**Preambul**

In acest proiect lucram pe codul sursă de la proiectul anterior, dar avem cazuri noi de testare (Project 3 – complex search), și formulăm probleme de căutare mai complexe. Trebuie să utilizați funcțiile de căutare scrise la proiectul anterior.

În mod similar ca și la proiect 2, execuția autograderului se face cu comanda

python autograder.py

Din arhiva pusa la dispoziție puteți edita următoarele fișiere, pe care trebuie să le transmiteți ca și răspuns la assignment:

|  |  |
| --- | --- |
| search.py | în care veți pune algoritmii de căutare |
| searchAgent.py | in care veți pune agenții voștri de căutare care se bazează pe algoritmii implementați in search.py |

Se vor modifica doar fișierele indicate. Pentru buna funcționare a autograderului pe Gradescope, recomandarea este sa nu se modifice fișiere suport.

Puteți lucra in echipe de maxim 2 persoane. Doar unul din membri echipei va face transmiterea fișierelor pe Gradescope si in assignment își va indica (pe Gradescope) colegul ca si membru al echipei. De asemenea, in comentariile funcțiilor din cadrul soluției transmise se va indica nominal autorul funcției respective.

Codul sursa transmis pe Gradescope va fi evaluat automat dpdv al corectitudinii tehnice. Nu schimbați numele funcțiilor respectiv a claselor deja existente in fișierele sursa. Pentru buna funcționare a Gradescope este necesar sa păstrați structura proiectului intacta. Scorul final va fi dat de corectitudinea implementării furnizate.

Gradescope permite detectarea similarităților de cod sursa si logica a codului. Prin urmare, va rugam sa lucrați independent si sa nu dați propriile coduri sursa altor colegi. Proiectele transmise care vor fi identificate cu grad ridicat de similitudine vor fi notate cu 0 puncte.

**Întrebări la care trebuie sa răspundeți (prin implementare de cod) in cadrul acestui proiect:**

**Întrebarea 1**. Găsirea colțurilor labirintului.

În colțurile labirintului se găsesc câte un punct / colț, deci în total 4 puncte. Problema pusă pentru această întrebare este să găsiți cea mai scurtă cale prin labirint care să atingă toate cele 4 colțuri (indiferent dacă in labirint găsim sau nu mâncare in fiecare din cele 4 colțuri). Putem observa că pentru unele configurații ale labirintului (precum tinyCorners) calea cea mai scurtă nu ne duce neapărat la colțul cel mai apropiat (pentru tinyCorners calea cea mai scurtă are 28 de pași).

Cerința este să implementați problema de căutare CornersProblem din fișierul searchAgent.py. Pentru funcționarea corectă, agentul trebuie să rezolve următoarele situații:

python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

Pentru a primi nota maximă trebuie să definiți **o reprezentare abstractă a stărilor** care să nu includă informație ne-relevantă (precum poziția strigoilor sau a punctelor de mâncare suplimentare – care nu ne interesează). Nu folosiți GameState pentru a reprezenta o stare de căutare, pentru că va face codul să ruleze foarte încet și incorect. De exemplu, puteți scrie o funcție care să returneze o structura de date care să reprezinte o stare a problemei de căutare. Din implementarea existentă a stării jocului aveți nevoie să obțineți doar poziția de start a lui Pacman și localizarea celor 4 colțuri. De asemenea, când scrieți implementarea pentru getSuccesors puteți adăuga copii la lista de succesori si aceștia să aibă cost 1.

Cu o implementare corectă a BFS, se vor expanda sub 2000 de noduri pentru mediumCorners. Cu euristici proiectate pentru A\* se poate reduce necesarul de stări expandate pentru căutare.

Puteți testa autograderul cu

python autograder.py -q q5

**Întrebarea 2**. Implementați o euristică **consistentă, non-trivială** pentru CornersProblem. Implementarea o veți face in cornersHeuristics. Testarea o puteți face cu

python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

Comanda de mai sus este un shortcut pentru

python pacman.py -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic -z 0.5

O euristică este o funcție care primește la intrare o stare a problemei și returnează un număr care este o estimare a costului către cea mai apropiată stare obiectiv. Euristica este cu atât mai bună cu cât returnează costuri cât mai apropiate de costul actual. Pentru a fi admisibilă, valorile euristicii trebuie să fie limite inferioare ale costului real al căii celei mai scurte către cel mai apropiat obiectiv. Pentru a fi consistentă euristica trebuie să îndeplinească suplimentar următoarea cerință: dacă o acțiune are costul c, atunci prin acțiunea respectivă poate să scadă valoarea euristicii cu maximum c.

În această problemă cel mai important este să găsiți o euristică admisibilă. Dacă aceasta funcționează bine, atunci puteți verifica dacă este si consistentă. Consistența se demonstrează teoretic, însă puteți vedea rapid dacă euristica este inconsistentă verificând valorile f ale nodurilor succesoare care se expandează de la un nod din căutare. De asemenea, dacă UCS și A\* returnează cai cu lungimi diferite, atunci euristica este inconsistentă.

Euristicile triviale sunt cea care returnează 0 in orice stare (adică se face căutare UCS) și euristica prin care se obține costul real. Aceasta ultima va face autograder-ul sa dea time-out. Prin urmare, autograder-ul verifică numărul de noduri și impune o limita rezonabilă de timp pentru căutare.

Pentru a primi puncte, euristica trebuie sa fie non-negativa, consistenta și non-triviala și trebuie sa returneze 0 pentru stările obiectiv. Nota primita va fi invers proporțională cu numărul de noduri expandate.

Puteți testa autograderul cu

python autograder.py -q q6

**Întrebarea 3**. Culegerea tuturor punctelor din labirint

Cerința este să culegeți toate punctele din labirint făcând un număr cat mai mic de pași. Problema este formalizată in clasa FoodSearchProblem din searchAgent.py. O soluție este definită prin calea pe care Pacman o urmează prin care se colectează toate punctele de mâncare din labirint. Nu se consideră strigoii, si toate punctele de mâncare au valoare egală. Dacă aveți implementarea corectă pentru A\*, atunci A\* cu euristica null sau UCS găsește soluția optimă pentru testSearch (cu cost 7).

python pacman.py -l testSearch -p AStarFoodSearchAgent

comandă echivalentă cu

python pacman.py -l testSearch -p SearchAgent -a fn=aStar,prob=FoodSearchProblem,heuristic=foodHeuristic

Deci trebuie sa scrieți foodHeuristic in searchAgent.py. Puteți încerca agentul pe trickySearch

python pacman.py -l trickySearch -p AStarFoodSearchAgent

Agentul UCS implementat corect explorează peste 16000 noduri. Orice euristica non-triviala primește puncte. In funcție de numărul de noduri expandate, se obține punctajul maxim pentru o euristică care explorează cel mult 9000 de noduri. Daca găsiți o euristică să exploreze mai puține noduri, primiți puncte suplimentare.

Testarea autograderului se face cu

python autograder.py -q q7

**Întrebarea 4**. Căutare suboptimală

In unele cazuri, A\* chiar si cu o euristica bună, va avea probleme să găsească calea optimă pentru a culege toate punctele. Deci avem nevoie să găsim o cale rezonabilă in timp scurt. Deci trebuie să scrieți un agent care întotdeauna mănâncă (in mod greedy) cel mai apropiat punct de mâncare. Aveți ClosestDotSearchAgent implementat, dar lipsește funcția care gâcește calea către cel mai apropiat punct de mâncare.

Implementați funcția findPathToClosestDot in searchAgent.py. Pentru labirintul bigSearch se găsește rapid o cale cu un cost de 350.

python pacman.py -l bigSearch -p ClosestDotSearchAgent -z 0.5

Remarcați faptul ca ClosestdotSearchAgent nu găsește întotdeauna cea mai scurta cale prin labirint.

Testarea autograderului se face cu

python autograder.py -q q8