Documentatie Inteligenta Artificala

Project 1: Search

Ilies Oana-Elena 30232

## Q1: Finding a Fixed Food Dot using Depth First Search

In acest caz facem expandarea celui mai adanc nod. Frontiera este reprezentata de o stiva(concept LIFO), avantajul acestei implementari este ca nodurile expandate fara succesori in frontiera sunt eliminate din memorie.

Este un algoritm complet care exploreaza cat mai mult posibil inainte de a reveni inapoi pe un drum. Pacman exploreaza continuu o cale, mergand cat mai adanc posibil pana cand nu mai poate merge, apoi se intoarce si incearca alte drumuri nexplorate, eliminand nodurile din memorie. DFS explorează în mod recursiv fiecare nod până când găsește soluția sau până când ajunge la o stare blocată (un nod fără succesori neexplorați). Odată ce a ajuns la un nod terminal sau blocat, se întoarce înapoi și continuă explorarea din ultimul nod neexplorat disponibil. FS poate găsi soluții rapid în structuri grafice de mari dimensiuni, dar nu garantează găsirea celei mai scurte căi către soluție. Deoarece explorează în profunzime, poate ajunge să urmeze o căi lungă până la o soluție, chiar dacă o soluție mai scurtă ar fi disponibilă pe altă ramură.

Este eficient cand bucata de mancata de mancare se gaseste la o adancime mica, cel mai neeficient este atunci cand bucata de mancare se gaseste foarte adanc, la disntante mari si poate duce la o salutie care nu este optima.

## Q2 : Breadth First Search

In acest caz facem expandarea celui mai superficial nod neexpandat. Frontiera este reprezentata de o coada(FIFO), succesorii noi sunt adaugati la final. Daca am gasit nodul scop nu ma opresc deoarece doar atunci cand scoatem din frontiera testam daca nodul e scop.

Este un algoritm complet care extinde nodurile adiacente ale nodului curent în ordine și le adaugă în coadă. Apoi, se elimină nodul curent din coadă și se continuă explorarea cu primul nod din coadă. Pentru a evita ciclurile în graf, BFS ține evidența nodurilor deja vizitate și nu le mai explorează din nou. BFS găsește o soluție de cost minim pentru căutările care sunt neponderate (toate muchiile au același cost) și pentru căutările unde costul drumului este direct proporțional cu adâncimea (costul este unitar pentru fiecare pas). În cazul grafurilor masive sau cu adâncime mare, BFS poate consuma multă memorie din cauza stocării tuturor nodurilor de pe un nivel în coadă. De asemenea, poate fi ineficient pentru grafuri cu costuri diferite pe muchii, deoarece nu este capabil să gestioneze aceste informații.

Exploreaza foarte mult(zona rosie foarte mare), dar gaseste solutia optima. Eficient atunci cand bucata de mancare se afla la distante mici, adica daca se gaseste la un nivel mic in arborele de cautare, dar la distante mari devine ineficient, deoarece expandeaza foarte mult si agentul cauta foarte mult. Dar nu parcurge decat solutia eficienta.

## Q3 : Varying the Cost Function

In acest caz facem expandarea celui mai mic cost si este folosit pentru a găsi cea mai ieftină cale posibilă de la Pacman la obiectivul său în labirint.. Frontiera este reprezentata de o coada ordonata dupa costul caii ce scoate nodul cu costul cel mai mic. Pastreaza doar calea cu costul cel mai mic. In cazul in care costul este egal cu 1, problema de cautare revine la o abordare BFS.

Este un algoritm optimal si complet care explorează spațiul de căutare pentru a găsi cea mai ieftină cale între un nod de start și un nod scop. Spre deosebire de căutarea în lățime (BFS) care se concentrează pe numărul de pași, UCS se concentrează pe costurile asociate fiecărui pas în cadrul căutării. Principiul de bază al UCS este că el extinde mereu nodurile cu cel mai mic cost de la nodul de start către nodurile vecine, ținând evidența costului total pentru a ajunge la fiecare nod. Acesta menține o coadă de priorități (de obicei implementată ca o coadă de tip minim sau o listă sortată) pentru a alege întotdeauna următorul nod cu cel mai mic cost pentru a-l explora.

Functioneaza la fel ca si BFS, dar agentul nostru nu cauta la fel de mult deoarece tine cont si de cost(astfel zona rosie este redusa).Este eficient pentru costuri variabile, dar devine ineficient atunci daca costurile cresc rapid sau exista mai multe cai cu costuri similare.

## Q4 : A\* search

Algoritmul A\* este o extensie a căutării uniform-cost (UCS), îmbunătățită prin adăugarea unei componente euristice. Acesta se concentrează și el pe găsirea celei mai ieftine căi între un nod de start și un nod țintă, dar utilizează informații euristice pentru a ghida căutarea către soluție într-un mod mai eficient.La fel ca UCS, A\* utilizează o coadă de priorități pentru a explora spațiul de stări, dar în plus, în calculul priorității fiecărui nod se adaugă și o estimare euristică a costului rămas până la soluție.Algoritmul A\* selectează nodurile pentru extindere combinând costul real de la nodul de start până la nodul curent cu o estimare euristică a costului rămas până la nodul țintă. Acest lucru se face în mod obișnuit prin adunarea costului real până la nodul curent cu o estimare euristică a costului rămas.

A\* este complet și optim în cazul în care euristica este admisibilă (nu supraestimează costurile) și consistentă (nu contrazice regulile jocului sau problemei). Folosind o euristică bună, A\* poate fi mult mai eficient decât UCS sau alte algoritmi de căutare, deoarece poate ghida căutarea către soluție într-un mod mai informat și mai direcționat.

Este cea mai eficienta varianta de implementare pentru jocul Pacman, daca euristica este aproape de adevarat(e complet pentru graph search si admisibil pentru three-search). Devine ineficient atunci cand euristica este supraestimata(exista posibilitatea sa nu gaseasca solutia optima). Deci algoritmul expandeaza cel mai putin si este optim pentru orice euristica consistenta.

## Q5: Finding All the Corners

La problema Q5, ne confruntăm cu un labirint pe care trebuie să-l străbatem pentru a atinge cele patru colțuri. Avem nevoie de o strategie eficientă pentru a gestiona această căutare.Pentru a face acest lucru, definim o reprezentare a stării care include poziția actuală a lui Pacman și o listă care indică dacă fiecare dintre cele patru colțuri a fost atins sau nu. Practic, starea noastră reflectă exact unde se află Pacman și dacă a ajuns sau nu la fiecare colț.În cadrul căutării, examinăm fiecare direcție posibilă în care Pacman poate merge. Verificăm dacă acea direcție este accesibilă și, dacă este, evaluăm dacă reprezintă un colț. Dacă da, actualizăm starea noastră pentru a reflecta că am atins un alt colț.În momentul inițial, stabilim poziția lui Pacman și notăm că nu a atins niciun colț. Apoi, în procesul de explorare, verificăm direcțiile posibile și, în cazul în care găsim un colț, actualizăm starea noastră pentru a reflecta acest lucru.Obiectivul final este să marcăm fiecare colț și să determinăm dacă Pacman a atins toate cele patru. Astfel, urmărim traseul lui Pacman și observăm dacă a reușit să acopere toate colțurile. Această abordare ne permite să navigăm eficient prin labirint și să identificăm momentul în care am terminat cu succes nivelul.

Funcția getStartState este responsabilă pentru returnarea stării de pornire pentru căutarea noastră. În acest caz, starea inițială constă în poziția de start a lui Pacman și un vector care indică dacă fiecare dintre cele patru colțuri a fost atins sau nu. Folosind (False, False, False, False), indicăm că la început Pacman nu a atins niciun colț.

Funcția isGoalState este utilizată pentru a verifica dacă starea dată este o stare scop sau nu, adică dacă am atins condiția finală în căutarea noastră. În acest caz, se verifică dacă toate elementele din vectorul care indică colțurile sunt True. Folosind funcția all(state[1]), verificăm dacă toate valorile din acest vector sunt True, ceea ce înseamnă că Pacman a ajuns la toate cele patru colțuri.

Funcția getSuccessors este responsabilă pentru generarea stărilor succesoare în timpul căutării drumului prin labirint. Ea explorează direcțiile posibile în care Pacman poate merge, verifică accesibilitatea acestora în raport cu peretele labirintului și actualizează stările succesoare, ținând cont de atingerea sau nu a colțurilor. Prin actualizarea informațiilor despre starea colțurilor și înregistrarea noilor stări succesoare, această funcție ajută la determinarea pașilor următori în rezolvarea problemei de găsire a drumului prin cele patru colțuri ale labirintului.

Colturi sunt trabatute rapid daca nu exista obstacole intre cele 4 colturi, devine mai lenta atunci cand exista multe obstacole, adica necestia un numar mare de explorari.Asadar eficienta/ineficienta este influentata de configuratia labirintului.

## Q6 : Corners Problem: Heuristic

Euristica pentru problema colțurilor are rolul de a estima cât de departe este cel mai îndepărtat colț nevizitat de poziția curentă a lui Pacman. Folosește informațiile despre poziția lui Pacman, starea colțurilor și configurația labirintului pentru a identifica colțurile care nu au fost atinse încă. Calculează distanța Manhattan (distanta pe orizontală și verticală) între poziția lui Pacman și fiecare colț nevizitat și returnează cea mai mare dintre aceste distanțe. Rezultatul returnat de euristică reprezintă valoarea maximă a acestor distanțe. Atunci când nu există colțuri nevizitate, euristica va returna 0, indicând că toate colțurile au fost atinse. În caz contrar, va furniza cea mai mare distanță dintre poziția lui Pacman și cel mai îndepărtat colț nevizita. Această valoare oferă o estimare a distanței până la cel mai îndepărtat colț nevizitat, ghidând căutarea către destinație într-un mod eficient și consistent.

Deci euristica folosita foloseste distanta maxima Manhattan intre pozitia actuala si colturile nezitate ale labirintului pentru a determina urmatoarea actiune pentru probelma. Astefel se priorizeaza colturile nevizitate, astfel agentul estimeaza mai eficient costul pana la acele puncte.Eficientizeaza cautarea pentru problema colturilor prin prioritizarea colturilor nevizitate astfel reduce timpul necesar pentru a obtine toate bucatile de mancare. La fel ca si la varianta fara euristica eficienta/ineficienta este influentata de configuratia labirintului.

Partea superioară a machetei

## Q7 : Eating All The Dots

Am folosit o euristică simplă și admisibilă, care estimează distanța maximă Manhattan între poziția curentă a lui Pacman și cea mai îndepărtată bucată de mâncare rămasă. Această euristică este admisibilă deoarece nu supraestimează niciodată distanța reală până la soluție și oferă o estimare pesimistă a distanței. În primul rând, extrage poziția lui Pacman și grila cu bucăți de mâncare din starea curentă. Apoi, verifică dacă mai există bucăți de mâncare neatinse. Dacă nu mai sunt bucăți de mâncare, întoarce 0, indicând că Pacman a ajuns la toate bucățile de mâncare. În caz contrar, calculează distanța Manhattan între poziția lui Pacman și fiecare bucată de mâncare rămasă. Aceste distanțe sunt stocate într-o listă. În final, euristica returnează cea mai mare distanță din această listă, indicând distanța maximă până la cea mai îndepărtată bucată de mâncare, oferind astfel o estimare a distanței până la următoarea bucată de mâncare pe care Pacman trebuie să o atingă.

Deci aceasta euristică calculează distanța Manhattan între poziția curentă a lui Pacman și cea mai îndepărtată bucată de mâncare. Euristică returnează cea mai mare dintre aceste valori, oferind astfel o estimare a distanței până la cea mai îndepărtată bucățică de mâncare, ghidând astfel Pacman-ul către următoarea sa destinație. Nu este cea mai eficienta varianta pentru pacman, această euristică poate duce la trasee mai lungi și poate fi ineficientă, la fel eficienta/ineficienta este influentata de configuratia labirintului. Asadar varianta implementata de mine expandeaza cel mult 12000 de noduri.

## Q8 : Suboptimal Search

Defineste o metodă care găsește un drum către cea mai apropiată bucată de hrană Pacman, plecând de la starea actuală a jocului. În primul rând, se extrag informațiile de bază despre poziția inițială a lui Pacman, grila cu bucăți de mâncare și configurația pereților labirintului. Problema de căutare a hranei este apoi definită utilizând starea curentă a jocului. Algoritmul începe prin adăugarea poziției de start a lui Pacman într-o coadă. Apoi, utilizează o buclă pentru a explora în lățime posibilele mișcări. Se verifică fiecare poziție pentru a determina dacă a fost deja vizitată și, în caz afirmativ, se continuă explorarea. Dacă se găsește o bucată de mâncare, algoritmul returnează lista de acțiuni necesare pentru a ajunge la acea bucată de mâncare. În absența unei bucăți de mâncare accesibile, se returnează o listă goală, indicând că nu există un drum disponibil la o bucată de mâncare din starea actuală a jocului Pacman. Scopul este de a găsi un drum rapid către cel mai apropiat punct de mâncare, chiar dacă nu întotdeauna este cea mai scurtă cale posibilă în labirint. Această funcție nu garantează găsirea celei mai scurte căi pentru a mânca toate punctele de mâncare și poate furniza soluții suboptimale în unele cazuri.

Această metodă de căutare este eficientă în situații în care labirinturile sunt relativ simple și fără multe obstacole. Funcționează bine atunci când există un singur drum clar și apropiat către o bucățică de mâncare și oferă rapid o soluție pentru a ajunge la acea bucată.

În schimb, devine ineficientă în labirinturi complexe sau dens populate cu obstacole, având tendința de a genera soluții suboptimale. Atunci când există multiple căi sau obstacole care influențează traseul către bucățile de mâncare, algoritmul poate să aleagă un drum care nu este cel mai scurt sau eficient, ducând la soluții care nu acoperă cea mai optimă cale pentru a ajunge la toate bucățile de mâncare disponibile.

Project 1: Multi-Agent Search

## Q1 : Reflex Agent

Am implementat o facuntie de evaluare mai eficienta pentru starea jocului Pacman. Primeste 2 stari, starea curenta si starea succesorului propus. Mai întâi, se generează starea următoare a jocului după aplicarea acțiunii, extrăgând informații despre poziția nouă a lui Pacman, pozițiile bucăților de mâncare rămase, starea actualizată a fantomelor și timpul rămas în care fiecare fantomă este speriată. Parcurgem lista de poziții ale bucăților de mâncare și calculează cea mai mică distanță Manhattan dintre poziția lui Pacman și acele bucăți de mâncare. Variabila closestFoodDistance retine distanța cea mai apropiată dintre bucățile de mâncare rămase. Apoi parcurge lista de stări ale fantomelor și calculează cea mai mică distanță Manhattan între poziția lui Pacman și fiecare poziție a fantomelor. Variabila closestGhostDistance este actualizată cu cea mai mică distanță găsită, reținând distanța cea mai apropiată dintre Pacman și fantomele din starea curentă a jocului.Urmand ca apoi sa estimam disnta pana la cea mia apropiata fantoma in functie de timpil ramas. Valoarea 70 este folosită pentru a scădea estimarea distanței până la cea mai apropiată fantomă daca este „speriata”, reducerea semnificativă a valorii estimării, indicând că Pacman-ul poate să se apropie mai mult de fantome fără a fi în pericol atunci când acestea sunt speriate. Dacă cea mai mică distanță până la bucata de mâncare sau cea mai mică distanță până la fantomă este zero (adică Pacman este chiar lângă o bucată de mâncare sau chiar lângă o fantomă), dacă distanța este zero, aceasta este setată la o valoare de 1, (adica 1.0 / closestFoodDistance sau 1.0 / closestGhostDistance), astefel se evita impartirea la 0.Actualizam valoarea starii. Se adaugat 10 pentru distanta pana la cea mai apropiata bucata de mancara, Acest lucru înseamnă că cu cât distanța până la mâncare este mai mică, cu atât va fi adăugată o valoare mai mare la evaluarea stării si se scade o pondere mai mică 7.0 pentru distanța până la cea mai apropiată fantomă Asta înseamnă că atunci când Pacman se apropie de o fantomă, evaluarea stării va scădea, oferind o penalizare mai mică pentru distanțele mici până la fantome. La final este returnata evaluarea.

## Q2 : Minimax

Algoritmul Minimax explorează recursiv posibilele acțiuni disponibile pentru Pacman și pentru fiecare fantomă, simulând mișcările lor și alternând între două tipuri de noduri: MAX pentru Pacman (care încearcă să maximizeze utilitatea) și MIN pentru fiecare fantomă (care încearcă să minimizeze utilitatea pentru Pacman). Acesta continuă explorarea până la o adâncime specificată sau până când se ajunge într-o stare finală (câștig, pierdere sau adâncimea maximă atinsă). În fiecare nod MAX, algoritmul alege acțiunea care maximizează scorul Pacman-ului, în timp ce în fiecare nod MIN, algoritmul ia în considerare toate acțiunile posibile ale unei fantome și alege acțiunea care minimizează scorul lui Pacman. În cele din urmă, algoritmul returnează cea mai bună acțiune posibilă pentru Pacman în starea de joc curentă, astfel încât să maximizeze șansele sale de câștig în jocul Pacman.

Deci MinMax se aplica pe o stare si se returneaza o actiune care duce in starea cu evaluarea cea mai mare. Min si Max returneaza o utilitate, in final se calculeaza min intre valoarea actuala si maximul valorii. E un algortim eficient si optimal impotriva unui oponent optimal.Odata castiga, odata pierde deoarece nu stim ce fac fontomele, ele nu sunt optimale. Cand fantomele aleg varianta optima pierdem. Cel mai eficient caz este atunci când adâncimea maximă de explorare este mică și numărul de acțiuni legale este redus la fiecare nivel, iar cel mai ineficient caz apare atunci când arborele de decizie este extrem de larg și adânc, iar numărul de acțiuni posibile crește rapid pe măsură ce se adâncește în arbore deoarece algoritmul are o abordare DFS. Datorita abordari DFS nu tine cont de toate valorile, cele care nu au frontiera sunt scoase din memorie.

## Q3 : Alpha-Beta Pruning