Pràctica 1
Pacman

Ferran Aran Domingo
Oriol Agost Batalla

48253731F
48257094N

Oriol Agost Batalla 48257094N
Ferran Aran Domingo 48253731F

Índex

Oriol Agost Batalla 48257094N
Ferran Aran Domingo 48253731F

Introducció

En aquesta pràctica se'ns demana demostrar els coneixements que hem aprés sobre els algorismes de cerca informada i no informada que se'ns han donat fins el moment. És per això que haurem de implementar 4 problemes, descrits a continuació, juntament amb la resposta que proporcionem.

Problema 1: Algorisme A*

En aquest exercici se'ns demana implementar l'algorisme A*, modificant la funció *aStarSearch* del fitxer *search.py*.

Per a fer això, el que hem fet es donat el codi del algorisme UCS, fer que ara la prioritat amb la que posem un node al *fringe* sigui no només el cost acumulat d'aquest com en l'UCS, sinò aquest cost sumat al valor que ens retorna l'heurística, que es passa per paràmetre:

```
fringe.push(succ_node, succ_node.cost + heuristic(state, problem))
```

Amb aquest canvi, podem transformar l'algorisme UCS en l'A*. La heurística serà aquella que passem per paràmetre a l'hora d'executar el programa. La crida per a fer-ho amb la distància de Manhattan serà:

```
$ python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

I amb la distància Euclidea:

```
$ python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=euclideanHeuristic
```

Canvi en UCS

Com que per definició, l'algorisme UCS és l'algorisme A* amb l'heurística nul·la, podem implementar precisament aquest canvi en el nostre codi, i evitar tenir dos fragments pràcticament iguals en el nostre programa. És per això que ara en la funció UCS posarem que retorni la pròpia funció A*, sense especificar cap heurística, ja que la que té per defecte és la nul·la, la que volem:

```
def uniformCostSearch(problem: SearchProblem):
    """Search the node of least total cost first."""
    return aStarSearch(problem)
```

Resultats

Mapa	Nodes expandits - manhattanHeuristic	Nodes expandits - euclideanHeuristic
tinyMaze	14	13
mediumMaze	221	226
bigMaze	549	557

Problema 2: Heurística Corners Problem

En aquest problema se'ns demanava implementar una heurística consistent i no trivial per al *Corners Problem*, modificant la funció *cornersHeuristic* del fitxer *searchAgents.py*.

Per a fer-ho, em implementat una heurística que partint del punt en el que es troba el *Pacman* en l'estat rebut per paràmetre, calcula la menor de les distàncies de Manhattan a les fruites que encara no ens hem menjat, i la suma al total. Després, repetim el mateix procediment, tenint en compte que aquella posició de la qual partim ara és la de la fruita que hem elegit.

Aquest procediment és repeteix mentre quedin fruites per les que encara no hem passat.

Finalment, retornem aquest cost acumulat; és a dir, la suma de les menors de les distàncies de Manhattan a cada possible fruita des d'on ens trobem, per cada fruita que no haguem menjat.

Si ens fixem, el valor que retorna la nostra heurística és el que s'obté aplicant una cerca 'El primer el millor' amb la restricció dels moviments del pacman (moure's amb el format de la distància de Manhattan), ja que sempre escollirem el menjar amb menor cost sobre qualsevol altra opció.

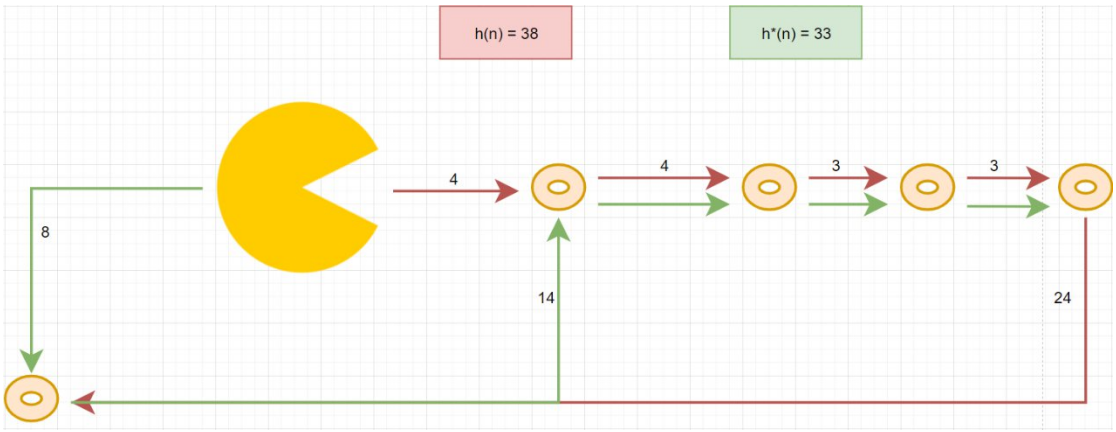
Admissibilitat

Recordem, una heurística és admissible si:

$$\forall n\ h(n) \leq h*(n)$$

Com que la nostra heurística fa una cerca 'El primer millor' imitant els moviments del Pacman, però tenint en compte sempre que no hi haurà cap mur al mapa, sempre estarem sent optimistes ja que considerarem el cas més favorable (no hi ha murs al mapa), i això aplicat al recorregut que passa per tots els punts suposa que sempre estarem donant un cost inferior al real.

Cal destacar que la heurística és admissible degut a la distribució dels menjars, ja que si aquests es trobessin distribuïts pel mapa enlloc d'estar a les quatre cantonades, una cerca 'El primer millor' podria donar-nos un cost superior al òptim tal com es veu a la següent figura, suposant un *layout* on no tenim cap mur:



Fifgura 1: La heurística retorna un valor superior al òptim degut a la distribució concreta de les peces de menjar.

Però en el cas del *Corners Problem* no es dona aquesta situació, ja que al estar els menjars situats a les quatre cantonades no és possible que es doni un escenari en que una cerca 'El primer millor' ens doni un cost superior al òptim, tenint en compte que el que valor que retornarà aquesta seria el cost òptim si no hi hagués murs, per tant sempre serà menor o igual al cost real.

Consistència

En quant a la consistència, sabem que una heurística serà consistent si per qualsevol parell de nodes:

$$h(n) \leq k(n,n') + h(n')$$

Sabem que es complirà pel fet que si ens trobem en un punt A i anem a un punt B, el valor de la heurística en el punt A serà el cost òptim del camí que seguiria el Pacman per menjar-se tots els menjars si no hi hagués murs. Per tant, si al valor de la heurística en el punt B li sumem el cost d'haver anat d'A a B, ara tindrem altra vegada el cost òptim del camí que seguiria el Pacman en les condicions esmentades **més** el cost d'haver anat d'A a B, per tant si A estava situat més a prop de

l'objectiu que B, sabem que h(A) serà inferior a k(A, B) + h(B). I si A estava situat a la mateixa distància que B de l'objectiu, h(A) serà igual a k(A, B) + h(B).

Les comandes seran del estil:

```
$ python3 pacman.py -l <layout> -p AStarCornersAgent -z 0.5
```

Resultats

Mapa	Nodes expandits
tinyCorners	153
mediumCorners	691
bigCorners	1716

Problema 3: Heurística Food Search Problem

En aquest problema se'ns demanava implementar una heurística consistent i no trivial per al *FoodSearchProblem*, modificant la funció *foodHeuristic* del fitxer *searchAgents.py*.

Com ja hem comentat en el problema anterior, no podem usar la heurística que havíem dissenyat per al *cornersProblem* ja que aquesta no seria admissible, per tant hem optat per fer-ne una de més simple, però que tot i així aconsegueix reduir el nombre de nodes expandits.

Aquest cop la heurística consisteix en, donada una posició inicial, calcular totes les distàncies de Manhattan als diferents menjars que quedin al mapa. I de totes aquestes distàncies obtingudes, quedar-nos amb la major d'elles.

Per tant, un cop més, hem de tornar a la heurística més simple que se'ns ha acudit, ja que malauradament és la única que podem demostrar com a consistent i admissible, encara que les dues anteriors donen bons resultats i un camí òptim en els escenaris del Pacman, on no es donen situacions com les de les figures.

Admissibilitat

En aquest cas és més fàcil demostrar l'admissibilitat, ja que el valor que retornem sempre és una distància de Manhattan a una casella amb menjar (a la més llunyana concretament). Per tant, en el millor dels casos, la casella destí serà la última amb menjar i no hi haurà murs entremig, es a dir que $h(n) = h^*(n)$. En la resta de casos, serà inferior, per tant compleix $\forall n\ h(n) \leq h^*(n)$.

Consistència

Serà consistent ja que al estar treballant amb distàncies tant en la heurística com en el cost real, si tenim tres punts A, B i C, on C és el punt més llunya tant d'A com de B, i B està més a prop de C que A, la diferència entre la heurística de A i B serà la distància entre aquests dos punts, per tant, tenim que per qualsevol parell de caselles estarem satisfent $h(n) = k(n,n') + h(n')$, que a la vegada satisfà $h(n) \leq k(n,n') + h(n')$.

Resultats

Mapa	Nodes expandits
testSearch	12
trickySearch	9551

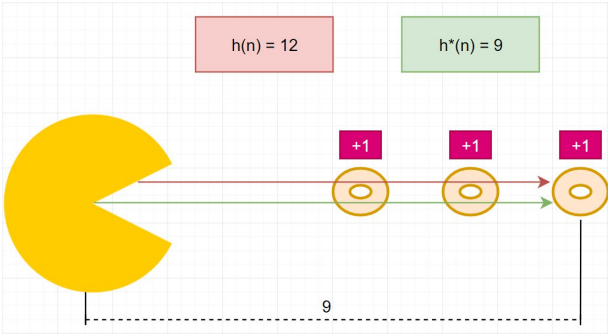
Heurístiques descartades

Híbrida distància Manhattan – quantitat de menjar restant

Vam estar pensant de fer anar la mateixa heurística que en el cas anterior, calculant les distàncies de Manhattan i posant-les en una llista, però a l'hora de retornar el resultat retornar la màxima d'aquestes sumat a la quantitat de menjar restant donat l'estat passat. D'aquesta manera aconseguíem donar una petita empenta a aquelles posicions que ja havien menjat més menjar que les anteriors, assolint un menor nombre de nodes expandits.

Va ser després quan ens vam adonar que aquesta modificació feia que la nostra heurística deixés de ser admissible. Per exemple, en el cas en el que ens queda només un ítem de menjar per menjar i aquest està a 100 unitats de distància, sense cap mur entremig, la heurística perfecta és la longitud del camí que ens queda per recórrer – 100. En el nostre cas però, la nostra heurística retornava 101, ja que encara quedava un ítem per menjar. És això el que ens va fer decidir descartar aquesta.

També vam provar a fer que fos la màxima de les distàncies més la quantitat de menjar restant menys 1, pensant que així potser arreglàvem el problema. Tampoc era el cas, ja que si per exemple ens quedaven per menjar tres menjars, que estaven davant nostre, sense murs en mig; la heurística nostra heurística hauria retornat un valor que superaria per 3 unitats al òptim tal com es veu a la figura. També vam descartar aquesta.



Fifgura 2: La heurística retorna un valor superior al òptim degut al nombre i la distribución de les peces de menjar restants

Sub-problema de cerca com heurística

Vam valorar que la millor heurística possible era aquella que assolia aproximar el que ens quedava per acabar el problema, i amb aquesta idea en ment, vam implementar una en que es calculava un sub-problema de cerca donat l'estat actual del problema, amb el BFS com a funció. D'aquesta manera assolia estimar correctament quan quedava per acabar el problema, expandint molts menys nodes.

Tot i així, els nodes que ens estalviàvem expandint aquest els expandíem en el sub-problema i a més a més, era una heurística exponencial, en la qual la magnitud del còmput d'aquesta creixia a mesura que creixien les posicions de menjar en el tauler.

A més a més, vam contactar amb el professorat i ens van indicar que a la fi i al cap, no obteníam cap millora i a més a més, una heurística havia de ser una aproximació ràpida que ens apropés al resultat final, no el càlcul d'aquest.

Per aquestes dues raons, també vam descartar-la.

Oriol Agost Batalla 48257094N
Ferran Aran Domingo 48253731F

Problema 4: Cerca subòptima

En aquest problema se'ns demanava implementar un agent que sempre vagi a menjar-se l'ítem de menjar que tingui més a prop, modificant la funció *findPathToClosestDot* de la classe *ClosestDotSearchAgent*, dins del fitxer *searchAgents.py*.

Test objectiu AnyFoodSearchProblem

Per a fer això, primerament ens hem fixat en que el tipus de problema que usava aquest agent era el *AnyFoodSearchProblem*, el qual representa un problema de cerca que busca un camí fins a qualsevol menjar. D'aquest, havíem de modificar el test objectiu, que no venia implementat i ens ajudaria amb la implementació del nostre agent.

Aquest problema té un atribut que és una graella que representa el tauler de la partida, on té cert o fals depenent de si hi ha o no menjar en aquella casella, respectivament.

Com que com hem dit abans el nostre objectiu és menjar-nos qualsevol ítem de fruita, fem que això quedi reflectit en el test objectiu, retornant cert si la posició del Pacman (que tenim en el paràmetre *state*) té cert en la graella que representa el menjar; és a dir, si està en una casella que tingui menjar:

```
def isGoalState(self, state):
    x, y = state

    return self.food[x][y]
```

Implementació findPathToClosestDot

Un cop determinat el test objectiu del nostre problema toca implementar l'agent.

La funció que nosaltres implementem ha de retornar un camí, una llista d'accions fins la fruita més propera. Si ens fixem, la funció *total_path()* de *node.py* fa precisament això. Per tant, podem fer que la nostra implementació simplement cridi a un algorisme de cerca de tots els que hem implementat.

Però quin?

- El DFS no és la millor opció, ja que no és òptim a l'explorar per branques
- El A* i UCS ens donarien la solució òptima però com que fan el test objectiu al treure els nodes del *fringe* i no en generar-los, no serà la millor opció.
- El BFS és la millor opció que tenim, ja que com que els costs de pas són idèntics (1), i aquest explora per nivells, sempre ens retornarà la solució òptima, que serà la primera que es trobarà. A més a més, com que té el test en successors, expandirà menys nodes que el UCS o el A*, per tant ens decantem per aquest algorisme

Finalment, l'únic que hem de fer és retornar la cerca en BFS del problema que se'ns proporciona:

```
def findPathToClosestDot(self, gameState):
    problem = AnyFoodSearchProblem(gameState)

    return search.bfs(problem)
```