**Projecte 2: Cerca dins del món Menjaclosques d'en Pacman**

**Introducció**

En aquest projecte, el vostre agent Pacman trobarà camins a través del seu laberíntic món, tant per arribar a una determinada posició com per recollir el menjar de forma eficient. Heu de construir algorismes de cerca generals i aplicar-los als escenaris de Pacman.

El codi per a aquest projecte consisteix en diversos fitxers Python, alguns dels quals haureu de llegir i comprendre de cara a completar el projecte i alguns dels quals podeu ignorar.

**Podeu descarregar el codi i els fitxers de suport** [aquí](https://campusvirtual2.ub.edu/mod/resource/view.php?id=209316) .

|  |  |
| --- | --- |
| **Fitxers que heu editar:** | |
| search.py | En el qual residiran tots els vostres algorismes de cerca. |
| searchAgents.py | En el qual residiran tots els vostres agents cercadors. |
| **Fitxers que és interessant que entengueu:** | |
| pacman.py | Fitxer principal que executa els jocs de Pacman. Aquest fitxer descriu un tipus GameState, que podeu utilitzar en aquest projecte. |
| game.py | La lògica que controla com funciona el món de Pacman. Aquest fitxer descriu diversos tipus de suport com AgentState, Agent, Direction i Grid. |
| util.py | Estructures de dades útils per implementar algorismes de cerca. |
| **Fitxers que no cal que entengueu:** | |
| graphicsDisplay.py | Gràfics del Pacman |
| graphicsUtils.py | Suport per als gràfics del Pacman |
| textDisplay.py | Gràfics ASCII del Pacman |
| ghostAgents.py | Agents per controlar els fantasmes |
| keyboardAgents.py | Interfície teclat per controlar Pacman |
| layout.py | Codi per llegir els fitxers de layout i guardar els seus continguts. |

**Què heu d'enviar?:** Omplireu fragments de search.py i searchAgents.py durant el projecte. Heu d'enviar únicament aquests dos fitxers, juntament amb un document descriptiu del treball realitzat, en un. Tar.gz. a través del campus virtual.

La memòria ha de contenir taules comparatives dels resultats en els diferents escenaris i diferents algorismes usats, en temps, cost i nodes expandits.

**Benvinguts a Pacman**

Després de baixar el codi , descomprimir-lo i entrar al directori *search,* hauríeu de poder jugar una partida de Pacman escrivint en la línia de comandes:

python pacman.py

El primer pas per en Pacman serà ser capaç de navegar en el seu món de forma eficient.

L'agent més senzill en searchAgents.py es diu GoWestAgent , i sempre va cap a l'Oest (és un agent reactiu trivial). Aquest agent pot guanyar en alguna ocasió:

python pacman.py --layout testMaze --pacman GoWestAgent

Però quan cal girar, les coses es posen lletges:

python pacman.py --layout tinyMaze --pacman GoWestAgent

Aviat, el vostre agent resoldrà no només tinyMaze , sinó qualsevol altre laberint que vulgueu. Observeu que pacman.py suporta un nombre d'opcions que es poden expressar d'una forma llarga (pe, --layout ) o d'una forma curta (pe, -l ). Podeu visualitzar la llista de totes les opcions i els seus valors per defecte mitjançant:

python pacman.py -h

Tots els comandaments que apareixen en aquest projecte apareixen també en commands.txt , perquè us sigui més senzill el copiar i enganxar. Fins i tot podríeu executar tots seguits si volguéssiu amb la instrucció bash commands.txt .

**Trobant un Floc de Menjar Fix utilitzant Algorismes de Cerca**

En searchAgents.py , trobareu un SearchAgent , completament implementat que planeja el seu camí a través del món de Pacman i després executa el camí pas a pas. Els algorismes de cerca per formular un pla no estan implementats - aquest és el vostre treball. Al final del document podeu trobar un glossari d'objectes en el codi que potser necessitareu a  mesura que avanceu al llarg de les següents preguntes. En primer lloc, verificar que el SearchAgent funciona correctament executant:

python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch

La comanda indica al SearchAgent que utilitzi tinyMazeSearch com el seu algorisme de cerca. tinyMazeSearch està implementat en search.py . Pacman hauria de navegar pel laberint amb èxit.

Ara ha arribat el moment d'implementar funcions de cerca genèriques per ajudar a en Pacman a planificar les seves rutes. El pseudocodi per als algorismes que heu d'escriure els podeu trobar a les transparències de teoria i en el llibre. Recordeu que un node de recerca no ha de contenir únicament un estat sinó també la informació necessària per reconstruir el camí (pla) que el porti a aquest estat.

*Nota important;* Totes les vostres funcions de cerca han de retornar una llista *d'accions* que portin a l'agent des de l'inici fins l'objectiu. Aquestes accions han de ser moviments legals (direccions vàlides, no ens podem moure travessant murs).

*Ajuda:* Tots els algorismes són molt similars. Els algorismes per DFS, BFS, UCS, i A \* es diferencien tan sols en els detalls de com es gestiona la frontera. Per tant, concentreu-vos en implementar DFS bé i la resta hauria de ser relativament fàcil. De fet, una possible implementació només necessitaria d'una funció genèrica de recerca que es configurés en cada cas amb un estratègia d'encuament diferent (la vostra implementació no té perquè ser així).

*Ajuda:* Assegureu-vos d'entendre bé les classes Stack, Queue i PriorityQueue posades a la vostra disposició en util.py !

***Qüestió 1:*** Implementeu l'algorisme de cerca de profunditat prioritària en la funció depthFirstSearch en search.py . Per fer que el vostre algorisme sigui *complet,* codifiqueu la versió de DFS que realitza recerca en un graf, evitant expandir estats ja visitats (llibre de text secció 3.5).

El vostre codi hauria de trobar fàcilment una solució per:

python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

python pacman.py -l bigMaze -z 0.5 -p SearchAgent

El tauler de Pacman us mostrarà com es van explorant els estats. Vermell brillant significa els estats explorats primerament. Segueix l'ordre d'exploració que haguéssiu esperat? En Pacman passa per tots les caselles explorades quan segueix el seu camí cap a l'objectiu?

*Ajuda:* Si utilitzeu un Stack com a estructura de dades, la solució trobada pel vostre algorisme DFS per mediumMaze hauria de tenir una longitud de 130 (suposant que afegiu els successors a la frontera en l'ordre en que ens els dóna getSuccessors; podríeu obtenir 244 si els afegiu en ordre invers). És una solució de cost mínim? En cas de no ser així, què està fent malament la recerca primer en profunditat?

***Qüestió 2:*** Implementa l'algorisme de cerca Amplitud prioritària (BFS) en la funció breadthFirstSearch en search.py . De nou, escriviu l'algorisme que realitzi la cerca en grafs, evitant expandir estats ja visitats. Verifiqueu el vostre codi de la mateixa manera que ho vau fer per a la recerca primer en profunditat.

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z 0.5

Troba BFS una solució de cost mínim? Si no, verifiqueu la vostra implementació.

*Ajuda:* Si en Pacman es mou lentament, proveu l'opció --frameTime 0 .

*Nota:* Si heu escrit el codi de cerca de forma genèrica, el vostre codi hauria de funcionar també amb el problema de cerca del 8-puzzle (llibre de text secció 3.2) sense que hageu de realitzar cap canvi.

python eightpuzzle.py

**Trobant Totes les Cantonades**

Ara anem a formular un nou problema. En *laberints amb cantonades,* hi ha quatre flocs de menjar, un a cada cantonada. El nostre nou problema de recerca és trobar el camí més curt a través del laberint que toqui les quatre cantonades (tant si el laberint realment té menjar allà com si no). Observeu que per a alguns laberints com tinyCorners , el camí més curt no sempre va primer al menjar més proper. *Ajuda:* el camí més curt a través de tinyCorners té 28 passos.

***Qüestió 3:*** Implementeu el problema de cerca CornersProblem en searchAgents.py . Necessitareu triar una representació d'estat que codifiqui tota la informació necessària per detectar si les quatre cantonades s'han visitat. Ara, els vostres agents de recerca hauran de resoldre:

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

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

Per tenir bona nota, heu de definir una representació d'estat abstracta que *no* inclogui informació irrellevant (com la posició dels fantasmes, on hi ha menjar extra, etc). En particular **no heu usar un** GameState **de Pacman com a estat de** cerca. Si ho feu vostre codi serà molt, molt lent (al mateix temps que incorrecte).

*Ajuda:* Les úniques parts de l'estat del joc que heu de referenciar a la vostra implementació són la posició inicial de Pacman i la localització de les quatre cantonades.

La nostra implementació de breadthFirstSearch s'expandeix just una mica menys de 2000 nodes en mediumCorners . Tot i això, l'ús d'heurístiques (juntament amb l'ús de A \* més endavant) pot reduir la quantitat de cerca requerida.

**Cerca A \***

***Qüestió 4:*** Implementeu la recerca en grafs A \* en la funció buida aStarSearch en search.py . A \* ha de rebre una funció heurística com a paràmetre. La heurística rep dos paràmetres: un estat del problema de recerca (el paràmetre principal) i el problema en si mateix (com a informació de referència, podria fer falta en algun cas). La funció heurística nullHeuristic en search.py és un exemple trivial.

Podeu provar la implementació d'A \* en el problema de trobar el camí en un laberint fins a una posició fixa utilitzant com heurística la distància Manhattan (heurística ja implementada com manhattanHeuristic en searchAgents.py ).

python pacman.py -l bigMaze -z 0.5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic. Què passa a openMaze per les diferents estratègies de cerca?

La potència d'A \* només quedarà de manifest amb un problema de recerca més complex proveu a dissenyar una heurística pel problema dels quatre corners:

***Qüestió 5:*** Implementeu una heurística per al CornersProblem en cornersHeuristic . Si expandís menys de 800 nodes és que ho esteu fent molt bé.

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

*Ajuda:* Recordeu, les funcions heurístiques únicament retornen números. Per ser admissibles, aquests números han inferiors al camí ral més curt a l'objectiu més proper.

*Nota:* AStarCornersAgent és un àlies per -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic .

**Menjant Tots Els Punts**

Ara resoldrem un problema de recerca difícil: menjar tots els flocs de menjar en tan pocs passos com sigui possible. Per això necessitarem definir un nou problema que formalitzi el problema d'acabar amb tot el menjar: FoodSearchProblem en searchAgents.py (heu de implementar-ho vosaltres). Una solució es defineix com un camí que recull tot el menjar en el món de Pacman. Per al projecte actual, les solucions no tenen en compte cap fantasma ni els flocs de poder. Les solucions només depenen de la col·locació dels murs, del menjar i d'en Pacman. Si heu escrit els vostres mètodes de cerca general correctament, A \* amb l'heurística nul (equivalent a la recerca de cost uniforme) hauria de trobar una solució òptima a testSearch sense cap canvi de codi per la vostra part (cost total de 7).

python pacman.py -l testSearch -p AStarFoodSearchAgent

*Nota:* AStarFoodSearchAgent és un àlies per -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic .

Haureu vist que la cerca comença a alentir-se fins i tot per l'aparentment simple tinySearch.

***Qüestió 6:*** Omplir foodHeuristic en searchAgents.py amb una heurística consistent per FoodSearchProblem . Proveu el vostre agent amb el tauler trickySearch :

python pacman.py -l trickySearch -p AStarFoodSearchAgent

La nostra implementació troba la solució òptima en uns 13 segons, explorant més de 16.000 nodes. La puntuació dependrà del nombre de nodes que expandeixi l'heurística.

Si l'heurística és inadmissible tindreu un 0, així que aneu amb compte. Penseu amb cura en l'admissibilitat, ja que heurístiques inadmissibles poden produir cerques ràpides i fins i tot camins òptims. Podeu resoldre mediumSearch en un curt espai de temps? Si és així o bé hauríeu d'estar molt sorpresos per l'excel·lent heurística o bé la vostra heurística és inadmissible.

*Admissibilitat vs.* *Consistència?* Tècnicament, l'admissibilitat no és suficient per garantir la correctesa en la recerca en grafs - necessitareu que l'heurística sigui a més consistent. Per a què una heurística sigui consistent ha de passar que si una acció té cost *c* llavors prenent aquesta acció, només pot causar una caiguda en l'heurística de com a molt *c.* Si la vostra heurística no és només admissible sinó també consistent, tindreu una millor puntuació.

Gairebé sempre, les heurístiques admissibles són també consistents, especialment si es deriven de relaxacions del problema. De manera que potser el millor és començar amb un brainstorming d'heurístiques admissibles. Un cop tingueu una heurística admissible que funcioni prou bé podeu verificar si també és consistent. La inconsistència pot en alguns casos ser detectada verificant que el valor de *f* de les solucions retornades no decreix. L'assumpte és delicat, així que heu de ser curosos.

**Glossari d'Objectes**

Aquí teniu un glossari dels objectes clau en el codi que estan relacionats amb problemes de cerca::

SearchProblem (search.py)

Un SearchProblem és un objecte abstracte que representa l'espai d'estats, la funció successor, costos i estat objectiu d'un problema. Interactuareu amb qualsevol SearchProblem únicament a través dels mètodes definits en la part superior de search.py

PositionSearchProblem (searchAgents.py)

Un tipus de SearchProblem específic amb el qual treballareu - correspon a buscar un únic floc en un laberint.

CornersProblem (searchAgents.py)

Un tipus de SearchProblem específic que definireu - correspon a buscar un camí que passi per les quatre cantonades d'un laberint.

FoodSearchProblem (searchAgents.py)

Un tipus de SearchProblem específic amb el qual treballareu - correspon a cercar la manera de menjar-se tots els flocs d'un laberint.

Search Function

Una funció de cerca és una funció que pren com a paràmetre una instància de SearchProblem, corre algun algorisme i retorna una seqüència d'accions que condueixen a l'objectiu. Exemples de funcions de cerca són depthFirstSearch i breadthFirstSearch , que heu d'escriure. S'inclou tinyMazeSearch que és una funció de cerca molt dolenta que només funciona correctament en tinyMaze.

SearchAgent

SearchAgent és una classe que implementa un Agent (un objecte que interactua amb el món) i realitza la seva planificació a través d'una funció de cerca. El SearchAgent utilitza en primer lloc la funció de cerca rebuda per realitzar un pla de les accions necessàries per aconseguir l'estat objectiu i, a continuació, executa les accions una a una.