DocumentatieProiectPacmanFabian

Fabian - Remus Cioban gr:30233

December 2023

1 Project 1: Search

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

Algoritmul DFS este un algoritm de cautare neinformat, el cautand solutia in adancime, altfel spus el va explora fiecare nod chiar daca toti fratii parintelui nodului curent inca nu au fost descoperiti. Acest algoritm foloseste ca structura de date o stiva. Se initializeaza o stiva (frontier) pentru a urmari nodurile de explorat. Se adauga nodul de start in stiva.Se initializeaza o lista (noduriExplorare) pentru a urmari nodurile deja explorate.Atata timp cat stiva nu este goala,se extrage nodul curent din varful stivei.Daca nodul curent reprezinta starea tinta (obiectivul cautarii), se reconstruieste calea de la nodul de start la nodul tinta si se returneaza aceasta cale.Daca nodul curent nu este tinta, acesta este adaugat in lista de noduri explorate si sunt explorati succesorii sai neexplorati.Acesti succesori sunt adaugati in stiva pentru explorare ulterioara.Daca nu este gasita nicio solutie, se returneaza o lista goala.Algoritmul mentine o lista a nodurilor explorate pentru a nu explora aceleasi noduri de mai multe ori.La final, cand se gaseste o solutie (obiectivul), algoritmul returneaza un vector de directii care reprezinta calea de la punctul de start catre obiectivul gasit (goalState).

1.2 Q2: Breadth First Search

Algoritmul BFS este foarte similar cu DFS, diferenta majora fiind faptul ca la BFS folosim o coada in locul stivei, astfel parcurgerea se va face in latime, nu in adancime.

1.3 Q3: Varying the Cost Function

Algoritmul de cautare uniforma cu costuri foloseste o metoda ce se bazeaza pe prioritate pentru a explora nodurile in functie de costul total pana la fiecare nod. Lista 'noduriExplorare' pastreaza nodurile ce au fost deja cercetate pentru a preveni explorarea de mai multe ori a acelorasi noduri. Incepe verificand daca starea de start (problem.getStartState()) coincide deja cu starea finala. Daca da, se returneaza o lista goala deoarece ambele stari sunt identice.'frontierPriorityQ' este o coada de prioritati utilizata pentru a explora nodurile. Starea de start este adaugata in aceasta coada cu un cost initial de 0. Procesul continua atata timp cat coada de prioritati nu este goala: se extrage urmatorul nod din coada impreuna cu directiile si costul minim necesar pentru a ajunge la acel nod. Se verifica daca nodul extras nu a fost explorat inca. Daca nu a fost, este adaugat in lista de noduri explorate.Daca nodul extras reprezinta starea finala, atunci se returneaza lista de directii necesare pentru a ajunge la nodul final. Algoritmul exploreaza succesiv nodurile succesoare ale nodului curent, iar informatiile despre fiecare succesor (noul nod, directia si costul pentru a ajunge la acesta) sunt adaugate in coada de prioritati.Procesul algoritmului continua sa exploreze si sa extinda nodurile pana cand gaseste nodul final sau pana cand coada de prioritati devine goala. In cazul in care nu exista un drum catre nodul final, se returneaza o lista goala. Scopul acestui algoritm este sa gaseasca cea mai ieftina ruta catre starea finala, comparand si actualizand costurile in functie de nodurile explorate si costurile asociate acestora.

1.4 Q4: A* search

Algoritmul A* este foarte popular cand vine vorba de cautarea in diferite stari. Functia ‘aStarSearch‘ primeste doua lucruri: ‘problem‘ si ‘heuristic‘ (care implicit este ‘nullHeuristic‘). Aceasta functie incearca sa gaseasca nodul care are cel mai mic cost combinat si evalueaza mai intai euristicile. Incepe prin a initializa o lista goala ‘noduriExplorare‘ pentru a urmari nodurile care au fost deja verificate. Verifica daca starea initiala este si o stare finala. Daca este, intoarce o lista goala pentru ca nu mai este necesar sa exploreze, fiind deja la solutie. Se creeaza o coada de prioritati ‘frontierPriorityQ‘ pentru a gestiona starile in functie de costul estimat. Starea initiala este adaugata in aceasta coada, avand un cost initial de 0. Intr-un ciclu ‘while‘, algoritmul continua sa exploreze starile pana cand coada de prioritati devine goala. In fiecare iteratie, se extrage urmatorul nod din coada de prioritati impreuna cu informatiile asociate (starea, directiile si costul). Se verifica daca nodul curent nu a fost deja explorat. Daca nu a fost, se adauga in lista ‘noduriExplorare‘. Daca nodul curent este un nod final, se intorc directiile pana la acea stare, reprezentand solutia cautarii. Se parcurg succesoriile nodului curent. Pentru fiecare succesor, se adauga in coada de prioritati informatiile acestuia (stare, directii si cost) impreuna cu costul estimat actualizat folosind euristica data si costul real pana la acel succesor. Algoritmul se opreste atunci cand coada de prioritati devine goala sau cand gaseste o solutie, intorcand o lista goala in cazul in care nu a gasit o solutie.

1.5 Q5: Finding All the Corners

Acest cod este esential pentru controlul si analiza starilor posibile in timpul cautarii unei solutii. Folosind functii precum getStartState, isGoalState si getSuccessors, acesta gestioneaza diferite problemei.

getStartState este functia responsabila cu returnarea starii initiale. Aceasta este reprezentata printr-o tupla ce contine pozitia de start si o colectie de colturi care inca nu au fost vizitate sau rezolvate.

isGoalState verifica daca starea furnizata ca argument este o stare finala, adica daca nu mai exista colturi nevizitate in acea stare. Aceasta este o functie cruciala pentru determinarea cand se incheie rezolvarea problemei.

getSuccessors este o functie complexa care primeste o stare specifica, formata dintr-o pozitie curenta si o lista de colturi ramase neexplorate. Ea genereaza stari succesoare prin evaluarea posibilitatilor de miscare. Pornind de la pozitia curenta, functia verifica in toate directiile posibile (nord, sud, est, vest) pentru a determina daca urmatoarea miscare este fezabila, fara a lovi un perete sau a iesi din limitele permise. In cazul in care miscarea este valida, se genereaza o noua stare succesoare. Aceasta va avea pozitia actualizata conform directiei de miscare si va elimina un colt din lista de colturi ramase, in situatia in care pozitia urmatoare contine un colt neexplorat. Fiecare stare succesoare generata este adaugata in lista de succesiuni impreuna cu actiunea care a condus la acea stare si un cost asociat. La final, functia returneaza lista de succesiuni, reprezentand astfel starile succesoare generate din starea data.

Pe langa aceste functii, in timpul generarii starilor succesoare, codul inregistreaza si numara numarul de noduri expandate intr-o variabila denumita self.expanded. Aceasta inregistrare a numarului de noduri expandate poate fi utila pentru analizarea si optimizarea algoritmilor de cautare.

1.6 Q6: Corners Problem: Heuristic

Aceasta functie, numita ‘cornersHeuristic‘, calculeaza o estimare a cat de aproape este o situatie intr-un joc de colturi de a fi considerata o solutie. Ea primeste doua informatii: ‘state‘ si ‘problem‘. ‘state‘ este o serie de date care include pozitia actuala si colturile ramase de vizitat in problema respectiva. ‘problem‘ este obiectul specific al problemei pentru colturi. Functia incepe prin extragerea pozitiei actuale si a colturilor ramase din informatia furnizata in ‘state‘. Daca toate colturile au fost deja vizitate (adica lista ‘colturiRamase‘ este goala), atunci functia va returna o valoare euristica de 0, deoarece am atins deja solutia.In caz contrar, functia incearca sa evalueze cat de departe se afla cel mai indepartat colt nevizitat fata de pozitia curenta. Acest lucru se realizeaza prin calcularea distantei Manhattan intre pozitia curenta

si fiecare colt nevizitat, retinand cea mai mare dintre aceste distante. Valoarea euristica returnata este distanta maxima dintre pozitia curenta si cel mai indepartat colt nevizitat, insa este limitata la maxim 1200. Aceasta limitare poate ajuta algoritmul de cautare sa se indrepte in directia corecta pentru a vizita colturile ramase, deoarece valoarea euristica indica cat de departe se afla cel mai indepartat colt nevizitat. Limitarea distantei la 1200 este menita sa evite valori nerealiste care ar putea influenta negativ algoritmul de cautare.In concluzie, functia va returna valoarea euristica calculata sau 0 in cazul in care apare o problema in procesul de calcul al acesteia, aceasta din urma fiind considerata o solutie implicita.

1.7 Q7: Eating All The Dots

Functia 'foodHeuristic' este utilizata in algoritmii de cautare pentru a estima cat de aproape sau cat de departe se afla cea mai indepartata hrana intr-un grid fata de pozitia curenta a jucatorului. Acest lucru se bazeaza pe distantele calculate pana la fiecare aliment in parte din grid. Ea primeste informatii despre pozitia jucatorului si gridul de alimente din starea curenta.Se initializeaza o lista numita 'distance' pentru a stoca distantele pana la fiecare aliment din grid. Apoi, algoritmul itereaza prin fiecare celula a gridului si verifica daca exista un aliment in acea celula. In caz afirmativ, se calculeaza distanta dintre pozitia alimentului si pozitia actuala a jucatorului, iar aceasta distanta este adaugata in lista 'distance'.Dupa aceea, se verifica daca nu a fost gasit niciun aliment in grid. In aceasta situatie, functia returneaza 0, indicand ca nu exista hrana in grid. In cazul in care au fost gasite alimente, functia returneaza cea mai mare distanta gasita pana la un aliment din lista 'distance'.

1.8 Q8: Suboptimal Search

Functia 'isGoalState' este o metoda in cadrul unei clase si isi propune sa determine daca starea actuala a jocului reprezinta un obiectiv, adica daca Pacman a ajuns la o pozitie unde exista mancare. Pentru a face acest lucru, functia primeste o stare 'state' sub forma de tuplu de coordonate (x, y), care reprezinta pozitia actuala a lui Pacman pe tabla. Pentru a verifica prezenta mancarii la acea pozitie, functia parcurge coordonatele posibile ale mancarii utilizand informatiile despre ziduri si obtine un tuplu de coordonate 'foodTuple', care cuprinde toate pozitiile cu mancare. Apoi, functia verifica daca pozitia data de 'state' se gaseste in 'foodTuple'. Daca 'state' (pozitia curenta a lui Pacman) se potriveste cu una dintre pozitiile cu mancare, functia returneaza True, semnaland ca Pacman a ajuns la mancare. In caz contrar, functia returneaza False, indicand ca in pozitia curenta nu exista mancare.Functia 'findPathToClosestDot' are rolul de a gasi o ruta catre cel mai apropiat punct disponibil din jocul Pacman, avand ca baza starea actuala a jocului reprezentata de 'gameState'. Foloseste o metoda de cautare in latime (breadthFirstSearch) pentru a identifica drumul catre cel mai apropiat punct disponibil, pe baza problemei definite anterior. Rezultatul acestei functii consta in ruta gasita catre cel mai apropiat punct.

2 Project 3: Multi-Agent Search

2.1 Q1: Reflex Agent

In functia de evaluare, am transformat mancarea ramasa intr-o lista si am calculat distanta Manhattan intre pozitia lui Pacman si fiecare bucatica de mancare. Apoi, am calculat distanta Manhattan intre pozitia lui Pacman si fiecare fantoma. In continuare, ne asiguram ca fantomele nu sunt aproape de Pacman; daca sunt aproape, il fortam sa se indeparteze de ele, adica returnam o distanta foarte mica. In lipsa mancarii, returnam o distanta foarte mare. In final, daca nu am intrat in niciunul dintre aceste cazuri, vom genera o distanta care variaza in functie de mancarea ramasa si scorul curent.

2.2 Q2: Minimax

Algoritmul Minimax incearca sa gaseasca cea mai buna miscare pentru Pacman, luand in considerare ce actiuni ar putea lua adversarii sai pentru a obtine cel mai bun scor posibil. Functioneaza prin exami-

narea si compararea diferitelor stari ale jocului intr-un mod de tip "cea mai buna optiune pentru mine, cea mai rea optiune pentru ei".Practic, acest algoritm exploreaza diferitele actiuni pe care Pacman le poate face si incearca sa maximizeze scorul sau. Algoritmul Minimax opereaza prin evaluarea recursiva a starilor jocului, alternand intre doua tipuri de functii: maxvalue si minvalue.Functia maxvalue evalueaza actiunile disponibile pentru Pacman si incearca sa maximizeze scorul acestuia. Pentru fiecare actiune legala, genereaza starea succesoare si apeleaza functia minvalue pentru a obtine scorul minim pentru actiunile agentilor adversi. Apoi, selecteaza actiunea care maximizeaza scorul pentru Pacman.Functia minvalue evalueaza actiunile agentilor adversi si incearca sa minimizeze scorul acestora. Pentru fiecare actiune legala, genereaza starea succesoare si, in functie de agentul curent sau de faptul ca sunt inca agenti adversi de evaluat, apeleaza recursiv functia minvalue sau maxvalue.In cele din urma, algoritmul Minimax este initiat prin apelul functiei maxvalue, care determina actiunea optima pentru Pacman in starea de joc data, intorcand acea actiune.

2.3 Q3: Alpha-Beta Pruning

Codul implementeaza un agent de cautare bazat pe algoritmul Minimax, imbunatatit cu taierea Alpha-Beta, o tehnica care imbunatateste eficienta algoritmului Minimax prin eliminarea explorarii unor ramuri inutile in arborele de decizie.Metoda getAction, determina cea mai buna actiune si scorul asociat pentru agent folosind adancimea specificata si o functie de evaluare.Functia getBestActionAndScore este responsabila pentru explorarea arborelui de decizie folosind algoritmul Minimax impreuna cu taierea Alpha-Beta. Aceasta decide intre a apela functia maxvalue (agentul maximizator Pacman) sau minvalue (agentul minimizator fantome).maxvalue si minvalue sunt functii care implementeaza logica algoritmului minimax, putin modificat fata de mai sus datorita noilor parametrii ai functiei,pentru agentul maximizator, respectiv minimizator. Acestea itereaza prin miscarile legale disponibile pentru agentul respectiv si genereaza succesorii starii curente pentru a evalua optiunile posibile. In timpul acestei explorari, algoritmul actualizeaza valorile alpha si beta pentru a elimina ramuri din arborele de decizie care nu vor afecta rezultatul final. Astfel, aceste functii utilizeaza taierea Alpha-Beta pentru a elimina explorarea unor ramuri care nu sunt relevante pentru alegerea finala a actiunii.Algoritmul se opreste in momentul in care se atinge o stare terminala (fara miscari legale disponibile sau adancimea maxima este atinsa), moment in care este returnata actiunea si scorul asociat acelei stari.In final, agentul va alege actiunea care maximizeaza scorul, conform algoritmului minimax imbunatatit cu taierea Alpha-Beta, si va returna acea actiune din metoda getAction.