MEMORIA PRÁCTICA 1 - A

Juan Larrondo y Abril Palanco

Grupo: 1311

Parte A: Búsqueda no informada e informada

A.1 Búsqueda en profundidad

El objetivo del apartado es implementar un algoritmo de búsqueda en profundidad y familiarizarse con el código fuente proporcionado por el equipo docente.

El problema principal que encontramos al enfrentarnos al primer apartado fue entender el código y recordar el lenguaje de la práctica: Python.

A.1.1

Que el algoritmo de búsqueda en profundidad sea completo significa que siempre que la solución exista el algoritmo dará con ella. Aplicado a nuestro caso particular del laberinto cómo se exploran todos los nodos nivel por nivel por lo que aunque puede que no sea eficiente en memoria y tiempo de ejecución siempre llegará a la solución.

A.1.2

En el momento en el que en BP se permite inspeccionar nodos ya visitados corremos el riesgo de quedar atrapados en un bucle infinito en el caso de los grafos que contengan ciclos.

Supongamos que un grafo A->B->C->A contiene un ciclo. Si bien es cierto que A ya fue visitado lo volverá a marcar como nuevo y lo seguirá explorando infinitamente.

A.1.3

El orden de exploración es el contrario (por ser un stack) al orden de la lista de sucesores: [north, south, east, west], es decir: [west, east, south, north]. Este orden de exploración concuerda con el orden de exploración de nodos en el mapa.

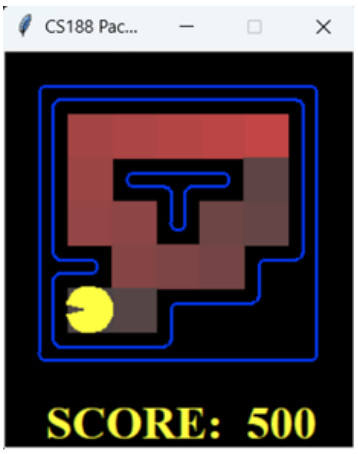
Pacman no va a todas las casillas exploradas, solo recorre el camino que lleva al resultado. Si se han explorado otros caminos que no llevan a ningún lado, pacman no los recorre en cuanto no coinciden con el camino solución.

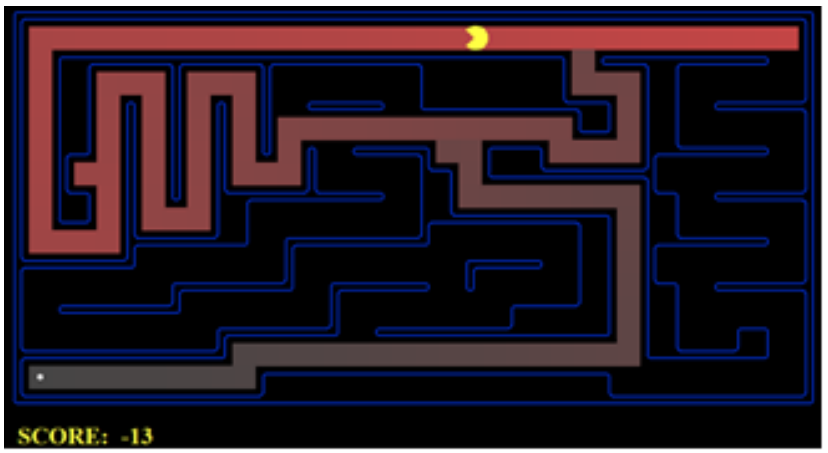
A.1.4

No, BP (o DFS) busca recorriendo el árbol hacia abajo, sin considerar otros caminos que puedan ser mejores, y el primer camino que llegue al destino (por profundo que sea el camino recorrido) se devuelve. No como BA (o BFS), que considera todos los

caminos posibles por cada nivel de profundidad en el árbol. Este sí garantiza que se encuentra la mejor solución

SALIDA:






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


Finished at 12:05:29

Provisional grades
==================
Question q1: 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.
```

A.2 Algoritmo de búsqueda genérico y búsqueda en anchura

El objetivo del apartado es implementar un algoritmo de búsqueda en anchura.

No nos resultó complicado el segundo apartado, pues una vez familiarizados con el código proporcionado y teniendo más soltura con Python (problema principal del primer apartado) con conocer el algoritmo de búsqueda en anchura no resulta complicado resolver este apartado.

A.2.1

BA explora el grafo nivel por nivel y expandiendo los nodos de nivel 1 primero, los de nivel 2 posteriormente y así sucesivamente. Esta estrategia garantiza que cuando se encuentre el primer camino óptimo éste se encuentre a la menor distancia posible,

A.2.2

Si, BA en su peor caso expande más nodos que BP. Para justificar esta idea pensemos en que el algoritmo BA busca un nodo en el nivel N, por consecuencia expandirá todos los nodos hasta el nivel N. En el caso de la búsqueda en profundidad al buscar un nodo de nivel N no habrá explorado TODOS los nodos de los niveles anteriores. La búsqueda en profundidad resulta mucho más eficiente si se

trata de árboles profundos pero no muy anchos. En términos de memoria es más eficiente BP.

A.2.3

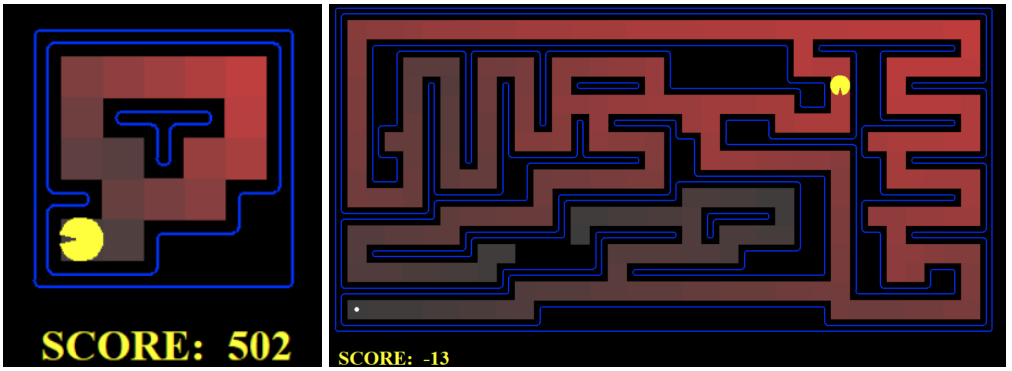
Pacman no entra en bucle porque hemos utilizado una técnica llamada “detección de ciclos”. Para implementar esta técnica registramos una lista de nodos visitados a la que vamos añadiendo el estado actual siempre que no esté ya explorado. Una vez añadido comprobamos los sucesores a los que podemos llegar desde este estado actual. De esta manera evitamos explorar los sucesores de un mismo estado dos veces (evitar ciclos).

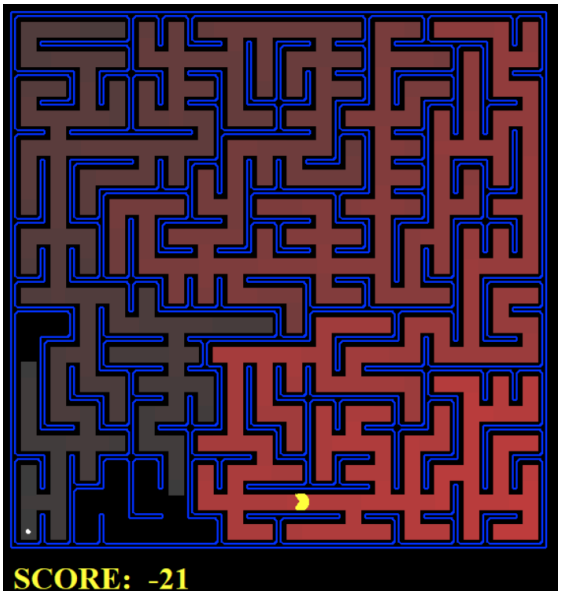
```
visited = []

while not structure.isEmpty():
    path = structure.pop()
    current_state = path[0] # INDEX THE CURRENT STATE
    #print(path)
    #print(current_state)
    if search_problem.isGoalState(current_state):
        return path[1] # RETURN THE PATH OF STATES

    if current_state not in visited:
        visited.append(current_state)
        successors = search_problem.getSuccessors(current_state)
```

SALIDA:





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


Finished at 21:26:02

Provisional grades
==================
Question q2: 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.
```

A.3 Variar la función de coste

El objetivo del apartado es que seamos capaces de variar la función de coste de los caminos en función de diferentes parámetros (enemigos, comida…) y que el algoritmo siga siendo capaz de encontrar una solución óptima a pesar de estas variaciones.

La dificultad principal que nos encontramos en este apartado fue la decisión que tuvimos que tomar sobre qué estructura de datos le pasábamos a la función de búsqueda genérica. Rápidamente descartamos Pila y Queue, pues son las estructuras que utilizan DFS y BFS. En segundo lugar pensamos en usar cola de prioridad sin prioridad, pero dado que no queremos ordenar según la tupla completa, sino solamente por el coste acumulado del camino necesitábamos una implementación que nos permitiera asignar una prioridad dinámicamente. A raíz de esta necesidad surge la idea de pasar una estructura de PriorityQueue With Function con su correspondiente expresión lambda que permite asignar esta prioridad dinámicamente.

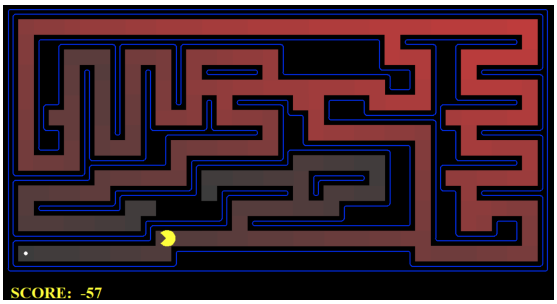
A.3.1

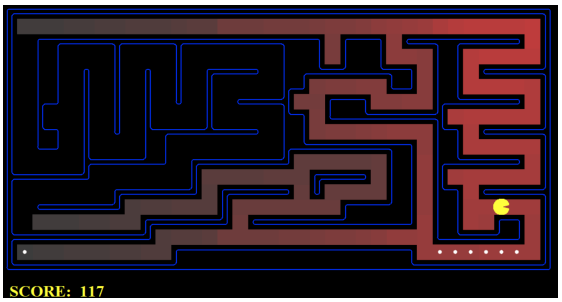
Si, esto es cierto dado que BCU ordena todos los nodos por coste acumulado, y cuando todos los movimientos tienen coste uniforme, el coste acumulado es directamente proporcional a la profundidad del nodo. Por tanto, BCU expande los nodos en el mismo orden que la búsqueda en anchura, primero los de profundidad 1, después los de profundidad 2, y así sucesivamente.

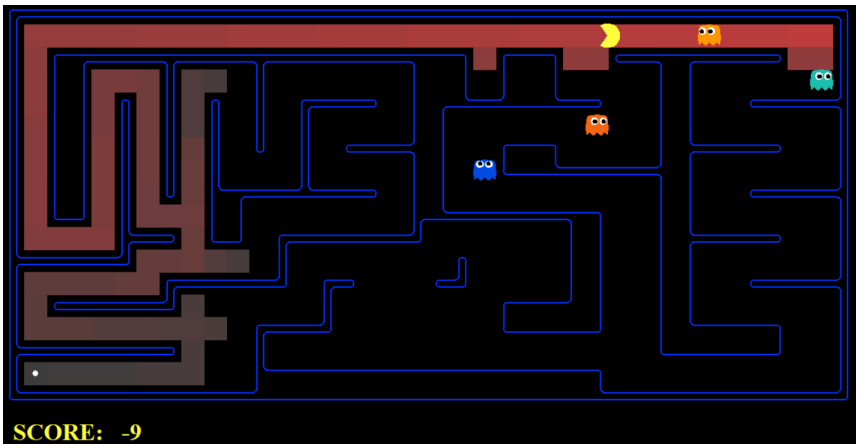
A.3.2

BP explora primero los nodos más profundos, mientras que BCU expande el nodo con menor coste acumulado. Para replicar el comportamiento de BP, los nodos más profundos deben tener menor coste que los menos profundos, de manera que BCU los seleccione primero.

SALIDA:








```
Finished at 9:34:27

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.

PS C:\Users\apala\IA\pacman>
```

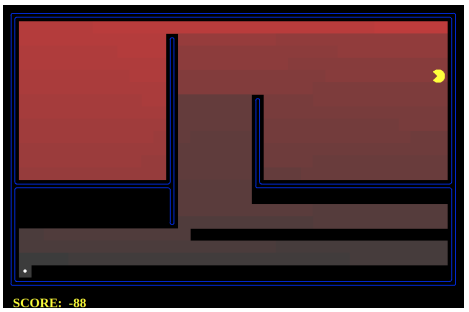
A.4 Búsqueda A*

El objetivo de apartado 4 es implementar una búsqueda en un grafo A* partiendo de la implementación y decisiones tomadas para UniformCostSearch.

A.4.1

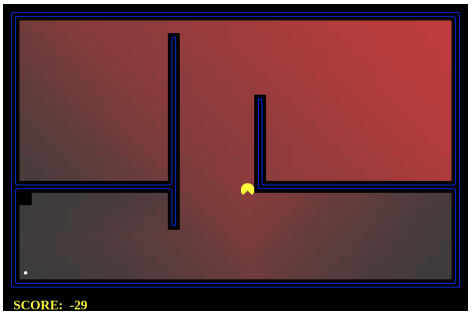
DFS: Expande pocos nodos pero encuentra una solución muy poco óptima.

```
juan@LAPTOP-UF2DBJK9:~/A3Q1/IA/pacman$ python pacman.py -l openMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 298 in 0.1 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
```



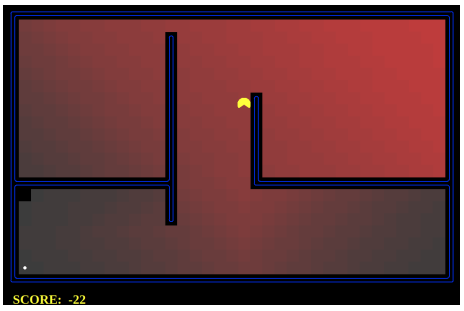
BFS: Expande muchos nodos, encontrando el camino más corto.

```
juan@LAPTOP-UF2DBJK9:~/A3Q1/IA/pacman$ python 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 0.1 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
```



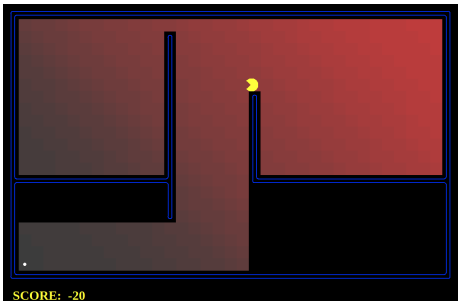
UCS: Se comporta de forma muy parecida a BFS, dando los mismos resultados.

```
juan@LAPTOP-UF2DBJK9:~/A3Q1/IA/pacman$ python 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
```



A*: Expande menos nodos que el DFS, tarda menos tiempo en encontrar la solución y encuentra el mejor camino.

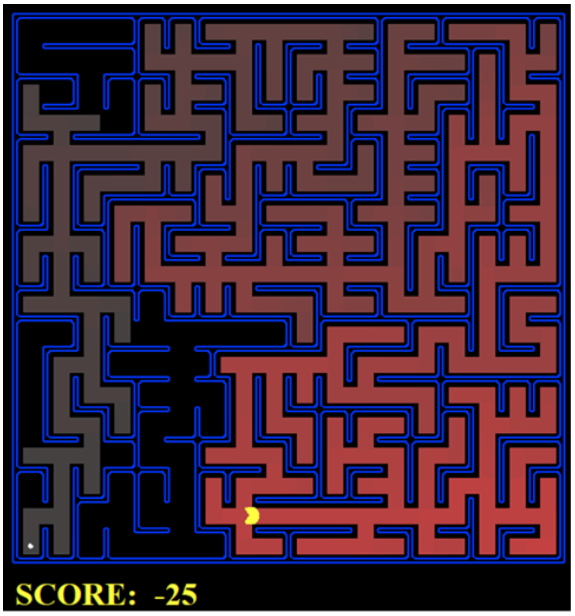
```
juan@LAPTOP-UF2DBJK9:~/A3Q1/IA/pacman$ python pacman.py -l openMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 535
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```



A.4.2

En el comportamiento utilizar una heurística u otra puede variar el tiempo de ejecución, su bien es cierto que usemos la que usamos garantizamos encontrar la solución.

SALIDA:




```
Finished at 14:15:08

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.
```

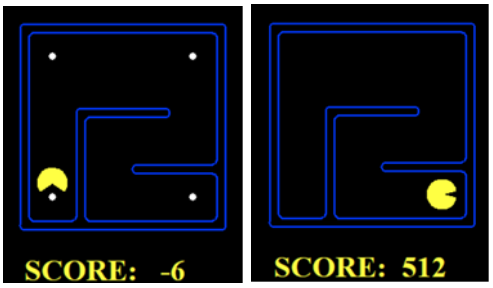
A.5 Definir un problema de búsqueda

En este apartado el objetivo es que seamos capaces de definir un problema de búsqueda. Este problema debe de cumplir que Pacman pase por las cuatro esquinas recorriendo el camino más corto.

El principal problema que hemos encontrado en la realización de este apartado fue el retroceso de Pacman una vez encontraba la primera esquina. Esto sucedía porque no se permitía pasar por un nodo ya visitado. Para solucionar este problema pensamos en varias posibilidades:

- Hacer BFS en cada esquina. Esto no solucionaba el problema de los nodos ya visitados anteriormente.
- Hacer una lista de 4 booleanos en la función de inicialización con todos inicializados a “False”, que se irían poniendo a “True” se fueran visitando. Esta fue la opción más similar a la finalista, pero no fue la elegida porque tampoco solucionaba que Pacman se quedara atascado en la esquina primera que encontrará.
- La opción elegida y que, por tanto, si resuelve el problema es la entregada en el código. En ella definimos el estado como un tupla de dos elementos: la posición inicial y el conjunto de esquinas YA visitadas. Según se vayan visitando se irán añadiendo al conjunto de esquinas visitadas. Esto soluciona el problema de la repetición de nodos porque no considerará dos estados iguales en los que varíe o bien la lista de esquinas visitadas o bien el estado inicial.

SALIDA:




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


Finished at 20:43:26

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.
```

A.6 Heurística para el problema de las cuatro esquinas

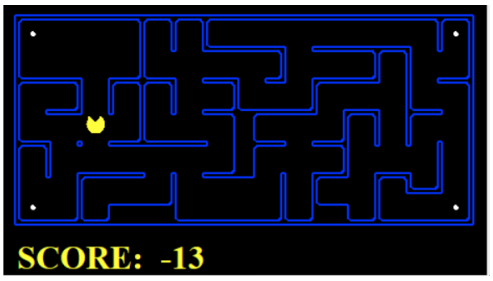
En este apartado el objetivo es que definamos una heurística que se adapte al problema de las cuatro esquinas de la forma más precisa posible.

A.6.1

En primer lugar pensamos en implementar una heurística euclidiana. La razón por la que la descartamos fue porque nos dimos cuenta que la heurística utilizada (heurística de Manhattan) es mucho más informativa que la de la distancia euclidiana, si bien es cierto que ambas son consistentes. Además, la heurística euclidiana calcula distancia en línea recta, hecho que rápidamente nos hizo descartarla, al no considerar los obstáculos del laberinto. En ningún momento consideramos la heurística nula por los requisitos del enunciado (cantidad de nodos expandidos e inconsistencia). Finalmente, tras haber elegido Manhattan y haberla implementado el problema fue la cantidad de nodos expandidos, pero finalmente dimos con la solución que cumplía los requisitos.

Esta solución, como se muestra en la salida de la imagen de la sección “SALIDA” se expanden 950 nodos únicamente. En primer lugar nos gustaría destacar que nuestra implementación busca entre todos los posibles órdenes de visita para las cuatro esquinas y se queda con el “mejor” de todos ellos. Para recorrer los caminos utiliza BFS. Esta idea surge a raíz de darnos cuenta de que pueden darse ocasiones en la que el camino óptimo comience visitando las esquina más lejana. Sin embargo, como esto tampoco es una regla que se cumple siempre comprobamos en cada caso el camino óptimo como se ha mencionado anteriormente.

SALIDA:




```
PS C:\Users\apala\IA\pacman> python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.2 seconds
Search nodes expanded: 950
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Users\apala\IA\pacman>
PS C:\Users\apala\IA\pacman>
```

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


Finished at 21:07:42

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

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

PS C:\Users\apala\IA\pacman>
```