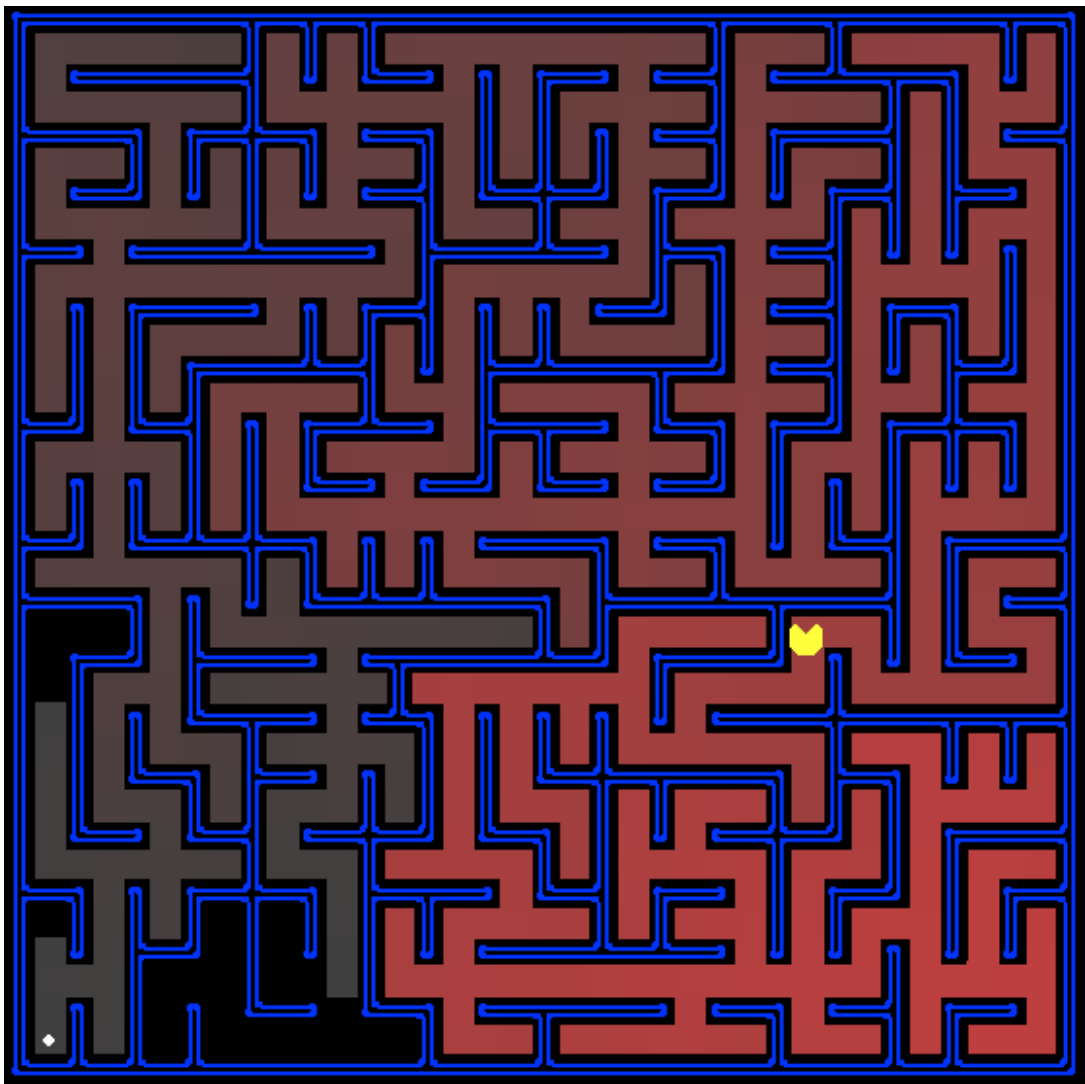


Técnicas de Inteligencia Artificial

Algoritmos de Búsqueda



Realizado por:

Cuesta Alario, David

Fernández Andrés, Cristian

Índice

Algoritmos de Búsqueda	3
Búsqueda en Profundidad DFS (Depth First Search)	4
Búsqueda en Anchura BFS (Breadth First Search)	5
Búsqueda Coste Uniforme UCS (Uniform Cost Search)	6
Búsqueda Heurística Voraz GS (Greedy Search)	7
Búsqueda Heurística en Haz BS (Beam Search)	7
Búsqueda Heurística con Coste Uniforme A*	8
Soluciones de las ejecuciones	10
Small Mace	11
Medium Mace	12
Big Mace	14
Conclusiones	16
Cuestiones a resolver	17
Problema de las cuatro esquinas	19
Resolución	19
Consideraciones adicionales:	20
Calculo del Heurístico	21
Soluciones de las ejecuciones	22
Sin heurístico	22
Con el heurístico calculado	24
Conclusiones	26
Cuestiones a resolver	26
Comiendo todos los puntos	27
Resolución	27
Calculo del Heurístico	27
Soluciones de las ejecuciones	28
Sin heurístico	28
Con el heurístico calculado	28
Conclusiones	28
Búsqueda subóptima (Suboptimal Search)	29
Apéndices:	31
Conceptos relacionados	31
Pilas de datos	31
Colas de datos	31
Colas de prioridad	31
Resultados del Autograder	32
Resultados	32
Apartado Q1	32
Apartado Q2	32
Apartado Q3	33
Apartado Q4	33
Apartado Q5	34
Apartado Q6	34
Apartado Q7	34
Apartado Q8	35
Referencias	36

Algoritmos de Búsqueda

En este proyecto nuestro agente de Pacman encontrará caminos a través de su mundo de laberintos, tanto para llegar a un lugar en particular como para recolectar alimentos de manera eficiente.

Aspectos a tener en cuenta:

- Todas las funciones de búsqueda deben devolver una lista de acciones que guiarán al agente desde el principio hasta el objetivo.
- Todas estas acciones tienen que ser movimientos legales ([no moverse a través de las paredes](#))
- Tenemos que utilizar las estructuras de datos Stack, Queue y PriorityQueue que se proporcionan en util.py

Búsqueda en Profundidad DFS (Depth First Search)

Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino regresa al último cruce de caminos en el que había otro camino disponible y repite el mismo proceso hasta dar con una solución válida o haber recorrido todos los elementos del grafo.

Este algoritmo de búsqueda se caracteriza porque

- **No está informado:** No tiene información acerca de la solución del problema
- **Es completo:** Permite recorrer todos los nodos de un grafo de manera ordenada pero no uniforme.
- **No es óptimo:** Encontrará al menos una solución si esta existe. Pero devolverá siempre la primera solución que encuentre

La complejidad computacional de esta solución es:

- **Tiempo Exponencial b^m**
- **Espacio Lineal $m * (b - 1)$**
 - o Factor de ramificación **b**: Número medio de arcos que parten de cada nodo.
 - o Profundidad máxima del grafo **m**: Numero de arcos desde el inicio del grafo hasta el punto más alejado

Para implementar este algoritmo de búsqueda utilizaremos una Pila

```
def depthFirstSearch(problem):  
    examinados = set()  
    camino = []  
    pendientes = util.Stack()  
  
    start = problem.getStartState()  
    pendientes.push( (start, []) )  
    meta = False  
  
    while ( not pendientes.isEmpty() and not meta ):  
        actual, ruta = pendientes.pop()  
        if problem.isGoalState(actual):  
            meta = True;  
            camino = ruta  
            print("META EN ", actual, "con camino", ruta)  
        else:  
            if actual not in examinados:  
                examinados.add(actual)  
                print("Ya examine ", actual)  
                print("el camino hasta aqui es: ", ruta)  
                hijos = problem.getSuccessors(actual)  
                for coord, direc, coste in hijos:  
                    pendientes.push( (coord, ruta+[direc]) )  
                    print(coord, "viene de ", actual, " en la direccion" , direc, " con coste ", coste)  
  
    return camino
```

Definimos las siguientes variables locales:

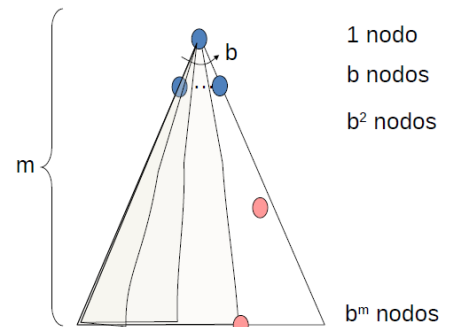
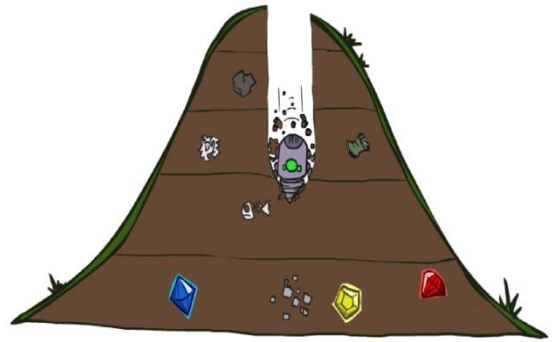
- **Examinados:** Es un ResultSet que permite conocer que nodos ya han sido revisados en un tiempo constante con respecto a su tamaño
- **Camino:** Se trata de una lista que contendrá las direcciones que deberá seguir el Pacman para llegar al destino
- **Pendientes:** Es una pila que contiene los nodos de la frontera que están pendientes de ser examinados en forma de dupla cuya clave son las coordenadas del nodo en cuestión y su atributo es una lista con el camino que ha sido necesario recorrer para llegar hasta el
- **Meta:** Es un flag de estado que permitirá conocer si hemos encontrado una solución válida para el problema

Inicializamos el problema:

- Obtenemos el nodo inicial con la función `getStartState()`
- Lo añadimos a la Pila de examinados indicando que el camino para llegar hasta él es una lista vacía
- Se inicializa el flag de la meta en falso.

Mientras que queden elementos por examinar (la Pila de examinados no está vacía) y no se halla activado el flag de la meta efectuaremos:

- Obtenemos de la Pila de examinados el primer nodo que este pendiente junto con la ruta hasta
- Comprobamos si dicho nodo es el resultado del problema con la función `isGoalState(NodoActual)`
 - o Si es el resultado:
 - Activamos el flag meta para indicar que hemos encontrado la solución
 - Definimos que el camino es la ruta que hemos seguido para llegar hasta ese punto
 - o Si no era el resultado
 - Comprobamos si el nodo extraído ya ha sido expandido comprobando si ese encuentra entre los nodos examinados
 - Si no ha sido expandido
 - o Lo añadimos a los nodos examinados para no volver a expandirlo
 - o Recorremos cada una de los movimientos legales disponibles con la función `getSucesors(NodoActual)` y lo añadimos a la Pila de pendientes junto con el camino con el que hemos llegado hasta el que será el camino que con el que hemos llegado hasta el nodo que ha llegado hasta el más la dirección con la se llegaba hasta el



Búsqueda en Anchura BFS (Breadth First Search)

Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos vecinos de los nodos que ya han sido expandidos. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta dar con una solución válida o que se recorra todo el grafo.

Este algoritmo de búsqueda se caracteriza porque

- **No está informado:** No tiene información acerca de la solución del problema
- **Es completo:** Permite recorrer todos los nodos de un grafo de manera ordenada y uniforme.
- **Es óptimo:** Siempre y cuando todos los arcos tengan el mismo coste de expansión se encontrara la solución de menor coste

La complejidad computacional de esta solución es:

- **Tiempo Exponencial** b^s
- **Espacio Exponencial** b^s
 - o Factor de ramificación **b**: Número medio de arcos que parten de cada nodo.
 - o Profundidad de la solución óptima del grafo **s**: Numero de arcos desde el inicio del grafo hasta el punto de la solución optima

Para implementar este algoritmo de búsqueda utilizaremos una Cola

```
def breadthFirstSearch(problem):  
    examinados = set()  
    camino = []  
  
    pendientes = util.Queue()  
  
    start = problem.getStartState()  
    pendientes.push( (start,[]) )  
    meta = False  
  
    while ( not pendientes.isEmpty() and not meta ):  
        actual,ruta = pendientes.pop()  
        if problem.isGoalState(actual):  
            meta = True;  
            camino = ruta  
            print("META EN ", actual,"con camino", ruta)  
        else:  
            if actual not in examinados:  
                examinados.add(actual)  
                print("Ya examine ", actual)  
                print("el camino hasta aqui es: ", ruta)  
                hijos = problem.getSuccessors(actual)  
                for coord, direc, coste in hijos:  
                    pendientes.push( (coord,ruta+[direc]) )  
                    print(coord," viene de ",actual," en la direccion" ,direc," con coste ",coste)  
  
    return camino
```

Definimos las siguientes variables locales:

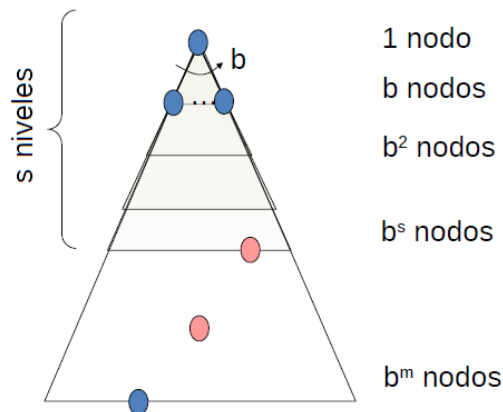
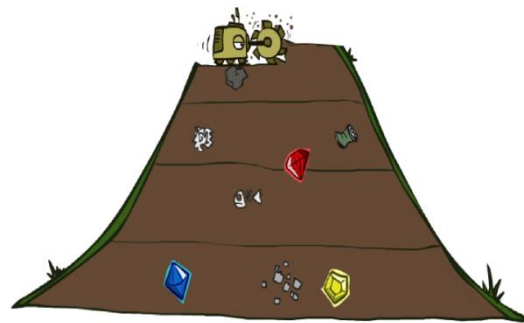
- **Examinados:** Es un ResultSet que permite conocer que nodos ya han sido revisados en un tiempo constante con respecto a su tamaño
- **Camino:** Se trata de una lista que contendrá las direcciones que deberá seguir el Pacman para llegar al destino
- **Pendientes:** Es una Cola que contiene los nodos de la frontera que están pendientes de ser examinados en forma de dupla cuya clave son las coordenadas del nodo en cuestión y su atributo es una lista con el camino que ha sido necesario recorrer para llegar hasta el
- **Meta:** Es un flag de estado que permitirá conocer si hemos encontrado una solución válida para el problema

Inicializamos el problema:

- Obtenemos el nodo inicial con la función `getStartState()`
- Lo añadimos a la Cola de examinados indicando que el camino para llegar hasta él es una lista vacía
- Se inicializa el flag de la meta en falso.

Mientras que queden elementos por examinar (la Cola de examinados no está vacía) y no se halla activado el flag de la meta efectuaremos:

- Obtenemos de la Cola de examinados el primer nodo que este pendiente junto con la ruta hasta
- Comprobamos si dicho nodo es el resultado del problema con la función `isGoalState(NodoActual)`
 - o Si es el resultado:
 - Activamos el flag meta para indicar que hemos encontrado la solución
 - Definimos que el camino es la ruta que hemos seguido para llegar hasta ese punto
 - o Si no era el resultado
 - Comprobamos si el nodo extraído ya ha sido expandido comprobando si ese encuentra entre los nodos examinados
 - Si no ha sido expandido
 - o Lo añadimos a los nodos examinados para no volver a expandirlo
 - o Recorremos cada una de los movimientos legales disponibles con la función `getSucesors(NodoActual)` y lo añadimos a la Cola de pendientes junto con el camino con el que hemos llegado hasta el que será el camino que con el que hemos llegado hasta el nodo que ha llegado hasta el más la dirección con la se llegaba hasta el



Búsqueda Coste Uniforme UCS (Uniform Cost Search)

Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos del grafo dando prioridad a aquellos nodos cuyo coste acumulado desde el nodo raíz sea mínimo. De este modo Los nodos son visitados de esta manera hasta que el nodo destino es alcanzado o se recorra todo el grafo

Este algoritmo de búsqueda se caracteriza porque

- **No está informado:** No tiene información acerca de la solución del problema
- **Es completo:** Permite recorrer todos los nodos de un grafo de manera ordenada y uniforme.
- **Es óptimo:** Encuentra la mejor solución por el camino de menor coste

La complejidad computacional de esta solución es:

- **Tiempo** Exponencial $b^{c/\epsilon}$
- **Espacio** Exponencial $b^{c/\epsilon}$
 - o Factor de ramificación b : Número medio de arcos que parten de cada nodo.
 - o Profundidad de la solución óptima del grafo c
 - o Coste máximo de los arcos ϵ
 - o Profundidad efectiva c/ϵ

Para implementar este algoritmo de búsqueda utilizaremos una Cola de prioridad

`def uniformCostSearch(problem):`

```
    examinados = set()
    camino = []
    costeFinal = 0
    pendientes = util.PriorityQueue()

    start = problem.getStartState()
    pendientes.push( (start, [], 0), 0 )
    meta = False

    while ( not pendientes.isEmpty() and not meta ):
        actual, ruta, coste2 = pendientes.pop()
        if problem.isGoalState(actual):
            meta = True
            camino = ruta
            costeFinal = coste2
            print("META EN ", actual, "con camino", ruta)
        else:
            if actual not in examinados:
                examinados.add(actual)
                print("Ya examine ", actual)
                print("el camino hasta aqui es: ", ruta)
                hijos = problem.getSuccessors(actual)
                for coord, direc, coste in hijos:
                    pendientes.push( (coord, ruta+[direc], coste2+coste), coste2+coste )
                    print(coord, " viene de ", actual, " en la direccion" , direc, " con coste ", coste)

    return camino
```

Definimos las siguientes variables locales:

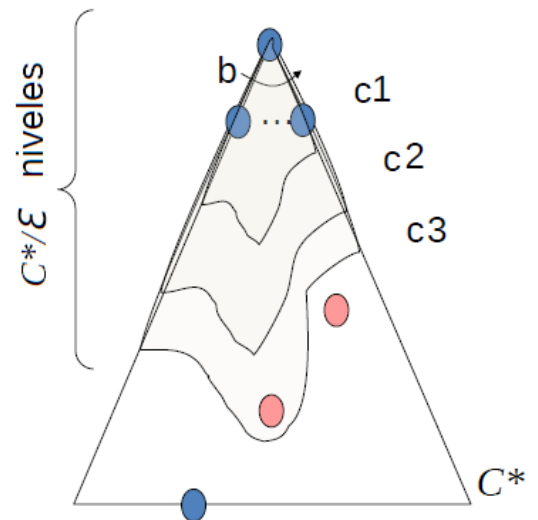
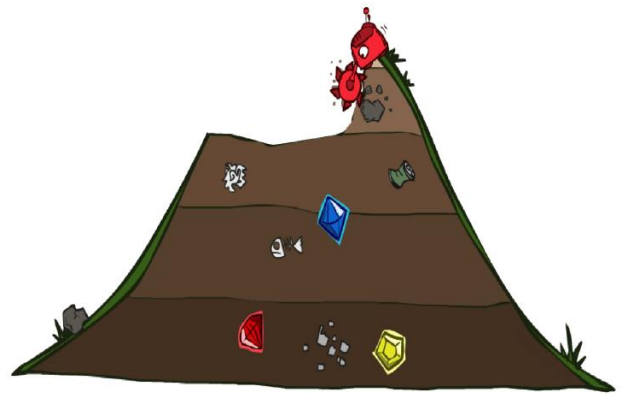
- **Examinados:** Es un ResultSet que permite conocer que nodos ya han sido revisados en un tiempo constante con respecto a su tamaño
- **Camino:** Se trata de una lista que contendrá las direcciones que deberá seguir el Pacman para llegar al destino
- **Pendientes:** Es una Cola de prioridad que contiene los nodos de la frontera que están pendientes de ser examinados en forma de dupla cuya clave son las coordenadas del nodo en cuestión y su atributo es una lista con el camino que ha sido necesario recorrer para llegar hasta él. Además la cola de prioridad exige que se le pase como parámetro el coste acumulado hasta dicho punto.
- **Meta:** Es un flag de estado que permitirá conocer si hemos encontrado una solución válida para el problema
- **Coste Final:** en verdad no es necesario pero almacenara la solución del coste optimo del camino con menor coste

Inicializamos el problema:

- Obtenemos el nodo inicial con la función `getStartState()`
- Lo añadimos a la Cola de prioridad indicando que el camino para llegar hasta él es una lista vacía y el coste inicial será cero
- Se inicializa el flag de la meta en falso

Mientras que queden elementos por examinar (la Cola de examinados no está vacía) y no se halla activado el flag de la meta efectuaremos:

- Obtenemos de la Cola de prioridad de examinados el primer nodo que este pendiente junto con la ruta hasta él y el coste acumulado hasta el
- Comprobamos si dicho nodo es el resultado del problema con la función `isGoalStatte(NodoActual)`
 - o Si es el resultado:
 - Activamos el flag meta para indicar que hemos encontrado la solución
 - Definimos que el camino es la ruta que hemos seguido para llegar hasta ese punto
 - Y el coste máximo es el coste acumulado hasta el



- Si no era el resultado
 - Comprobamos si el nodo extraído ya ha sido expandido comprobando si ese encuentra entre los nodos examinados
 - Si no ha sido expandido
 - Lo añadimos a los nodos examinados para no volver a expandirlo
 - Recorremos cada una de los movimientos legales disponibles con la función `getSucesors(NodoActual)` y lo añadimos a la Cola de prioridad de pendientes junto con
 - El camino con el que hemos llegado hasta el que será el camino que con el que hemos llegado hasta el nodo que ha llegado hasta el más la dirección con la se llegaba hasta él
 - El coste acumulado hasta él que será el coste con el que hemos llegado hasta el nodo que ha llegado hasta el más el coste del arco con el que hemos llegado hasta él.
 - La prioridad con a que se añade la dupla a la cola de prioridad será el coste acumulado hasta el nodo

Búsqueda Heurística Voraz GS (Greedy Search)

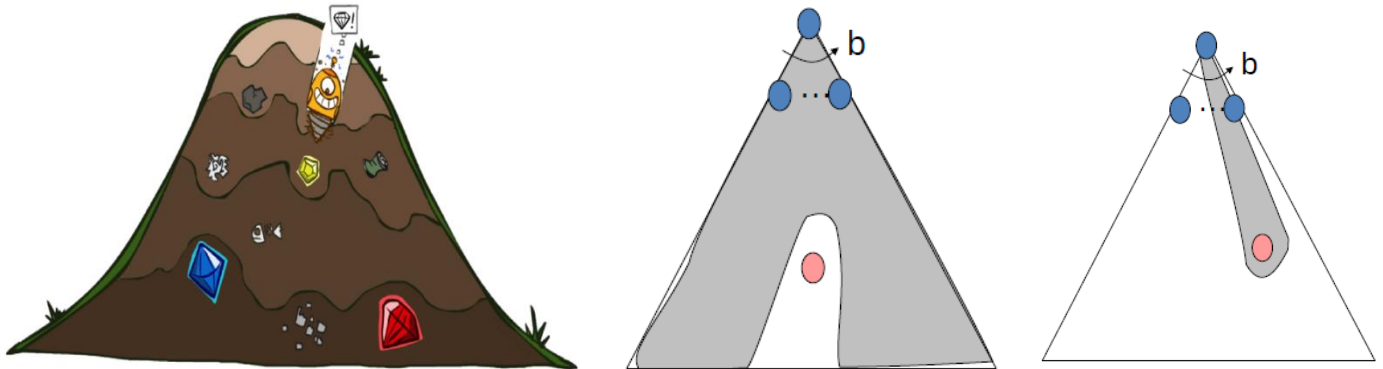
Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando expandiendo primero al mejor candidato posible con la esperanza de llegar a una solución general óptima.

Al mejor candidato se le selecciona con respecto a un conjunto de reglas internas denominadas Heurísticos.

Este algoritmo de búsqueda se caracteriza porque

- **Está informado:** Los heurísticos se establecen conforme a una estimación de la distancia relativa entre la posición actual y la solución más cercana.
- **Es completo:** Permite recorrer todos los nodos de un grafo de manera ordenada pero no uniforme.
- **No es óptimo:** En muchas ocasiones, dependiendo de las características del grafo y más concretamente de los obstáculos que plantee, el heurístico nos lleva a tomar caminos que nos alejan de la solución óptima.

Para implementar este algoritmo de búsqueda utilizaremos una **Cola de prioridad**



Búsqueda Heurística en Haz BS (Beam Search)

Su funcionamiento consiste en ir expandiendo exclusivamente aquellos nodos que sean mejores candidatos con la esperanza de llegar a una solución general óptima. De esta forma el número de nodos en la frontera se mantiene constante

Al mejor candidato se le selecciona con respecto a un conjunto de reglas internas denominadas Heurísticos.

Este algoritmo de búsqueda se caracteriza porque

- **Está informado:** Los heurísticos se establecen conforme a una estimación de la distancia relativa entre la posición actual y la solución más cercana.
- **No es completo:** Solo recorre los nodos que considera mejores candidatos por lo que podría no llegar a encontrar ninguna solución
- **No es óptimo:** En muchas ocasiones, dependiendo de las características del grafo y más concretamente de los obstáculos que plantee, el heurístico nos lleva a tomar caminos que nos alejan de la solución óptima.

Para implementar este algoritmo de búsqueda utilizaremos una **Cola de prioridad**

Búsqueda Heurística con Coste Uniforme A*

Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando expandiendo primero al mejor candidato posible con la esperanza de llegar a una solución general óptima.

Al mejor candidato se le selecciona con respecto la ponderación del coste acumulado desde el nodo raíz y el conjunto de reglas internas denominadas Heurísticos

Este algoritmo de búsqueda se caracteriza porque

- **Está informado:** Los heurísticos se establecen conforme a una estimación de la distancia relativa entre la posición actual y la solución más cercana.
- **Es completo:** Permite recorrer todos los nodos de un grafo de manera ordenada pero no uniforme.
- **Es óptimo:** Siempre y cuando se cumplan las siguientes reglas:
 - o **Test de admisibilidad del heurístico:** El heurístico utilizado para ponderar la idoneidad de un camino será menor que el coste necesario para llegar por ese camino al nodo destino. Como consecuencia el heurístico de mayor magnitud asignado al nodo más alejado de la solución será menor que el coste mínimo acumulado necesario para alcanzar la solución óptima. Y dado que estamos hablando de distancias estos siempre serán positivos
$$0 \leq h(n) \leq c(n)$$
 - o **Test de consistencia del heurístico** El heurístico utilizado para ponderar la idoneidad de un camino será coherente con respecto al coste real de las acciones para llegar a cada nodo. Por ejemplo, debe suceder que si una acción ha costado x, entonces tomar esa acción solo puede causar una caída en la heurística de a lo sumo x.
 - o **Trivialidad:** los heurísticos triviales son aquellos que no aportan ninguna información acerca del objetivo (ni buena ni mala). Por ejemplo un heurístico que siempre devuelva cero será admisible pero también trivial.Cualquier problema que utilice un heurístico trivial se asemejará más a un algoritmo de UCS que de búsqueda heurística



Para implementar este algoritmo de búsqueda utilizaremos una **Cola de prioridad**

```
def aStarSearch(problem, heuristic=nullHeuristic):  
    examinados = set()  
    camino = []  
    costeFinal = 0  
    pendientes = util.PriorityQueue()  
  
    start = problem.getStartState()  
    pendientes.push( (start, [], 0), 0 )  
    meta = False  
  
    while ( not pendientes.isEmpty() and not meta ):  
        actual, ruta, costeAct = pendientes.pop()  
        if problem.isGoalState(actual):  
            meta = True  
            camino = ruta  
            costeFinal = costeAct  
            print("META EN ", actual, "con camino", ruta, "con coste", costeFinal)  
        else:  
            if actual not in examinados:  
                examinados.add(actual)  
                print("Ya examine ", actual)  
                print("el camino hasta aqui es: ", ruta)  
                hijos = problem.getSuccessors(actual)  
                for coord, direc, coste in hijos:  
                    euristico = heuristic(coord, problem)  
                    costeNuevo = costeAct + coste  
                    pendientes.push( (coord, ruta + [direc], costeNuevo), costeNuevo + euristico )  
                    print(coord, "viene de ", actual, "en la direccion", direc, "con coste ", coste, "con euristico", euristico)  
  
    return camino
```

Definimos las siguientes variables locales:

- **Examinados:** Es un ResultSet que permite conocer que nodos ya han sido revisados en un tiempo constante con respecto a su tamaño
- **Camino:** Se trata de una lista que contendrá las direcciones que deberá seguir el Pacman para llegar al destino
- **Pendientes:** Es una Cola de prioridad que contiene los nodos de la frontera que están pendientes de ser examinados en forma de dupla cuya clave son las coordenadas del nodo en cuestión y su atributo es una lista con el camino que ha sido necesario recorrer para llegar hasta él. Además la cola de prioridad exige que se le pase como parámetro el coste acumulado hasta dicho punto.
- **Meta:** Es un flag de estado que permitirá conocer si hemos encontrado una solución válida para el problema
- **Coste Final:** en verdad no es necesario pero almacenara la solución del coste optimo del camino con menor coste

Inicializamos el problema:

- Obtenemos el nodo inicial con la función `getStartState()`
- Lo añadimos a la Cola de prioridad indicando que el camino para llegar hasta él es una lista vacía y el coste inicial será cero
- Se inicializa el flag de la meta en falso.

Mientras que queden elementos por examinar (la Cola de examinados no está vacía) y no se halla activado el flag de la meta efectuaremos:

- Obtenemos de la Cola de prioridad de examinados el primer nodo que este pendiente junto con la ruta hasta él y el coste acumulado hasta el
- Comprobamos si dicho nodo es el resultado del problema con la función *isGoalState(NodoActual)*
 - o Si es el resultado:
 - Activamos el flag meta para indicar que hemos encontrado la solución
 - Definimos que el camino es la ruta que hemos seguido para llegar hasta ese punto
 - Y el coste máximo es el coste acumulado hasta el
 - o Si no era el resultado
 - Comprobamos si el nodo extraído ya ha sido expandido comprobando si ese encuentra entre los nodos examinados
 - Si no ha sido expandido
 - o Lo añadimos a los nodos examinados para no volver a expandirlo
 - o Recorremos cada uno de los movimientos legales disponibles con la función *getSucesors(NodoActual)* y lo añadimos a la Cola de prioridad de pendientes junto con:
 - El camino con el que hemos llegado hasta el que será el camino que con el que hemos llegado hasta el nodo que ha llegado hasta el más la dirección con la se llegaba hasta él
 - El coste acumulado hasta él que será el coste con el que hemos llegado hasta el nodo que ha llegado hasta el más el coste del arco con el que hemos llegado hasta él.
 - La prioridad con a que se añade la dupla a la cola de prioridad será el coste acumulado hasta el nodo más la cifra indicada por el heurístico que viene indicado como parámetro, cero si no se indica ningún heurístico

Soluciones de las ejecuciones

Comprobamos su funcionamiento ejecutando los siguientes comandos:

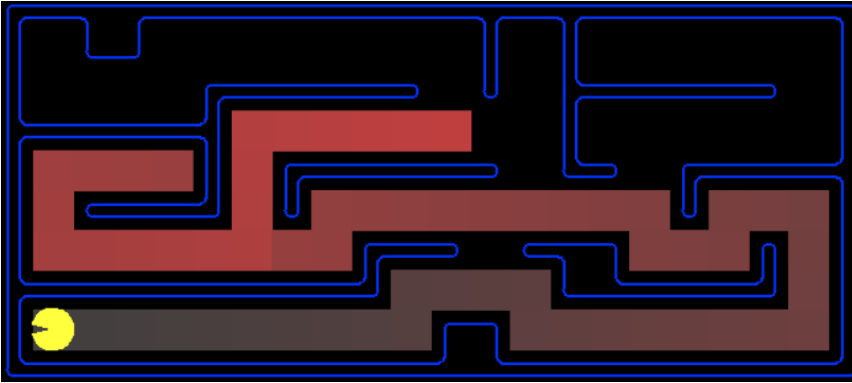
python pacman.py -l tinyMaze -p SearchAgent

```
Anaconda Powershell Prompt (Anaconda3) CS188 Pacm...  
  
python pacman.py -l tinyMaze -p SearchAgent  
[SearchAgent] using function depthFirstSearch  
[SearchAgent] using problem type PositionSearchProblem  
Ya examine (5, 5)  
(5, 4) viene de (5, 5) en la direccion South con coste 1  
(4, 5) viene de (5, 5) en la direccion West con coste 1  
Ya examine (4, 5)  
(5, 5) viene de (4, 5) en la direccion East con coste 1  
(3, 5) viene de (4, 5) en la direccion West con coste 1  
Ya examine (3, 5)  
(4, 5) viene de (3, 5) en la direccion East con coste 1  
(2, 5) viene de (3, 5) en la direccion West con coste 1  
Ya examine (2, 5)  
(3, 5) viene de (2, 5) en la direccion East con coste 1  
(1, 5) viene de (2, 5) en la direccion West con coste 1  
Ya examine (1, 5)  
(1, 4) viene de (1, 5) en la direccion South con coste 1  
(2, 5) viene de (1, 5) en la direccion East con coste 1  
Ya examine (1, 4)  
(1, 5) viene de (1, 4) en la direccion North con coste 1  
(1, 3) viene de (1, 4) en la direccion South con coste 1  
Ya examine (1, 3)  
(1, 4) viene de (1, 3) en la direccion North con coste 1  
(2, 3) viene de (1, 3) en la direccion East con coste 1  
Ya examine (2, 3)  
(2, 2) viene de (2, 3) en la direccion South con coste 1  
(1, 3) viene de (2, 3) en la direccion West con coste 1  
Ya examine (2, 2)  
(2, 3) viene de (2, 2) en la direccion North con coste 1  
(2, 1) viene de (2, 2) en la direccion South con coste 1  
(3, 2) viene de (2, 2) en la direccion East con coste 1  
Ya examine (3, 2)  
(4, 2) viene de (3, 2) en la direccion East con coste 1  
(2, 2) viene de (3, 2) en la direccion West con coste 1  
Ya examine (4, 2)  
(4, 3) viene de (4, 2) en la direccion North con coste 1  
(3, 2) viene de (4, 2) en la direccion West con coste 1  
Ya examine (4, 3)  
(4, 2) viene de (4, 3) en la direccion South con coste 1  
(5, 3) viene de (4, 3) en la direccion East con coste 1  
Ya examine (5, 3)  
(5, 4) viene de (5, 3) en la direccion North con coste 1  
(4, 3) viene de (5, 3) en la direccion West con coste 1  
Ya examine (5, 4)  
(5, 5) viene de (5, 4) en la direccion North con coste 1  
(5, 3) viene de (5, 4) en la direccion South con coste 1  
Ya examine (2, 1)  
(2, 2) viene de (2, 1) en la direccion North con coste 1  
(1, 1) viene de (2, 1) en la direccion South con coste 1  
META EN (1, 1) con camino ['West', 'West', 'West', 'West', 'South', 'South', 'East', 'South', 'South',  
'West']  
Path found with total cost of 10 in 0.0 seconds  
Search nodes expanded: 15  
Pacman emerges victorious! Score: 500  
Average Score: 500.0  
Scores: 500.0  
Win Rate: 1/1 (1.00)  
Record: Win
```

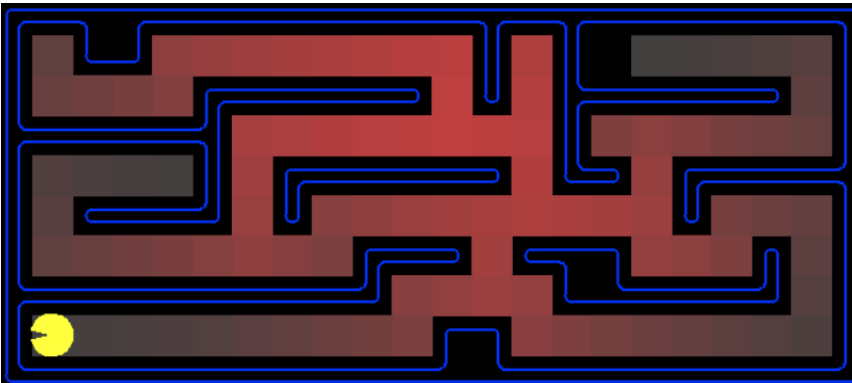


Small Maze

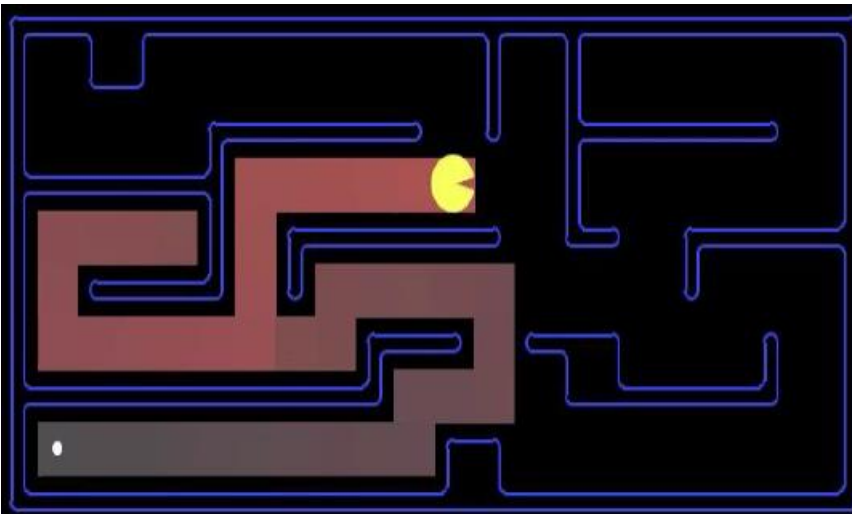
DFS



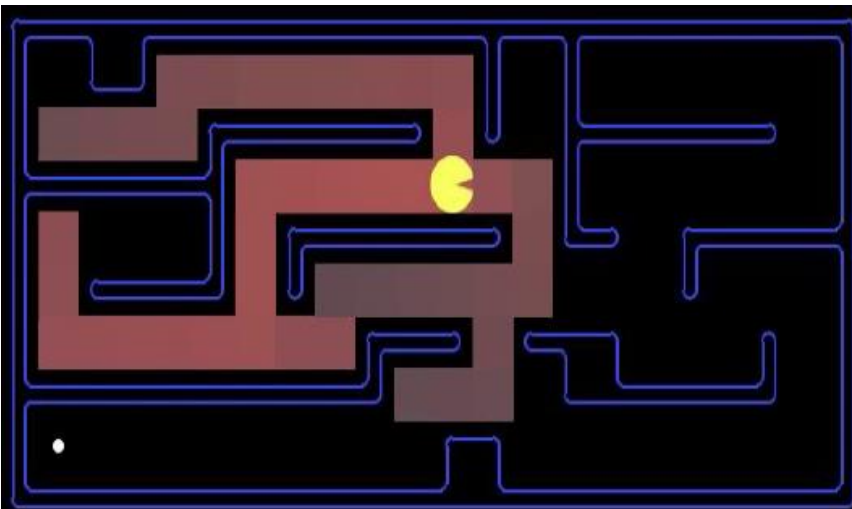
UCS



Greedy



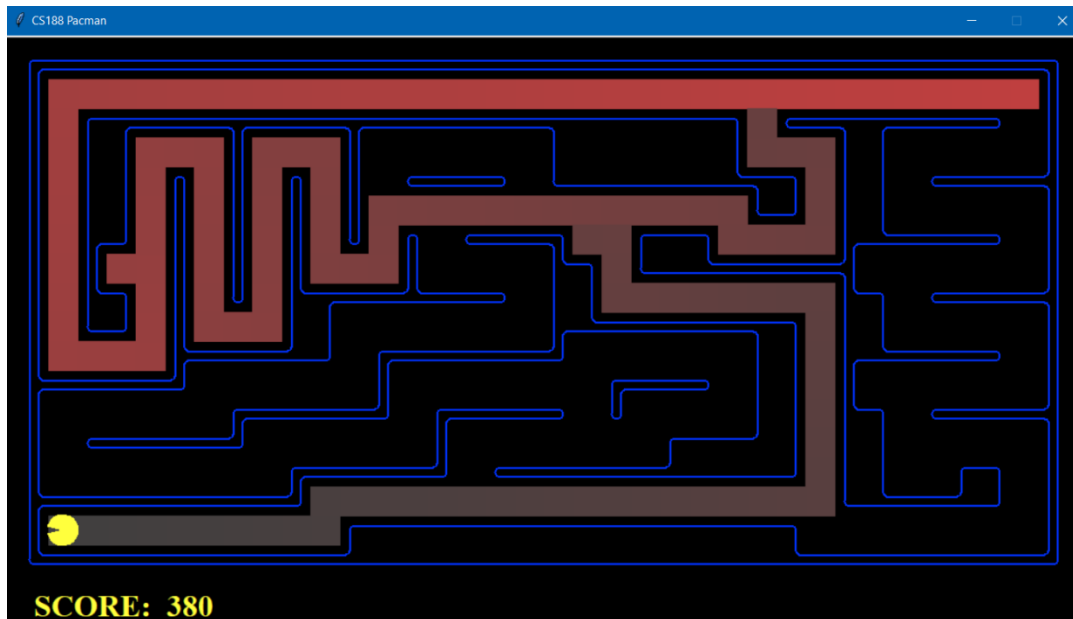
A*



Medium Maze

Profundidad

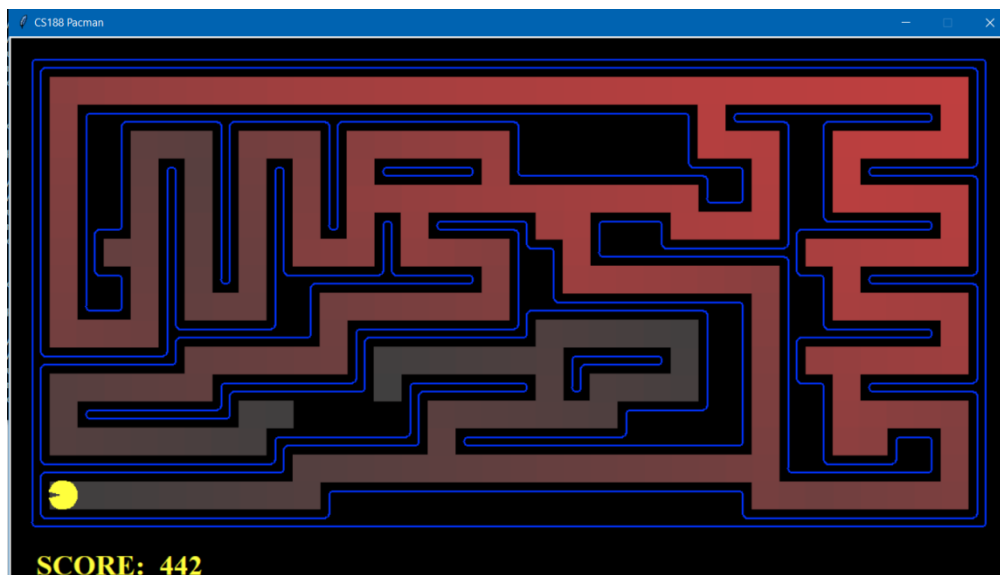
`python pacman.py -l mediumMaze -p SearchAgent`



```
Anaconda Powershell Prompt (Anaconda3)
search> python pacman.py -l mediumMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:      380.0
Win Rate:    1/1 (1.00)
Record:      Win
```

Anchura

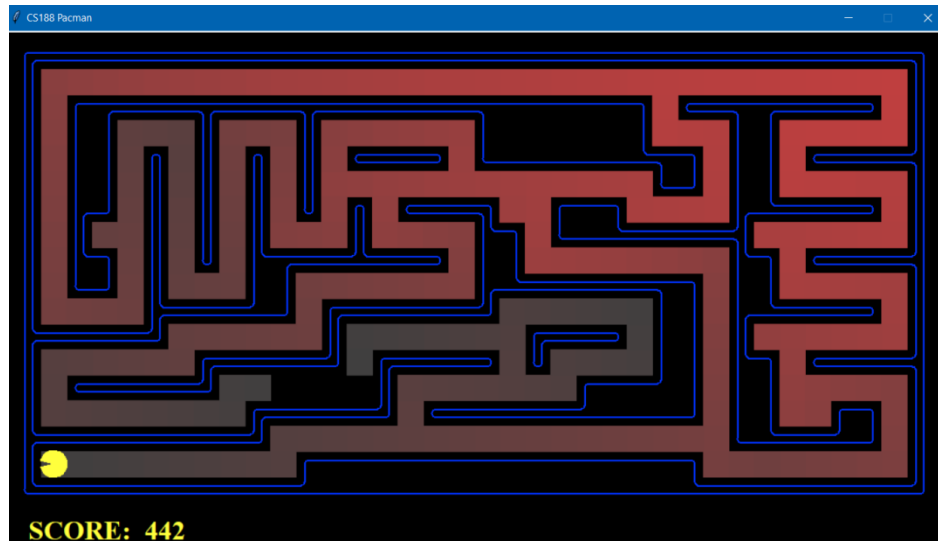
`python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs`



```
Anaconda Powershell Prompt (Anaconda3)
search> python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:      442.0
Win Rate:    1/1 (1.00)
Record:      Win
```

Coste Uniforme

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

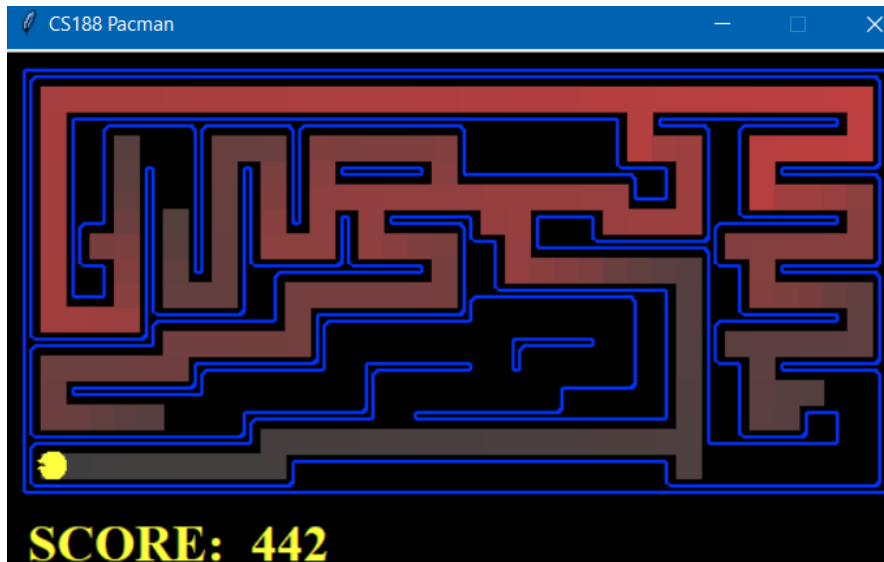


```
Anaconda Powershell Prompt (Anaconda3)

search> python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:      442.0
Win Rate:    1/1 (1.00)
Record:      Win
```

A*

```
python pacman.py -l mediumMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```



```
Anaconda Powershell Prompt (Anaconda3)

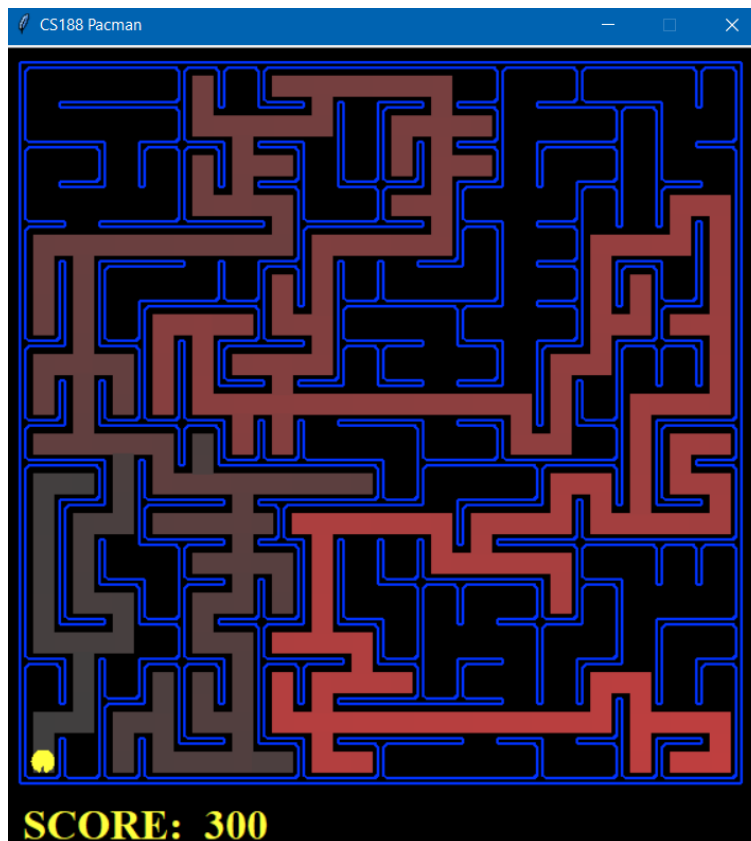
python pacman.py -l mediumMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 221
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:      442.0
Win Rate:    1/1 (1.00)
Record:      Win
```

Big Maze

Comprobamos su funcionamiento ejecutando los siguientes comandos:

Profundidad

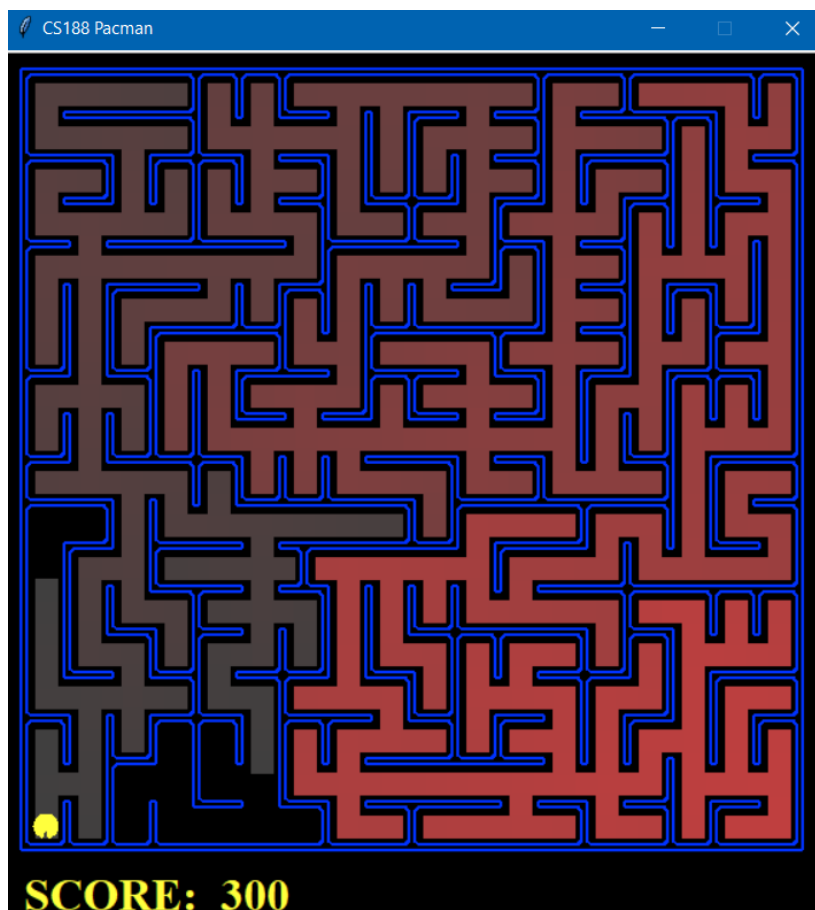
```
python pacman.py -l bigMaze -z .5 -p SearchAgent
```



```
Anaconda Powershell Prompt (Anaconda3)
search> python pacman.py -l bigMaze -z .5 -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores: 300.0
Win Rate: 1/1 (1.00)
Record: Win
```

Anchura

```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```



```
Anaconda Powershell Prompt (Anaconda3)
search> python pacman.py -l bigMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores: 300.0
Win Rate: 1/1 (1.00)
Record: Win
```

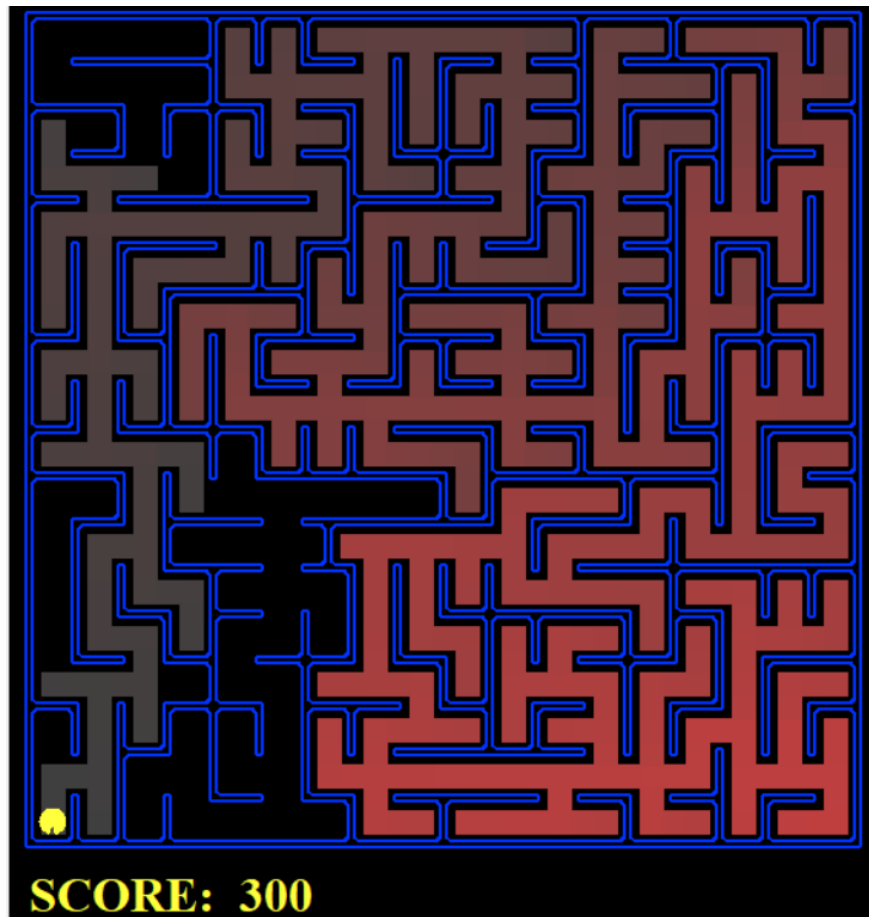
Coste uniforme

`python pacman.py -l bigMaze -p SearchAgent -a fn=ucs -z.5`



A*

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



```
Anaconda Powershell Prompt (Anaconda3)
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores: 300.0
Win Rate: 1/1 (1.00)
Record: Win
```

Conclusiones

Profundidad

Resultados de mediumMaze:

- Nodos expandidos: 146
- Coste del recorrido: 130

Resultados de bigMaze:

- Nodos expandidos: 549
- Coste del recorrido: 210

Observaciones

- En este caso Profundidad es el método más eficiente dado que explora menos nodos que todos los demás algoritmos.
- No encuentra la solución óptima.

Conclusiones

- por las características del algoritmo de profundidad, es altamente probable que este resultado sea una excepción dado que el hecho de encontrar una solución rápidamente depende más de la suerte que del propio algoritmo

Anchura y Coste uniforme

Resultados de mediumMaze:

- Nodos expandidos: 269
- Coste del recorrido: 68

Resultados de bigMaze:

- Nodos expandidos: 620
- Coste del recorrido: 210

Observaciones

- Ambos exploran el mismo recorrido y llegan a la meta por la solución óptima.
- El número de nodos expandidos es casi igual al número de nodos del laberinto

Conclusiones

- En este ejemplo el coste de todos los nodos era uniforme por lo que no debería existir ninguna diferencia entre estos dos algoritmos
- Estos algoritmos dan con la solución óptima pero son bastante ineficientes. Si el laberinto fuera infinitamente más grande podría no llegar a resolverse nunca con este método.

A*

Resultados de mediumMaze:

- Nodos expandidos: 221
- Coste del recorrido: 68

Resultados de bigMaze:

- Nodos expandidos: 549
- Coste del recorrido: 210

Observaciones

- La búsqueda Heurística de coste uniforme con el heurístico Manhattan expande unos pocos menos nodos que los algoritmos de Anchura y coste uniforme y también alcanza la solución por el camino óptimo.

Conclusiones

- Esto se debe a que el algoritmo A* es muy similar al de coste uniforme ([realiza una búsqueda en anchura priorizando los nodos con menor coste](#)) con la diferencia de en este caso explora primero los nodos que le aconseja el heurístico.
- Se podría decir que es una búsqueda en anchura guiada, lo cual mejora los resultados siempre y cuando el heurístico utilizado sea bueno.
- Podemos observar que en laberintos más grandes la diferencia entre UCS y A* se acentúa

Cuestiones a resolver

¿Es el orden de exploración lo que esperábamos?

La búsqueda en profundidad DFS

Ha explorado todos los nodos en el mismo camino hasta completarlo, tras lo cual volvió al último cruce y siguió explorando

La búsqueda en anchura BFS y La búsqueda de coste uniforme UCS

Ha explorado todos los nodos del laberinto de forma constante y uniforme hasta dar con la solución

La búsqueda Heurística de coste constante

Han explorado todos los nodos del laberinto de forma constante y uniforme priorizando los recomendados por el heurístico hasta dar con la solución

¿Encuentra BFS una solución de coste mínimo?

Debido a que el peso de todos los arcos es el mismo no hay ninguna diferencia entre UCS y BFS.

Por lo que este caso BFS si encuentra una solución de coste mínimo.

¿Qué diferencia hay entre los algoritmos de búsqueda informados y los no informados?

Si nos fijamos en el código de los tres primeros algoritmos podemos observar que en realidad son el mismo excepto por la estrategia de tratamiento de la frontera. La cosa cambia con los algoritmos informados que reciben información acerca de la ubicación de la solución

¿Cuándo debería terminar un algoritmo de búsqueda?

En los algoritmos de búsqueda DFS y BFS podríamos detener la búsqueda en el momento en que encolamos la solución y el resultado obtenido seria el mismo al resultado esperado. Esto se debe a las características de la estructura de las pilas y las colas que solo permiten extraer los elementos de forma proporcional a como se introducen.

Por el contrario UCS, GS y A* utilizan colas de prioridad, que no mantienen la linealidad de la información sino que la estructuran en función del coste. De este modo podría encolarse la solución por un camino y posteriormente desencolar otra que se encolo por otra ruta diferente. Como consecuencia, en estos algoritmos solo obtendremos la solución esperada si se termina el algoritmo en el momento en que desencolamos la solución

¿Qué algoritmo resuelve mejor el problema?

La búsqueda en profundidad DFS

Puede llegar a encontrar una solución muy rápidamente aunque no siempre será la óptima.

No obstante en ocasiones el algoritmo puede “irse por las ramas” y expandir muchos nodos antes de encontrar una solución

La búsqueda en anchura BFS y La búsqueda de coste uniforme UCS

Siempre encuentra la solución óptima

EL coste de llegar a la solución óptima puede llegar a ser muy elevado si el número de caminos crece de forma exponencial

La primera se utilizara si los costes de todos los arcos son uniformes y la segunda cuando no lo sean

La búsqueda Heurística voraz

Si no hay obstáculos y el heurístico es bueno encuentra una solución muy rápidamente, aunque no tiene por qué ser la optima

La búsqueda Heurística en haz

Si no hay obstáculos y el heurístico es bueno encuentra una solución muy rápidamente, aunque no tiene por qué ser la optima

Este algoritmo es más eficiente que la búsqueda voraz dado que la información guardada en la frontera está limitada a un valor previamente especificado. Por lo que el coste de expansión es constante

Podría no llegar a obtener ninguna solución

La búsqueda Heurística de coste constante

Es una combinación lineal de la búsqueda Heurística voraz y la búsqueda de coste uniforme, lo cual se refleja c claramente en los resultados. Obtiene la solución óptima expandiendo todos los nodos de forma uniforme con respecto al coste de los arcos pero dando prioridad a los nodos recomendados por el heurístico.

Conclusión

Para este problema concreto (Pacman) considero que el mejor algoritmo de búsqueda es **La búsqueda Heurística de coste constante** a pesar de que los resultados indican que DFS ha sido menos costoso en todas las ocasiones y en el bigMace obtuvo el camino óptimo. Consideramos que esto no es más que una de las casualidades que a veces se dan en dicho algoritmo y que para laberintos de mayor tamaño La búsqueda Heurística de coste constante obtendría los mejores resultados

¿Qué pasaría si utilizáramos el algoritmo de búsqueda de A* sin ningún heurístico?

Si el heurístico de la búsqueda Heurística de coste uniforme es cero nos quedaría la búsqueda de coste uniforme

$A^* = UCS + GS$ Si quitamos **GS** nos queda $A^* = UCS$

$C(A^*) = C(Arco) + h(Nodo)$ Si $h(Nodo) = 0$ nos queda $C(A^*) = C(Arco)$

¿Pacman realmente va a todos los cuadrados explorados en su camino hacia la meta?

No, Pacman solo recorre los nodos de la ruta que el algoritmo considera la solución del problema

¿Qué pasa en openMaze para las distintas estrategias de búsqueda?

DFS: Coste 298, nodos expandidos 808.

BFS: Coste 54, nodos expandidos 683.

UCS: Coste 54, nodos expandidos 683.

A*: Coste 54, nodos expandidos 535.

Problema de las cuatro esquinas

Resolución

En este ejercicio utilizaremos los algoritmos de búsqueda implementados en el ejercicio anterior para buscar cuatro nodos que se ubicaran en las cuatro esquinas del laberinto. De este modo, solo consideraremos la posición inicial de Pacman y la ubicación de las cuatro esquinas.

Es decir, los estados de este problema contendrán la posición (nodo) de Pacman y una lista que contenga los nodos esquina ya visitados. Cabe destacar que esta última en realidad no será una lista ni un set (ya que necesitamos que sea hashable), sino una tupla de tuplas (cada una de las coordenadas de la esquina visitada), y será necesaria una concatenación especial cuando hay que añadir una nueva: `tuplaAnterior + ((x, y),)`.

```
class CornersProblem(search.SearchProblem):

    def __init__(self, startingGameState):

        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()

        top = self.walls.height-2
        right = self.walls.width-2

        self.corners = ((1,1), (1,top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner' + str(corner))

        self._expanded = 0
```

Para inicializar el problema deberemos cargar la ubicación de los muros y la posición inicial del Pacman que se encuentran en la clase `startingGameState`

A continuación marcamos las coordenadas de las cuatro esquinas partiendo de las coordenadas de los muros y teniendo en cuenta que estos tienen dos de ancho en el borde.

La variable `expanded` **/*TODO*/**

```
def getStartState(self):
    return (self.startingPosition, ())

def isGoalState(self, state):
    return (len(state[1]) == 4)
```

La función `getStartState` devuelve el nodo inicial donde Pacman empieza el recorrido que está recogido en la variable `startingPosition`

La función `isGoalState` recibe como parámetro el estado del problema y devuelve true cuando el número de elementos contenido en el primer elemento del estado sea igual a cuatro, es decir, cuando hayamos pasado por las cuatro esquinas. Lo utilizaremos para saber cuándo ha terminado el problema.

```
def getSuccessors(self, state):

    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:

        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)

        hitsWall = self.walls[nextx][nexty]

        if not hitsWall:
            if (nextx, nexty) in self.corners and not ((nextx, nexty) in state[1]):
                successors.append((((nextx, nexty), (state[1] + ((nextx, nexty),))), action, 1))
            else:
                successors.append((((nextx, nexty), state[1]), action, 1))

    self._expanded += 1
    return successors
```

La función `isGoalState` recibe como parámetro el estado inicial del problema y devuelve una lista con los estados finales de los nodos que son accesibles desde el mediante movimientos legales del juego. Para obtenerlos deberemos comprobar:

Definimos las siguientes variables locales:

- **sucesors**: Es una lista que contendrá los estados finales de los nodos que son accesibles desde el recibido como parámetro mediante movimientos legales del juego

Para cada una de las cuatro direcciones posibles [North, South, East, West]

- Obtenemos las coordenadas actuales del Pacman que se encuentran en el primer elemento del estado del problema
- Obtenemos las coordenadas finales tras realizar el movimiento en la dirección que estamos estudiando
- Comprobamos si existe algún muro comprendido entre ambas posiciones
 - o Si no hay ningún muro
 - Comprobamos si la posición final es una de las esquinas del laberinto y no ha sido ya añadida a la dupla de elementos que ya han sido visitados
 - En caso afirmativo añadimos a la solución el estado formado por:
 - o las coordenadas finales del nodo al que se accede
 - o concatenamos a la lista de las esquinas visitadas las coordenadas de la esquina que se acaba de visitar
 - o La dirección que se ha tomado para llegar a dicho nodo
 - o El coste que en este problema siempre es uno
 - En caso negativo añadimos a la solución el estado formado por:
 - o las coordenadas finales del nodo al que se accede
 - o La lista de las esquinas que ya han sido visitadas sin cambiar nada
 - o La dirección que se ha tomado para llegar a dicho nodo
 - o El coste que en este problema siempre es uno

```
def getCostOfActions(self, actions):  
  
    if actions == None: return 999999  
    x,y= self.startingPosition  
    for action in actions:  
        dx, dy = Actions.directionToVector(action)  
        x, y = int(x + dx), int(y + dy)  
        if self.walls[x][y]: return 999999  
    return len(actions)
```

La función **getCostOfActions** recibe como parámetro la lista de acciones que debe realizar el Pacman para resolver el laberinto y devuelve el sumatorio de los costes de realizar cada una de las acciones

Si la lista de acciones esta vacía consideramos que el problema no está resuelto y devolvemos una cantidad que equivale a un coste infinito

Obtenemos las coordenadas iniciales del Pacman llamando a la variable global **startingPosition**

Para cada acción contenida en la lista recibida como parámetro

- Obtenemos las coordenadas finales tras realizar el movimiento en la dirección indicada
- Modificamos las coordenadas del Pacman a las coordenadas finales calculadas
- Comprobamos si existe algún muro comprendido entre ambas posiciones
 - o En caso afirmativo consideramos que el problema no está resuelto y devolvemos una cantidad que equivale a un coste infinito

Si el bucle que recorre todas las acciones termina con éxito (no devuelve la longitud infinita para ninguna acción) se devuelve como resultado el número de elementos de la lista recibida como parámetro. Debido a que para el problema del Pacman estamos considerando que el coste de cada acción es uniforme

Consideraciones adicionales:

En este problema, al contrario que en el anterior, la variable de estado tendrá cuatro atributos

- **0** – Las coordenadas en las que se encuentra Pacman actualmente
- **1** – Es una dupla que contendrá las coordenadas de las esquinas que ya han sido comprobadas
- **2** – La dirección que ha de tomar para llegar a otro estado
- **3** – El coste del arco

Calculo del Heurístico

Antes de calcular nuestro heurístico echaremos un vistazo a los diferentes tipos de distancia que se pueden obtener en un plano bidimensional:

```
def manhattanHeuristic(position, problem, info={}):
    x1 = position
    xy2 = problem.goal
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])

def euclideanHeuristic(position, problem, info={}):
    x1 = position
    xy2 = problem.goal
    return ( (xy1[0] - xy2[0]) ** 2 + (xy1[1] - xy2[1]) ** 2 ) ** 0.5
```

Distancia Minkowsky:

Hemos implementado una función que realiza una generalización de métricas permitiendo elegir una u otra distancia en función de un parámetro

```
def distanceToNode(n1, n2, p):
    x1,y1 = n1
    x2,y2 = n2
    distance = ((abs(x1 - x2) ** p) + (abs(y1 - y2) ** p)) ** (1 / p)
    return distance
```

- P = 1 Distancia Manhattan
- P = 2 Distancia Euclidea
- P = infinito distancia Chebychev

Para resolver nuestro problema deberemos implementar un heurístico no trivial y consistente

La admisibilidad no es suficiente para garantizar la corrección en la búsqueda en grafo, sin embargo los heurísticos admisibles suelen ser también consistentes cuando se derivan de soluciones a problemas.

La única forma de garantizar la consistencia es con una prueba. Sin embargo, la inconsistencia a menudo se puede detectar verificando:

- Que para cada nodo que expandamos sus nodos sucesores tengan un heurístico igual o mayor.
- Que UCS y A* devuelvan siempre el mismo camino como resultado optimo

Definimos las siguientes variables locales:

- **corners**: Contendrá las coordenadas de las cuatro esquinas
- **walls**: Contendrá toda la información relativa a los muros del laberinto
- **cornersVisited** Contendrá las coordenadas de las esquinas que ya han sido visitadas, esta información la rescataremos del estado del problema.
- **p** Indica el tipo de distancia que utilizaremos. En este caso Manhattan por su mayor simplicidad
- **distances**: Se trata de una lista que contendrá la distancia calculada hasta cada una de las esquinas que quedan por visitar

En primer lugar calculamos la diferencia entre todas las esquinas del problema y las esquinas que ya han sido visitadas para quedarnos solo con las esquinas que quedan por visitar.

A continuación calculamos la distancia entre la posición actual del Pacman y cada una de las esquinas que quedan por visitar y las guardamos en una lista de duplas

Si la lista de distancias está vacía:

- Implica que ya hemos visitado todas las esquinas y el problema terminaría. Por lo que el heurístico sería cero

En caso contrario:

- Obtenemos la esquina que está más cerca del Pacman, es decir la de menor distancia.
- Eliminamos dicha esquina de la lista de esquinas pendientes
- Si no quedan más esquinas por recorrer
 - o Terminaría el cálculo y devolveríamos el heurístico calculado, que sería la distancia hasta la esquina más cercana del Pacman
- Si quedan otras esquinas por visitar
 - o Cogemos la primera esquina, calculamos todas las distancias a las esquinas no visitadas restantes, y las añadimos a una nueva lista junto a sus coordenadas. Ordenamos dicha lista como antes, y volvemos a coger la distancia más cercana (es decir, la esquina más cercana a la anterior esquina más cercana). Sumamos esa distancia a la anterior, y repetimos hasta que no queden esquinas sin visitar.

```
def cornersHeuristic(state, problem):
    corners = list(problem.corners)
    walls = problem.walls

    cornersVisited = state[1]
    distances = []

    for visited in cornersVisited:
        corners.remove(visited)

    p = 1

    for corner in corners:
        distance = distanceToNode(state[0], corner, p)
        distances.append((corner, distance))

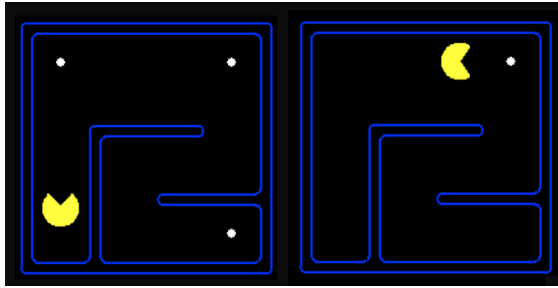
    if len(distances) == 0:
        return 0
    else:
        distances.sort(key=lambda tup: tup[1])
        minCorner = distances[0]
        distances.remove(minCorner)
        h = minCorner[1]
        newCorners = [x[0] for x in distances]
        if len(newCorners) != 0:
            d = minCorner[0]
            while len(newCorners) != 0:
                newList = []
                for c in newCorners:
                    if c != d:
                        newList.append((c, distanceToNode(d, c, p)))
                newList.sort(key=lambda tup: tup[1])
                if len(newList) != 0:
                    h += newList[0][1]
                    d = newList[0][0]
                newCorners.remove(d)
            return h
```

Soluciones de las ejecuciones

Sin heurístico

Profundidad

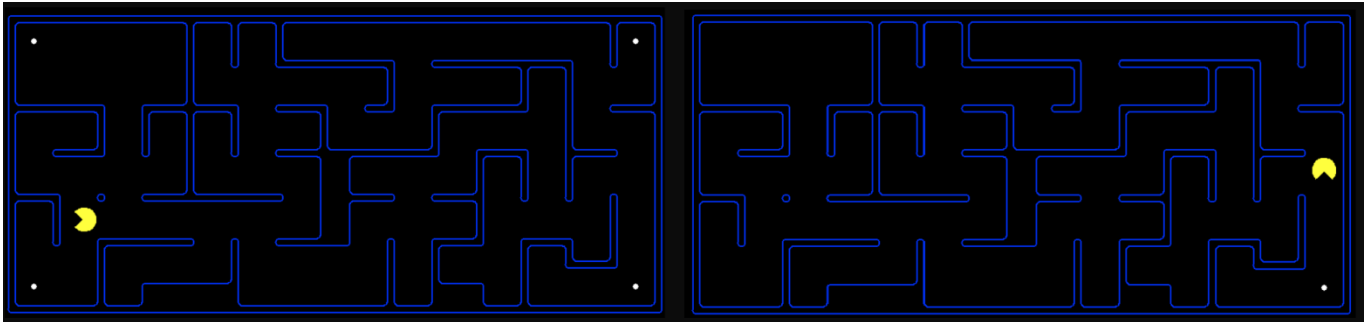
python pacman.py -l tinyCorners -p SearchAgent -a fn = dfs,prob = CornersProblem



```
Anaconda Powershell Prompt (Anaconda3)

python pacman.py -l tinyCorners -p SearchAgent -a fn=dfs,prob=CornersProblem
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 47 in 0.0 seconds
Search nodes expanded: 51
Pacman emerges victorious! Score: 493
Average Score: 493.0
Scores:      493.0
Win Rate:    1/1 (1.00)
Record:      Win
```

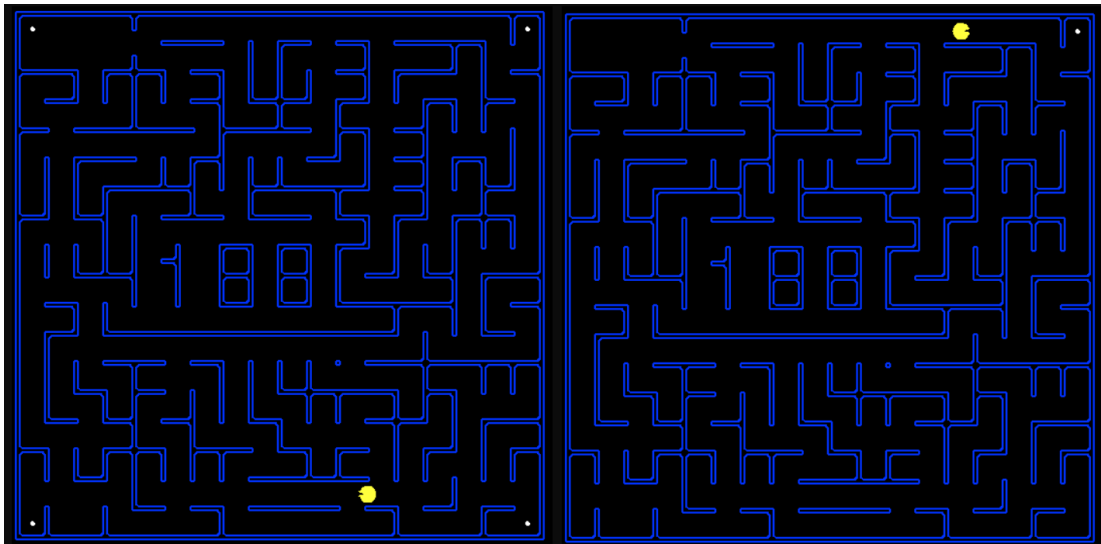
python pacman.py -l mediumCorners -p SearchAgent -a fn = dfs,prob = CornersProblem



```
Anaconda Powershell Prompt (Anaconda3)

(base) PS C:\Users\david\Desktop\searchCom\search> python pacman.py -l mediumCorners -p SearchAgent -a fn=dfs,prob=CornersProblem
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 221 in 0.0 seconds
Search nodes expanded: 380
Pacman emerges victorious! Score: 319
Average Score: 319.0
Scores:      319.0
Win Rate:    1/1 (1.00)
Record:      Win
```

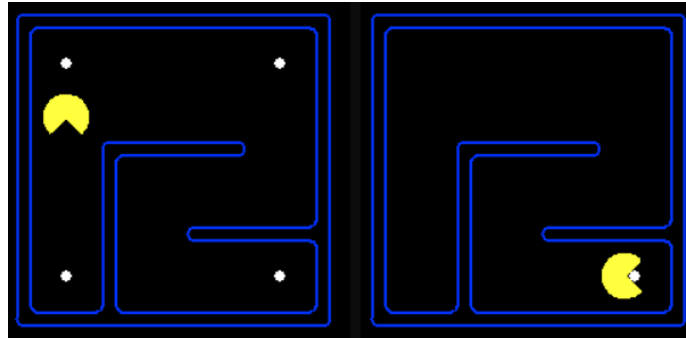
python pacman.py -l bigCorners -p SearchAgent -a fn = dfs,prob = CornersProblem -z 0.5



```
Anaconda Powershell Prompt (Anaconda3)

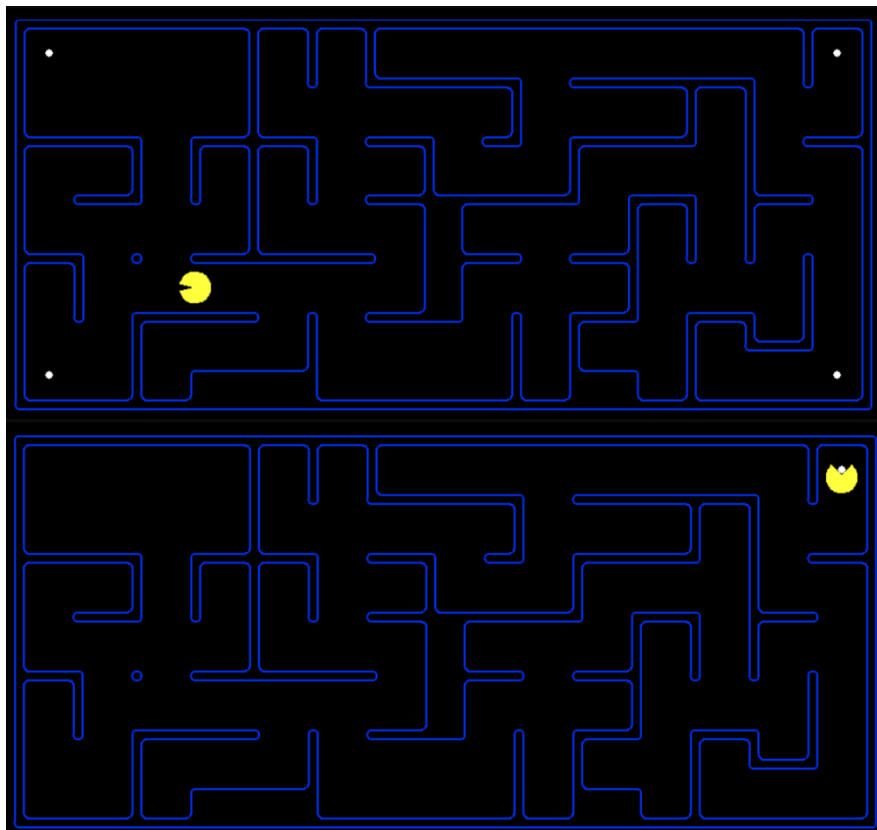
python pacman.py -l bigCorners -p SearchAgent -a fn=dfs,prob=CornersProblem -z 0.5
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 302 in 0.0 seconds
Search nodes expanded: 515
Pacman emerges victorious! Score: 238
Average Score: 238.0
Scores:      238.0
Win Rate:    1/1 (1.00)
Record:      Win
```

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



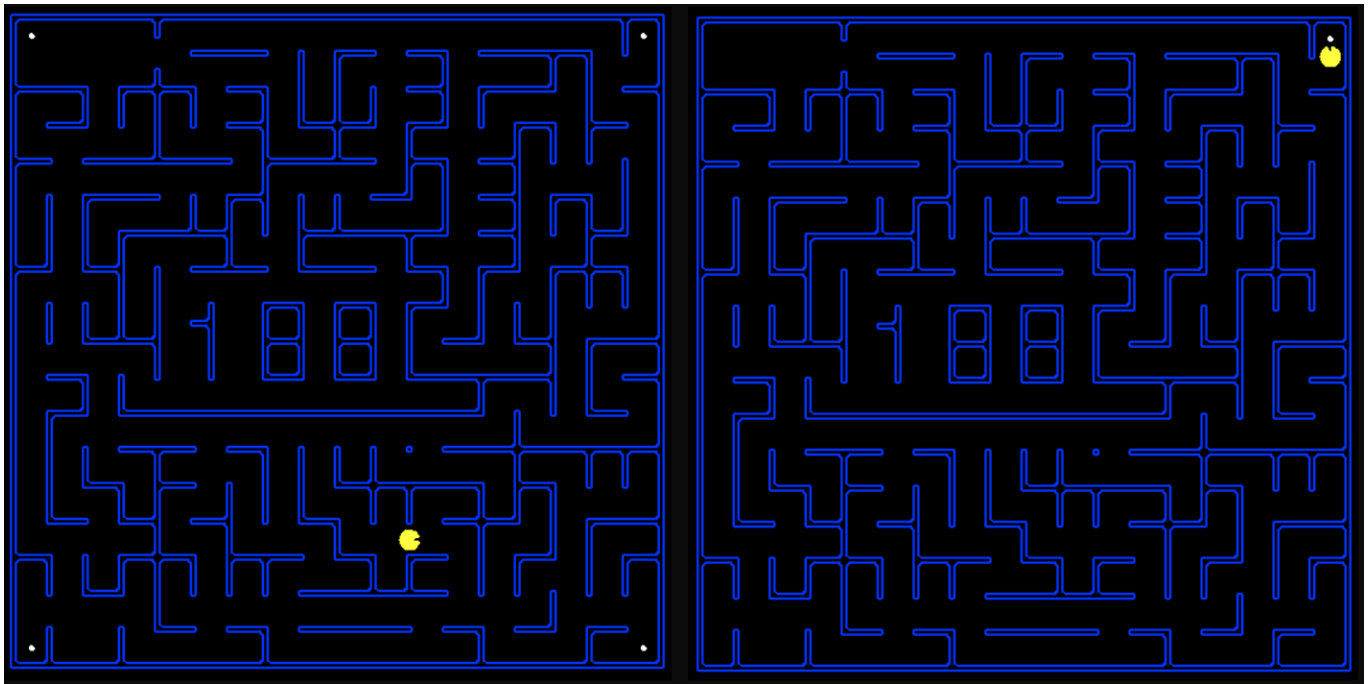
```
Anaconda Powershell Prompt (Anaconda3)
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 436
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:      512.0
Win Rate:    1/1 (1.00)
Record:      Win
```

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



```
Anaconda Powershell Prompt (Anaconda3)
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 2449
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:      434.0
Win Rate:    1/1 (1.00)
Record:      Win
```

```
python pacman.py -l bigCorners -p SearchAgent -a fn = bfs,prob = CornersProblem -z 0.5
```

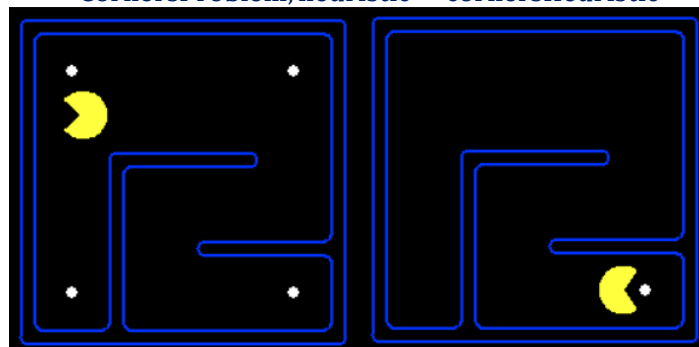


```

Anaconda Powershell Prompt (Anaconda3)
python pacman.py -l bigCorners -p SearchAgent -a fn=bfs,prob=CornersProblem -z 0.5
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 162 in 0.1 seconds
Search nodes expanded: 9905
Pacman emerges victorious! Score: 378
Average Score: 378.0
Scores:      378.0
Win Rate:    1/1 (1.00)
Record:      Win
  
```

Con el heurístico calculado

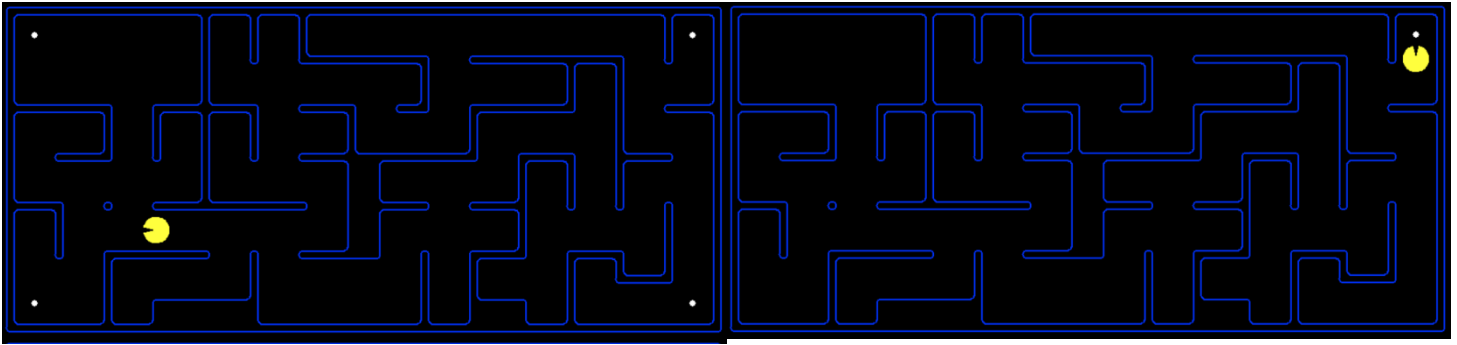
```
python pacman.py -l tinyCorners -p SearchAgent -a fn = aStarSearch,prob
= CornersProblem, heuristic = cornersHeuristic
```



```

Seleccionar Anaconda Powershell Prompt (Anaconda3)
python pacman.py -l tinyCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
[SearchAgent] using function aStarSearch and heuristic cornersHeuristic
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 218
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:      512.0
Win Rate:    1/1 (1.00)
Record:      Win
  
```

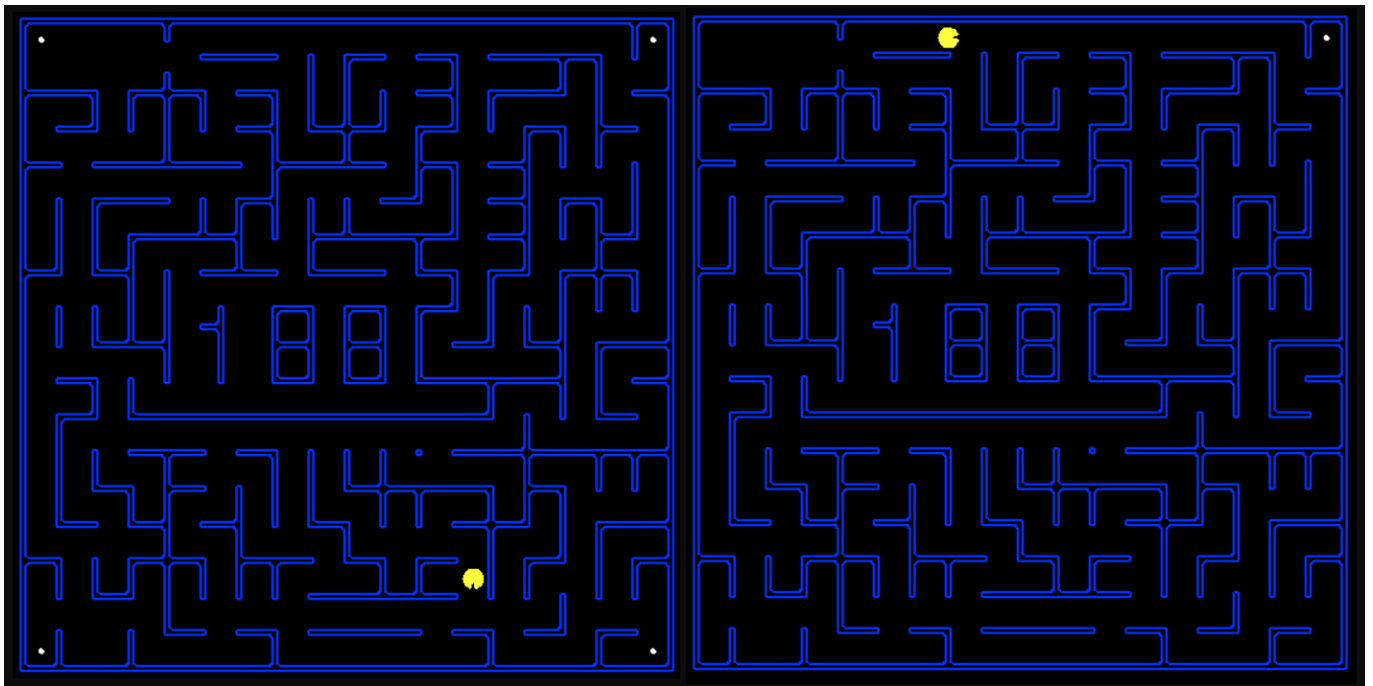

python pacman.py -l mediumCorners -p SearchAgent -a fn = aStarSearch,prob = CornersProblem,heuristic = cornersHeuristic



```
Anaconda Powershell Prompt (Anaconda3)

python pacman.py -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
[SearchAgent] using function aStarSearch and heuristic cornersHeuristic
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 902
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:      434.0
Win Rate:    1/1 (1.00)
Record:      Win
```

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



```
Anaconda Powershell Prompt (Anaconda3)

python pacman.py -l bigCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic -z 0.5
[SearchAgent] using function aStarSearch and heuristic cornersHeuristic
[SearchAgent] using problem type CornersProblem
Path found with total cost of 162 in 0.0 seconds
Search nodes expanded: 1721
Pacman emerges victorious! Score: 378
Average Score: 378.0
Scores:      378.0
Win Rate:    1/1 (1.00)
Record:      Win
(base) PS C:\Users\david\Desktop\searchCom\search>
```

Conclusiones

Sin heurístico en profundidad

Resultados de tinyCorners

- Nodos expandidos: 51
- Coste del recorrido: 593

Resultados de mediumCorners

- Nodos expandidos: 380
- Coste del recorrido: 319

Resultados de bigCorners

- Nodos expandidos: 512
- Coste del recorrido: 238

Sin heurístico en anchura

Resultados de tinyCorners

- Nodos expandidos: 436
- Coste del recorrido: 512

Resultados de mediumCorners

- Nodos expandidos: 2449
- Coste del recorrido: 434

Resultados de bigCorners

- Nodos expandidos: 9905
- Coste del recorrido: 378

A* Con el heurístico calculado

Resultados de tinyCorners

- Nodos expandidos: 218
- Coste del recorrido: 512

Resultados de mediumCorners

- Nodos expandidos: 902
- Coste del recorrido: 434

Resultados de bigCorners

- Nodos expandidos: 1721
- Coste del recorrido: 378

Observaciones

/* TODO */ en profundidad dan resultados que no son coherentes. En la ejecución da una vuelta muy grande pero después los resultados son mejores de lo que cabría esperar, mejores incluso que con el heurístico.

/* TODO */ Pregunte al profesor y me dijo que dejara los resultados estos extraños y que miraría cual era la razón

Cuando no se utilizan heurísticos se expanden muchísimos más nodos en orden exponencial con respecto al tamaño del laberinto.

No obstante el tiempo en alcanzar el objetivo siempre es óptimo dado que UCS y A* obtenían siempre caminos óptimos

Conclusiones

El heurístico que hemos calculado aporta información útil al algoritmo de búsqueda reduciendo considerablemente el tiempo de búsqueda del camino óptimo

Cuestiones a resolver

¿Si usáramos un heurístico inadmisibile en búsqueda en árbol A* podría cambiar la completitud de la búsqueda?

No, Encontraría una solución igualmente debido a que el algoritmo A* guarda todos los estados

¿Si usáramos un heurístico inadmisibile en búsqueda en árbol A* podría cambiar la optimalidad de la búsqueda?

Si, podría desempacarse una solución que no fuera la óptima al no estar siguiendo un buen criterio como guía.

Indica una ventaja general que podría tener un heurístico inadmisibile sobre uno admisibile

Si bien es cierto que un buen heurístico reduce el coste de la búsqueda al reducir el número de nodos que es necesario expandir, también cabe desatacar que no es el único factor que influye en el coste final de la solución. Un heurístico muy costoso de calcular incrementaría el tiempo necesario para realizar cada expansión. Como consecuencia, aunque el problema expanda menos nodos la solución podría no compensar.

$$\text{CosteTotal} = \text{CosteExpansion} * \text{NºNodosExpandidos}$$

Comiendo todos los puntos

Resolución

Se trata de un problema de búsqueda en el que intentaremos recorrer todos los puntos por la ruta que menos tiempo le lleve al Pacman

Las siguientes funciones ya están implementadas:

class FoodSearchProblem:

```
def __init__(self, startingGameState):
    self.start = (startingGameState.getPacmanPosition(), startingGameState.getFood())
    self.walls = startingGameState.getWalls()
    self.startingGameState = startingGameState
    self._expanded = 0 # DO NOT CHANGE
    self.heuristicInfo = {} # A dictionary for the heuristic to store information

def getStartState(self):
    return self.start

def isGoalState(self, state):
    return state[1].count() == 0

def getSuccessors(self, state):
    successors = []
    self._expanded += 1
    for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
        x,y = state[0]
        dx, dy = Actions.directionToVector(direction)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            nextFood = state[1].copy()
            nextFood[nextx][nexty] = False
            successors.append( ( (nextx, nexty), nextFood), direction, 1) )
    return successors

def getCostOfActions(self, actions):
    x,y = self.getStartState()[0]
    cost = 0
    for action in actions:
        # figure out the next state and see whether it's legal
        dx, dy = Actions.directionToVector(action)
        x, y = int(x + dx), int(y + dy)
        if self.walls[x][y]:
            return 999999
        cost += 1
    return cost
```

Calculo del Heurístico

Deberemos implementar un heurístico no trivial y consistente

Definimos las siguientes variables locales:

- **position:** Coordenadas actuales del Pacman
- **foodGrid:** Matriz con información sobre los puntos con comidas
- **foodCoordsList:** Contendrá las coordenadas todos los puntos objetivo que deben ser visitados, esta información la rescataremos del estado del problema.
- **mean** /*TODO*/
- **distances:** Se trata de una lista que contendrá la distancia calculada hasta cada uno de los puntos que quedan por visitar

En primer lugar calculamos la distancia entre la posición actual del Pacman y cada una de los puntos que quedan por visitar y las guardamos en una lista

Si la lista de distancias está vacía:

- Implica que ya hemos visitado todas las esquinas y el problema terminaría. Por lo que el heurístico sería cero

En caso contrario:

- Ordenamos las distancias de menor a mayor
- Las dividimos en dos bloques que representan las distancias más cercanas y lejanas respectivamente
- Si el segundo bloque no está vacío, devolvemos la media de sus distancias (es decir, la media de las distancias más lejanas). Si no, devolvemos la media de las distancias del primer bloque.

def foodHeuristic(state, problem):

```
    position, foodGrid = state

    p = 1 # Manhattan distance

    foodCoordsList = foodGrid.asList()
    mean = 0
    distances = []
    for foodCoords in foodCoordsList:
        distance = distanceToNode(position, foodCoords, p)
        distances.append(distance)

    import statistics
    if len(distances) == 0:
        return 0
    else:
        distances.sort()
        d1 = distances[:len(distances)//2]
        d2 = distances[len(distances) // 2:]
        if len(d2) != 0:
            h = statistics.mean(d2)
        else:
            h = statistics.mean(d1)

    return h
```

Soluciones de las ejecuciones

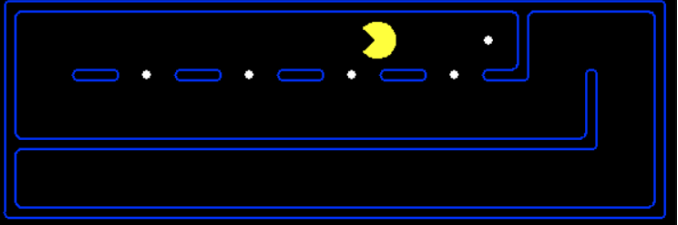
Sin heurístico

Profundidad

python pacman.py -l trickySearch -p SearchAgent -a fn=dfs,prob=FoodSearchProblem

```
Anaconda Powershell Prompt (Anaconda3)

python pacman.py -l trickySearch -p SearchAgent -a fn=dfs,prob=FoodSearchProblem
[SearchAgent] using function dfs
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 216 in 0.0 seconds
Search nodes expanded: 362
Pacman emerges victorious! Score: 414
Average Score: 414.0
Scores:      414.0
Win Rate:    1/1 (1.00)
Record:      Win
```

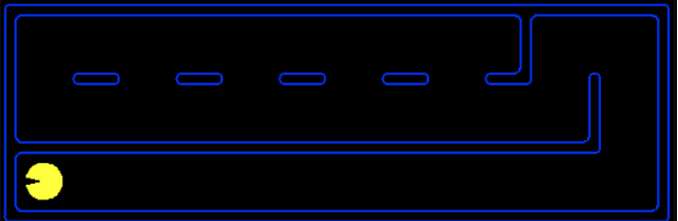


Anchura

python pacman.py -l trickySearch -p SearchAgent -a fn=bfs,prob=FoodSearchProblem

```
Anaconda Powershell Prompt (Anaconda3)

python pacman.py -l trickySearch -p SearchAgent -a fn=bfs,prob=FoodSearchProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 60 in 1.6 seconds
Search nodes expanded: 16689
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:      570.0
Win Rate:    1/1 (1.00)
Record:      Win
```

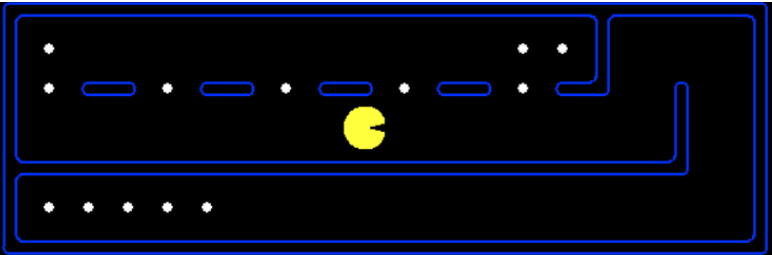


Con el heurístico calculado

python pacman.py -l trickySearch -p AStarFoodSearchAgent

```
Anaconda Powershell Prompt (Anaconda3)

python pacman.py -l trickySearch -p AStarFoodSearchAgent
Path found with total cost of 60 in 1.7 seconds
Search nodes expanded: 10181
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:      570.0
Win Rate:    1/1 (1.00)
Record:      Win
```



Conclusiones

Sin heurístico en profundidad

Resultados de trickySearch

- Nodos expandidos: 362
- Coste del recorrido: 414

Sin heurístico en anchura

Resultados de trickySearch

- Nodos expandidos: 16689
- Coste del recorrido: 570

A* Con el heurístico calculado

Resultados de trickySearch

- Nodos expandidos: 10181
- Coste del recorrido: 570

Observaciones

/* TODO */ en profundidad dan resultados que no son coherentes. En la ejecución da una vuelta muy grande pero después los resultados son mejores de lo que cabría esperar, mejores incluso que con el heurístico.

/* TODO */ Pregunte al profesor y me dijo que dejara los resultados estos extraños y que miraría cual era la razón

Cuando no se utilizan heurísticos se expanden muchísimos más nodos en orden exponencial con respecto al tamaño del laberinto.

No obstante el tiempo en alcanzar el objetivo siempre es óptimo dado que UCS y A* obtenían siempre caminos óptimos

Conclusiones

El heurístico que hemos calculado aporta información útil al algoritmo de búsqueda reduciendo considerablemente el tiempo de búsqueda del camino óptimo

Búsqueda subóptima (Suboptimal Search)

Incluso con A* y una buen heurístico en algunas ocasiones resulta complicado encontrar la ruta óptima a través de todos los puntos.

En estos casos puede ser mejor opción encontrar un camino razonablemente bueno rápidamente, que aunque no sea el óptimo encuentre no requiera muchos cálculos.

Resolución

```
class ClosestDotSearchAgent(SearchAgent):
    def registerInitialState(self, state):
        self.actions = []
        currentState = state
        while(currentState.getFood().count() > 0):
            nextPathSegment = self.findPathToClosestDot(currentState) # The missing piece
            self.actions += nextPathSegment
            for action in nextPathSegment:
                legal = currentState.getLegalActions()
                if action not in legal:
                    t = (str(action), str(currentState))
                    raise Exception('findPathToClosestDot returned an illegal move: %s!\n%s' % t)
                currentState = currentState.generateSuccessor(0, action)
            self.actionIndex = 0
            print('Path found with cost %d.' % len(self.actions))

    def findPathToClosestDot(self, gameState):
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        return search.breadthFirstSearch(problem)
```

La función que faltaba en el algoritmo era la que devolvía el resultado.

En nuestro caso optamos por utilizar el algoritmo de la búsqueda en anchura pasándole como parámetro el AnyFoodSearchProblem.

```
class AnyFoodSearchProblem(PositionSearchProblem):
    def __init__(self, gameState):
        # Store the food for later reference
        self.food = gameState.getFood()

        # Store info for the PositionSearchProblem (no need to change this)
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        self._visited, self._visitedlist, self._expanded = {}, [], 0

    def isGoalState(self, state):
        x,y = state
        return state in self.food.asList()
```

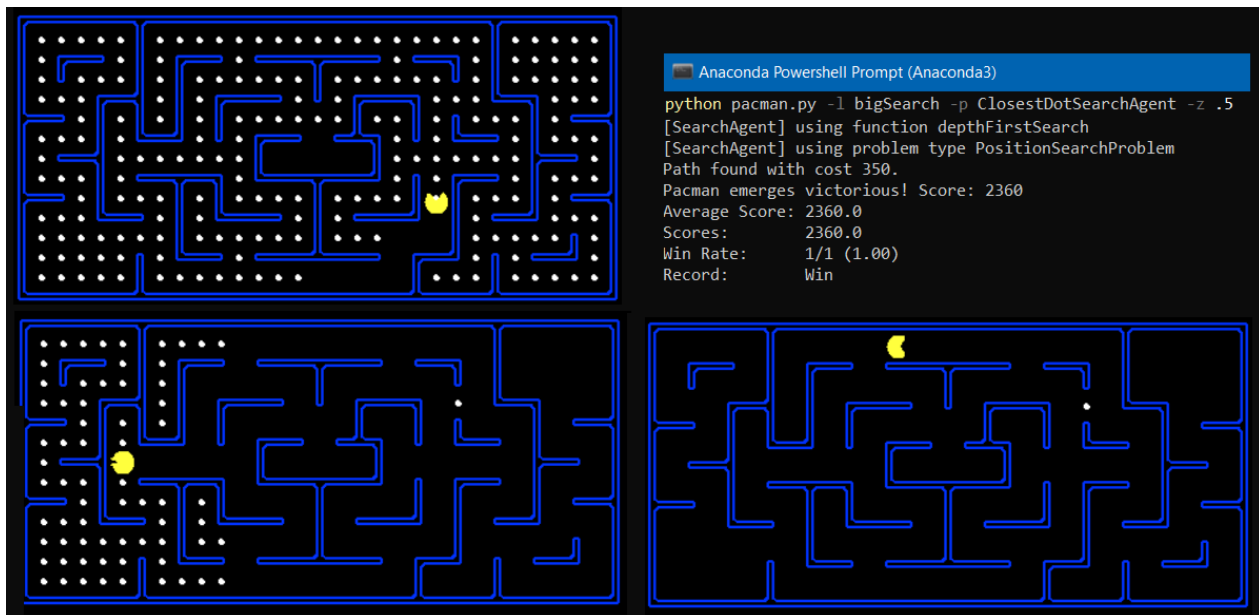
La función que faltaba en el algoritmo era la que determinaba si el problema ha terminado

En nuestro caso el problema termina cuando todos los puntos han sido comidos

Juntando estos dos elementos, se genera una búsqueda en la que a cada paso tenemos un nuevo problema que buscará en anchura la comida más cercana

Soluciones de las ejecuciones

`python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5`



Conclusiones

Sin heurístico en profundidad

Resultados de trickySearch

- Coste del recorrido: 2360

Observaciones

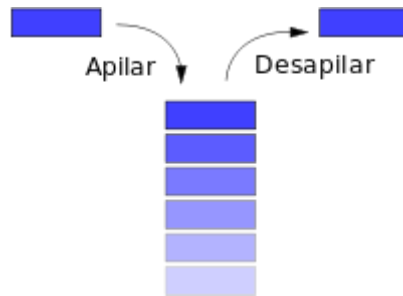
Podemos observar que Pacman se dejó un punto un punto clave para el final del recorrido, por lo que claramente la solución no es óptima. Pero el cálculo del problema fue casi instantáneo

Apéndices:

Conceptos relacionados

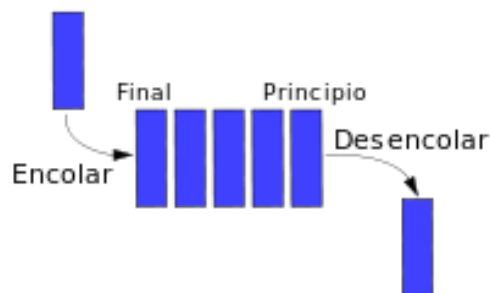
Pilas de datos

Se trata de una colección lineal de elementos en la que sólo se pueden añadir y retirar elementos por uno de sus extremos. De este modo se consigue asignar una mayor prioridad a aquellos elementos que se hallan añadido recientemente



Colas de datos

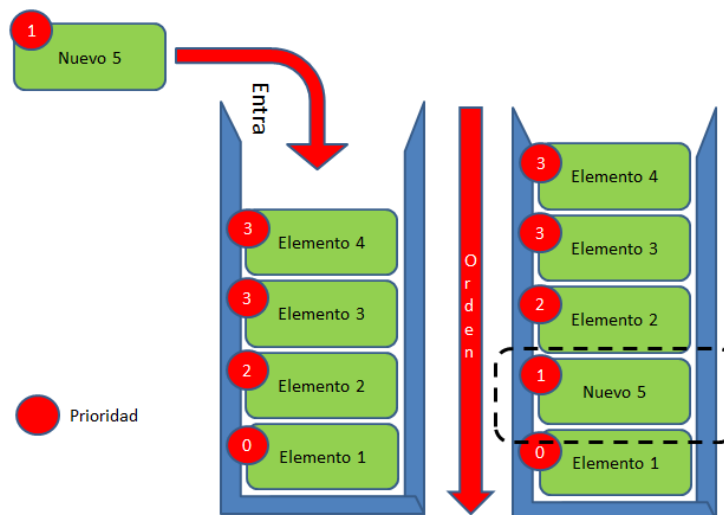
Se trata de una colección lineal de elementos en la que se puede añadir elementos por un extremo pero se retiran por el otro. De este modo se consigue asignar una mayor prioridad a aquellos elementos que se hallan añadido primero



Colas de prioridad

Se trata de una colección de elementos de dato abstracto similar a una cola en la que los elementos tienen adicionalmente, una prioridad asignada.

- En una cola de prioridades un elemento con mayor prioridad será desencolado antes que un elemento de menor prioridad.
- Si dos elementos tienen la misma prioridad, se desencolarán siguiendo el orden de cola.



Resultados del Autograder

Resultados

Finished at 10:08:21

Provisional grades

=====

Question q1: 3/3

Question q2: 3/3

Question q3: 3/3

Question q4: 3/3

Question q5: 3/3

Question q6: 3/3

Question q7: 3/4

Question q8: 3/3

Total: 24/25

Your grades are NOT yet registered. To register your grades, make sure to follow your instructor's guidelines to receive credit on your project.

Apartado Q1

Question q1

=====

*** PASS: test_cases\q1\graph_backtrack.test

*** solution: ['1:A->C', '0:C->G']

*** expanded_states: ['A', 'D', 'C']

*** PASS: test_cases\q1\graph_bfs_vs_dfs.test

*** solution: ['2:A->D', '0:D->G']

*** expanded_states: ['A', 'D']

*** PASS: test_cases\q1\graph_infinite.test

*** solution: ['0:A->B', '1:B->C', '1:C->G']

*** expanded_states: ['A', 'B', 'C']

*** PASS: test_cases\q1\graph_manypaths.test

*** solution: ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']

*** expanded_states: ['A', 'B2', 'C', 'D', 'E2', 'F']

*** PASS: test_cases\q1\pacman_1.test

*** pacman layout: mediumMaze

*** solution length: 130

*** nodes expanded: 146

Question q1: 3/3

Apartado Q2

Question q2

=====

*** PASS: test_cases\q2\graph_backtrack.test

*** solution: ['1:A->C', '0:C->G']

*** expanded_states: ['A', 'B', 'C', 'D']

*** PASS: test_cases\q2\graph_bfs_vs_dfs.test

*** solution: ['1:A->G']

*** expanded_states: ['A', 'B']

*** PASS: test_cases\q2\graph_infinite.test

*** solution: ['0:A->B', '1:B->C', '1:C->G']

*** expanded_states: ['A', 'B', 'C']

*** PASS: test_cases\q2\graph_manypaths.test

*** solution: ['1:A->C', '0:C->D', '1:D->F', '0:F->G']

*** expanded_states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

*** PASS: test_cases\q2\pacman_1.test

*** pacman layout: mediumMaze

*** solution length: 68

*** nodes expanded: 270

Question q2: 3/3

Apartado Q3

```
Question q3
=====
*** PASS: test_cases\q3\graph_backtrack.test
***   solution:      ['1:A->C', '0:C->G']
***   expanded_states: ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_bfs_vs_dfs.test
***   solution:      ['1:A->G']
***   expanded_states: ['A', 'B']
*** PASS: test_cases\q3\graph_infinite.test
***   solution:      ['0:A->B', '1:B->C', '1:C->G']
***   expanded_states: ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_manypaths.test
***   solution:      ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***   expanded_states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q3\ucs_0_graph.test
***   solution:      ['Right', 'Down', 'Down']
***   expanded_states: ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q3\ucs_1_problemC.test
***   pacman layout: mediumMaze
***   solution length: 68
***   nodes expanded: 270
*** PASS: test_cases\q3\ucs_2_problemE.test
***   pacman layout: mediumMaze
***   solution length: 74
***   nodes expanded: 261
*** PASS: test_cases\q3\ucs_3_problemW.test
***   pacman layout: mediumMaze
***   solution length: 152
***   nodes expanded: 173
*** PASS: test_cases\q3\ucs_4_testSearch.test
***   pacman layout: testSearch
***   solution length: 7
***   nodes expanded: 15
*** PASS: test_cases\q3\ucs_5_goalAtDequeue.test
***   solution:      ['1:A->B', '0:B->C', '0:C->G']
***   expanded_states: ['A', 'B', 'C']

### Question q3: 3/3 ###
```

Apartado Q4

```
Question q4
=====
*** PASS: test_cases\q4\astar_0.test
***   solution:      ['Right', 'Down', 'Down']
***   expanded_states: ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q4\astar_1_graph_heuristic.test
***   solution:      ['0', '0', '2']
***   expanded_states: ['S', 'A', 'D', 'C']
*** PASS: test_cases\q4\astar_2_manhattan.test
***   pacman layout: mediumMaze
***   solution length: 68
***   nodes expanded: 222
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***   solution:      ['1:A->B', '0:B->C', '0:C->G']
***   expanded_states: ['A', 'B', 'C']
*** PASS: test_cases\q4\graph_backtrack.test
***   solution:      ['1:A->C', '0:C->G']
***   expanded_states: ['A', 'B', 'C', 'D']
*** PASS: test_cases\q4\graph_manypaths.test
***   solution:      ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***   expanded_states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###
```

Apartado Q5

```
Question q5
=====
*** PASS: test_cases\q5\corner_tiny_corner.test
***      pacman layout:      tinyCorner
***      solution length:      28

### Question q5: 3/3 ###
```

Apartado Q6

```
Question q6
=====
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'West', 'West', 'South', 'South', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'South', 'West', 'West', 'South', 'South', 'South', 'West', 'West', 'East', 'East', 'North', 'North', 'North', 'East', 'East', 'East', 'East', 'East', 'East', 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'South', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'North', 'North', 'East', 'East', 'North', 'North']
path length: 106
*** PASS: Heuristic resulted in expansion of 902 nodes

### Question q6: 3/3 ###
```

Apartado Q7

```
Question q7
=====
*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** FAIL: test_cases\q7\food_heuristic_grade_tricky.test
***      expanded nodes: 10181
***      thresholds: [15000, 12000, 9000, 7000]

### Question q7: 3/4 ###
```

Apartado Q8

```
Question q8
=====
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_1.test
***   pacman layout:      Test 1
***   solution length:    1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_10.test
***   pacman layout:      Test 10
***   solution length:    1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_11.test
***   pacman layout:      Test 11
***   solution length:    2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_12.test
***   pacman layout:      Test 12
***   solution length:    3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_13.test
***   pacman layout:      Test 13
***   solution length:    1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_2.test
***   pacman layout:      Test 2
***   solution length:    1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_3.test
***   pacman layout:      Test 3
***   solution length:    1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_4.test
***   pacman layout:      Test 4
***   solution length:    3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_5.test
***   pacman layout:      Test 5
***   solution length:    1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_6.test
***   pacman layout:      Test 6
***   solution length:    2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_7.test
***   pacman layout:      Test 7
***   solution length:    1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_8.test
***   pacman layout:      Test 8
***   solution length:    1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_9.test
***   pacman layout:      Test 9
***   solution length:    1
### Question q8: 3/3 ###
```

Referencias

Todo el contenido teórico así como las imágenes utilizadas para este guion de prácticas han sido obtenidos de las siguientes fuentes:

Apuntes de la asignatura de IA (Ekaitz Jauregi, Eneko Agirre, Juanma Pikatza)

<https://inst.eecs.berkeley.edu/~cs188/sp19>

Guion de prácticas de los laboratorios