Pacman – Project 1, Project 3

1. Algoritm de Căutare în Adâncime (DFS)

1.1. Descriere

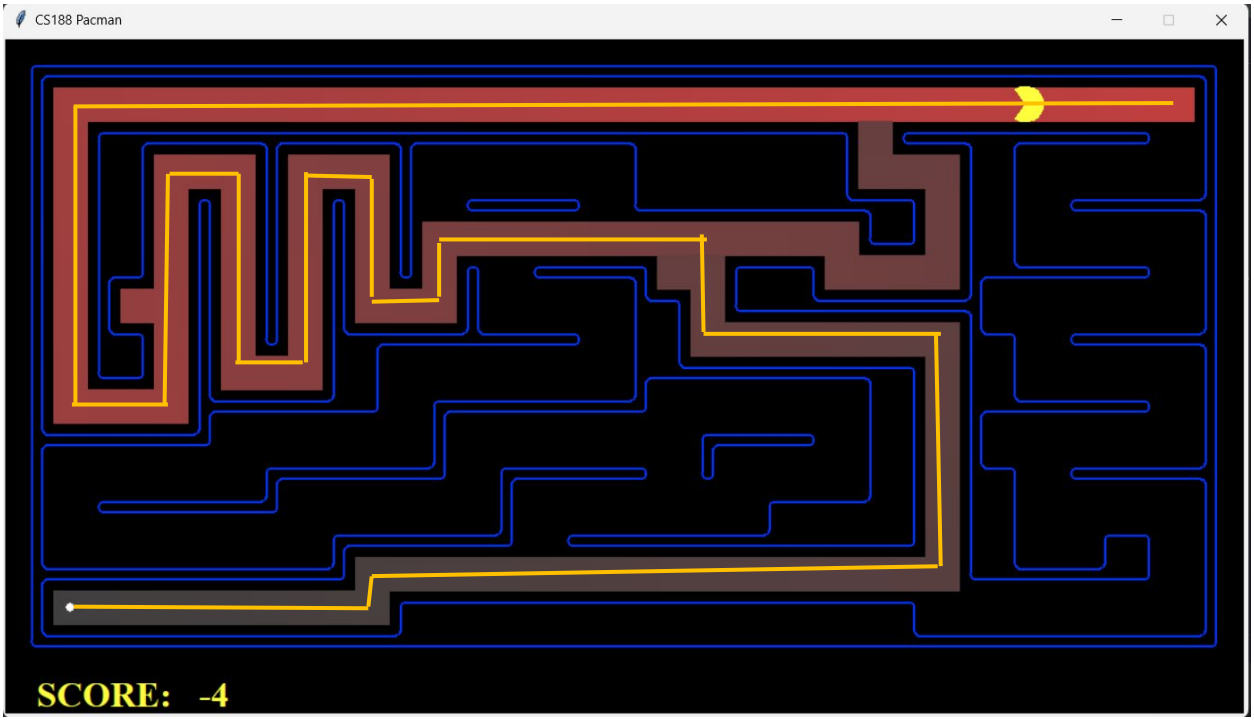
Algoritmul de căutare în adâncime (DFS) explorează cât mai adânc într-un arbore de stări, mergând pe un singur braț al arborelui până la capăt, apoi înapoi. Utilizează o structură de tip stack pentru a gestiona starea curentă și acțiunile asociate.

1.2. Funcționare

- Pornind de la starea inițială, algoritmul alege o acțiune disponibilă și o aplică, ajungând într-o nouă stare.
- Algoritmul adaugă noul nod în stack și repetă procesul până când găsește soluția sau explorează întregul arbore.
- Punerea starilor in stack se face in ordinea in care apar starile folosind functia getSuccesors(), respectiv accesarea lor se face in sens invers

1.3. Avantaje si dezavantaje

- Marele dezavantaj al acestei abordari este ca nu garanteaza ca gaseste un drum optim din niciun punct de vedere, acesta gaseste o cale de la starea initiala si pana la bucata de mancare (starea finala)
- Avantajul este ca pentru a gasi un drum pana la mancare acesta nu are nevoie sa calculeze toate starile jocului



2. Algoritm de Căutare în Lățime (BFS)

3.1. Descriere

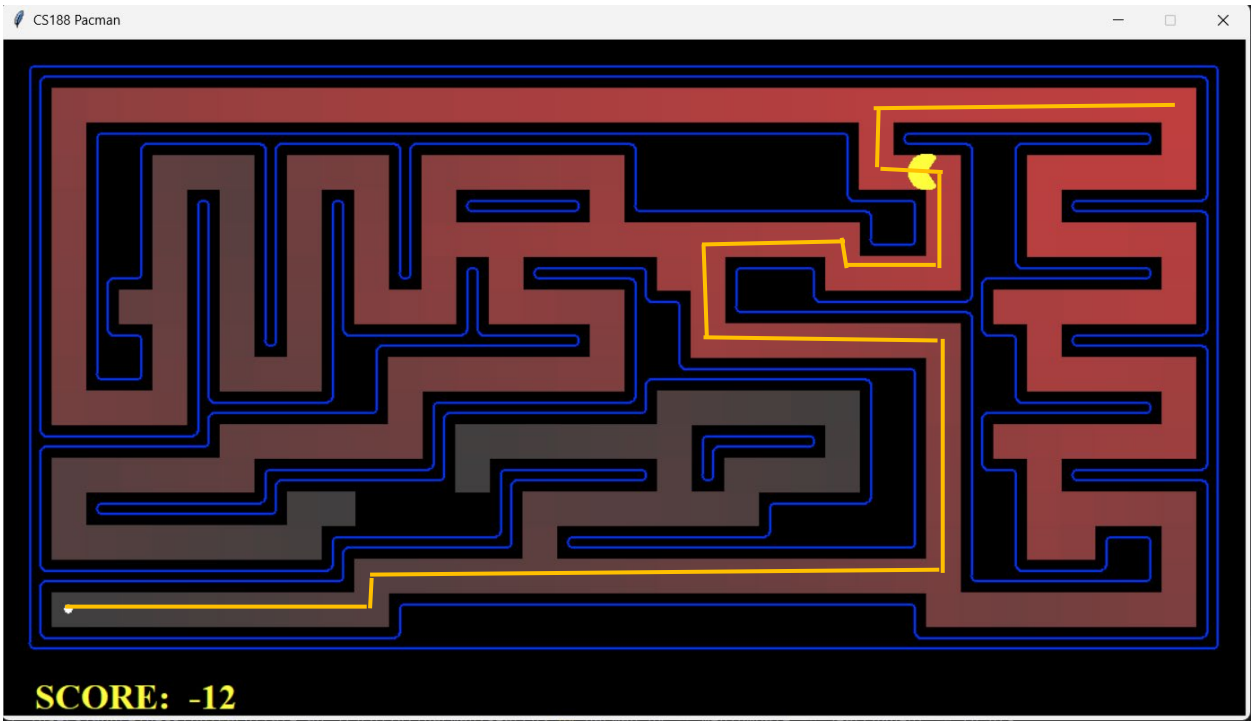
Algoritmul de căutare în lățime (BFS) explorează toate nodurile de pe un nivel înainte de a trece la nivelul următor. Utilizează o structură de tip coadă pentru a gestiona starea curentă și acțiunile asociate.

3.2. Funcționare

- Pornind de la starea inițială, algoritmul adaugă toate acțiunile disponibile în coadă și le explorează pe rând.
- Algoritmul repetă procesul până când găsește soluția sau explorează întregul arbore.

3.3. Avantaje si dezavantaje

- Marele avantaj al acestei abordari este ca gaseste un drum optim din punct de vedere al numarului de stari parcurse, acesta gaseste calea cea mai scurta, care are cel mai mic numar de actiuni de la a trece de la o stare la alta din starea initiala si pana la bucata de mancare (starea finala)
- Dezavantajul consta in parcurgerea unui numar mare de stari pana la a gasi starea finala, algoritmul de bfs expandeaza graful de stari foarte mult in latime => parcurge multe stari inutile



3. Algoritm de Căutare cu Cost Uniform (UCS)

3.1. Descriere

Algoritmul de căutare cu cost uniform (UCS) explorează nodurile în funcție de costul total până la acel moment. Utilizează o coadă de priorități pentru a gestiona starea curentă și acțiunile asociate, ținând cont de costurile. Prioritatea cozii de prioritati este data de costul actiunii de la a trece dintr-o stare in alta, astfel coada se modifica la fiecare adugare a unui element si se sorteaza dupa prioritate

3.2. Funcționare

- Pornind de la starea inițială, algoritmul adaugă nodurile în coada de priorități în funcție de costul total.
- Algoritmul repetă procesul până când găsește soluția sau explorează întregul arbore.

3.3. Avantaje si dezavantaje

- Avantajul acestei abordari este ca gaseste un drum optim din punct de vedere al costului actiunii, acesta gaseste calea cu cel mai mic cost, pornind din starea initiala si pana la bucata de mancare (starea finala)
- Dezavantajul, la fel ca si in cazul algoritmului de cautare in latime(BFS) consta in parcurgerea unui numar mare de stari pana la a gasi starea finala, algoritmul expandeaza graful de stari foarte mult in latime => parcurge multe stari inutile

4. Algoritm A* (Astar)

4.1 Descriere

Algoritmul A* este o extensie a algoritmului UCS care utilizează o euristică pentru a estima costul rămas până la soluție. Euristicile influențează ordinea în care nodurile sunt explorate.

4.2 Funcționare

- Pornind de la starea inițială, algoritmul adaugă nodurile în coada de priorități în funcție de costul total și euristică.
- Algoritmul repetă procesul până când găsește soluția sau explorează întregul arbore.

4.3 Avantaje si dezavantaje

- La fel ca in cazul algoritmului de UCS

5. CornersProblem

5.1. Metoda getStartState

Această metodă returnează starea inițială a problemei. Starea inițială este o tuplă care conține poziția inițială a lui Pacman și o listă de stări pentru fiecare colț, inițializată cu "False" pentru a indica faptul că Pacman nu a trecut încă prin niciun colț.

5.2. Metoda isGoalState

Această metodă verifică dacă starea dată este o stare finală (obiectiv). În acest caz, returnează True dacă nu există False în lista de colțuri (state[1]), indicând că Pacman a trecut prin toate colțurile.

5.3. Metoda getSuccessors

Această metodă returnează o listă de succesori pentru o anumită stare dată (state). Pentru fiecare acțiune posibilă (deplasare într-o direcție), calculează o noua poziție (successorX, successorY) și verifică dacă aceasta pozitie este a unui perete din harta jocului si daca duce pacman-ul in peretele respectiv (hitsWall). Dacă nu, actualizează starea colțului, adăugând True pentru colțurile atinse. Acești succesori sunt adăugați la lista de succesori care va fi returnată.

6. Corners heuristic

Funcția cornersHeuristic reprezintă o euristică utilizată în algoritmul de căutare A* pentru problema CornersProblem. Această euristică estimează costul de a ajunge la starea scop pornind de la o anumită stare, luând în considerare colțurile nevizitate rămase. Valoarea euristicii este suma distanțelor Manhattan de la poziția curentă la cele mai îndepărtate colțuri nevizitate:

- **Verificare Stare Terminala:** Mai întâi, se verifică dacă starea curentă este deja o stare terminala (toate colțurile au fost vizitate). Dacă este adevărat, valoarea euristică este 0.
- **Calcul Estimare:** Dacă scopul nu a fost atins, se calculează valoarea euristică. Pentru fiecare colț nevizitat, se calculează distanța Manhattan de la poziția curentă și se ține evidența celei mai îndepărtate distanțe.
- **Valoare Euristică:** Valoarea euristică este suma distanțelor maxime către colțurile nevizitate. Aceasta reprezintă o estimare optimistă a costului rămas pentru a atinge scopul.

Scopul acestei euristici este să ghideze algoritmul A* pentru a explora căi care probabil duc mai întâi la colțurile nevizitate, îmbunătățind eficiența căutării. Ea încurajează algoritmul să prioritizeze stări care sunt mai aproape de colțurile nevizitate în termeni de distanță Manhattan.

7. Food Heuristic

Funcția foodHeuristic este o euristică utilizată în algoritmul de căutare A* pentru problema FoodSearchProblem. Această euristică estimează costul rămas pentru a atinge starea finala, luând în considerare distribuția rămasă a punctelor de mancare pe harta jocului. Valoarea euristicii este suma distanței maxime între două puncte de mancare și distanța minimă de la poziția curentă la cele două alimente:

- Verificare Stare Scop: Mai întâi, se verifică dacă starea curentă este deja o stare scop (pacman a mancat toate punctele de pe harta). Dacă este adevărat, valoarea euristică este 0.
- Calcul Estimare: Dacă scopul nu a fost atins, se calculează valoarea euristică. Se iau toate perechile posibile de mancare rămase și se calculează distanța Manhattan între acestea. Se ține evidența celei mai mari distanțe și celei mai mici distanțe de la poziția curentă la aceste perechi.

- Valoare Euristică: Valoarea euristică este suma dintre distanța maximă între două puncte și distanța minimă de la poziția curentă la acele două puncte. Aceasta oferă o estimare a costului rămas pentru a atinge scopul.

Scopul acestei euristici este să ghideze algoritmul A* pentru a explora căi care duc probabil către regiuni ale hărții cu grupuri de mancare apropiate.

8. Suboptimal Search

Funcția findPathToClosestDot este utilizată în cadrul agentului ClosestDotSearchAgent pentru a găsi un drum către cel mai apropiat punct de hrană (food) pe harta jocului. Această funcție utilizează căutarea în lățime (breadthFirstSearch) pentru a explora spațiul stărilor și a găsi un drum către cel mai apropiat punct de hrană:

- Inițializare: Se primește starea curentă a jocului gameState, iar problema de căutare asociată acestuia este creată utilizând clasa AnyFoodSearchProblem. Această problemă implică găsirea unui drum către orice punct de hrană.
- Căutare în Lățime: Se aplică algoritmul de căutare în lățime (breadthFirstSearch) pe problema de căutare. Acest algoritm explorează succesiv stările vecine, nivel cu nivel, până când găsește o stare care satisface condiția scopului definită în AnyFoodSearchProblem. În acest caz, scopul este atins când orice punct de hrană este atins.
- Reconstruirea Drumului: După ce s-a găsit o soluție, se reconstruiește drumul de la starea inițială la starea scop. Acest drum reprezintă o secvență de acțiuni care ghidă agentul către cel mai apropiat punct de hrană.
- Returnarea Drumului: Drumul găsit este returnat ca rezultat al funcției și va fi utilizat mai târziu de către agentul pentru a decide următoarea sa acțiune.

Această funcție joacă un rol important în strategia agentului ClosestDotSearchAgent, care își propune să colecteze toată hrana de pe hartă, navigând întotdeauna către cel mai apropiat punct de hrană disponibil.

9. Reflex Agent

Funcția evaluationFunction din clasa ReflexAgent are rolul de a evalua starea curentă a jocului și de a returna un scor asociat acestei stări. Această funcție este folosită pentru a ghida agenții reflex în alegerea celei mai bune acțiuni într-o anumită stare.

- **Input:**
    - **currentGameState**: Starea curentă a jocului.
    - **action**: Acțiunea pe care agentul reflex intenționează să o efectueze în starea curentă.

- **Output:**
    - Un scor numeric care reprezintă evaluarea stării curente. Cu cât scorul este mai mare, cu atât starea este considerată mai favorabilă.
- **Funcționalitate:**
    - Funcția începe prin generarea stării succesoare (**successorGameState**) bazată pe acțiunea specificată.
    - Se extrag informații relevante din starea curentă și starea succesoare, cum ar fi poziția noului Pacman (**newPos**), lista curentă a hranei (**currentFood**), și stările fantomelor (**newGhostStates**).
    - Se calculează distanța Manhattan minimă de la poziția Pacman la cel mai apropiat punct de mancare (**minim**). Distanța este negată pentru a fi utilizată în evaluare.
    - Se verifică dacă oricare dintre fantome este în imediata vecinătate a noii poziții ale lui Pacman sau dacă acțiunea este "Stop". În acest caz, se returnează un scor foarte mic pentru a descuraja astfel de acțiuni.
    - Scorul final este scorul minim negat, ceea ce face ca Pacman să încerce să maximizeze distanța față de cel mai apropiat punct de hrană.

Această funcție servește ca o euristică simplă pentru agentul reflex, oferindu-i informații despre starea curentă și îndrumându-l să ia decizii care să maximizeze o anumită calitate a stării jocului (în acest caz, distanța față de punctul de hrană).

10. Algoritm Minimax (MinimaxAgent)

10.1. Descriere generală:

- Algoritmul Minimax este o metodă de luare a deciziilor folosită în jocurile cu doi jucători, unde scopul este minimizarea pierderii posibile în scenariul cel mai defavorabil.
- Explorează arborele de joc pentru a găsi cea mai bună mutare pentru jucător, presupunând că și adversarul joacă optim.

10.2. Implementare:

- Metoda getAction a clasei MinimaxAgent este punctul de intrare pentru algoritmul Minimax.
- Începe prin a obține acțiunile legale disponibile pentru jucătorul Pacman (agentIndex=0) în starea curentă a jocului.
- Pentru fiecare acțiune, se apelează funcția minValue, care inițiază explorarea recursivă a arborelui de joc.

10.3. **Funcții cheie:**

- getAction**(**gameState**)**: Inițiază algoritmul Minimax, returnând cea mai bună acțiune pentru Pacman.
- maxValue**(**gameState**,** depth**)**: Explorează stratul maxim al arborelui de joc pentru Pacman.
- minValue(gameState, agentIndex, depth): Explorează stratul minim al arborelui de joc pentru fantome.

10.4. **Condiții de oprire:**

- Algoritmul se oprește când se atinge adâncimea maximă (depth == 0) sau când jocul este într-o stare terminală (câștig/pierdere).
- Se apelează funcția evaluationFunction pentru a atribui un scor nodurilor frunză.

10.5. **Utilizarea lui self.depth și self.evaluationFunction:**

- self.depth: Reprezintă adâncimea până la care algoritmul ar trebui să exploreze arborele de joc.
- self.evaluationFunction: Furnizează funcția de evaluare pentru a atribui scor nodurilor frunză.

11. Algoritm Alpha-Beta Pruning (AlphaBetaAgent)

11.1. Descriere generală:

- Alpha-Beta Pruning este o tehnică de optimizare pentru algoritmul Minimax, care reduce numărul de noduri evaluate în arbore.
- Menține două valori, alpha și beta, care reprezintă scorul minim asigurat pentru jucătorul de maximizare și scorul maxim asigurat pentru jucătorul de minimizare, respectiv.

11.2. Implementare:

- Metoda getAction a clasei AlphaBetaAgent este punctul de intrare pentru algoritmul Alpha-Beta Pruning.