

Práctica 1: Búsqueda

Autores: Carlos Aliaga y Eduardo Terrés

Sección 1 (1 punto)

1.1. Enfoque y decisiones de la solución

Cabe destacar que este apartado contiene explicaciones de los siguientes tres, de forma que será referenciado en secciones posteriores.

Se plantea el problema de implementar diferentes algoritmos de búsqueda para ayudar a PacMan navegar el laberinto con éxito. Dado lo parecidas que son las implementaciones de las secciones 1.1., 1.2., 1.3., y 1.4. –aunque las dos últimas conlleven la necesidad adicional de implementar una heurística –, hemos decidido, tal y como se propone, crear una función de búsqueda general. La función en cuestión es `busquedaGeneral`, que presenta la siguiente declaración:

```
def busquedaGeneral(problem, listaAbiertos)
```

Los argumentos de entrada son el problema y la lista de abiertos. La variable problema se pasa directamente de las llamadas a la correspondiente función del apartado.

Cabe destacar que la definición de nodo que hemos dado consiste en una tupla, conteniendo el estado de PacMan – coordenada X e Y dentro de la matriz de posiciones – y una lista, que incluye las acciones a realizar desde el comienzo de la partida hasta la posición de PacMan. De esta forma, resulta muy directo encontrar la solución – vista como el conjunto de acciones que debe realizar PacMan – ya que coincide con el segundo elemento de la tupla del estado objetivo.

La lógica de la función consiste en una serie de controles de errores y un bucle infinito principal. Se empieza explorando el nodo inicial, y desde ahí comienza la lógica del algoritmo general, que va sacando los nodos de la lista de abiertos – independientemente de si es pila, cola o cola de prioridad –, los expande y añade los sucesores a la lista de abiertos en caso de que el nodo en cuestión no se encontrase ya en la lista de cerrados, de forma que existe eliminación de estados repetidos. La función tiene dos puntos de salida: devuelve una lista vacía en el caso de que la lista de abiertos se encuentre vacía – este escenario implica que no se ha encontrado solución al problema –; devuelve la lista con las acciones a realizar, en caso de que un nodo ha cumplido la función de estado objetivo – este escenario implica que se ha encontrado solución –.

La particularización del algoritmo de búsqueda empleado en cada apartado se determinará mediante el tipo de objeto al que pertenezca `listaAbiertos`, y será explicada en el apartado correspondiente. En el caso de búsqueda en profundidad, `listaAbiertos` es una pila, de forma que los nodos insertados más recientemente sean los siguientes en explorar, obteniendo así el funcionamiento característico del algoritmo, que prioriza llegar hasta el final de una rama antes de explorar nodos de menor profundidad. De esta manera, la función `depthFirstSearch()` tan solo hace una llamada a la de búsqueda general, con una pila como lista de abiertos.

1.1.1. Funciones del framework usadas

Las funciones del framework usadas para el método general son las siguientes:

- *getStartState()*: definida en la clase *SearchProblem*, dentro de *search.py*, aunque la implementación se da en *searchAgents.py*. La finalidad es obtener el estado inicial del problema.
- *isGoalState()*: definida en la clase *SearchProblem*, dentro de *search.py*. Se trata de un método que comprueba si un nodo es objetivo, devolviendo *True* en ese caso y *False* en caso contrario.
- *push()*: un método definido para la lista de abiertos. Introduce en la lista de abiertos un objeto.
- *isEmpty()*: un método definido para la lista de abiertos. Devuelve *True* en caso de que la lista de abiertos este vacía.
- *pop()*: un método definido para la lista de abiertos. Extrae el siguiente objeto en función del orden de extracción implementado.
- *getSuccessors()*: definida en la clase *SearchProblem*, dentro de *search.py*. Devuelve una lista de 3-tuplas de todos los estados que derivan del estado introducido como parámetro.

Específicamente en este caso usamos *Stack()*, una clase implementada en *util.py* para crear una pila que usaremos como implementación de la lista de abiertos.

1.1.2. Incluye el código añadido

```
def busquedaGeneral(problem, lista_abiertos):
    """
    Algoritmo de busqueda general.
    Se le pasa como argumento la lista de abiertos que ha de ser la estructura de datos
    que define la estrategia de busqueda.
    Devuelve la lista de acciones que se ha de realizar para resolver el problema.
    """

    # Control de errores
    if problem.isGoalState(problem.getStartState()) \
        or lista_abiertos.isEmpty() == False:
        return []

    #Inicializamos lista con la raiz
    lista_abiertos.push((problem.getStartState(),[]))
    lista_cerrados = []

    #Iteramos
    while (True):
        #Si la lista esta vacia no hemos encontrado solucion
        if (lista_abiertos.isEmpty()):
            return []

        #Elegimos un nodo conforme a estrategia
        estado,camino = lista_abiertos.pop()
        #Si el nodo cumple el test devolvemos el camino hacia el
        if (problem.isGoalState(estado)):
            return camino
```

```

if (estado not in lista_cerrados):
    #Añadimos el nodo a la lista de cerrados
    lista_cerrados.append(estado)
    # Expande el nodo
    # El nodo ya fue eliminado mediante pop
    sucesores = problem.getSuccessors(estado)
    # Añade la información del estado del hijo y el camino que conduce hasta él
    for (estado_hijo, accion_hijo, coste_hijo) in sucesores:
        camino_hijo = []
        camino_hijo.extend((camino+[accion_hijo] if camino != None\
                             else accion_hijo))
        lista_abiertos.push((estado_hijo, camino_hijo))

```

```

def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    pila = util.Stack()
    return busquedaGeneral(problem, pila)

```

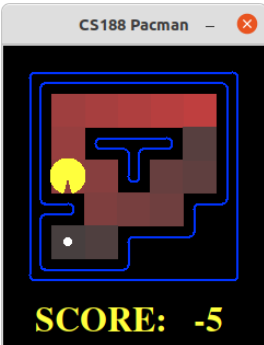
1.1.3. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

Resultados de la ejecución de los comandos:

```

edu@edu:~/Escritorio/search$ python3 pacman.py -l tinyMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
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

```



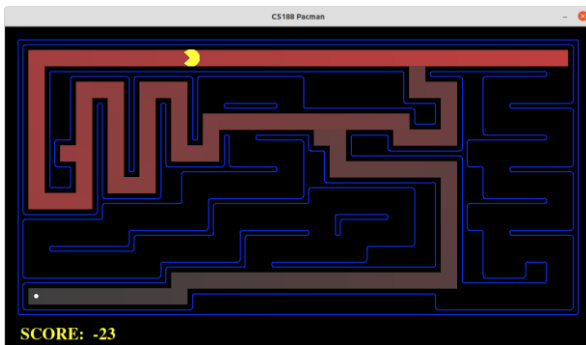
SCORE: -5

python pacman.py -l tinyMaze -p SearchAgent

```

edu@edu:~/Escritorio/search$ python3 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

```

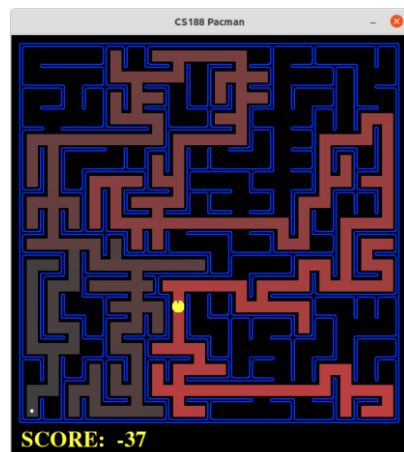


```
python pacman.py -l mediumMaze -p SearchAgent
```

```

edu@edu:~/Escritorio/search$ python3 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

```



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

```

edu@edu:~/Escritorio/search$ python3 autograder.py -q q1
Starting on 2-26 at 20:36:51

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_imp.test
***     solution:      []
***     expanded_states: ['A', 'C', 'B']
*** 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
*** PASS: test_cases/q1/pacman_imp.test
***     pacman layout:  trapped
***     solution length: 0
***     nodes expanded:  6

### Question q1: 7/7 ###

Finished at 20:36:51

Provisional grades
=====
Question q1: 7/7
-----
Total: 7/7

```

python autograder.py -q q1

Los resultados se concentran en la siguiente tabla:

	Nº nodos expandidos	Nº nodos camino	Ratio
TinyMaze	15	10	1,500
MediumMaze	146	130	1,123
BigMaze	390	210	1,857

Estos datos se compararán con el algoritmo de búsqueda en anchura, ya que arrojan información del mayor coste computacional necesario para encontrar la solución óptima.

1.2. Conclusiones en el comportamiento de pacman

El comportamiento de PacMan es correcto, en el sentido que **sí** encuentra una solución al problema. Se debe a que el algoritmo de búsqueda en profundidad es completo. No obstante, la solución encontrada **no** es óptima. Esto no implica que la implementación dada sea incorrecta, ya que el camino obtenido mediante el algoritmo de búsqueda en profundidad no es necesariamente óptimo. Intuitivamente, siempre que el nodo objetivo óptimo no se encuentre lo más a la izquierda

del árbol de exploración, no se podrá asegurar optimalidad, ya que el algoritmo cogerá el camino a profundizar simplemente por convenio, sin lógica alguna.

Este fenómeno es fácilmente visible en el recorrido y orden de exploración del laberinto mediano. Al comienzo, el camino de la izquierda se explora antes que el de abajo, por lo que toda la búsqueda continuará por esa rama – dado que existe una solución por esa rama -. En la siguiente intersección sucede lo mismo, el camino de la izquierda se explora primero y si no llega a ser porque ambas direcciones son conexas, no se habría llegado a explorar el camino de abajo.

1.2.1. ¿El orden de exploración es el que esperabais?

Sí, como se explicó en el párrafo anterior, al llegar una intersección, la dirección que se explore en primer lugar será investigada a fondo.

1.2.2. ¿Pacman realmente va a todas las casillas exploradas en su camino hacia la meta?

No, siguiendo la lógica del algoritmo, que coincide con el resultado de la ejecución observado, un nodo explorado no es necesariamente parte del camino solución. De no ser así, al explorar la solución, PacMan estaría obligado a recorrer todos los caminos sin salida del laberinto, de forma que estaría obligado a dar media vuelta y volver por el mismo lugar, cosa imposible ya que no se expanden nodos que ya se encuentren en la lista de cerrados. Por esta razón, el camino resultante es un camino directo.

1.2.3. ¿Es esta una solución de menor coste?

No para los laberintos tinyMaze y mediumMaze, es suficiente para responder a esta pregunta leer la siguiente sección, donde se encuentran caminos de menor coste para los mismos laberintos, de forma que los caminos de esta sección no son de menor coste.

No obstante, aunque no es una garantía, se consigue obtener el camino de menor coste para el laberinto de mayor tamaño.

Sección 2 (1 punto)

2.1. Enfoque y decisiones de la solución

Haciendo referencia al punto 1.1., la función *busquedaGeneral()* implementa la totalidad de la lógica necesaria para la función del algoritmo de búsqueda en anchura.

En el caso de búsqueda en anchura, listaAbiertos es una cola, de forma que los nodos que lleven más tiempo en la cola serán explorados antes. Intuitivamente, esta lógica conduce a una exploración de todas las ramas a una misma velocidad. De esta manera, la función *breadthFirstSearch()* tan solo hace una llamada a la de búsqueda general, con una cola como lista de abiertos.

2.1.1. Funciones del framework usadas


Además de los métodos y funciones ya explicados en el apartado 1.1.1., en este caso usamos *Queue()*, una clase implementada en *util.py* para crear una cola que usaremos como implementación de la lista de abiertos.

2.1.2. Incluye el código añadido

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    cola = util.Queue()
    return busquedaGeneral(problem, cola)
```

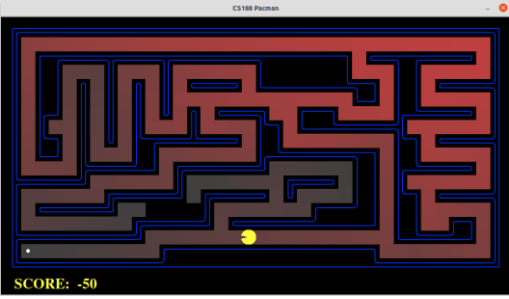
2.1.3. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

```
edu@edu:~/Escritorio/search$ python3 pacman.py -l tinyMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:      502.0
Win Rate:    1/1 (1.00)
Record:      Win
```



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

```
edu@edu:~/Escritorio/search$ python3 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
```



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

```
edu@edu:~/Escritorio/search$ python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 --frameTime 0
[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
```



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

Resultado de la ejecución del puzle propuesto en el enunciado :

```
edu@edu:~/Escritorio/search$ python3 eightpuzzle.py
A random puzzle:
| 3 | 1 | 4 |
|   | 5 | 2 |
| 6 | 7 | 8 |
-----
BFS found a path of 7 moves: ['up', 'right', 'right', 'down', 'left', 'up', 'left']
After 1 move: up
|   | 1 | 4 |
| 3 | 5 | 2 |
| 6 | 7 | 8 |
-----
Press return for the next state...
After 2 moves: right
| 1 |   | 4 |
| 3 | 5 | 2 |
| 6 | 7 | 8 |
-----
Press return for the next state...
After 3 moves: right
| 1 | 4 |   |
| 3 | 5 | 2 |
| 6 | 7 | 8 |
-----
Press return for the next state...
After 4 moves: down
| 1 | 4 | 2 |
| 3 | 5 |   |
| 6 | 7 | 8 |
-----
Press return for the next state...
After 4 moves: down
| 1 | 4 | 2 |
| 3 | 5 |   |
| 6 | 7 | 8 |
-----
Press return for the next state...
After 5 moves: left
| 1 | 4 | 2 |
| 3 |   | 5 |
| 6 | 7 | 8 |
-----
Press return for the next state...
After 6 moves: up
| 1 |   | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
-----
Press return for the next state...
After 7 moves: left
|   | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
-----
Press return for the next state...
```

python eightpuzzle.py

```
edu@edu:~/Escritorio/search$ python3 autograder.py -q q2
Starting on 2-26 at 20:41:39

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_imp.test
***   solution:          []
***   expanded_states:   ['A', 'B', 'C']
*** 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:    269
*** PASS: test_cases/q2/pacman_imp.test
***   pacman layout:     trapped
***   solution length:   0
***   nodes expanded:    6

### Question q2: 7/7 ###

Finished at 20:41:39
Provisional grades
=====
Question q2: 7/7
-----
Total: 7/7
```

python autograder.py -q q2

Los resultados se concentran en la siguiente tabla:

	Nº nodos expandidos	Nº nodos camino	Ratio
TinyMaze	15	8	1,875
MediumMaze	269	68	3,956
BigMaze	620	210	2,952

Comparación con el algoritmo de búsqueda en profundidad

Se observa que los caminos obtenidos son de menor o igual coste, en el caso de bigMaze. No obstante ha sido necesaria una mayor cantidad de exploración. No solo el ratio de nodos explorados por nodos en el camino final es mayor, sino que en los tres laberintos, el número de nodos explorados es mayor con este tipo de búsqueda en comparación con la búsqueda de la sección 1.

2.2. Conclusiones en el comportamiento de pacman

El comportamiento de PacMan es correcto, en el sentido que **sí** encuentra una solución al problema, y además esta solución es óptima. Se debe a que el algoritmo de búsqueda en anchura es completo y optimo.

Este fenómeno es fácilmente visible en el recorrido y orden de exploración del laberinto mediano. El gradiente de colores disminuye la intensidad a medida que se acerca a la esquina inferior izquierda. Los dos caminos que derivan del comienzo tienen misma tonalidad, ya que han sido explorados a la vez.

2.2.1. ¿BA encuentra una solución de menor coste?

Como se ha explicado en el párrafo anterior, el algoritmo de búsqueda en anchura es óptimo. No obstante, se dará una demostración intuitiva.

La exploración por niveles de profundidad nunca dejará atrás caminos alternativos para adentrarse en otros más profundos de forma que, suponiendo que existen dos soluciones, una requiriendo de menos pasos, el estudio de ambas se hará al mismo nivel de profundidad cada vez. De esta forma nunca se podrá dar como solución la de mayor profundidad, ya que no se podrá haber llegado al nodo final de este camino sin que el algoritmo haya concluido la búsqueda en el camino de menor coste de la hipótesis, ya que la solución por este camino se encuentra a una menor profundidad.

Es condición necesaria para que se pueda garantizar la optimalidad del algoritmo que el coste de todos los movimientos sea el mismo. Solo bajo esta condición se puede asemejar el coste de un camino a su profundidad dentro del árbol.

Sección 3 (1 punto)

3.1. Enfoque y decisiones de la solución

Haciendo referencia al punto 1.1., la función *busquedaGeneral()* implementa la totalidad de la lógica necesaria para la sección 3.

En el caso de algoritmo de búsqueda en grafo de coste uniforme, listaAbiertos es una cola de prioridad. De esta manera, la función *uniformCostSearch()* hace una llamada a la de búsqueda general, con una cola de prioridad como lista de abiertos.

La función de prioridad para la cola se declara como sigue, donde item es el nodo al que se asigna la prioridad dentro de la cola:

```
def funcionCosteUniforme (item)
```

En esencia, esta función es una cobertura para el método *getCostOfActions*, explicado en el siguiente subapartado. Es este método el que implementa toda la lógica necesaria para parametrizar la prioridad que se debe dar a distintos nodos. Cabe destacar que hemos optado por declarar una variable global en la que almacenar el problema que se pasa como parámetro de entrada a la función *uniformCostSearch*, de forma que se puede emplear en la función de prioridad. Hemos preferido esta implementación frente a usar una expresión lambda, ya que así queda un código más legible y fragmentado.

3.1.1. Funciones del framework usadas

Además de los métodos y funciones ya explicados en el apartado 1.1.1., en este caso usamos *PriorityQueueWithFunction()*, una clase implementada en *util.py* para crear una cola que usaremos como implementación de la lista de abiertos, a la que se introduce una función de prioridad.

También hacemos uso del método *getCostOfActions()*, que dará mayor prioridad a caminos que contengan comida para PacMan, y menor para caminos que contengan fantasmas.

3.1.2. Incluye el código añadido

Al principio de *search.py*:

```
global g_problem
```

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

    #El objeto problem es necesario para la función heurística
    global g_problem
    g_problem = problem

    cola_prioridad = util.PriorityQueueWithFunction(funcionCosteUniforme)
    return busquedaGeneral(problem, cola_prioridad)
```

```
def funcionCosteUniforme(item):
    """
    Devuelve la prioridad del nodo a la hora de expandirlo para la búsqueda de coste
    uniforme.
    El problema ha de ser registrado previamente en la variable global
    correspondiente.
    """
    global g_problem
    estado, camino = item
    return g_problem.getCostOfActions(camino)
```

3.1.3. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

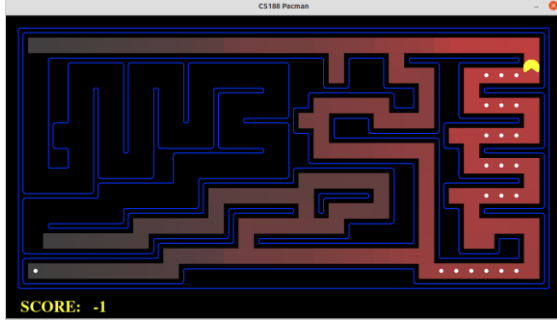
```
edu@edu:~/Escritorio/search$ python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:      442.0
Win Rate:    1/1 (1.00)
Record:      Win
```



SCORE: -28

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

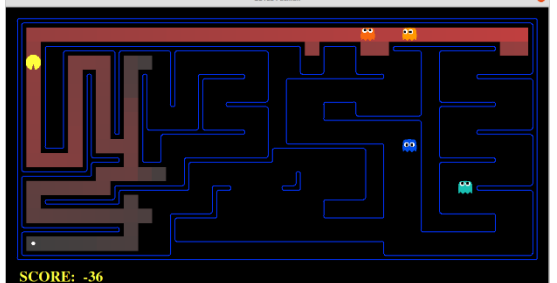
```
edu@edu:~/Escritorio/search$ python3 pacman.py -l mediumDottedMaze -p StayEastSearchAgent --frameTime 0
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:      646.0
Win Rate:    1/1 (1.00)
Record:      Win
```



SCORE: -1

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

```
edu@edu:~/Escritorio/search$ python3 pacman.py -l mediumScaryMaze -p StayWestSearchAgent
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:      418.0
Win Rate:    1/1 (1.00)
Record:      Win
```



SCORE: -36

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

```
edu@edu:~/Escritorio/search$ python3 autograder.py -q q3
Starting on 2-26 at 20:48:00
```

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_imp.test
***   solution:      []
***   expanded_states: ['A', 'B', 'C']
*** 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/pacman_imp.test
***   pacman layout:   trapped
***   solution length: 0
***   nodes expanded:   6
*** 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:   269
*** PASS: test_cases/q3/ucs_2_problemE.test
***   pacman layout:   mediumMaze
***   solution length: 74
***   nodes expanded:   260
*** 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:   14
*** 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: 7/7

Finished at 20:48:00

Provisional grades

=====

Question q3: 7/7

Total: 7/7

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

python autograder.py -q q3

Los resultados se concentran en la siguiente tabla:

	Nº nodos expandidos	Nº nodos camino
MediumMaze	269	68
MediumDottedMaze	186	1
MediumScaryMaze	108	68719479864

3.2. Conclusiones en el comportamiento de pacman

Cuando PacMan se encuentra en un laberinto sin fantasmas ni alimento, todos los movimientos tienen el mismo coste, ya que los nodos insertados en la cola tienen la misma prioridad. Una cola de prioridad donde la prioridad es siempre la misma es, en esencia, una cola. Así pues, el resultado de la ejecución del primer comando – en *mediumMaze* - debe coincidir con la análoga de la sección 2, usando el algoritmo de búsqueda en anchura. En todos los casos el algoritmo llega a la solución. Los costes del camino encontrado dependen del retorno de la función *getCostOfActions*. En el caso del laberinto con alimento, el camino solución tiene un coste de 1. Expande más nodos que en el caso del laberinto de fantasmas, ya que la diferencia absoluta entre el coste de un camino normal y un camino con comida es menor que entre un camino normal y un camino con fantasmas. Es así que la ejecución en *mediumScaryMaze*, explora la primera recta del camino prácticamente sin explorar otros nodos, ya que la existencia de un fantasma eleva el coste de ese camino significativamente.

Sección 4 (2 puntos)

4.1. Enfoque y decisiones de la solución

Haciendo referencia al punto 1.1., la función *busquedaGeneral()* implementa la totalidad de la lógica necesaria para la función del algoritmo de búsqueda A*.

En el caso de algoritmo de búsqueda en grafo con A*, *listaAbiertos* es una cola de prioridad. De esta manera, la función *aStarSearch()* hace una llamada a la de búsqueda general, con una cola de prioridad como lista de abiertos.

La función de prioridad para la cola se declara como sigue, donde *item* es el nodo al que se asigna la prioridad dentro de la cola:

```
def funcionPrioridadAEstrella (item)
```

En esencia, esta función consigue el valor *f* del nodo *item*, como suma del valor *g* (el coste de llegar hasta el nodo, que es el mismo valor que devolvía la función de prioridad de la sección anterior) y el valor *h* de la heurística deseada (ésta será por defecto la trivial que otorga a todos los nodos el valor de 0). Es este método el que implementa toda la lógica necesaria para parametrizar la prioridad que se debe dar a distintos nodos. Cabe destacar que, como en la sección anterior, hemos optado por declarar una variable global en la que almacenar el problema y otra para almacenar la heurística que se pasan como parámetros de entrada a la función *aStarSearch*, de forma que se puedan emplear en la función de prioridad. Hemos preferido esta implementación frente a usar una expresión lambda, ya que así queda un código más legible y fragmentado.

4.1.1. Funciones del framework usadas

En esta sección utilizamos los mismos métodos y funciones que se explican en el apartado 1.1.1., así como las explicadas en el 3.1.1.

4.1.2. Incluye el código añadido

Al principio de search.py:

```
global g_problem
global g_heuristic
```

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""

    global g_heuristic
    g_heuristic = heuristic

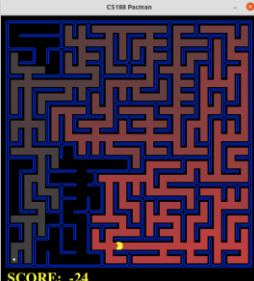
    global g_problem
    g_problem = problem
    cola_prioridad = util.PriorityQueueWithFunction(funcionPrioridadAEstrella)
    return busquedaGeneral(problem, cola_prioridad)
```

```
def funcionPrioridadAEstrella(item):
    """
    Devuelve la prioridad del nodo a la hora de expandirlo para la búsqueda de A
    estrella.
    El problema y la heurística han de ser registrados previamente en las variables
    globales correspondientes.
    """
    global g_heuristic
    global g_problem

    estado, camino = item
    h = g_heuristic(estado, g_problem)
    g = g_problem.getCostOfActions(camino)
    f = g + h
    return f
```

4.1.3. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

```
edu@edu:~/Escritorio/search$ python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic --frameTime 0
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 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
```




CS188 Pacman

SCORE: -24

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

```
edu@edu:~/Escritorio/search$ python3 pacman.py -l openMaze -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 54 in 0.0 seconds
Search nodes expanded: 535
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores: 456.0
Win Rate: 1/1 (1.00)
Record: Win
```



CS188 Pacman

SCORE: -14

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

```

edu@edu:~/Escritorio/search$ python3 autograder.py -q q4
Starting on 2-26 at 20:49:15

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: 221
*** 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 ###

Finished at 20:49:15

Provisional grades
=====
Question q4: 3/3
-----
Total: 3/3

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

```

python autograder.py -q q4

4.2. Conclusiones en el comportamiento de pacman

El comportamiento de PacMan es correcto, en el sentido que **sí** encuentra una solución al problema, y además esta solución es óptima. Se debe a que el algoritmo de búsqueda A* es completo y optimo si la heurística utilizada es admisible y monótona.

Debido a que la heurística utilizada es la distancia de Manhattan y el nodo final se encuentra en la esquina inferior izquierda, se puede observar como PacMan tiende a ir hacia esa dirección, por ejemplo, en la parte del final del recorrido se observa una gran zona sin visitar a la derecha de la solución.

	Nº nodos expandidos	Nº nodos camino	Ratio
BigMaze	549	210	2. 61428

4.2.1. Respuesta a pregunta 4

Podemos observar como en todas las estrategias de búsqueda encuentran una solución, aunque no todas ellas óptima. Este es el caso de la búsqueda en profundidad, que como ya hemos explicado en la sección 1, no asegura que la solución encontrada sea óptima, de hecho difiere bastante de ella.

Cabe destacar también que la búsqueda en anchura y la búsqueda de coste uniforme dan resultados equivalentes, debido a que la cola de prioridad que usa el coste uniforme en nuestro caso, ya que todas las acciones tienen el mismo coste, actúa como una cola normal, que es la implementación para la lista de abiertos utilizada en la búsqueda en anchura. La solución obtenida en estos casos sí es óptima.

El último caso a estudiar es la búsqueda mediante A*. Este algoritmo asegura encontrar una solución y que ésta sea óptima debido a que usamos una heurística admisible y monótona. Además, con la ayuda de dicha heurística, la cantidad de nodos expandidos es menor a los casos anteriores.

	Nº nodos expandidos	Nº nodos camino	Ratio
Profundidad	576	298	1.93289
Anchura	682	54	12. 6296
Coste uniforme	682	54	12. 6296
A*	535	54	9.90741

```
edu@edu:~/Escritorio/search$ python3 pacman.py -l openMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 298 in 0.0 seconds
Search nodes expanded: 576
Pacman emerges victorious! Score: 212
Average Score: 212.0
Scores: 212.0
Win Rate: 1/1 (1.00)
Record: Win

edu@edu:~/Escritorio/search$ python3 pacman.py -l openMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores: 456.0
Win Rate: 1/1 (1.00)
Record: Win

edu@edu:~/Escritorio/search$ python3 pacman.py -l openMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.1 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores: 456.0
Win Rate: 1/1 (1.00)
Record: Win
```

Sección 5 (2 puntos)

5.1. Enfoque y decisiones de la solución

Haciendo referencia al punto 1.1., la función *busquedaGeneral()* implementa la totalidad de la lógica necesaria. Debido a la generalidad con que fue diseñada esta función, también nos servirá para esta sección.

A la hora de inicializar el problema de búsqueda de las esquinas añadimos en ella un diccionario que guardará las esquinas y un booleano asociado a ellas. Hemos decidido usar un diccionario ya que es bastante sencillo trabajar con él y facilita y compacta el código, por lo que es más manejable y corregible en caso de error.

Además, ahora el estado no será simplemente la posición del laberinto en la que nos encontremos, sino también este diccionario que lleva cuenta de las esquinas ya visitadas, ya que no es el mismo estado encontrarse en la misma posición habiendo visitado una esquina o no. Con esta descripción del listado de estados del problema, el estado inicial será la posición inicial junto con el diccionario con la información de que no se ha visitado ninguna esquina todavía y para saber si el estado actual es objetivo basta con comprobar si la información sobre las esquinas visitadas así lo dice.

Los sucesores se consiguen mediante todas las acciones legales que puede realizar PacMan desde la posición actual. Estos sucesores heredan del padre el diccionario con la información de las esquinas y lo actualizan en caso de ser necesario, es decir, si la posición del sucesor es una esquina. Cabe destacar que no es en este momento cuando se evaluará si el sucesor es estado objetivo, sino cuando él vaya a ser expandido, por tanto esta operación tiene sentido realizarla en este momento.

5.1.1. Funciones del framework usadas

En esta sección hacemos uso de la función *hasFood()* definida en *pacman.py*, que devuelve si una posición del laberinto contiene “comida”. También utilizamos el método *directionToVector()* para obtener el desplazamiento en x e y asociado a cada una de las direcciones que puede generar un sucesor.

5.1.2. Incluye el código añadido

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor function
    """

    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, 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 # DO NOT CHANGE; Number of search nodes expanded
# Please add any code here which you would like to use
# in initializing the problem
self.cornerFood = dict()
for corner in self.corners:
    if startingGameState.hasFood(*corner):
        self.cornerFood[corner] = True

def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    return (self.startingPosition, self.cornerFood)

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    (pacman_state, cornerFood) = state
    return not any(cornerFood.values())

def getSuccessors(self, state):
    """
    Returns successor states, the actions they require, and a cost of 1.

    As noted in search.py:
    For a given state, this should return a list of triples, (successor,
    action, stepCost), where 'successor' is a successor to the current
    state, 'action' is the action required to get there, and 'stepCost'
    is the incremental cost of expanding to that successor
    """
    successors = []
    pacman_state, cornerFood = state

    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, \
        Directions.WEST]:
        # Comprueba que la accion es legal
        x,y = pacman_state
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        state_hijo = (nextx, nexty)

        # Añade el movimiento a la lista si no es un muro
        if(self.walls[nextx][nexty] == False):
            cornerFood_hijo = dict(cornerFood)
            # Si ha llegado a la esquina, elimina la comida
            if(state_hijo in self.corners):

```

```

cornerFood_hijo[state_hijo] = False
successors.append(((state_hijo,cornerFood_hijo),action,1))

self._expanded += 1 # DO NOT CHANGE
return successors

```

5.1.3. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

```

edu@edu:~/Escritorio/search$ python3 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: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:      512.0
Win Rate:    1/1 (1.00)
Record:      Win

```

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

```

edu@edu:~/Escritorio/search$ python3 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 0.1 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:      434.0
Win Rate:    1/1 (1.00)
Record:      Win

```

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

```

edu@edu:~/Escritorio/search$ python3 autograder.py -q q5
Note: due to dependencies, the following tests will be run: q2 q5
Starting on 2-26 at 20:51:47

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_imp.test
***     solution:          []
***     expanded_states:   ['A', 'B', 'C']
*** 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:    269
*** PASS: test_cases/q2/pacman_imp.test
***     pacman layout:     trapped
***     solution length:   0
***     nodes expanded:    6

### Question q2: 7/7 ###

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

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

Finished at 20:51:47

Provisional grades
=====
Question q2: 7/7
Question q5: 3/3
-----
Total: 10/10

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

```

python autograder.py -q q5

5.2. Conclusiones en el comportamiento de pacman

El problema que estudiamos en esta sección es diferente a los estudiados en las anteriores en su descripción, pero a la hora de computar es equivalente al resto, es decir, los estados tienen descripciones diferentes pero se trabaja sobre ellos como si fueran el mismo que en las secciones diferentes. Debido a esto, el comportamiento observado es el esperado, con la función de búsqueda en anchura (como se le pasa como argumento en los ejemplos propuestos) el algoritmo de búsqueda en grafo es completo y óptimo.

	Nº nodos expandidos	Nº nodos camino	Ratio
tinyCorners	252	28	9
mediumCorners	1966	106	18.5472

Sección 6 (3 puntos)

6.1. Enfoque y decisiones de la solución

En este apartado se pide diseñar una heurística para el problema de las esquinas. El enfoque que debe tener el diseño se fundamenta en ir a las esquinas que se encuentren más cerca de PacMan. De forma natural, la primera heurística que planteamos consistía en que PacMan se acercase a la esquina más cercana respecto a su posición – midiendo la distancia con la métrica de Manhattan -, pero no resultaba ser una lógica suficientemente buena como para evitar expandir menos de 1200 nodos. Esto es, en parte, evidente, ya que solo se hacía uso de la información de una única esquina. Nuestra heurística – descrita en la pregunta de esta sección - es la evolución natural de la recientemente explicada.

6.1.1. Explicación de las funciones del framework

No hemos hecho uso de ninguna función del framework, más allá de las clases incluidas en Python – diccionarios y listas -.

6.1.2. Incluye el código añadido

```
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

    state:      The current search state
                (a data structure you chose in your search problem)

    problem:    The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e. it should be
    admissible (as well as consistent).
    """

    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
    esquinas_no_visitadas = []
    posicion, cornerFood = state

    for corner in cornerFood.keys():
        if cornerFood[corner] == True:
            esquinas_no_visitadas.append(corner)

    if (len(esquinas_no_visitadas) == 0):
        return 0
```

```

# Inicializaciones
h = 0
esquina_actual, count = posicion, 0

# Busca la esquina mas cercana a PacMan y la esquina mas cercana
# a esa esquina
while len(esquinas_no_visitadas) > 0 and count < 2:
    count += 1
    #print(esquinas_no_visitadas)
    l_coin = dict()
    distancias = dict()
    posx, posy = esquina_actual

    # Calcula la distancia a la esquina mas cercano
    for (esqx, esqy) in esquinas_no_visitadas:
        distancias[(esqx, esqy)] = abs(posx - esqx) + abs(posy - esqy)
    # Coge la esquina mas cercana
    estado_min_dist = min(distancias, key=distancias.get)
    min_dist = distancias[estado_min_dist]

    esquina_actual = estado_min_dist
    h += min_dist
    esquinas_no_visitadas.remove(esquina_actual)

return h

```

6.1.3. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

```

edu@edu:~/Escritorio/search$ python3 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 919
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:      434.0
Win Rate:    1/1 (1.00)
Record:      Win

```

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

```

Note: due to dependencies, the following tests will be run: q4 q6
Starting on 3-1 at 12:36:06

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: 221
*** 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 ###

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
*** 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', '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', 'East', 'East', 'South', 'South', 'South', 'South', 'East', 'East',
'East', 'North', 'North', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'South', 'North', 'North', 'North', 'North',
'North', 'North', 'North', 'North', 'West', 'West', 'North', 'North', 'East', 'East', 'North', 'North']
path length: 106
*** PASS: Heuristic resulted in expansion of 919 nodes
path: ['East', 'South', 'South', 'West', 'West', 'South', 'West', 'South', 'South', 'East', 'East', 'East', 'West',
'West', 'West', 'West', 'West', 'North', 'North', 'North', 'North', 'North']
path length: 22
*** PASS: Heuristic resulted in expansion of 24 nodes

### Question q6: 6/6 ###

Finished at 12:36:06

Provisional grades
=====
Question q4: 3/3
Question q6: 6/6
-----
Total: 9/9

```

python autograder.py -q q6

6.2. Conclusiones en el comportamiento de pacman

El comportamiento de PacMan es completo, ya que llega al estado solución, y óptimo, ya que encuentra la solución en el menor número de movimientos posibles – 106 -. Expande 919 nodos, que es menor que 1200, por lo que pasa todas las pruebas del *autograder*.

6.2.1. Respuesta a pregunta 5: heurística

La lógica de nuestra heurística consiste en calcular el coste del camino, haciendo uso de la métrica de Manhattan, de PacMan a la esquina más cercana. Seguidamente, le sumamos la distancia de esa esquina a la siguiente esquina más cercana. En la implementación, primeramente, creamos una

lista de esquinas no visitadas. En un bucle realizamos dos iteraciones – marcadas por la variable *count* -.

En la primera iteración, mediante un diccionario, obtenemos la distancia de PacMan a todas las esquinas y nos quedamos con la menor. En la segunda iteración, el papel que antes jugaba PacMan dentro del bucle ahora lo juega la esquina encontrada en la primera iteración, de forma que, habiendo eliminado esta de la lista de esquinas no visitadas, calculamos la distancia al resto de esquinas aún no visitadas. Devolvemos la suma de ambas distancias, acumulada sobre la variable *h*. Como se trata de una relajación del problema, la heurística es admisible, ya que se calcula el camino más corto, sin tener en cuenta existencia de muros, tan solo a dos de las esquinas. La consistencia se puede derivar también de este hecho, ya que generalmente las heurísticas derivadas de relajación del problema suelen ser consistentes, y el *autograder* afirma este hecho.

Sección 7

Comentarios personales de la realización de esta práctica

La parte más interesante ha sido el diseño de la heurística para el problema de visitar todas las esquinas. A medida que incluíamos más información sobre la posición de las esquinas que faltaban por visitar, la cantidad de nodos expandidos se reducía y la solución se volvía más eficiente.

Esta práctica ha servido para poder relacionar el comportamiento abstracto de algoritmos de búsqueda con soluciones a problemas o juegos conocidos. Asimismo, ha sido útil para entender que una buena solución no solo se mide en términos de optimalidad, sino que es necesario tener en cuenta el coste computacional, medido en nodos expandidos, necesitado para llegar a la solución.