cu suma a altor 2 parametrii: g si h)
* g -> costul necesar deplasarii din spatiul de start pana intr-un spatiu al patratului
* h -> costul estimativ necesar mutarii din acel nou spatiu pana in spatiul destinatie

h este de obicei cunoscut sub numele de euristica, ceea ce nu este altceva decat un mod d de a “ghici” distanta pana la destinatie.

Avem o multime de posibiliatati de a calcula eusristica (strict in proiectul nostru, ne vom folosi de nullHeuristic sau de ManhattanHeuristic).

Ca si implementare, facand abstractie de existenta unei euristici, algoritmul este similar cu UCS.

3. Prezentarea algoritmilor in cadrul Pacman

In urma prezentarii ideii fiecarui algoritm de implementat, se poate incepe o descriere particulara pentru a fiecaruia din acestia in cadrul jocului de Pacman:

* DFS

Pentru a stoca nodurile pe care le avem de visitat, ne vom folosi de o structura de tip STIVA, iar aceasta structura de acum incolo in cadrul DFS se va numi frontiera.

Dupa cum a fost prezentat si in capitolul 2, avem nevoie de un nod de start, in cazul nostru,

startState. De asemenea, pentru memorarea nodului curent, ne vom folosi de variabila currentNode (momentan acesta fiind nodul de start), variabila pe care o vom adauga in stiva. Pentru a putea tine urma nodurilor explorate, vom crea o lista exploredNodes, unde vom adauga toate nodurile deja explorate.

In urma acestor initializari, vom continua cu parcugerea grafului, dupa cum urmeaza:

1. Vom verifica daca frontiera nu este goala (daca este goala, nu am avea ce noduri sa exploram)
2. Daca nu este goala, vom incepe prin a explora nodul din coada frontierei (adica cel mai recent adaugat) si il vom salva in variabila currentState.
3. Vom verifica daca acest nod se afla in lista nodurilor explorate. Daca nu, atunci:

* Il vom adauga in lista
* Vom verifica daca nodul tocmai adaugat este nodul nostru destinatie (adica goalState)
* Daca da, cautarea s-a terminat si vom returna calea pana la acest nod
* **Daca nu:**

1. – intr-o variabila numita succesors, vom stoca toti succesorii nodului curent explorat, sub forma (succesor, action, stepCost), iar pe fiecare din acesti succesori, ii vom pune in frontiera.
2. Vom lua apoi cel mai recent nod adaugat in frontiera, si vom realiza toti acesti pasi, pana cand frontiera este goala.
3. Returam calea

* BFS

Ca si implementare, BFS este asementaror cu DFS, insa, in cadrul acestui algoritm, frontiera nu va mai fi o stiva, ci o COADA, din pricina faptului ca nu vom mai avea sa exploram cele mai recent adaugate noduri in frontiera, ci cele mai vechi.

De asemenea, in cadrul DFS, aveam nevoie sa stocam in cadrul nodului curent, doar starea, si actiunea, insa, in cadrul BFS, vom avea nevoie sa stocam si parintele si costul (conform prezentarii din capitolul 2).

De observat este faptul ca nodul curent, se scoate din frontiera imediat dupa ce a fost verificat continutul acestuia, si nu la finalul verificarii, ca si in cadrul DFS.

* UCS

In continuare, se prezenta algoritmul UCS, algoritm asemanator ca si implementare cu BFS si DFS, avand urmatoarele diferente (pe langa cele prezentate in capitolul 2):

* Frontiera este reprezentata de o coada de prioritati
* Nodurile explorate nu mai sunt memorate ca si o lista, ci ca si un dictionar, care memoreaza starea si costul starii respective. Aceasta metoda de stocare este folosita in acest caz pentru a vedea daca in graful nostru exista cicluri. De asemena, nodul curent este retinut drept nod ce are stare, actiune, parinte, cost.
* Algoritmul incepe prin a explora nodul cu cel mai mic cost din coada de prioritati, urmand sa se faca aceleasi verificari asupra nodului ca si la BFS, cu mentiunea ca:

Verificarea se face fie asupra apartenentei nodului in dicitonarul nodurilor explorate, sau:

Asupra conditiei costului, adica, nodul curent explorat sa aiba un cost mai mic decat a celor explorate deja.

* A\*

Dupa cum am prezentat anterior, A\* este o imbunatatire a algoritmului Dijkstra, si implicit si a algoritmului UCS, astfel ca:  
- frontiera va fi tot o coada de prioritati

* Nodurile se vor pastra intr-o lista
* Nodul curent se va pastra intr-o structura identica cu cea din UCS sau BFS
* In frontiera vom pune si nodul curent, dar si un cost initial (0)
* Fata de UCS, A\* face si verificari asupra succesorilor, pentru a aduce o imbunatatire din punct de vedere al eficientei, asadar:  
  - in loc sa punem un nou nod in frontiera dupa “calcularea lui” ii vom verifica succesori, astfel:

Daca succesorul nu a fost inca explorat, doar atunci il vom pune in frontiera.

Verificarea se face in modul urmator:  
 - Vom lua toate nodurile deja explorate din lista de noduri explorate.

- Vom lua starea si costul fiecarui nod mai sus mentionat, si il vom compara cu starea si codul succeorului curent. Daca starea succesorului este egala cu starea nodului curent, iar noul cost calculat pentru a ajunge la succesor este mai mare sau egala cu costul nodului explorat, atunci succesorul a fost explorat.

- Daca succesorul nu a fost explorat, doar atunci il vom pune in frontiera, si nodul il vom adauga in lista nodurilor explorate, cu ajutorul unei eurisitici.

Pe langa euristica euclidiana si cea Manhattan, vom prezenta si urmatoarele:

**CornersHeuristic:**

In primul rand, ne vom axa pe clasa CornersProblem, unde avem definit urmatoarele functii:

1. Init -> functie pentru a initiliaza pozitia de start, zidurile, top, right, colturile grid-ului, dar si pozitia initiala a Pacman-ului(vom spune ca un colt este 0 daca nu a fost vizitat, si 1 daca a fost).
2. getStartState -> returneaza punctul de inceput, dar doar din spatiul starilor initiale(initialState)
3. isGoalState -> functie care ne spune daca am ajuns la goal-ul nostru, verificand fiecare colt din spatiul starilor (prin verificarea cu 0 sau 1 din initialState)
4. getSuccesors -> aceasta functie ne returneaza starea, actiunea si un cost de 1.

In final, vom prezenta euristica mentionata mai sus:

Vom lua colturile, si peretii labirintului nostru.

Daca am starea la care suntem este goal, atunci vom retuna 0, asta insemannd ca Pacman-ul se afla din start intr-unul din colturile labirintului.

Daca nu, atunci vom pastra distanta pana la toate colturile nevizitate.

Vom lua fiecare index si colt al starii, pentru a ne permite sa vedem daca coltul respectiv a fost vizitat, iar in cazul in care nu a fost, il vom adauga in lista mentionata mai sus, cu ajutorul euristicii Manhattan.

Pentru ca euristica sa fie valida, va fi nevoie sa returnanam coltul aflat la distanta maxima fata de Pacman.

**FoodHeuristic**:

Pentru aceasta euristica, ne vom folosi de clasa AstarSearchFoodAgent.

Aceasta euristica ne ajuta sa ajungem la mancare in cel mai eficient mod posibil, iar ideea de baza este asemanatoare cu cea de la Cornersheusristic. Practic, vom incepe tot prin a merge la cea mai indepartata mancare. Ne vom folosi tot de o lista in care sa pastram distanta pana la manacre, in care vom adauga mancarea folisndu-ne de eusristica Manhattan. Pentru ca eusristica sa functioneze corect, va trebui sa returnam mancarea care este cea mai indepartata de Pacman.

De asemnea, pentru a putea sa gasim cel mai apropiat punct de mancare fata de pozitia actuala a Pacman-ului, ne vom folosi de bfs in clasa ClosestDotSearchAgent.

Pentru formarea unui nod curent, ne vom folosi de clasa Node, care va mentine urmatoarele atribute:

* Stare
* Actiune
* Parinte
* Cost

De asemnea, pentru a putea returna calea parcursa de Pacman, ne vom folosi de functia getPath, care va adauga intr-o lista path fiecare nod care indeplineste conditiile necesare, si apoi o returneaza.

Toate aceste modificari au fost facute in clasa **search.py**

**Adversarial Search**

In continuare, vom prezenta modul in care a fost implementata cautarea adversariala, in clasa multiAgents.py.

Pentru acest capitol, vom descrie clasele MinimaxAgent si AlphaBetaAgent.

MinimaxAgent

Aceasta clasa are la baza algorimul minimax, descris de functia cu acelasi nume. Acest algoritm are nevoie de pozitia agentului, de agentul curent, dar si de adancimea arboreului asupra caruia va fi aplicat algorimtul.

In primul rand, algoritmul va fi aplicat doar daca jocul nu este finalizat (adica a fost pierdut sau castigat) sau avem o adancime a arborelui). Daca se poate aplica algoritmul, atunci vom avea 2 cazuri:

1. Cazul in care Pacman va realiza o miscare
2. Cazul in care fantoma va realiza o miscare

Randul lui Pacman:

In cadrul acestui algoritm, il vom considera pe Pacman avand corespondenta in arbore drept max, asadar, il vom initiliza la inceput cea mai mica valoare posibila (-infinit), dupa care ii vom lua in considerare toate posibilelel mutari (N, S, E, W), si apoi le vom verifica pe rand, astfel:

* Vom explora fiecare pozitie posibila din jurul Pacman-ului (adica vom lua starea succesrilor sai, luand in considerare tipul de agent si actiunea pe care o poate realiza.
* Dupa aceasta initializare, insemana ca s-a facut o mutare de catre Pacman, si va urma randul primei fantome(lucru pe care il vom face apeland functia minimax, insa pe agentul urmator), si vom stoca rezultatul in variabila newMax.
* Apoi, ca si Ia orice verificare de maxim, vom face o comparatie intre noul maxim si vechiul maxim, dar vom stoca si actiunea facuta de agentul caruia ii corespunde maximul.
* In final, vom returna maximul si actiunea.

Randul fantomei

* Fantomele au numarul agentului corespunzator 1.
* Ele au corespondent in arborele minimax valorile minime, asadar, vom initializa minimul cu o valoare cat mai mare (infinit). De asemenea, vom avea nevoie si de numarul fantomelor, acesta fiind dat de numarul agentilor din joc – 1(Pacman-ul).
  + In continuare, vom vedea agentul la care ne aflam, pentru a sti stadiul jocului:  
    -> daca agentul este egal cu numarul fantomelor, insemna ca am ajuns la ultima fantoma, deci vom scadea adancimea arboreului cu 1, si vom da “controlul” Pacman-ului, altfel, vom trece la urmatoarea fantoma.
  + Dupa acestea, algorimtul va lua actiunile posibile ale fantomelor, si va face aceleasi verificari ca in cadrul maximului(dar cu valorile minime).
  + De mentinoat este faptul ca, la apelul functiei minimax, aceasta este apelata cu indexul 0, index ce reprezinta valoarea minima anterioara. De asemenea, in functia getAction, am mai definit variabila bestAction, ce face apel al functiei minimax, insa cu index 1, acesta indicand “cea mai buna actiune” anterioara, atribuita minimului.

Pentru a eficientiza algorimtul minimax, ne vine in ajutor algorimul alphaBeta, care este asemenator cu minimax, insa in cadrul arboreului, nu va face toate verificarile, o data ce se asigura ca nu poate obtine de pe o ramura o valoare mai buna decat cea pe care o are in momentul curent. Asta duce la prunning (ramura respectiva nu va mai fi luata in calcul).

Dupa cum am spus anterior, algorimtul este o imbunatatire pentru minimax, asadar, vom mai tine cont de inca 2 parametrii: alpha si beta.

Alpha: initial va fi initializat cu cea mai mare valoare (inifinit), iar apoi va fi reinitializat in functie de noul maxim gasit

Beta: initial va lua valorea -infinit, iar apoi se va reinitializa in functie de noul minim gasit.

De mentionat este faptul ca, in functi getActions, am mai creat o variabila ajutatoare, ca si la minimax, numita bestAction, ce va avea index 1, iar dupa cum am prezentat anterior, la inceput va apela functia alphabeta cu paremetrii pentru alpha si beta egali cu inifinit si -inifinit.