**Documentație Pacman**

**Am implementat de la proiectul P1 întrebările Q1,Q2,Q3,Q4,Q5,Q6,Q7, iar de la proiectul P3 întrebările Q1 și Q2.**

**Proiect P1:Search**

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

Aici am avut de implementat algoritmul de căutare în adâncime(DFS). Mai întâi am folosit o stivă frontier pentru a stoca nodurile care urmează să fie explorate. Cu ajutorul unei liste ținem evidența stărilor care au fost deja explorate pentru a evita explorarea repetată a acelorași noduri. Se execută un ciclu până când stiva nu mai conține noduri. Dacă stiva conține încă noduri atunci se extrage un nod din vârful stivei, iar dacă starea nodului extras este starea scop atunci se reconstruiește și se returnează traseul de la starea de start la starea scop. Apoi se adaugă starea nodului extras în lista și se adaugă succesorii neexplorați a nodului extras în stivă. Dacă niciun traseu nu a fost găsit până în acest moment atunci se returnează o listă goală.

Algoritmii de DFS,BFS,UCS și A\* sunt foarte asemănători așa că m-am concentrat tare pe obținerea unui DFS corect, iar mai apoi algoritmii de BFS,UCS și A\*, i-am făcut după modelul DFS.

* **Q2: Breadth First Search**

Diferența implementarii algoritmului de BFS față de DFS este ca folosim o coadă, utilizăm un set pentru a memora stările deja explorate și a evita explorarea lor în mod repetat. O altă diferență este apariția unei linii în plus : expanded.add(ns). Această instrucțiune evită adăugarea repetată a stărilor în set și asigură că doar stările noi sunt adăugate în coadă și în set.

* **Q3: Varying the Cost Function**

Funcția uniformCostSearch(UCS) explorează nodurile în ordinea cea mai mică a costurilor totale(expandează nodul cu cel mai mic cost). Se utilizează o coadă de priorități pentru a se adăuga în PriorityQueue fiecare nod cu o prioritate egală cu costul total până la acel nod și un dicționar numit closed\_set pentru a ține evidența costurilor minime până la fiecare stare și pentru a evita explorarea repetată a unor stări. Această funcție spre deosebire de implementarea lui DFS și BFS alege stările cu cele mai mici costuri și oferă o soluție cu costul minim.

* **Q4: A\* search**

Funcția aStarSearch(A\*) combină costul actual a unui nod cu o euristică pentru a ghida căutarea spre soluții eficiente din punct de vedere al costului. Se utilizează la fel ca și la UCS o coadă de priorități. De asemenea se utilizează un set pentru a ține evidența stărilor care au fost deja explorate. Euristica folosită e nullHeuristic care este o euristică trivială. O funcție euristică estimează costul de la starea curentă până la cea mai apropiată stare. Diferența principală dintre A\* și UCS este modul în care se calculează prioritățile nodurilor. La UCS se bazează doar pe costurile reale pentru a determina ordinea în care se explorează, în timp ce la A\* se combină costurile reale cu o euristică anume.

* **Q5: Finding All the Corners**

Aici, noua noastră problemă este de a găsi cea mai scurtă cale prin labirint care să atingă toate cele patru colțuri (indiferent dacă labirintul are sau nu mâncare acolo). Starea inițială este definită ca o tuplă ce conține poziția inițială a lui Pacman și o tuplă cu coordonatele celor patru colțuri. Starea scop este considerată scop atunci când lista cu colțuri rămase este goală. În cadrul metodei getSuccessors se generează stările succesoare pentru o anumită stare. Pentru fiecare acțiune posibilă, adică mișcarea lui Pacman la nord, sud, est și vest se verifică dacă noua poziție lovește sau nu un zid. În cazul în care nu lovește un zid(if not h\_wall: ), atunci se actualizează poziția lui Pacman și lista de colțuri rămase și se generează o nouă stare și se adaugă la lista de succesori. La final se incrementează numărul de noduri explorate(self.\_expanded += 1). Această metodă întoarce o listă de stări, acțiunile asociate acestora și costul.

* **Q6: Corners Problem: Heuristic**

Implementarea euristică cornersHeuristic pentru Corners Problem este o euristică care furnizează o estimare a costului miim al căii de la starea curentă la scopul problemei, acela de a vizita toate colțurile. Aici definesc o funcție auxiliară manhattan\_distance\_to\_pacman care primește un colț și returnează distanța Manhattan de la poziția actuală a lui Pacman la acel colț. Se parcurg toate colțurile rămase în timp ce se actualizează valoarea euristicii în funcție de distanța Manhattan de la Pacman la cel mai apropiat colț, pe care îl găsim cu ajutorul funcției min. După fiecare colț vizitat se actualizează poziția și se elimină colțul din lista colțurilor rămase. La final se returnează valoarea euristicii.

* **Q7: Eating All The Dots**

Implementarea euristică foodHeuristic pentru problema FoodSearchProblem constă în estimarea costului minim al căii de la starea curentă la colectarea tuturor bucăților de alimente din labirint. Se utilizează distanța Manhattan. Definesc o funcție auxiliară manhattan\_distance\_to\_pacman care primește o bucată de hrană ca argument și returnează distanța până la acea bucată. Se inițializează lista remainfood cu toate pozițiile de hrană rămase, se găsește poziția celei mai apropiate bucăți de hrană utilizând funcția min și se returnează la final distanța Manhattan de la poziția actuală a lui Pacman până la cea mai apropiată hrană. La rularea cu autograder am primit 2/4.

**Proiect P3: Multi-Agent Search**

* **Q1: Reflex Agent**

Am implementat un agent reflex pentru joc care folosește o funcție de evaluare simplă pentru a determina cea mai bună acțiune de luat la fiecare pas. Această parte se ocupă de evaluarea stării jocului, acordând scoruri în funcție de distanța până la bucățile de mâncare și cât de aproape se află față de fantome, contribuiind la decizia agentului reflex în alegerea celei mai bune acțiuni. La început am inițializat o varibilă val cu o valoare mare în cazul în care nu există nicio bucată de mâncare disponibilă. Apoi verific dacă numărul de bucăți de hrană este același ca în starea curentă atunci înseamnă că nu au fost consumate noi bucăți de mâncare între starea curentă și starea următoare și atunci se va calcula distanța Manhattan până la cea mai apropiată bucată de mâncare și se calculeaza scorul, alfel scorul va fi setat la 0. Am folosit atunci când atribui distanța Manhattan variabilei the\_score funcțiile min si default care asigură că dacă nu există nicio bucată de hrană scorul va fi setat la valoarea inițială. Apoi am parcurs toate fantomele din starea viitoare și calculăm distanța Manhattan până la poziția lui Pacman și se atribuie un anumit scor în funcție de o distanță: the\_score\_according\_to\_distance=2 if distance==0 else 1 if distance==1 else 0. Ce am abordat eu face ca scorul să crească pe măsură ce Pacman se apropie de fantome și adaug la scorul total o valoare: the\_score+=5\*\*the\_score\_according\_to\_distance (o valoare ridicată la putere), astfel încât distanțele mici să aiba un impact semnificativ asupra scorului. La final returnez scorul final negat, deoarece negarea scorului poate fi o opțiune pentru a face ca agentul să prefere scoruri mai mari.

* **Q2: Minimax**

Aici am avut de implementat algoritmul Minimax pentru Pacman si pentru fantome. Algoritmul Minimax este o strategie de joc perfect pentru jocuri deterministe, cu informație perfecta. Acesta alege mutarea către poziția cu cea mai mare valoare minimax, adică cel mai bun rezultat ce se poate obține împotriva unui jucător perfect. Mai întâi testăm cazurile de bază, adică când adâncimea e 0 sau e o stare de victorie sau înfrângere(în toate aceste cazuri se returnează scorul și o acțiune de stop(se oprește). Apoi obținem toate acțiunile legale pe care le poate efectua agentul curent. Apoi vedem pe rând a cui îi este rândul, a lui Pacman sau a unei fantome:

* Dacă este rândul lui Pacman atunci se explorează toate acțiunile posibile și se alege acțiunea care maximizează scorul. Apoi se generează stările succesoare pentru fiecare acțiune și se calculează scorurile, ca în final să se aleagă cel mai mare scor și acțiunea corespunzătoare acestuia
* Dacă este rândul unei fantome la fel se explorează și se generează stările succesoare, însă se alege acțiunea care minimizează scorul, iar la final se alege cel mai mic scor și acțiunea corespunzătoare

La final se returnează scorul și acțiunea asociată acestuia pentru starea curentă a jocului. Pe scurt acest algoritm Minimax explorează toate posibilitățile de joc până la o anume adâncime și decide cea mai bună acțiune pentru agentul curent luând în considerare mișcările adversarilor.