**DOCUMENTATIE**

**PACMAN**

**Chetan Anca**

**Dragus Andreea**

**grupa 30236**

***PACMAN***

In cadrul acestui proiect am implementat diferite strategii de cautare si si diferite euristici pentru anumite probleme de cautare.

Vom incepe prin a dezvolta ce strategii de cautare am abordat in cadrul proiectului.

Pentru inceput am creat o structura de nod care sa ne permite sa retinem starea curenta, parintele (ce stare a generat starea curenta) si actiunea prin care a fost generata aceasta stare si de asemenea si costul.

1. Prima strategie de cautare este ***DFS*** care foloseste pentru frontiera o structura de date de tip stiva. Am folosit graph search ceea ce inseamna ca am introdus si o lista de noduri expandate. De asemenea pentru a putea returna calea pe care a urmat-o Pacman in urma acestei cautari vom avea si o lista care contine actiunile facute de acesta pentru a ajunge la starea de goal. Astfel pornim cu Pacman din pozitia de start, adaugam acest nod in frontiera, iar apoi cat timp frontiera nu este goala scoatem pe rand cate un nod din aceasta, verificam daca nu cumva am ajuns la goal, iar daca este cazul apelam metoda constructPath() din cadrul clasei Node care ne va construi mergand in sus pe parinte lista de actiuni pe care Pacman a urmat-o sa ajunga in starea de goal. In cazul in care nu am atins starea noastra finala, vom verifica daca nodul tocmai scos din frontiera a fost deja expandat, in caz contrar il vom adauga noi acum in lista de noduri expandate, vom calcula succesorii acestuia cu ajutorul metodei getSuccessors(), iar pentru fiecare successor care nu este deja expandat il vom adauga in frontiera. La final vom retura calea pe care Pacman a urmat-o.
2. A doua strategie de cautare este ***BFS*** care foloseste pentru frontiera o structura de date de tip coada. Am folosit graph search ceea ce inseamna ca am introdus si o lista de noduri expandate. De asemenea pentru a putea returna calea pe care a urmat-o Pacman in urma acestei cautari vom avea si o lista care contine actiunile facute de acesta pentru a ajunge la starea de goal. Astfel pornim cu Pacman din pozitia de start, adaugam acest nod in frontiera, iar apoi cat timp frontiera nu este goala scoatem pe rand cate un nod din aceasta, verificam daca nu cumva am ajuns la goal, iar daca este cazul apelam metoda constructPath() din cadrul clasei Node care ne va construi mergand in sus pe parinte lista de actiuni pe care Pacman a urmat-o sa ajunga in starea de goal. In cazul in care nu am atins starea noastra finala, vom verifica daca nodul tocmai scos din frontiera a fost deja expandat, in caz contrar il vom adauga noi acum in lista de noduri expandate, vom calcula succesorii acestuia cu ajutorul metodei getSuccessors(), iar pentru fiecare successor care nu este deja expandat il vom adauga in frontiera. La final vom retura calea pe care Pacman a urmat-o.
3. A treia strategie de cautare este ***UCS*** care foloseste pentru frontiera o structura de date de tip coada de prioritati, iar elementele din aceasta structura vor fi tuple formate din nod si costul acumulat pana la acesta, acest cost reprezentand prioritatea. Am folosit graph search ceea ce inseamna ca am introdus si o lista de noduri expandate. De asemenea pentru a putea returna calea pe care a urmat-o Pacman in urma acestei cautari vom avea si o lista care contine actiunile facute de acesta pentru a ajunge la starea de goal. Astfel pornim cu Pacman din pozitia de start, adaugam acest nod in frontiera, iar apoi cat timp frontiera nu este goala scoatem pe rand cate un nod din acesta, verificam daca nu cumva am ajuns la goal, iar daca este cazul apelam metoda constructPath() din cadrul clasei Node care ne va construi mergand in sus pe parinte lista de actiuni pe care Pacman a urmat-o sa ajunga in starea de goal. In cazul in care nu am atins starea noastra finala, vom verifica daca nodul tocmai scos din frontiera a fost deja expandat, in caz contrar il vom adauga noi acum in lista de noduri expandate, vom calcula succesorii acestuia cu ajutorul metodei getSuccessors(), iar pentru fiecare successor care nu este deja expandat il vom adauga in frontiera impreuna cu costul acumulat pana in acest punct (cost + costParinte). La final vom retura calea pe care Pacman a urmat-o.
4. A patra strategie de cautare este ***A\**** care foloseste pentru frontiera o structura de date de tip coada de prioritati, iar elementele din aceasta structura vor fi tuple formate din nod si suma dintre costul acumulat pana la acesta si euristica in aceasta stare, acesta suma reprezentand prioritatea. Am folosit graph search ceea ce inseamna ca am introdus si o lista de noduri expandate. De asemenea pentru a putea returna calea pe care a urmat-o Pacman in urma acestei cautari vom avea si o lista care contine actiunile facute de acesta pentru a ajunge la starea de goal. Astfel pornim cu Pacman din pozitia de start, adaugam acest nod in frontiera, iar apoi cat timp frontiera nu este goala scoatem pe rand cate un nod din acesta, verificam daca nu cumva am ajuns la goal, iar daca este cazul apelam metoda constructPath() din cadrul clasei Node care ne va construi mergand in sus pe parinte lista de actiuni pe care Pacman a urmat-o sa ajunga in starea de goal. In cazul in care nu am atins starea noastra finala, vom verifica daca nodul tocmai scos din frontiera a fost deja expandat, in caz contrar il vom adauga noi acum in lista de noduri expandate, vom calcula succesorii acestuia cu ajutorul metodei getSuccessors(), iar pentru fiecare successor care nu este deja expandat il vom adauga in frontiera impreuna cu suma dintre costul acumulat pana in acest punct (cost + costParinte) si euristica in aceasta stare. La final vom retura calea pe care Pacman a urmat-o.

***Corners Problem***

Pentru aceasta problema de cautare am ales ca starea sa fie reprezentata de pozitia lui Pacman si de o tupla in care adaugam pe masura ce ajungem intr-un corner pozitia acestuia. Pentru noi starea de goal este atinsa atunci cand tupla noastra are lungimea 4 adica am adaugat toate corner-urile in aceasta. Metoda de getSuccessors() ne returneaza succesorii starii curente astfel:

1. Extragem pozitia curenta a lui Pacman
2. Extragem lista de de corner-uri vizitate pana in acest moment
3. Recalculam pozitia lui Pacman daca ar face o anumita actiune
4. Verificam sa nu cumva sa avem un perete la pozitia tocmai calculate
5. Verificam daca la pozitia respectiva este un corner, apoi daca acesta nu este deja adaugat in lista il adaugam
6. La final adaugam succesorul impreuna cu actiunea si costul

***Corners Heuristic***

La inceput luam pozitia lui Pacman si lista de corner-uri parcurse. Avem si o lista cu corner-uri nevizitate. Cat timp aceasta nu este goala vom calcula distanta Manhattan de la pozitia lui Pacman la fiecare corner nevizitat si selectam minimul. Vom actualiza pozitia lui Pacman in corner-ul cu distanta Manhattan cea mai mica, vom scoate din lista de corner-uri nevizitate, corner-ul la care tocmai am obtinut distanta minima si vom adauga la suma aceasta distanta. La final vom returna aceasta suma.

***Food Heuristic***

Pentru fiecare bucata de mancare din grid vom calcula distanta reala (mazeDistance) de la pozitia lui Pacman la pozitia mancarii, iar la final vom returna distanta maxima.

***Any Food Search Problem***

In cadrul acestei probleme de cautare am atins starea de goal daca starea in care ne aflam coincide cu pozitia unei bucati de mancare.

***Find Path to Closest Dot***

Pentru a gasi cea mai apropiata bucata de mancare vom porni un BFS, iar problema pentru care aplicam acesta strategie de cautare este Any Food Search Problem unde goal state este cel mentionat mai sus.

***MULTIAGENT***

In cadrul acestui proiect am inceput prin implementarea unei functii de evualuare pentru ReflexAgent. Am extras la inceput cateva informatii utile care ne vor ajuta pe parcurs: succesorul lui Pacman daca se efectueaza o anumita actiune, noua pozitie a lui Pacman dupa ce s-a efectuat aceasta actiune (pozitia succesorului), bucatile de mancare ramase dupa efectuarea actiunii, pozitiile fantomelor si scared time-ul aferent fiecarei fantome. Prima oara am calculate distantele de la noua pozitie a lui Pacman pana la toate fantomele si distantele pana la toate bucatile de mancare, de asemenea am extras si scorul noii stari. Verificam daca nu cumva pozitia succesorului nu coincide cu pozitia curenta a lui Pacman, ceea ce inseamna ca acesta sta pe loc, vom returna o valoare foarte mica. Daca avem mancare ramasa dupa efectuarea actiunii este destul de rau si atunci din scorul nostru vom scadea distanta minima pana la mancare. Apoi vom alege distanta pana la cea mai apropiata fantoma, daca pozitia succesorului coincide cu aceasta distanta minima este foarte rau, deoarece inseamna ca am mers fix pe o fantoma deci vom scadea din scor un numar foarte mare, daca distanta minima nu este 0 vom scadea din scor 1/distanta minima, cu cat aceasta distanta este mai mare cu atat vom scadea mai putin (depinde cat de aprope este cea mai apropiata fantoma).

***MinMax***

Pentru aceasta strategie de cautare adversariala in clasa MinMaxAgent am avut nevoie de 3 metode:

1. **minmax\_decision():** aceasta metoda stabileste la inceput daca starea in care am ajuns s-a atins adancimea sau suntem intr-o stare in care am pierdut sau in care am castigat, daca este asa vom returna scorul obtinut cu ajutorul functiei de evaluare. Apoi vom verifica daca pozitia agentului este 0 inseamna ca Pacman detine controlul si vom returna apelul metodei max\_value(), in caz contrar o fantoma detine controlul si vom returna apelul metodei min\_value().
2. **max\_value():** in cadrul acestei metode vom obtine toate actiunile legale pe care le poate face agentul curent, vom parcurge aceasta lista de actiuni si pentru fiecare vom genera succesorul aferent. Vom stabili indexul succesorului si de asemenea si adancimea acestuia. Apoi vom apela metoda minmax\_decision() care se va comporta dupa cum am explicat mai sus. La final vom calcula valoarea maxima in functie de valoarea maxima curenta si ceea ce ne-a returnat apelul metodei minmax\_decision() si o vom returna impreuna cu actiunea care a generat acest maxim. Am returnat si aceasta actiune pentru a putea sa o folosim in metoda getAction().
3. **min\_value():** in cadrul acestei metode vom obtine toate actiunile legale pe care le poate face agentul curent, vom parcurge aceasta lista de actiuni si pentru fiecare vom genera succesorul aferent. Vom stabili indexul succesorului si de asemenea si adancimea acestuia. Apoi vom apela metoda minmax\_decision() care se va comporta dupa cum am explicat mai sus. La final vom calcula valoarea minimul in functie de valoarea minima curenta si ceea ce ne-a returnat apelul metodei minmax\_decision() si o vom returna impreuna cu actiunea care a generat acest minim.

Am returnat si aceasta actiune pentru a putea sa o folosim in metoda getAction().

***Alpha Beta Pruning***

Aceasta strategie de cautare adversariala se comporta aproape la fel ca MinMax, dar apar in plus alpha si beta care ne permit sa reducem din numarul de noduri evaluate. Pentru acesta strategie avem nevoie de 3 metode in cadrul clasei AlphaBetaAgent:

1. **alpha\_beta\_search():** in cadrul acestei metode facem exact ce facem si in metoda minmax\_decision() explicata mai sus
2. **max\_value():** in cadrul acestei metode procedam la fel ca in cazul max\_decision() explicata mai sus, dar introducem in plus actualizarea lui alpha atunci cand este cazul si mai exact atunci cand beta este mai mare decat valoare maxima actuala.
3. **min\_value():** in cadrul acestei metode procedam la fel ca in cazul min\_decision() explicata mai sus, dar introducem in plus actualizarea lui beta atunci cand este cazul si mai exact atunci cand alpha este mai mic decat valoare minima actuala.

***Expectimax***

Aceasta stratgie de cautare adversariala nu mai presupune ca jucam impotriva unui adversar perfect. Pentru aceasta avem nevoie de 3 metode in cadrul calsei ExpectimaxAgent

1. **expectimax():** acesta metoda se comporta la fel ca minmax\_decision() explicata mai sus, doar ca atunci cand agentul este o fantom ava apela metoda prob\_value()
2. **metoda max\_value()**: face exact acelasi lucru ca si in cazul metodei cu acelasi nume din cadrul clasei MinMaxAgent.
3. **prob\_value():** parcurge lista de actiuni posibile pentru agentul nostru, calculeaza succesorul aferent actiunii si apoi apeleaza metoda expectimax(), dupa revenirea din acesta adauga la suma valoarea returnata de apelul anterior si de asemenea vom incrementa numarul de optiuni pentru ca la final sa returnam raportul dintre suma calculate si numarul de optiuni.

**BetterEvaluationFunction**

In cadrul acestei functii am definit cateva criterii pe baza carora se vor evalua starile jocului. Pentru a realiza acest lucru, am folosit trei tipuri de scor: ghost\_score, food\_score si scared\_score. Stategia aplicata in cadrul evaluarii este urmatoarea:

1. Calculam distantele Manhattan dintre Pacman si fantome, pentru a o extrage pe cea mai mica, numita minGhostDistance.
2. Procedam in maniera similara in ceea ce priveste bucatie de mancare, obtinand astfel minFoodDistance.
3. De asemenea, am calculat timpul minim in care fantomele au fost inactive ( adica au fost speriate de Pacman); aceasta valoare a fost retinuta in variabila minScaredTime.
4. Dupa aceea, in functie de valorile celor 3 parametrii mentionati mai sus am calculat scorurile aferente astfel:
5. Daca fantoma il mananca pe Pacman, ghost\_score ia o valoare foarte mica deoarece inseamna ca pierdem.
6. Daca Pacman se afla in apropierea unei fantome active, ghost\_score va primi un scor negativ invers proportional cu distanta pana la fantoma.
7. Daca Pacman se afla in aporopierea unei fantoma inactive, ghost\_score va primi un scor pozitiv invers proportional cu distanta pana la fantoma. De asemenea, faptul ca am speriat fantoma este un lucru bun, motiv pentru care scared\_score primeste o valoare pozitiva direct proportionala cu jumatatea minScaredTime.
8. Daca in starea curenta au mai ramas bucati de mancare, acest lucru se reflecta intr-un scor negativ direct proportional cu numarul acestora. La acestea se adauga si un punctaj pozitiv direct proportional cu jumatatea distantei minime pana la cea mai apropiata bucata de mancare.
9. La final, se returneaza suma celor 3 scoruri mentionate mai sus adunate cu un procent de 60% din scorul actual al starii.