UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Pacman

Inteligenta Artificiala

Autori: Zubascu Maria si Zubascu Ileana
Grupa: 30234

Facultatea de Automatica
si Calculatoare

09 Ianuarie 2024

Cuprins

1 Pacman

1.1 Introducere

Pac-Man, numit inițial Puck Man în Japonia, este un joc video de acțiune în labirint din 1980, dezvoltat și lansat de Namco pentru arcade. În America de Nord, jocul a fost lansat de Midway Manufacturing ca parte a acordului de licență cu Namco America. Jucătorul îl controlează pe Pac-Man, care trebuie să mănânce toate punctele din interiorul unui labirint închis, evitând în același timp patru fantome colorate. Mâncatul punctelor mari și intermitente numite "Power Pellets" face ca fantomele să devină temporar albastre, permițându-i lui Pac-Man să le mănânce pentru puncte bonus.

Dezvoltarea jocului a început la începutul anului 1979, regizat de Toru Iwatani cu o echipă de nouă oameni. Iwatani a vrut să creeze un joc care să atragă atât femeile, cât și bărbații, deoarece majoritatea jocurilor video din acea vreme aveau teme de război sau sport. Deși sursa de inspirație pentru personajul Pac-Man a fost imaginea unei pizza cu o felie scoasă, Iwatani a declarat că a rotunjit și caracterul japonez pentru gură, kuchi. Personajele din joc au fost făcute să fie drăguțe și colorate pentru a atrage jucătorii mai tineri. Titlul original japonez Puck Man a fost derivat din expresia japoneză paku paku taberu, care se referă la a înghiți ceva; titlul a fost schimbat în Pac-Man pentru lansarea din America de Nord.

2 Implementare Project 1

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

Acest cod implementează o variantă a algoritmului de căutare în adâncime (DFS) pentru a explora un spațiu de stări într-un anumit context.

Initializare:

- Se inițializează o stivă pentru a gestiona stările ce urmează a fi explorate
- Se adaugă starea de start în stivă
- Se inițializează o listă pentru a verifica dacă o stare a fost deja vizitată
- Se inițializează o listă pentru a reține direcțiile finale care duc de la starea de start la starea scop.
- Se inițializează o a doua stivă pentru a menține calea de la starea de start la o anumită stare.

Loop DFS:

- Se extrage o stare din stivă
- Se verifică dacă starea curentă este starea scop
- Dacă starea nu a fost vizitată: Se adaugă starea la lista de vizitate, se obțin succesori pentru starea curentă, iar fiecare succesor este adăugat în stivă, se actualizează lista temporară a direcțiilor (tempPath) cu noile direcții generate de succesori, lista temporară a direcțiilor este adăugată în a doua stivă pentru a menține calea de la starea de start la starea curentă.

Pasul Următor:

- Se mută la următoarea stare sau se revine în spate (backtrack), extrăgând o nouă stare din stivă și actualizând lista direcțiilor

Rezultat Final:

- Algoritmul continuă această buclă până când se ajunge la starea scop
- Soluția finală constă în lista de direcții (sau acțiuni) care reprezintă calea de la starea de start la starea scop

- Prin urmare, rezultatul returnat este o listă de acțiuni care ajung de la starea de start la starea scop a problemei de căutare specificată

2.2 Q2: Breadth First Search

Acest cod implementează algoritmul de căutare în lățime (BFS), care explorează în primul rând nodurile cele mai superficiale ale arborelui de căutare. Iată o explicație pas cu pas a codului:

Importarea Modulului și Inițializarea:

- Se importă clasa Queue din modulul util
- Se inițializează o coadă pentru a gestiona stările ce urmează să fie explorate
- Se adaugă starea de start în coadă
- Se inițializează o listă pentru a verifica dacă o stare a fost deja vizitată
- Se inițializează o listă pentru a reține direcțiile finale care duc de la starea de start la starea scop
- Se inițializează o a doua coadă pentru a menține calea de la starea de start la o anumită stare

Bucle BFS:

- Se extrage o stare din coadă
- Se verifică dacă starea curentă este starea scop
- Dacă starea nu a fost vizitată: se adaugă starea la lista de vizitate, se obțin succesori pentru starea curentă, iar fiecare succesor este adăugat în coadă, se actualizează lista temporară a direcțiilor (tempPath) cu noile direcții generate de succesori, dar doar dacă succesorul nu a fost deja vizitat, lista temporară a direcțiilor este adăugată în a doua coadă pentru a menține calea de la starea de start la starea curentă.

Pasul Următor:

- Se mută la următoarea stare sau se revine în spate (backtrack), extrăgând o nouă stare din coadă și actualizând lista direcțiilor.

Rezultat Final:

- Algoritmul continuă această buclă până când se ajunge la starea scop
- Soluția finală constă în lista de direcții (sau acțiuni) care reprezintă calea de la starea de start la starea scop.

Rezultatul returnat este o listă de acțiuni care ajung de la starea de start la starea scop a problemei de căutare specificate. Algoritmul BFS explorează în lățime, extinzând succesiv toate stările la adâncimi egale în arborele de căutare.

2.3 Q3: Varying the Cost Function

Acest cod implementează algoritmul de căutare a costului uniform (Uniform Cost Search - UCS), care este utilizat pentru a găsi calea de la starea de start la starea scop cu costul total minim. Iată o explicație pas cu pas a codului:

Initializare:

- Se inițializează o coadă de priorități (PriorityQueue) pentru a gestiona stările cu costul asociat
- Starea de start, o listă goală de acțiuni și un cost total de 0 sunt introduse în coada de priorități
- Se inițializează o listă goală (visited) pentru a ține evidența stărilor vizitate

- Se inițializează o listă goală (path) pentru a stoca secvența finală de acțiuni.

Bucla Principală:

- Algoritmul continuă să exploreze stări până când coada de priorități este goală

Extragerea Stării cu Cel Mai Mic Cost:

- Se extrage starea cu cel mai mic cost total din coada de priorități

Verificarea Dacă Starea a Fost Vizitată:

- Dacă starea nu a fost vizitată:

Vizitarea și Verificarea Stării Scop:

- Se marchează starea ca vizitată
- Se verifică dacă starea curentă este starea scop. Dacă este, soluția este găsită și algoritmul iese din buclă

Explorarea Successorilor:

- Se obțin succesori pentru starea curentă.
- Pentru fiecare succesor, dacă nu a fost vizitat, se calculează costul total pentru a ajunge la acel succesor și se adaugă în coada de priorități.

Returnarea Căii Finale:

- Odată ce starea scop este atinsă, se returnează calea finală ca soluție

Algoritmul de căutare a costului uniform explorează căi în ordine crescătoare a costului total, asigurându-se că se găsește calea optimă (cu cost minim). Folosește o coadă de priorități pentru a da prioritate stărilor cu costuri mai mici.

2.4 Q4: A* search

Acest cod implementează algoritmul A* (A-star), care este utilizat pentru a găsi calea optimă de la o stare de start la o stare scop într-un spațiu de căutare, ținând cont de costul actual și o euristică estimată pentru fiecare stare. Iată o explicație fără cod:

Importul Modulului și Inițializarea:

- Se importă structura de date PriorityQueue din modulul util
- Se inițializează o coadă de priorități pentru a gestiona stările cu costul total și euristica asociate
- Se adaugă starea de start, o listă goală de acțiuni și un cost total de 0 în coada de priorități.

Bucle A:*

- Algoritmul continuă să exploreze stări până când coada de priorități este goală

Extragerea Stării cu Cel Mai Mic Cost și Euristică:

- Se extrage starea cu cel mai mic cost total și euristică din coada de priorități

Verificarea Dacă Starea a Fost Vizitată:

- Dacă starea nu a fost vizitată:

Vizitarea și Verificarea Stării Scop:

- Se marchează starea ca vizitată
- Se verifică dacă starea curentă este starea scop. Dacă este, soluția este găsită, și algoritmul iese din buclă

Explorarea Successorilor cu Euristică:

- Se obțin succesori pentru starea curentă
- Pentru fiecare succesor, dacă nu a fost vizitat, se calculează costul total până la acel succesor, inclusiv euristica pentru acel succesor, și se adaugă în coada de priorități

Returnarea Căii Finale:

- Odată ce starea scop este atinsă, se returnează calea finală ca soluție

Algoritmul A* îmbină costurile găsite până în acel moment cu o euristică pentru a ghida căutarea către soluția optimă. Coada de priorități asigură că stările cu cel mai mic cost total combinat și euristică sunt explorate prima dată. A* este utilizat în probleme de căutare unde optimizarea costului este importantă.

2.5 Q5: Finding All the Corners

Clasa CornersProblem este o problemă de căutare în care obiectivul este ca Pac-Man să găsească o cale care să treacă prin toate cele patru colțuri ale unui labirint. Aceasta extinde clasa search.SearchProblem și definește metodele necesare pentru a implementa această căutare. Constructorul primește starea inițială a jocului, inclusiv poziția de start, pereții labirintului și colțurile. Metodele principale includ getStartState, isGoalState, și getSuccessors. Metoda getStartState returnează starea inițială în spațiul de stări al problemei, isGoalState verifică dacă o anumită stare este o stare scop, iar getSuccessors furnizează stările succesoare posibile, acțiunile necesare pentru a ajunge la ele și costul asociat. Este de asemenea inclusă o metodă auxiliară getCostOfActions pentru a calcula costul unei secvențe de acțiuni.

2.6 Q6: Corners Problem: Heuristic

Funcția cornersHeuristic este o euristică pentru problema CornersProblem definită anterior. Această euristică furnizează o estimare inferioară a celei mai scurte căi de la starea curentă la unul dintre colțurile nevizitate. Explicația pentru codul furnizat este următoarea:

- Se inițializează o listă unvisitedCorners pentru a ține evidența colțurilor care nu au fost încă vizitate.
- Se parcurg colțurile, iar cele nevizitate se adaugă în lista unvisitedCorners.
- Se importă funcția manhattanDistance din modulul util. Această funcție calculează distanța Manhattan între două puncte.
- Se inițializează o listă heuristicvalue cu o valoare inițială de 0, care va fi utilizată pentru a stoca valorile euristice calculate pentru fiecare colț nevizitat.
- Pentru fiecare colț nevizitat, se calculează distanța Manhattan între poziția curentă (poz) și colțul respectiv, iar rezultatul este adăugat în lista heuristicvalue.
- Funcția întoarce valoarea maximă din lista heuristicvalue.

În esență, euristica calculează distanța Manhattan maximă între poziția curentă și colțurile nevizitate, astfel încât să furnizeze o estimare inferioară a costului minim pentru a ajunge la oricare dintre aceste colțuri. Este important ca o euristică să fie admisibilă și consistentă, și această funcție respectă aceste criterii. Admisibilitatea se referă la faptul că euristica întoarce întotdeauna o valoare mai mică sau egală cu costul real al celei mai scurte căi către un obiectiv, iar consistența presupune că valoarea euristicii pentru un nod și un succesor al său nu va depăși niciodată costul dintre aceste două noduri.

2.7 Q7: Eating All The Dots

Despachetarea Stării:

1. **Despachetarea Stării:**
   - `state` este o tupla care conține poziția curentă a lui Pac-Man (`position`) și o grilă cu alimente (`foodGrid`).

- `position` este poziția actuală a lui Pac-Man, iar `foodGrid` este o grilă care indică locurile unde se găsesc alimentele.

2. **Inițializarea Listei de Poziții cu Alimente:**
   - Se obține o listă (`foodPosition`) care conține coordonatele pozițiilor unde se găsesc alimentele din grila `foodGrid`.
3. **Importul Funcției de Distanță Manhattan:**
   - Se importă funcția `manhattanDistance` din modulul `util`, care calculează distanța Manhattan între două puncte într-un plan.
4. **Inițializarea Listei de Valori Euristice:**
   - Se inițializează o listă (`heuristicValues`) cu o valoare inițială de 0, care va stoca valorile euristice calculate pentru fiecare poziție a alimentelor.
5. **Calculul Distanței Manhattan pentru Fiecare Poziție a Alimentelor:**
   - Pentru fiecare poziție a alimentelor, se calculează distanța Manhattan între poziția curentă a lui Pac-Man și poziția alimentului.
   - Valorile calculate sunt adăugate în lista `heuristicValues`.
6. **Întoarcerea Valorii Maximale din Listă:**
   - Funcția întoarce valoarea maximă din lista `heuristicValues`, care reprezintă o estimare a celei mai îndepărtate poziții a alimentelor față de poziția curentă a lui Pac-Man.

Prin estimarea distanței maxime până la cea mai îndepărtată sursă de hrană, euristica încearcă să ghidă algoritmul A* să exploreze mai întâi acele căi care conduc către obiectivele mai îndepărtate. Aceasta contribuie la o eficiență sporită a căutării și la găsirea unei soluții cât mai rapide. Este important să menționăm că această euristică respectă proprietățile de admisibilitate și consistență pentru a asigura corectitudinea algoritmului A*.

3 Implementare Project 2

3.1 Q1: Reflex Agent

1. **Succesor GameState:**
   - Generează starea următoare (`successorGameState`) aplicând acțiunea la starea curentă a jocului.
2. **Extrage Informații:**
   - Extrage informații utile din `successorGameState`, cum ar fi noua poziție a lui Pac-Man (`newPos`), distribuția alimentelor rămase (`newFood`), starea fantomelor (`newGhostStates`), și timpul rămas până la revenirea la starea normală pentru fiecare fantomă (`newScaredTimes`).
3. **Verificare Stare de Victorie:**
   - Dacă starea succesoare este o stare de victorie (toată hrana a fost consumată), returnează un scor foarte mare.
4. **Calcul Distanțe:**
   - Calculează distanțele Manhattan de la noua poziție a lui Pac-Man la hrana rămasă (`foodDistance`) și la poziția fiecărui fantom (`ghostDistance`). De asemenea, calculează distanțele la fantome din starea curentă (`ghostDistanceCurrent`).
5. **Scor Inițial:**
   - Inițializează scorul general.
6. **Scor Relativ:**
   - Adaugă diferența relativă de scor între starea succesoare și starea curentă.

7. **Penalizare pentru Oprire:**
   - Dacă acțiunea este oprire, se aplică o penalizare la scor.
8. **Scor pentru Consumarea Power Pellet:**
   - Dacă Pac-Man consumă o Power Pellet în starea următoare, adaugă un scor semnificativ.
9. **Scor pentru Mai Puține Alimente:**
   - Dacă există mai puține alimente în starea succesoare decât în starea curentă, adaugă un scor.
10. **Penalizare pentru Alimente Rămase:**
    - Pentru fiecare aliment rămas, se scade o penalizare din scor.
11. **Gestionare Fantome:**
    - Dacă fantomele sunt speriate, se preferă o distanță mai scurtă la ele. Dacă nu sunt speriate, se preferă o distanță mai mare la ele.
12. **Returnare Scor Final:**
    - Returnează scorul final calculat ca rezultat al funcției de evaluare.

3.2 Q2: Minimax

- Inițializare:
  - Se obține numărul total de agenți în joc, inclusiv Pac-Man și fantome.
- Funcție Max:
  - Se definește o funcție ‘maxLevel‘ pentru a evalua stările maxime (adversarul încearcă să minimizeze scorul).
  - Dacă starea curentă este o stare de victorie, de înfrângere sau s-a atins adâncimea maximă (‘self.depth‘), se întoarce scorul evaluat folosind funcția de evaluare (‘self.evaluationFunction‘).
  - Se initializează ‘maxvalue‘ cu (infinit negativ).
  - Se parcurg acțiunile posibile ale lui Pac-Man în starea curentă și se calculează scorul maxim dintre succesori (stările minime).
- Funcție Min:
  - Se definește o funcție ‘minLevel‘ pentru a evalua stările minime (Pac-Man încearcă să maximizeze scorul).
  - Dacă starea curentă este o stare de victorie sau înfrângere, se întoarce scorul evaluat folosind funcția de evaluare.
  - Se initializează ‘minvalue‘ cu (infinit pozitiv).
  - Se parcurg acțiunile posibile ale fiecărui agent fantom și se calculează scorul minim dintre succesori (stările maxime).
- Acțiunea Root:
  - Se obțin acțiunile posibile ale lui Pac-Man în starea curentă.
  - Pentru fiecare acțiune, se generează starea succesoare și se apelează ‘minLevel‘ pentru a obține scorul minim al succesivilor.
  - Se alege acțiunea care maximizează scorul succesivilor și se întoarce această acțiune ca rezultatul algoritmului Minimax.

3.3 Q3: Alpha-Beta Pruning

- **Funcția Max (maxLevel):**
  - Evaluează starea maximă (Pac-Man încearcă să maximizeze scorul).

* Dacă starea curentă este o stare de victorie sau înfrângere sau s-a atins adâncimea maximă (`self.depth`), returnează scorul evaluat folosind funcția de evaluare (`self.evaluationFunction`).
* Inițializează `maxvalue` cu $-\infty$ și parcurge acțiunile posibile ale lui Pac-Man în starea curentă.
* Pentru fiecare acțiune, generează starea succesoare și apelează funcția `minLevel` pentru a obține scorul minim al succesivilor.
* Implementează pruning Alpha-Beta: dacă `maxvalue` depășește `beta`, oprește evaluarea și returnează scorul actual. Actualizează `alpha` dacă `maxvalue` este mai mare.

– **Funcția Min (minLevel):**
* Evaluează starea minimă (adversarul încearcă să minimizeze scorul).
* Dacă starea curentă este o stare de victorie sau înfrângere, returnează scorul evaluat folosind funcția de evaluare.
* Inițializează `minvalue` cu ∞ și parcurge acțiunile posibile ale fiecărui agent fantom.
* Pentru fiecare acțiune, generează starea succesoare și apelează funcția `maxLevel` pentru a obține scorul maxim al succesivilor.
* Implementează pruning Alpha-Beta: dacă `minvalue` este mai mic decât `alpha`, oprește evaluarea și returnează scorul actual. Actualizează `beta` dacă `minvalue` este mai mic.

– **Algoritmul Alpha-Beta Pruning (`getAction`):**
* Parcurge acțiunile posibile ale lui Pac-Man în starea curentă și, pentru fiecare acțiune, generează starea succesoare.
* Apelează funcția `minLevel` pentru a obține scorul minim al succesivilor și alege acțiunea care maximizează scorul succesivilor.
* Actualizează `alpha` cu `score` în timpul parcurgerii.
* Returnează acțiunea optimă conform algoritmului Alpha-Beta Pruning.

3.4 Q4: Expectimax

– **Funcția Max (`maxLevel`):**
* Evaluează starea pentru Pac-Man, care încearcă să maximizeze scorul.
* Verifică dacă starea curentă este o stare de victorie sau înfrângere sau dacă adâncimea maximă a fost atinsă. În aceste cazuri, returnează scorul evaluat folosind funcția de evaluare (`self.evaluationFunction`).
* Inițializează `maxvalue` cu $-\infty$ (infinit negativ) și parcurge acțiunile legale ale lui Pac-Man.
* Pentru fiecare acțiune, generează starea succesoare și apelează funcția `expectLevel` pentru a obține valoarea așteptată a succesivilor (fantome).
* Returnează valoarea maximă așteptată.

– **Funcția Expect (`expectLevel`):**
* Evaluează starea pentru un agent (fantome), care aleg acțiuni cu probabilitate uniformă din mișcările legale disponibile.
* Verifică dacă starea curentă este o stare de victorie sau înfrângere. În aceste cazuri, returnează scorul evaluat folosind funcția de evaluare (`self.evaluationFunction`).
* Inițializează `totalexpectedvalue` cu 0 și parcurge acțiunile legale ale agentului curent (fantome).

    - Pentru fiecare acțiune, generează starea succesoare și apelează recursiv funcția `expectLevel` pentru a obține valoarea așteptată.
    - Calculează suma valorilor așteptate și returnează media acestora.
    - Dacă nu există acțiuni posibile, returnează 0.
- **Algoritmul Expectimax (`getAction`):**
    - Parcurge acțiunile legale ale lui Pac-Man în starea curentă și, pentru fiecare acțiune, generează starea succesoare.
    - Apelează funcția `expectLevel` pentru a obține valoarea așteptată a succesivilor și alege acțiunea care maximizează valoarea așteptată.
    - Returnează acțiunea optimă conform algoritmului Expectimax.

3.5 Q5: Evaluation Function

Funcția de Evaluare Îmbunătățită

Această funcție implementează o euristică pentru evaluarea stării curente a jocului Pac-Man într-un mediu de tip *capture-the-flag*. Iată o explicație pas cu pas:

1. **Extrage Informații Despre Starea Jocului:** Funcția extrage informații relevante despre starea curentă a jocului, cum ar fi poziția lui Pac-Man, distribuția hranei, informații despre fantome și timpul rămas până la revenirea la starea normală pentru fiecare fantom.
2. **Calculează Distanța Manhattan la Hrană:** Se calculează distanța Manhattan de la poziția curentă a lui Pac-Man la fiecare bucățică de hrană rămasă în joc.
3. **Calculează Distanța Manhattan la Fantome:** Se calculează distanța Manhattan de la poziția curentă a lui Pac-Man la fiecare fantom din joc.
4. **Componente ale Scorului:** Se calculează numărul de power pellet-uri (capsule) rămase în joc.
5. **Calculul Scorului:** Se initializează variabilele și se pregătește pentru calculul scorului.
6. **Scor Bazat pe Starea Jocului:** Se adaugă la scor scorul curent al jocului, un termen legat de reciproca distanței totale la hrană și numărul de spații care nu conțin hrană.
7. **Ajustarea Scorului în Funcție de Fantome și Power Pellet-uri:** Se ajustează scorul în funcție de timpul rămas până la revenirea la starea normală pentru fantome. Dacă acestea sunt speriate, se încurajează mâncarea lor, în caz contrar, se încurajează evitarea lor.
8. **Returnează Scorul Final:** Funcția returnează scorul calculat, reprezentând dorința stării curente de joc pentru agent.