

# Práctica 1

## Búsqueda

Junco de las Heras Valenzuela  
y  
Marta Vaquerizo Núñez

Pareja 09

# Índice

<b>Sección 1</b>	<b>3</b>
1.1. Comentario personal en el enfoque y decisiones de la solución propuesta . . .	3
1.2. Lista y explicación de las funciones del framework usadas . . . . .	3
1.3. Incluye el código añadido . . . . .	4
1.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados . . . . .	5
1.5. Conclusiones en el comportamiento de pacman . . . . .	6
1.6. Respuesta a pregunta 1.1 . . . . .	7
1.7. Respuesta a pregunta 1.2 . . . . .	7
1.8. Respuesta a pregunta 2 . . . . .	8
<b>Sección 2</b>	<b>8</b>
2.1. Comentario personal en el enfoque y decisiones de la solución propuesta . . .	8
2.2. Lista y explicación de las funciones del framework usadas . . . . .	8
2.3. Incluye el código añadido . . . . .	9
2.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados . . . . .	9
2.5. Conclusiones en el comportamiento de pacman . . . . .	10
2.6. Respuesta a pregunta 3 . . . . .	10
<b>Sección 3</b>	<b>11</b>
3.1. Comentario personal en el enfoque y decisiones de la solución propuesta . . .	11
3.2. Lista y explicación de las funciones del framework usadas . . . . .	11
3.3. Incluye el código añadido . . . . .	11
3.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados . . . . .	12
3.5. Conclusiones en el comportamiento de pacman . . . . .	13
<b>Sección 4</b>	<b>14</b>
4.1. Comentario personal en el enfoque y decisiones de la solución propuesta . . .	14
4.2. Lista y explicación de las funciones del framework usadas . . . . .	14
4.3. Incluye el código añadido . . . . .	14
4.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados . . . . .	15
4.5. Conclusiones en el comportamiento de pacman . . . . .	16
4.6. Respuesta a pregunta 4 . . . . .	16
<b>Sección 5</b>	<b>17</b>
5.1. Comentario personal en el enfoque y decisiones de la solución propuesta . . .	17
5.2. Lista y explicación de las funciones del framework usadas . . . . .	17
5.3. Incluye el código añadido . . . . .	17

5.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados . . . . .	19
5.5. Conclusiones en el comportamiento de pacman . . . . .	20
<b>Sección 6</b>	<b>20</b>
6.1. Comentario personal en el enfoque y decisiones de la solución propuesta . . .	20
6.2. Lista y explicación de las funciones del framework usadas . . . . .	21
6.3. Incluye el código añadido . . . . .	21
6.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados . . . . .	24
6.5. Conclusiones en el comportamiento de pacman . . . . .	24
6.6. Respuesta a pregunta 5 . . . . .	24
<b>Sección 7</b>	<b>25</b>

# Sección 1

## 1.1. Comentario personal en el enfoque y decisiones de la solución propuesta

Para las cuatro primeras secciones se ha creado la función `searchGoalState(problem, structure)` para no repetir tanto código. Esta función tiene dos argumentos: `problem`, que es el problema de búsqueda que se plantea; y `structure`, que es la estructura de datos donde se almacenarán los estados generados, y dependerá del algoritmo de búsqueda.

Se ha definido un estado como el conjunto: (posición, acciones), donde acciones es una lista de acciones hasta llegar a ese estado desde el inicial. Con esta definición de estado, el camino a devolver cuando se llega al estado objetivo está en el propio estado.

Para implementar DFS se ha utilizado como estructura de datos la pila.

## 1.2. Lista y explicación de las funciones del framework usadas

En esta sección se han usado las siguientes funciones del framework:

- `util.Stack()`: es la estructura de datos usada para almacenar los estados generados (abiertos).
- `problem.getStartState()`: estado inicial desde el cuál se empieza la búsqueda.
- `problem.isGoalState(state)`: test objetivo para saber si `state` es el estado que lleva a la solución del problema de búsqueda.
- `problem.getSuccessors(state)`: devuelve los sucesores de `state` que se meterán en la pila de abiertos para luego ser explorados.
- `util.Stack.push(state)`: añade el elemento `state` a la pila de estados generados.
- `util.Stack.pop()`: extrae el último elemento que se ha añadido a la pila de estados generados para ser explorado.
- `util.Stack.isEmpty()`: se usa como condición para seguir explorando nodos o no, ya que si la pila de estados generados está vacía no se puede explorar nada, y entonces no hay solución.

### 1.3. Incluye el código añadido

Se ha creado la función `searchGoalState(problem, structure)`.

```
def searchGoalState(problem, structure):
    """
    problem: El problema de búsqueda a resolver.
    structure: La estructura de datos, en la que se guardan los nodos a
    explorar, ejemplo: pila, cola...

    Devuelve una lista con las acciones que llevan al estado objetivo.

    Nota:
    Los elementos de abiertos tienen la forma (a, b, c) donde:
    a son las coordenadas (x, y) de la casilla.
    b es una lista que guarda el camino desde la casilla inicial
    hasta la casilla actual.
    c es el coste acumulado desde la casilla inicial hasta la casilla
    actual.

    Se ha creado esta funcion para ser lo mas generica posible y asi no
    repetir codigo en las funciones a desarrollar despues.
    """
    abiertos = structure
    # Se obtiene la estado de busqueda inicial.
    inicial = problem.getStartState()
    # Si la estructura es una Cola de Prioridad y no tiene funcion (para
    # A*) entonces el abierto tiene 3 elementos.
    if isinstance(structure, util.PriorityQueue) and not
    isinstance(structure, util.PriorityQueueWithFunction):
        abiertos.push((inicial, [], 1)
    # Sino, solo tiene 2.
    else:
        abiertos.push((inicial, []))
    cerrados = []
    while abiertos.isEmpty() is False:
        state = abiertos.pop()
        # Si el estado de busqueda es meta se devuelve la lista con los
        #movimientos a hacer.
        if problem.isGoalState(state[0]) is True:
            return state[1]
        if state[0] not in cerrados:
            cerrados.append(state[0])
            sucesores = problem.getSuccessors(state[0])
            for suc in sucesores:
                # Cada sucesor se ania de a la lista de abiertos segun el
```

```

        #tipo de estructura que se ha proporcionado.

        if isinstance(structure, util.PriorityQueue) and not
        isinstance(structure, util.PriorityQueueWithFunction):
            abiertos.push((suc[0], state[1] + [suc[1]]),
                problem.getCostOfActions(state[1]) + suc[2])
        else:
            abiertos.push((suc[0], state[1] + [suc[1]]))
    # En caso de que no haya solucion el Pacman se para.
    from game import Directions
    return [Directions.STOP]

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.
    """
    return searchGoalState(problem, util.Stack())

```

## 1.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

```

(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
ini (5, 5)
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

(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
ini (34, 16)
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

(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l bigMaze -p SearchAgent -a fn=dfs
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
ini (35, 1)
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300

```

Figura 1: Captura de la ejecución en los tres laberintos.

En la ejecución en los tres tipos de laberintos, se puede observar que la diferencia entre el coste del camino solución y el número de nodos expandidos es bastante pequeña, es decir, prácticamente se visitan todos los nodos expandidos para llegar a la solución.

```

Question q1
=====
ini A
*** PASS: test_cases\q1\graph_backtrack.test
***   solution:      ['1:A->C', '0:C->G']
***   expanded_states: ['A', 'D', 'C']
ini A
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***   solution:      ['2:A->D', '0:D->G']
***   expanded_states: ['A', 'D']
ini A
*** PASS: test_cases\q1\graph_infinite.test
***   solution:      ['0:A->B', '1:B->C', '1:C->G']
***   expanded_states: ['A', 'B', 'C']
ini A
*** 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']
ini (34, 16)
*** PASS: test_cases\q1\pacman_1.test
***   pacman layout:   mediumMaze
***   solution length: 130
***   nodes expanded:  146
### Question q1: 3/3 ###

```

Figura 2: Captura del test en la primera sección.

En la Figura 2 se puede observar que el algoritmo pasa todos los casos de prueba del autograder.

## 1.5. Conclusiones en el comportamiento de pacman

En este apartado contestamos a las preguntas: ¿es óptimo?, ¿es completo?, ¿cuántos nodos expande?

La búsqueda en profundidad es completa, es decir, siempre se encuentra una solución, pero no es la óptima. Se puede observar en las figuras del apartado 1,7 que el camino solución que recorre el Pacman se ve que no es el de menor coste, sobre todo en el caso de *mediumMaze*. Cabe destacar que, a pesar de que en general no es óptima, en el caso de *bigMaze* si lo es.

Como se ha observado antes, el número de nodos que se expanden prácticamente conforman el camino que lleva a la solución. El peor de los casos ocurre cuando el nodo meta es el último nodo que se explora, habiendo expandido antes todos los nodos restantes.

## 1.6. Respuesta a pregunta 1.1

**Pregunta 1.1:** ¿El orden de exploración era el que esperabais?

Sí, el algoritmo implementado explora los nodos en el orden que se esperaba, que es el mismo que se ha visto en clase.

## 1.7. Respuesta a pregunta 1.2

**Pregunta 1.2:** ¿Pacman realmente va a todas las casillas exploradas en su camino hacia la meta?

El pacman no va a pasar por todos y cada uno de los nodos que se expanden (casillas de color distinto de negro), solo el camino que lleva en el espacio de búsqueda, desde el nodo inicial al nodo meta. Este efecto se ve claramente en el *mediumMaze*:

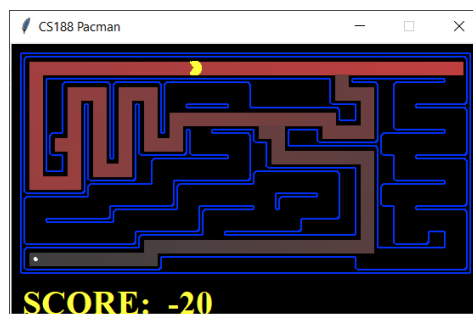


Figura 3: Visualización de nodos expandidos *mediumMaze*.

En el caso de *tinyMaze*, al haber pocas casillas, se expanden todas, por lo que no se ve nada este efecto. Y por último, en el caso de *bigMaze* también se ve el camino que lleva a la solución:

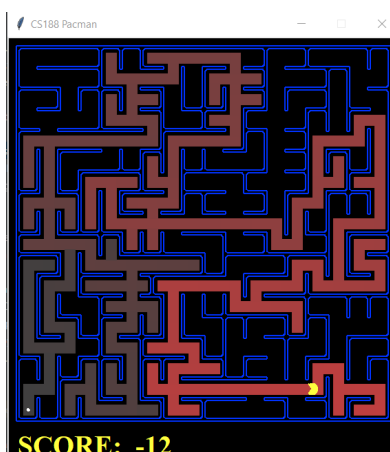


Figura 4: Visualización de nodos expandidos *bigMaze*.



## 1.8. Respuesta a pregunta 2

**Pregunta 2:** ¿Es esta una solución de menor coste? Si no es así, pensad qué está haciendo mal la búsqueda en profundidad.

En general, la solución que encuentra este algoritmo no es de menor coste, y esto se refleja en los casos de *tinyMaze* y *mediumMaze*. Pero, como ya se ha comentado antes, en el caso de *bigMaze* si se encuentra la solución de menor coste.

Lo que el algoritmo de búsqueda en profundidad hace “mal” es que si en el grafo de búsqueda empieza por un estado que no es óptimo, y de ese estado hay un camino a la solución, esta es la solución que va a reportar, sin mirar que al principio haya cogido todos los estados de forma óptima.

## Sección 2

### 2.1. Comentario personal en el enfoque y decisiones de la solución propuesta

Dado que ya teníamos la función `searchGoalState`, ahora solo falta llamarla con una estructura de cola para implementar el BFS.

### 2.2. Lista y explicación de las funciones del framework usadas

A continuación, se muestra una lista con las funciones usadas para implementar esta sección.

- `util.Queue()`: es la estructura de datos usada para almacenar los estados generados.
- `problem.getStartState()`: estado inicial desde el cuál se empieza la búsqueda.
- `problem.isGoalState(state)`: test objetivo para saber si `state` es el estado que lleva a la solución del problema de búsqueda.
- `problem.getSuccessors(state)`: devuelve los sucesores de `state` que se meterán en la cola de abiertos para luego ser explorados.
- `util.Queue.push(state)`: añade el elemento `state` a la cola de estados generados.
- `util.Queue.pop()`: extrae el primer elemento de la cola de estados generados para ser explorado.
- `util.Queue.isEmpty()`: se usa como condición para seguir explorando nodos o no, ya que si la cola de estados generados está vacía no se puede explorar nada, y entonces no hay solución.

## 2.3. Incluye el código añadido

La función `searchGoalState(problem, structure)` también está presente en la ejecución de la función `breadthFirstSearch(problem)`.

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

## 2.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

```
(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs  
[SearchAgent] using function bfs  
[SearchAgent] using problem type PositionSearchProblem  
ini (5, 5)  
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  
  
(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs  
[SearchAgent] using function bfs  
[SearchAgent] using problem type PositionSearchProblem  
ini (34, 16)  
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  
  
(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l bigMaze -p SearchAgent -a fn=bfs  
[SearchAgent] using function bfs  
[SearchAgent] using problem type PositionSearchProblem  
ini (35, 1)  
Path found with total cost of 210 in 0.0 seconds  
Search nodes expanded: 620  
Pacman emerges victorious! Score: 300
```

Figura 5: Captura de la ejecución en los tres laberintos.

En la Figura 5, se puede observar que el número de nodos expandidos es mucho mayor que en el caso de DFS. También se puede observar que el coste en *tinyMaze* y en *mediumMaze* es menor que en DFS, mientras que en el caso de *bigMaze* es el mismo coste.

```

Question q2
=====
ini A
*** PASS: test_cases/q2/graph_backtrack.test
***   solution:      ['1:A->C', '0:C->G']
***   expanded_states: ['A', 'B', 'C', 'D']
ini A
*** PASS: test_cases/q2/graph_bfs_vs_dfs.test
***   solution:      ['1:A->G']
***   expanded_states: ['A', 'B']
ini A
*** PASS: test_cases/q2/graph_infinite.test
***   solution:      ['0:A->B', '1:B->C', '1:C->G']
***   expanded_states: ['A', 'B', 'C']
ini A
*** 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']
ini (34, 16)
*** PASS: test_cases/q2/pacman_1.test
***   pacman layout:      mediumMaze
***   solution length: 68
***   nodes expanded:      269

### Question q2: 3/3 ###

```

Figura 6: Captura del test en la primera sección.

En esta última figura se muestra que el algoritmo pasa todos los casos de prueba del autograder.

## 2.5. Conclusiones en el comportamiento de pacman

En este apartado contestamos a las preguntas: ¿es óptimo?, ¿es completo?, ¿cuántos nodos expande?

La búsqueda en anchura encuentra es completa y óptima, es decir, encuentra la solución de menor coste. Este algoritmo va expandiendo nodos según la distancia del nodo al estado inicial, en orden creciente, así que cada vez que se explora un nodo nuevo este nodo tiene la distancia mínima posible, por lo que cuando se explora el nodo meta la solución encontrada va a ser óptima. Esto es así ya que, como el coste de las acciones es igual a 1, esta búsqueda se comporta como la de coste uniforme. En este caso, el número de nodos que se expanden es mucho mayor en comparación con el de búsqueda en profundidad, y por lo tanto, no se marca el camino de la solución.

## 2.6. Respuesta a pregunta 3

**Pregunta 3:** ¿BFS encuentra una solución de menor coste?

Sí, como se ha comentado anteriormente, el algoritmo es óptimo, por lo que encuentra la solución de menor coste.

## Sección 3

### 3.1. Comentario personal en el enfoque y decisiones de la solución propuesta

Dado que ya teníamos la función `searchGoalState`, ahora solo falta llamarla con una estructura de cola de prioridad para implementar la búsqueda con coste uniforme. Cabe destacar que en la función `searchGoalState` hay que hacer la distinción para este caso, ya que en el push había que indicar la prioridad del nodo a añadir, que es el coste de las acciones que llevan al nodo.

### 3.2. Lista y explicación de las funciones del framework usadas

A continuación, se muestra una lista con las funciones usadas para implementar esta sección.

- `util.PriorityQueue()`: es la estructura de datos usada para almacenar los estados generados.
- `problem.getStartState()`: estado inicial desde el cuál se empieza la búsqueda.
- `problem.isGoalState(state)`: test objetivo para saber si `state` es el estado que lleva a la solución del problema de búsqueda.
- `problem.getSuccessors(state)`: devuelve los sucesores de `state` que se meterán en la cola de prioridad de abiertos para luego ser explorados.
- `util.PriorityQueue.push(state, priority)`: añade el elemento `state` a la cola de prioridad de estados generados según su prioridad `priority`.
- `util.PriorityQueue.pop()`: extrae el primer elemento de la cola de prioridad de estados generados para ser explorado.
- `util.PriorityQueue.isEmpty()`: se usa como condición para seguir explorando nodos o no, ya que si la cola de prioridad de estados generados está vacía no se puede explorar nada, y entonces no hay solución.
- `problem.getCostOfActions(acciones)`: devuelve la suma de los costes de las acciones pasadas por argumento. Como estamos en coste uniforme, este coste se trata como prioridad.

### 3.3. Incluye el código añadido

```
def uniformCostSearch(problem):  
    """Search the node of least total cost first."""  
  
    return searchGoalState(problem, util.PriorityQueue())
```

### 3.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

```
(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l tinyMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
ini (5, 5)
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

(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
ini (34, 16)
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

(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l bigMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
ini (35, 1)
Path found with total cost of 210 in 0.2 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
```

Figura 7: Captura de la ejecución en los tres laberintos.

La ejecución en esta sección es igual que la de la sección anterior.

```
Question q3
=====
ini A
*** PASS: test_cases\q3\graph_backtrack.test
***   solution:      ['1:A->C', '0:C->G']
***   expanded_states: ['A', 'B', 'C', 'D']
ini A
*** PASS: test_cases\q3\graph_bfs_vs_dfs.test
***   solution:      ['1:A->G']
***   expanded_states: ['A', 'B']
ini A
*** PASS: test_cases\q3\graph_infinite.test
***   solution:      ['0:A->B', '1:B->C', '1:C->G']
***   expanded_states: ['A', 'B', 'C']
ini A
*** 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']
ini A
*** PASS: test_cases\q3\ucs_0_graph.test
***   solution:      ['Right', 'Down', 'Down']
***   expanded_states: ['A', 'B', 'D', 'C', 'G']
ini (34, 16)
*** PASS: test_cases\q3\ucs_1_problemC.test
***   pacman layout:      mediumMaze
***   solution length: 68
***   nodes expanded:      269
ini (34, 16)
*** PASS: test_cases\q3\ucs_2_problemE.test
***   pacman layout:      mediumMaze
***   solution length: 74
```

Figura 8: Parte 1 de la captura del test de la sección 3.

```

ini (34, 16)
*** PASS: test_cases\q3\ucs_3_problemW.test
***   pacman layout:      mediumMaze
***   solution length: 152
***   nodes expanded:      173
ini ((2, 3), <game.Grid object at 0x00000216C18E8F70>)
*** PASS: test_cases\q3\ucs_4_testSearch.test
***   pacman layout:      testSearch
***   solution length: 7
***   nodes expanded:      14
ini A
*** 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 ###

```

Figura 9: Parte 2 de la captura del test de la sección 3.

En las figuras 8 y 9, se puede observar que el algoritmo pasa todos los casos del autograder. Debido a que había muchos casos de prueba, se han hecho dos capturas.

### 3.5. Conclusiones en el comportamiento de pacman

En este apartado contestamos a las preguntas: ¿es óptimo?, ¿es completo?, ¿cuántos nodos expande?

La búsqueda de coste uniforme es completa y óptima, es decir, encuentra la solución de menor coste. Este algoritmo va expandiendo nodos según la distancia del nodo al estado inicial, en orden creciente, así que cada vez que se explora un nodo nuevo este nodo tiene la distancia mínima posible, por lo que cuando se explora el nodo meta la solución encontrada va a ser óptima. Al igual que ocurre en la búsqueda en anchura, el número de nodos que se expanden es mucho mayor, y para nada marcan un camino como ocurre en la búsqueda en profundidad.

## Sección 4

### 4.1. Comentario personal en el enfoque y decisiones de la solución propuesta

Dado que ya teníamos la función `searchGoalState`, ahora solo falta llamarla con una estructura de cola de prioridad, dada por una función, para implementar el A\*. La función que se pasa a la cola de prioridad es la función  $f(n) = g(n) + h(n)$ , donde  $g(n)$  es el coste para llegar hasta el nodo, y  $h(n)$  es la función heurística.

### 4.2. Lista y explicación de las funciones del framework usadas

A continuación, se muestra una lista con las funciones usadas para implementar esta sección.

- `util.PriorityQueueWithFunction(priorityFunction)`: es la estructura de datos usada para almacenar los estados generados, donde `priorityFunction` sirve para calcular la prioridad de los estados que se añaden a la cola.
- `problem.getStartState()`: estado inicial desde el cuál se empieza la búsqueda.
- `problem.isGoalState(state)`: test objetivo para saber si `state` es el estado que lleva a la solución del problema de búsqueda.
- `problem.getSuccessors(state)`: devuelve los sucesores de `state` que se meterán en la cola de prioridad de abiertos para luego ser explorados.
- `util.PriorityQueueWithFunction.push(state)`: añade el elemento `state` a la cola de prioridad de estados generados.
- `util.PriorityQueueWithFunction.pop()`: extrae el primer elemento de la cola de prioridad de estados generados para ser explorado.
- `util.PriorityQueueWithFunction.isEmpty()`: se usa como condición para seguir explorando nodos o no, ya que si la cola de prioridad de estados generados está vacía no se puede explorar nada, y entonces no hay solución.

### 4.3. Incluye el código añadido

```
def aStarSearch(problem, heuristic=nullHeuristic):  
    """Search the node that has the lowest combined cost and heuristic first."""  
  
    "Se crea la función evaluación para que la cola de prioridad la use"  
    def priority_function(x): return heuristic(x[0], problem) +  
        problem.getCostOfActions(x[1])  
  
    return searchGoalState(problem, util.PriorityQueueWithFunction(priorityFunction))
```

## 4.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

```
(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l tinyMaze -p SearchAgent -a fn=astar
[SearchAgent] using function astar and heuristic nullHeuristic
[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

(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l mediumMaze -p SearchAgent -a fn=astar
[SearchAgent] using function astar and heuristic nullHeuristic
[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

(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l bigMaze -p SearchAgent -a fn=astar
[SearchAgent] using function astar and heuristic nullHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.2 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:      300.0
Win Rate:    1/1 (1.00)
```

Figura 10: Captura de la ejecución de los tres laberintos.

La ejecución es la misma que la de las dos últimas secciones (BFS y UCS).

```
Question q4
=====
ini A
*** PASS: test_cases/q4\astar_0.test
***   solution:      ['Right', 'Down', 'Down']
***   expanded_states: ['A', 'B', 'D', 'C', 'G']
ini S
*** PASS: test_cases/q4\astar_1_graph_heuristic.test
***   solution:      ['0', '0', '2']
***   expanded_states: ['S', 'A', 'D', 'C']
ini (34, 16)
*** PASS: test_cases/q4\astar_2_manhattan.test
***   pacman layout:      mediumMaze
***   solution length: 68
***   nodes expanded:      221
ini A
*** PASS: test_cases/q4\astar_3_goalAtDequeue.test
***   solution:      ['1:A->B', '0:B->C', '0:C->G']
***   expanded_states: ['A', 'B', 'C']
ini A
*** PASS: test_cases/q4\graph_backtrack.test
***   solution:      ['1:A->C', '0:C->G']
***   expanded_states: ['A', 'B', 'C', 'D']
ini A
*** 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 ###
```

Figura 11: Captura del test de la sección 4.



## 4.5. Conclusiones en el comportamiento de pacman

En este apartado contestamos a las preguntas: ¿es óptimo?, ¿es completo?, ¿cuántos nodos expande?

El algoritmo A\* llega a la solución, aunque no tiene por qué ser óptima, ya que depende de la heurística que se le da como argumento. Si se le da una heurística admisible y consistente entonces está garantizado a encontrar la solución óptima, sino podría no ser óptima. El algoritmo A\* es óptimo en el sentido de que dada una heurística, no hay otro algoritmo que expanda menos nodos que este. Expande los nodos de igual manera que el BFS, salvo que en vez de tener la distancia del BFS tiene esa distancia más la heurística, pero si es admisible y consistente entonces cuando se expanda la meta esta va a ser la de menor coste posible.

## 4.6. Respuesta a pregunta 4

**Pregunta 4:** ¿Qué sucede en openMaze para las diversas estrategias de búsqueda?

Lo que ocurre es que tanto BFS, UCS como A\* expanden 682 nodos, mientras que DFS expande 576 nodos. Todos encuentran el camino óptimo, excepto DFS, que no lo encuentra, como esperabamos. A continuación podemos ver los resultados de las ejecuciones:

```
(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l openMaze -p SearchAgent -a fn=dfs
[SearchAgent] using function dfs
[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

(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>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
```

Figura 12: Resultados en openMaze.

```
(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>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

(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l openMaze -p SearchAgent -a fn=astar
[SearchAgent] using function astar and heuristic nullHeuristic
[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
```

Figura 13: Resultados en openMaze.

## Sección 5

### 5.1. Comentario personal en el enfoque y decisiones de la solución propuesta

En primer lugar, se ha definido el nodo para un estado de búsqueda como una tupla. La primera componente es la coordenada (x, y) en el tablero, y la segunda componente es una lista con variables booleanas, en la que la i-ésima posición es True si el Pacman ha visitado la i-ésima esquina, False en otro caso. Por ejemplo, el estado inicial, si el Pacman no está en ninguna esquina, será ((pos inicial x, pos inicial y), [False, False, False, False]).

### 5.2. Lista y explicación de las funciones del framework usadas

A continuación, se muestra una lista con las funciones usadas para implementar esta sección.

- `Actions.directionToVector(action)`: Dada una acción devuelve una pareja con valores cada miembro de +1, 0 o -1, dependiendo de cuál es la siguiente casilla.
- Hemos usado atributos de la clase en la que se ha implementado el código como `self.startingPosition`, `self.walls` y `self._expanded`.

### 5.3. Incluye el código añadido

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"
    # El estado t es una pareja con 2 elementos:
    # t[0] la posicion (x, y) de la casilla en el tablero.
    # t[1] Un vector con 4 elementos, t[1][i] es True si se ha visitado la
    # esquina self.corners[i], sino False.

    # Se comprueba si el Pacman esta inicializado en una esquina.
    corner = [False, False, False, False]
    for i in range(4):
        if self.startingPosition == self.corners[i]:
            corner[i] = True

    return self.startingPosition, corner
```

```

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    """*** YOUR CODE HERE ***"""
    # El estado actual es meta si ha visitado a todas las esquinas.
    for i in range(4):
        if not state[1][i]:
            return False

    return True

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 = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
        # Add a successor state to the successor list if the action is legal
        # Here's a code snippet for figuring out whether a new position hits
        # a wall:
        #   x,y = currentPosition
        #   dx, dy = Actions.directionToVector(action)
        #   nextx, nexty = int(x + dx), int(y + dy)
        #   hitsWall = self.walls[nextx][nexty]

        """*** YOUR CODE HERE ***"""
        x,y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            nextState = (nextx, nexty)
            # Cada movimiento tiene coste 1.
            cost = 1
            # Si nextState es una esquina se marca como visitada (para ese
            # estado de busqueda).
            next_corner = list(state[1])

```

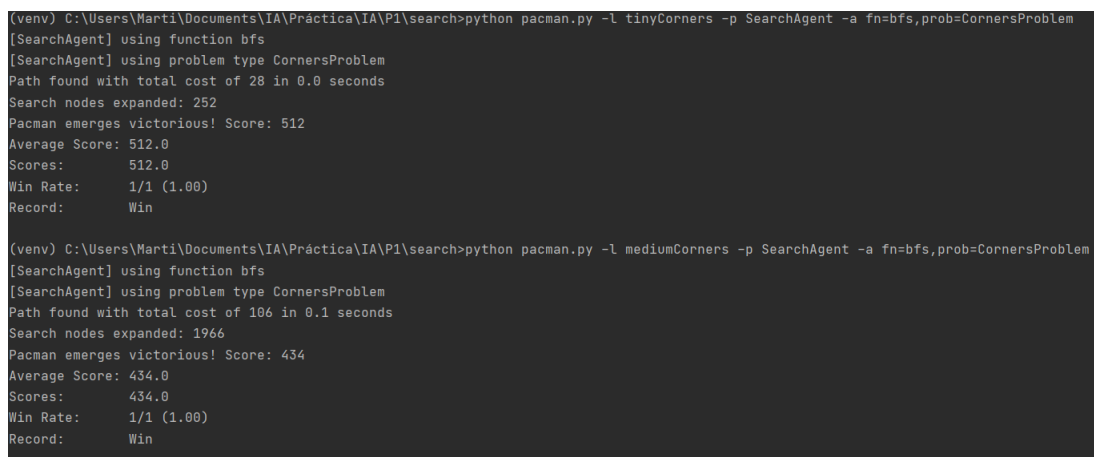
```

for i in range(4):
    if nextState == self.corners[i]:
        next_corner[i] = True
    successors.append( ( (nextState, next_corner), action, cost) )

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

```

## 5.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados



```

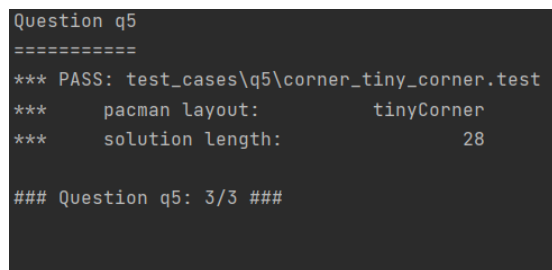
(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:      512.0
Win Rate:    1/1 (1.00)
Record:      Win

(venv) C:\Users\Marti\Documents\IA\Práctica\IA\P1\search>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 0.1 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

```

Figura 14: Captura de la ejecución en los dos laberintos.

Se puede observar que se expanden muchos más nodos que por los que realmente pasa el Pacman para llegar a la solución.



```

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

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

```

Figura 15: Captura de los resultados del test.

En esta figura se puede observar que el código implementado pasa los casos de prueba del autograder.

## 5.5. Conclusiones en el comportamiento de pacman

En este apartado contestamos a las preguntas: ¿es óptimo?, ¿es completo?, ¿cuántos nodos expande?

El algoritmo, dado que usamos en la llamada a la función la búsqueda en anchura, va a encontrar una solución óptima, como ya hemos visto en apartados anteriores. Asimismo, expande los mismos nodos que ésta. Usando otra representación del nodo en el espacio de búsqueda u otra búsqueda (a DFS en vez de a BFS) puede dar otros resultados.

## Sección 6

### 6.1. Comentario personal en el enfoque y decisiones de la solución propuesta

La función heurística que se ha implementado se basa en la relajación de las restricciones del problema. En este caso, se han quitado las paredes del laberinto. Se podrían hacer heurísticas mejores considerando las paredes más cercanas a la posición actual del Pacman y hacer una búsqueda con límite reducido y luego la heurística, pero hemos considerado que no íbamos a usar las paredes.

Primero, se guarda en una lista las esquinas que no se han visitado. La función se divide en casos, en función de cuántas esquinas faltan por visitar.

**Nota:** cuando se habla de distancia se hace referencia a la distancia Manhattan.

- Caso 0: Devuelve 0 la heurística (se ha conseguido el objetivo de visitar las cuatro esquinas).
- Caso 1: Devuelve la distancia entre la posición del Pacman a la esquina que falta.
- Caso 2: Devuelve el mínimo entre la distancia entre la posición del Pacman a una esquina y de esa esquina a la otra, y entre la posición del Pacman a la otra esquina y luego a la primera esquina.
- Caso 3: Se generaliza el caso 2 con fuerza bruta, se calcula la distancia de todos los caminos que primero van a la esquina  $i$ , luego a la  $j$  y luego a la  $k$  y se toma el mínimo (para que la distancia sea admisible). El camino es  $state[0] - corner[i] - corner[j] - corner[k]$ .
- Caso 4: Igual que en el caso 3, calcula la distancia de todos los caminos que primero van a la esquina  $i$ , luego a la  $j$  y luego a la  $k$  y se toma el mínimo (para que la distancia sea admisible). El camino es  $state[0] - corner[i] - corner[j] - corner[k] - corner[z]$ .

Es decir, lo que hace la heurística es calcular todos los posibles caminos del Pacman a las esquinas (a todas las permutaciones de esquinas) y coger la que resulta en la mínima distancia. Se puede comprobar que no hay una heurística consistente mejor que esta cuando se relaja el problema y no se usan las paredes, ya que cualquier otra heurística consistente  $h_2(\text{estado})$  es necesariamente  $\leq h(\text{estado})$ , porque si  $h_2(\text{estado}) > h(\text{estado})$  para algún estado, entonces existe un laberinto (por ejemplo un laberinto sin paredes) tal que la heurística  $h_2$  no es consistente.

## 6.2. Lista y explicación de las funciones del framework usadas

- `util.manhattan_dist(pos1, pos2)`: calcula la distancia Manhattan entre dos puntos del laberinto. La usamos para calcular nuestra heurística.
- `min(num1, num2)`: calcula el mínimo entre dos cantidades. Nos sirve a la hora de calcular la distancia mínima.

## 6.3. 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)
    #return 0 # Default to trivial solution
    "*** YOUR CODE HERE ***"

    # Si el estado es meta entonces el valor heurístico es 0.
    if problem.isGoalState(state):
        return 0

    manhattan_dist = util.manhattanDistance

    # Esquinas no se han visitado todavía.
    corners_not_visited = []
```

```

for i in range(4):
    if not state[1][i]:
        corners_not_visited += [corners[i]]

# n es el numero de esquinas no visitadas.
n = len(corners_not_visited)

# Si solo falta 1 esquina entonces el valor heuristico es la distancia
# Manhattan
#del estado a la esquina.
if n == 1:
    return manhattan_dist(state[0], corners_not_visited[0])

# Si falta mas de una esquina por descubrir, el valor heuristico, que se
# guarda en min_distance
# es el minimo valor entre todos los caminos manhattan posibles.
# Por ejemplo, si hay 4 esquinas sin visitar, un camino manhattan posible es
# ir de la posicion actual (del estado) a la primera esquina, de la primera
# esquina a la segunda etc
# usando como distancia la manhattan porque sabemos que es <= que la real, y
# como tomamos el minimo
# nos aseguramos de que es admisible y consistente.

# El camino es min(state[0] -> corner[0] -> corner[1], state[0] -> corner[1]
# -> corner[0]).
if n == 2:
    return min(manhattan_dist(state[0], corners_not_visited[0]),
               manhattan_dist(state[0], corners_not_visited[1])) + \
           manhattan_dist(corners_not_visited[0], corners_not_visited[1])

# El camino es state[0] -> corner[i] -> corner[j] -> corner[k].
if n == 3:
    min_distance = 9999999 # Infinito.
    for i in range(n):
        for j in range(n):
            if i == j:
                continue
            for k in range(n):
                if i == k or j == k:
                    continue
            min_distance = min(min_distance,
                               manhattan_dist(state[0],
                                                  corners_not_visited[i]) +
                               manhattan_dist(corners_not_visited[i],
                                                  corners_not_visited[j]) +

```

```

        manhattan_dist(corners_not_visited[j],
                        corners_not_visited[k]))

    return min_distance

# El camino es state[0] -> corner[i] -> corner[j] -> corner[k] -> corner[z].
if n == 4:
    min_distance = 9999999 # Infinito.
    for i in range(n):
        for j in range(n):
            if i == j:
                continue
            for k in range(n):
                if i == k or j == k:
                    continue
                for z in range(n):
                    if i == z or j == z or k == z:
                        continue
                    min_distance = min(min_distance,
                                       manhattan_dist(state[0],
                                                       corners_not_visited[i]) +
                                       manhattan_dist(corners_not_visited[i],
                                                       corners_not_visited[j]) +
                                       manhattan_dist(corners_not_visited[j],
                                                       corners_not_visited[k]) +
                                       manhattan_dist(corners_not_visited[k],
                                                       corners_not_visited[z]))

    return min_distance

```



#### 6.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

```

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', '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', 'West', 'East', 'East', 'North', 'North', 'North', 'East', 'East', 'East', 'East', 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'North', 'North', 'East', 'East', 'North', 'North']
path length: 106
*** FAIL: Heuristic resulted in expansion of 741 nodes

### Question q6: 2/3 ###

```

Figura 16: Captura de los resultados del test.

En la figura se muestran los resultados al pasar el autograder, como ya se ha comprobado. la heurística diseñada es admisible y consistente. Expande 741 nodos, lo cuál nos da una puntuación de 2 sobre 3. Como ya hemos demostrado que no hay ninguna heurística mejor que la dada en caso general, se podrían conseguir los 3 puntos haciendo una heurística para este problema en concreto.

## 6.5. Conclusiones en el comportamiento de pacman

En este apartado contestamos a las preguntas: ¿es admisible?, ¿es consistente?, ¿cuántos nodos expande?

Como se ha aclarado en apartados anteriores, nuestra heurística es admisible y consistente, expandiendo para el caso de prueba 741 nodos.

## 6.6. Respuesta a pregunta 5

**Pregunta 5:** Explica la lógica de tu heurística.

Nuestra heurística, como se ha comentado anteriormente, se basa en la estrategia de relación de restricciones del juego, en nuestro caso, las paredes del laberinto. Una vez que se “quitan” las paredes, se usa la distancia Manhattan que sabemos que es una buena heurística.

Nuestra función heurística es admisible, ya que de todos los caminos que contemplamos, los calculamos con la distancia Manhattan, y tomamos el mínimo de esas distancias, por lo que si hay muros la distancia que tiene que recorrer será igual o mayor que la que si no hubiese muros.

La heurística es consistente debido a que se usa la distancia Manhattan, por lo que al no considerar los muros,  $h(\text{estado})$  es menor o igual que el coste de moverse del estado al estado nuevo más  $h(\text{estado nuevo})$ , y es cierto que se da menor o igual porque al tomar la distancia mínima se toma en consideración ir del estado al estado nuevo.

## Sección 7

Esta práctica ha sido nuestro primer contacto con la inteligencia artificial, y con ello recordando algunos algoritmos como BFS, DFS, UCS, y aprendiendo cosas nuevas como el algoritmo  $A^*$  y las heurísticas.

Hemos comprendido lo importante que es elegir una buena función heurística para el ejercicio 6. Para llegar a la heurística, primero íbamos pensando ideas para relajar el problema o abstraerlo, y poco a poco, probando estas ideas llegamos a una heurística que expandía cada vez menos nodos.