MEMORIA PRÁCTICA 1

INTELIGENCIA ARTIFICIAL

Autores:
Juan Moreno Díez juan.morenod@estudiante.uam.es
Inés Martín Mateos ines.martinmateos@estudiante.uam.es

GRUPO **2323** - PAREJA 05

Introducción

En la práctica realizada hemos implementado diversos métodos de búsqueda informada y no informada. El principal objetivo ha sido analizar los resultados y observar los comportamientos de los algoritmos en distintas situaciones del juego de **pacman**. Estas observaciones nos han servido para poder deducir en qué situaciones es óptimo cada algoritmo de búsqueda y así saber cuándo utilizar cada uno.

La segunda parte de la práctica trata de implementar un problema que nos venía ya definido en el código. Este ha sido el problema de las esquinas. El principal objetivo de este problema es que pacman llegue a las esquinas del problema de la forma más óptima. Por último, hemos implementado una heurística consistente y no trivial para este problema.

Sección 1

1.1. Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)

Consideramos que hemos propuesto una solución de la búsqueda primero en profundidad muy clara y concisa. Es muy intuitiva y fácil de leer. Debido a la forma en la que lo hemos estructurado nos ha sido muy sencillo crear un algoritmo de búsqueda general y luego realizar cambios.

1.2. Lista y explicación de las funciones del framework usadas (1pt)

Hemos implementado la función **depthFirstSearch** del fichero **search.py** a la que se le pasa un problema por argumento que intentará resolver según la implementación que demos. Otro de los ficheros que hemos utilizado ha sido el **utils.py**. Nos ha servido para poder crear la estructura de datos de tipo Pila definido en ese fichero. La pila se caracteriza por ser una estructura de tipo LIFO (Last In First Out) por lo tanto nos ha sido muy útil para la realización de este algoritmo de búsqueda. Del resto del framework hemos utilizado **getStartState** (nos saca el estado inicial del problema), **getSuccessors** (nos saca los sucesores de un nodo que le pasamos por argumento), **isGoalState** (comprueba si un estado es el meta), **getCostOfActions** (nos saca el coste de las posibles acciones de un camino) y por último las funciones de las estructuras de **utils.py (isEmpty, push y pop)**.

1.3. Incluye el código añadido (0.25 pts)

Método de búsqueda general que recibirá el problema y una instancia de la clase correspondiente a cada estructura que se utilice en el algoritmo.

```python
def busquedaGeneral(problem, util, heuristic):
    lista_cerrados = []
    path = []

    sucesores_inicial = problem.getSuccessors(problem.getStartState())

    for s in sucesores_inicial:
        totalpath = path + [s[1]]
        cost = problem.getCostOfActions(totalpath)
        if heuristic is None:
            util.push((s, totalpath, cost))
        else:
            util.update((s, totalpath, cost), cost + heuristic(s[0], problem))

    lista_cerrados.append(problem.getStartState())

    while not util.isEmpty():
        siguiente, path = util.pop()

        if problem.isGoalState(siguiente[0]):
            # devolvemos el camino
            return path

        if siguiente[0] not in lista_cerrados:
            lista_cerrados.append(siguiente[0])
            sucesores = problem.getSuccessors(siguiente[0])

            for s in sucesores:
                # s[1] es la accion que te lleva al sucesor
                totalpath = path + [s[1]]
                cost = problem.getCostOfActions(totalpath)
                if heuristic is None:
                    util.push((s, totalpath, cost))
                else:
                    util.update((s, totalpath, cost), cost + heuristic(s[0], problem))

    # si no encuentra el destino devolvemos None
    return None
```

Como podemos observar, utilizamos la búsqueda general para todos los algoritmos (profundidad, anchura, coste uniforme y A*).

Método de búsqueda primero en profundidad que recibirá un problema a resolver. Utilizará la pila como estructura de datos para la lista de abiertos.

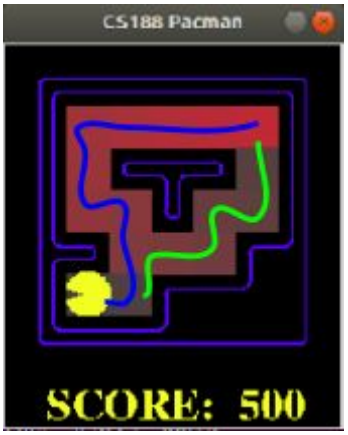
```python
def depthFirstSearch(problem):

    stack = util.Stack()

    return busquedaGeneral(problem, stack, None)
```

1.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)

Ejecución DFS tinyMaze

```
[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
```

El número de nodos que se expanden = 15



Para la búsqueda primero en profundidad en **tinyMaze** obtiene un coste total de **10** para el punto de comida. En la segunda imagen podemos observar que hay dos caminos posibles a la solución y el camino que realiza(línea azul) no es el óptimo(camino verde). Los estados que antes se han explorado han sido los que recorren el camino azul ya que están indicados en el tablero con un rojo más intenso.

Ejecución DFS mediumMaze

```
[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
```

El número de nodos que se expanden = 146

Para el laberinto **mediumMaze** obtenemos un coste total de **130** como se nos indica que nos debería salir en el enunciado de la sección 1. Como ocurre en el anterior caso, la solución óptima no es la que se proporciona para la búsqueda en profundidad, el camino más óptimo es el recorrido por la línea verde y el que toma pacman es el de la línea azul.

Ejecución DFS bigMaze

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

El número de nodos que se expanden = 390

Para el **bigMaze** no nos ha sido posible obtener una captura de todo el laberinto, pero al observarlo, nos ocurre lo mismo que en los laberintos previos. La solución que escoge DFS no es la mejor por razones que discutiremos en las próximas cuestiones. En este caso se obtiene un coste de **210**.

1.5. Conclusiones en el comportamiento de pacman, es óptimo (s/n), llega a la solución (s/n), nodos que expande, etc (1pt)

Como hemos podido comprobar en los distintos laberintos de ejemplo, el comportamiento de pacman **NO** es el óptimo y **SÍ** que llega a la solución. El número de nodos que expande va incrementando según el problema se va haciendo más grande. DFS encuentra la solución si lo utilizamos en espacios finitos, puesto que todos los nodos serán expandidos. Este algoritmo no asegura que la solución encontrada sea la óptima ya que podría encontrar una solución más profunda en otra rama que no haya sido expandida.

1.6. Respuesta a pregunta 1.1. ¿El orden de exploración es el que esperabais? ¿Pacman realmente va a todas las casillas exploradas en su camino hacia la meta? (1pt)

Sí es el orden de exploración que esperábamos, no el más corto a la meta pero si el indicado para el método de búsqueda que estamos utilizando. Pacman no va a todas las casillas exploradas, sino tardaría mucho más en ir a la meta. Las casillas exploradas antes corresponden con un rojo más intenso en el tablero y como podemos ver en el recorrido que realiza, pacman no va a todas las casillas que tengan algo de rojo. Lo que hace el algoritmo es explorar el laberinto y escoger un camino a la meta sin necesariamente pasar por todos los nodos explorados.

1.7. Respuesta a pregunta 1.2. ¿Es esta una solución de menor coste? Si no es así, pensad qué está haciendo mal la búsqueda en profundidad. (1pt)

No es la solución de menor coste. Al usar búsqueda en profundidad lo que hace es expandir el último nodo que haya en la lista de abiertos. Por lo tanto esto puede hacer que el camino elegido no sea el óptimo. Esto lo hace mediante el uso de la pila y provoca que llegue a la solución pero no necesariamente será la más óptima.

Sección 2

2.1. Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)

Al igual que en la anterior sección, la resolución y las decisiones de la búsqueda en anchura han sido muy claras y concisas porque al tener la base de la búsqueda en profundidad sólo había que utilizar una Cola en vez de una Pila. También, tal y como lo hemos estructurado nos ha sido muy sencillo crear un algoritmo de búsqueda general y luego realizar dichos cambios.

2.2. Lista y explicación de las funciones del framework usadas (1pt)

Hemos implementado la función **breadthFirstSearch** del fichero **search.py** a la que se le pasa un problema por argumento que intentará resolver según la implementación que demos. Otro de los ficheros que hemos utilizado ha sido el **utils.py**. Nos ha servido para poder crear la estructura de datos de tipo Cola definido en ese fichero. La pila se caracteriza por ser una estructura de tipo FIFO (Fist In First Out) por lo tanto nos ha sido muy útil para la realización de este algoritmo de búsqueda.

Y al igual que en búsqueda en profundidad hemos utilizado: **getStartState**, **getSuccessors**, **isGoalState**, **getCostOfActions** y por último las funciones de las estructuras de **utils.py (isEmpty, push y pop)**.

2.3. Incluye el código añadido (0.25 pts)

Método de búsqueda general que recibirá el problema y una instancia de la clase correspondiente a cada estructura que se utilice en el algoritmo.

```python
def busquedaGeneral(problem, util, heuristic):
    lista_cerrados = []
    path = []

    sucesores_inicial = problem.getSuccessors(problem.getStartState())

    for s in sucesores_inicial:
        totalpath = path + [s[1]]
        cost = problem.getCostOfActions(totalpath)
        if heuristic is None:
            util.push((s, totalpath, cost))
        else:
            util.update((s, totalpath, cost), cost + heuristic(s[0], problem))

    lista_cerrados.append(problem.getStartState())

    while not util.isEmpty():
        siguiente, path = util.pop()

        if problem.isGoalState(siguiente[0]):
            # devolvemos el camino
            return path

        if siguiente[0] not in lista_cerrados:
            lista_cerrados.append(siguiente[0])
            sucesores = problem.getSuccessors(siguiente[0])

            for s in sucesores:
                # s[1] es la accion que te lleva al sucesor
                totalpath = path + [s[1]]
                cost = problem.getCostOfActions(totalpath)
                if heuristic is None:
                    util.push((s, totalpath, cost))
                else:
                    util.update((s, totalpath, cost), cost + heuristic(s[0], problem))

    # si no encuentra el destino devolvemos None
    return None
```

Como podemos observar, utilizamos la búsqueda general para todos los algoritmos (profundidad, anchura, coste uniforme y A*).

Método de búsqueda en anchura que recibirá un problema a resolver. Utilizará la cola como estructura de datos para la lista de abiertos.

```python
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    queue = util.Queue()

    return busquedaGeneral(problem, queue, None)
```

2.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)

Ejecución BFS tinyMaze

```
[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
```

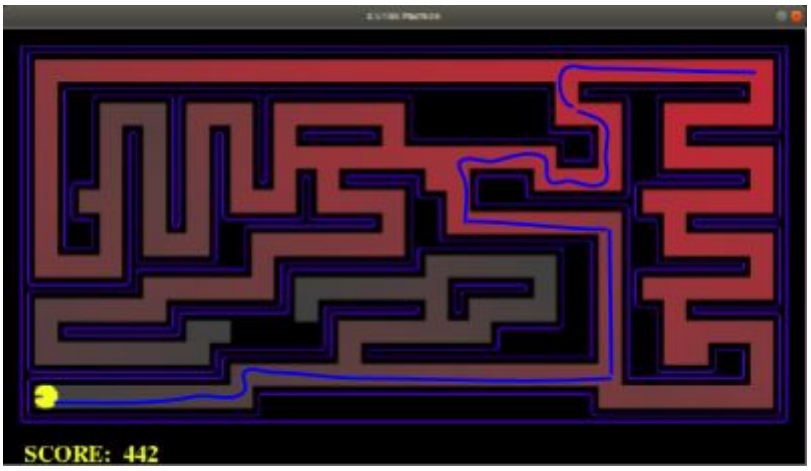
El número de nodos que se expanden = 15

Para la búsqueda en anchura en **tinyMaze** obtiene un coste total de **8** para el punto de comida. En este caso, en la segunda imagen podemos observar que el camino que realiza es el óptimo (línea azul).

Ejecución BFS mediumMaze

```
[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
```

El número de nodos que se expanden = 269



Para el laberinto **mediumMaze** obtenemos un coste total de **68**. En este caso, el camino que recorre también es la solución óptima.

Ejecución BFS bigMaze

```
[SearchAgent] using function bfs
[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)
Record:        Win
```

El número de nodos que se expanden = 620

Para el **bigMaze** no nos ha sido posible obtener una captura de todo el laberinto, pero al observar la solución es la óptima. En este caso se obtiene un coste de **210**, igual que en profundidad.

2.5. Conclusiones en el comportamiento de pacman, es óptimo (s/n), llega a la solución (s/n), nodos que expande, etc (1pt)

Como hemos podido comprobar en los distintos laberintos de ejemplo, el comportamiento de pacman **SÍ** es el óptimo y **SÍ** que llega a la solución. Al igual que en profundidad el número de nodos que expande va incrementando según el problema se va haciendo más grande. desventajas BA. Existen dos posibles caminos a la meta con costes distintos, comparando este con el de búsqueda en profundidad, vemos claramente que si obtiene el camino óptimo ya que tiene menor coste que el elegido por búsqueda en profundidad.

2.6. Respuesta a pregunta 2. ¿BA encuentra una solución de menor coste? Si no es así, verificad vuestra implementación. (1pt)

Como podemos ver en los resultados, BA sí que encuentra una solución de menor coste comparada con BP. Esto se debe a que BA recorre el árbol de forma horizontal y encuentra la primera solución existente (la más corta). Pero esto no implica que la solución sea la de menor coste. Al coger la primera solución que existe en el árbol, puede que haya alguna solución más profunda/ ancha que obtenga un menor coste que la elegida. El comportamiento de BA se debe a que utiliza la cola(estructura de datos FIFO) para ir recorriendo el problema.

Sección 3

3.1. Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)

Como en los algoritmos anteriores, ha sido sencillo implementar este algoritmo ya que hemos abstraído el algoritmo de búsqueda general a una función externa lo cual nos ha permitido aplicar coste uniforme tan solo modificando la estructura de datos utilizada, utilizando una Cola de Prioridad con función que establece una prioridad en función del coste acumulado.

3.2. Lista y explicación de las funciones del framework usadas (1pt)

Hemos implementado la función **uniformCostSearch** del fichero **search.py** a la que se le pasa un problema por argumento que intentará resolver según la implementación que demos. Otro de los ficheros que hemos utilizado ha sido el **utils.py**.

Y al igual que en las anteriores búsquedas hemos utilizado: **getStartState**, **getSuccessors**, **isGoalState**, **getCostOfActions** y por último las funciones de las estructuras de **utils.py (isEmpty, push y pop)**.

3.3. Incluye el código añadido (0.25 pts)

Método de búsqueda general que recibirá el problema y una instancia de la clase correspondiente a cada estructura que se utilice en el algoritmo.

Método de búsqueda general que recibirá el problema y una instancia de la clase correspondiente a cada estructura que se utilice en el algoritmo.

```python
def busquedaGeneral(problem, util, heuristic):
    lista_cerrados = []
    path = []

    sucesores_inicial = problem.getSuccessors(problem.getStartState())

    for s in sucesores_inicial:
        totalpath = path + [s[1]]
        cost = problem.getCostOfActions(totalpath)
        if heuristic is None:
            util.push((s, totalpath, cost))
        else:
            util.update((s, totalpath, cost), cost + heuristic(s[0], problem))

    lista_cerrados.append(problem.getStartState())

    while not util.isEmpty():
        siguiente, path = util.pop()

        if problem.isGoalState(siguiente[0]):
            # devolvemos el camino
            return path

        if siguiente[0] not in lista_cerrados:
            lista_cerrados.append(siguiente[0])
            sucesores = problem.getSuccessors(siguiente[0])

            for s in sucesores:
                # s[1] es la accion que te lleva al sucesor
                totalpath = path + [s[1]]
                cost = problem.getCostOfActions(totalpath)
                if heuristic is None:
                    util.push((s, totalpath, cost))
                else:
                    util.update((s, totalpath, cost), cost + heuristic(s[0], problem))

    # si no encuentra el destino devolvemos None
    return None
```

Como podemos observar, utilizamos la búsqueda general para todos los algoritmos (profundidad, anchura, coste uniforme y A*)..

Método de **costeUniforme** que recibirá un problema a resolver. Utilizará una cola de prioridad con función como estructura de datos para la lista de abiertos. Implementación del método costeCamino que se le pasará a Cola de Prioridad con función (devuelve el coste en ese instante).

```python
def costeCamino(item):
    return item[2]

def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    pQueueF = util.PriorityQueueWithFunction(costeCamino)

    return busquedaGeneral(problem, pQueueF, None)
```

3.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)

Ejecución UCS mediumMaze

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.1 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
```

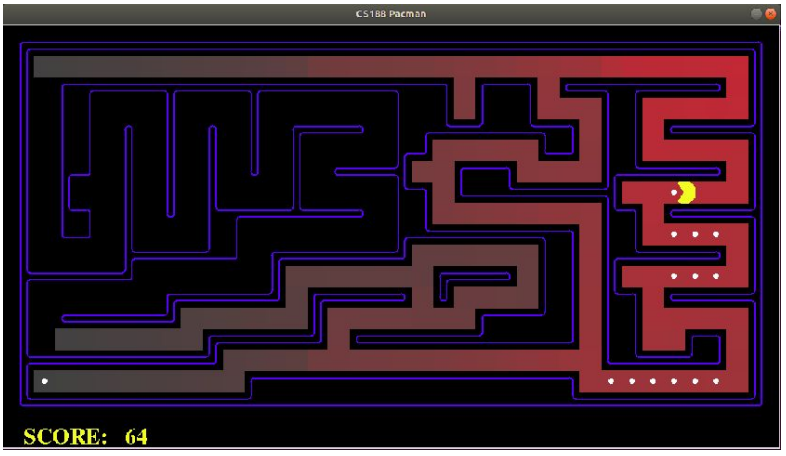
El número de nodos que se expanden = 269

Para el laberinto **mediumMaze** obtenemos un coste total de **68** (al igual que en BFS). En este caso, el camino que recorre es la solución óptima.

Ejecución UCS mediumDottedMaze

```
Path found with total cost of 1 in 0.1 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
```

El número de nodos que se expanden = 186

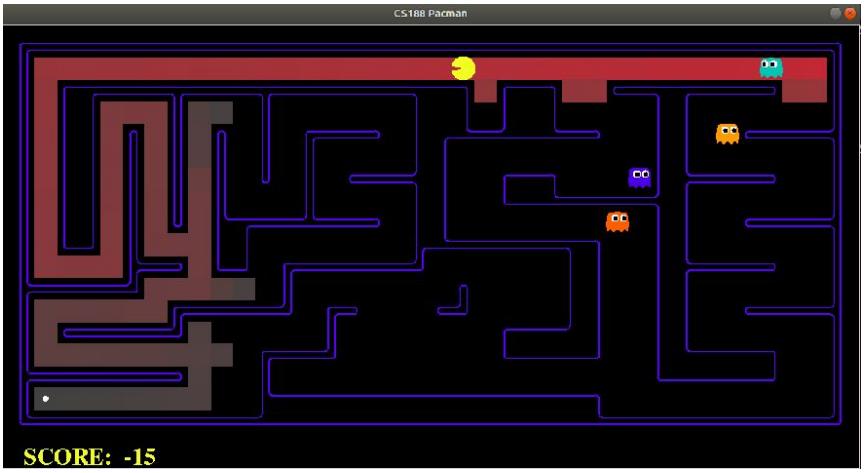


Para el laberinto **mediumDottedMaze** obtenemos un coste total de **1**. En este caso, el camino que recorre es la solución óptima.

Ejecución UCS mediumScaryMaze

```
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
```

El número de nodos que se expanden = 108



Para el laberinto **mediumScaryMaze** obtenemos un coste total de **68719479864** (valor algo extraño, no tenemos muy claro el porqué). En este caso, el camino que recorre es la solución óptima.

3.5. Conclusiones en el comportamiento de pacman, es óptimo (s/n), llega a la solución (s/n), nodos que expande, etc (1pt)

Como hemos podido comprobar en los distintos laberintos de ejemplo, el comportamiento de pacman **SÍ** es el óptimo y **SÍ** que llega a la solución. Además, el número de nodos que expande va decrementando según el problema se va haciendo más pequeño. Como se puede observar al realizar la ejecución, pacman no explora los estados en los que haya fantasmas cerca y realiza una mayor exploración en los estados en los que haya comida cercana.

Sección 4

4.1. Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)

La solución que hemos propuesto para A* ha sido muy similar a lo utilizado en la búsqueda de coste uniforme. Para optimizar código y no repetirlo, hemos incluido los siguientes aspectos en la búsqueda general: le hemos añadido un parámetro que "**heuristic**" que valdrá None para todos los algoritmos de búsqueda menos para A*. Más tarde, a la hora de realizar un push, en el caso de que "**heuristic**" contenga un valor que no sea None, en vez de push haremos un update y como segundo argumento le pasaremos el coste sumado a la heurística del sucesor y el problema.

Debido a la estructuración que hemos diseñado, nos ha sido mucho más sencillo depurar código y cuando tuvimos una versión funcional correcta, añadir nuevas cosas para demás algoritmos de búsqueda no nos ha sido una gran complicación.

4.2. Lista y explicación de las funciones del framework usadas (1pt)

Para este ejercicio hemos implementado la función "**aStarSearch**" con el uso de la cola de prioridad incluida en **utils.py**. Hemos utilizado las mismas funciones que en ejercicios previos para la obtención de sucesores y estados del problema. La única novedad incluida ha sido la comentada previamente, el uso del **update** de la cola de prioridad si se indica un valor específico para la heurística.

4.3. Incluye el código añadido (0.25 pts)

Método de búsqueda general que recibirá el problema y una instancia de la clase correspondiente a cada estructura que se utilice en el algoritmo.

```python
def busquedaGeneral(problem, util, heuristic):
    lista_cerrados = []
    path = []

    sucesores_inicial = problem.getSuccessors(problem.getStartState())

    for s in sucesores_inicial:
        totalpath = path + [s[1]]
        cost = problem.getCostOfActions(totalpath)
        if heuristic is None:
            util.push((s, totalpath, cost))
        else:
            util.update((s, totalpath, cost), cost + heuristic(s[0], problem))

    lista_cerrados.append(problem.getStartState())

    while not util.isEmpty():
        siguiente, path = util.pop()

        if problem.isGoalState(siguiente[0]):
            # devolvemos el camino
            return path

        if siguiente[0] not in lista_cerrados:
            lista_cerrados.append(siguiente[0])
            sucesores = problem.getSuccessors(siguiente[0])

            for s in sucesores:
                # s[1] es la accion que te lleva al sucesor
                totalpath = path + [s[1]]
                cost = problem.getCostOfActions(totalpath)
                if heuristic is None:
                    util.push((s, totalpath, cost))
                else:
                    util.update((s, totalpath, cost), cost + heuristic(s[0], problem))

    # si no encuentra el destino devolvemos None
    return None
```

Como podemos observar, utilizamos la búsqueda general para todos los algoritmos (profundidad, anchura, coste uniforme y A*).

Método de búsqueda A* que recibirá un problema a resolver. Utilizará una cola de prioridad como estructura de datos para la lista de abiertos.

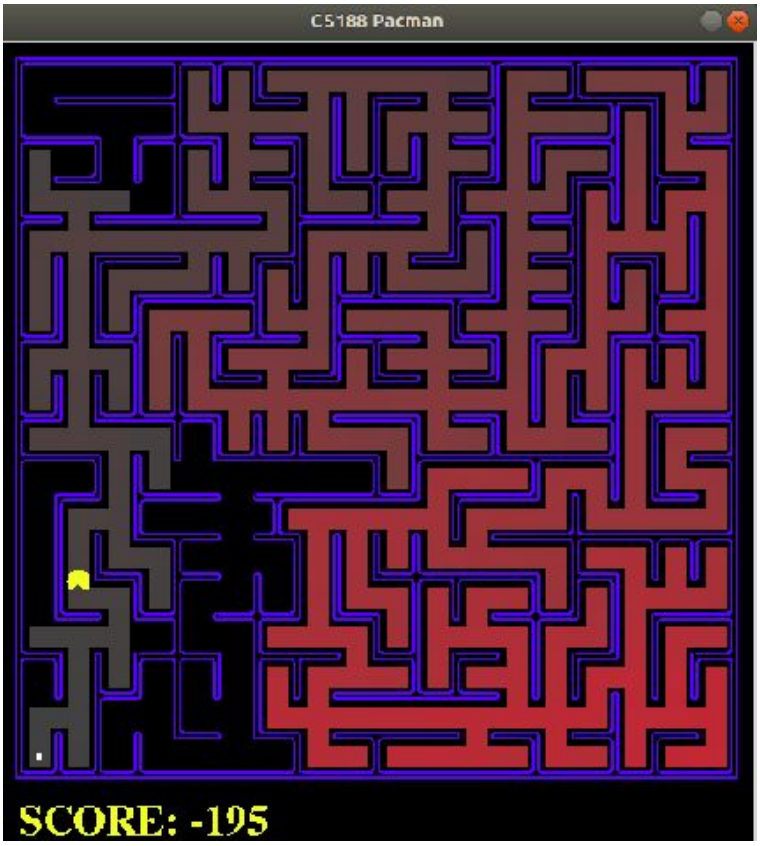
```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    pQueue = util.PriorityQueue()

    return busquedaGeneral(problem, pQueue, heuristic)
```

4.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)

Ejecución Manhattan A*

```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.2 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
```

El número de nodos que se expanden = 549



Para el laberinto bigMaze utilizando A* con la heurística de Manhattan obtenemos un coste de 210 y comparando con otros algoritmos, no se obtienen mejores resultados.

4.5. Conclusiones en el comportamiento de pacman, es óptimo (s/n), llega a la solución (s/n), nodos que expande, etc (1pt)

En este caso, el comportamiento de pacman **SÍ** es óptimo y **SÍ** que llega a la solución. Los algoritmos que van recorriendo el árbol según el menor coste son los que obtienen las soluciones más óptimas. Es probable que no se obtenga la solución más corta a la meta, pero con respecto a coste, tanto A* como UCS, obtendrán el menor coste para llegar a la meta. El número de nodos expandidos se puede observar al lado de las capturas de pantalla.

4.6. Respuesta a pregunta 3. ¿Qué sucede en openMaze para las diversas estrategias de búsqueda? (1pt)

Hemos ejecutado las búsquedas DFS, BFS y UCS para openMaze. La conclusión más clara que hemos sacado es que el camino óptimo lo realizan BFS y UCS.

En cuanto a los costes y nodos, para DFS hemos obtenido 298 y 576 nodos expandidos y para BFS y UCS 54 de coste y 682 nodos expandidos.

El resultado tan pobre de búsqueda en profundidad se debe a que escoge la solución más profunda en el árbol generado y no otras posibles soluciones que podrían obtenerse antes de llegar a ese nodo.

Sección 5

5.1. Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)

Para abordar el problema de búsqueda de esquinas lo primero que hemos hecho ha sido codificar la representación del estado que más adecuase al problema, la cual ha sido la siguiente: por un lado la posición actual de pacman, y por el otro una tupla que contiene las coordenadas de las esquinas restantes por visitar.

A la hora de generar sucesores, se comprueba si el sucesor en concreto es una esquina que no hayamos visitado, es decir, que sus coordenadas estén presentes en la tupla del estado y en ese caso se eliminan de esta.

La comprobación de estado objetivo será ver si la tupla está vacía (ya se han visitado todas las esquinas).

5.2. Lista y explicación de las funciones del framework usadas (1pt)

Para completar las funciones anteriores, hemos cogido la estructura ya existente en otras clases que generaban otros problemas y hemos incluido el código que comprobaba todo lo mencionado anteriormente en las búsquedas, relacionado con el manejo del estado.

5.3. Incluye el código añadido (0.25 pts)

```
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

    # Codificacion de estado: Posicion de pacman + esquinas restantes por visitar
    self.state = (self.startingPosition, self.corners)
```

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

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    if len(state[1]) == 0:
        return True

    return False
```

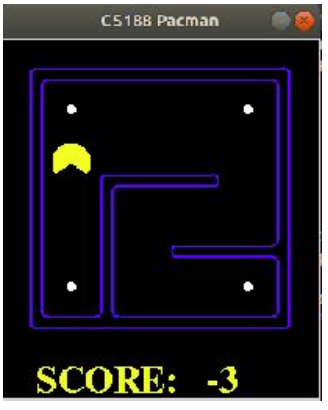
```python
def getSuccessors(self, state):
    """
    Returns successor states, the actions they require, and a cost of 1.

     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 = []
    self._expanded += 1 # DO NOT CHANGE
    for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
        x,y = state[0]
        dx, dy = Actions.directionToVector(direction)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            corners = deepcopy(state[1])
            next = (nextx,nexty)
            if next in corners:
                list_corners = list(corners)
                list_corners.remove(next)
                corners = tuple(list_corners)
            successors.append( ( ((nextx, nexty), corners), direction, 1) )
    return successors
```

```python
def getCostOfActions(self, actions):
    """
    Returns the cost of a particular sequence of actions.  If those actions
    include an illegal move, return 999999.  This is implemented for you.
    """
    if actions == None: return 999999
    x,y= self.startingPosition
    for action in actions:
        dx, dy = Actions.directionToVector(action)
        x, y = int(x + dx), int(y + dy)
        if self.walls[x][y]: return 999999
    return len(actions)
```

5.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)

Ejecución tinyCorners

```
[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
```



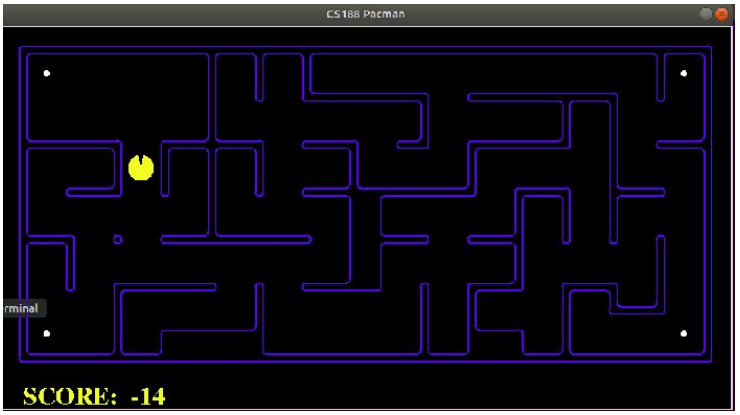
El número de nodos que se expanden = 252

Para la búsqueda de todas las esquinas en **tinyCorners** se obtiene un coste total de **28.** En este caso, en la segunda imagen podemos observar que el camino que realiza es el óptimo y lo recorre entero hasta encontrar toda la comida en las esquinas.

Ejecución mediumCorners

```
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.6 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
```

El número de nodos que se expanden = 1966



Para la búsqueda de todas las esquinas en **mediumCorners** se obtiene un coste total de **106.** En este caso, en la segunda imagen podemos observar que también el camino que realiza es el óptimo y lo recorre entero hasta encontrar toda la comida en las esquinas.

5.5. Conclusiones en el comportamiento de pacman, es óptimo (s/n), llega a la solución (s/n), nodos que expande, etc (1pt)

Como hemos podido comprobar en los distintos laberintos de ejemplo, el comportamiento de pacman **SÍ** es el óptimo y **SÍ** que llega a la solución. Además, el número de nodos que expande va incrementando según el problema se va haciendo más grande. Los recorridos además son los óptimos con el objetivo de que consiga llegar a todas las esquinas con la mayor facilidad y utilizando el menor recorrido posible. Todo esto teniendo en cuenta las distancias desde el origen hasta las esquinas y posteriormente desde la esquina que vaya a visitar hasta las siguientes.

Sección 6

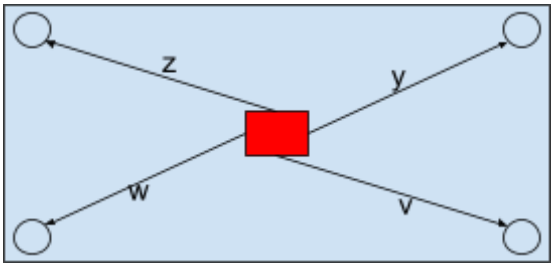
6.1. Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)

Para abordar este último problema, hemos empezado por plantear varias heurísticas que pudiesen obtener un resultado óptimo para el problema de las esquinas.

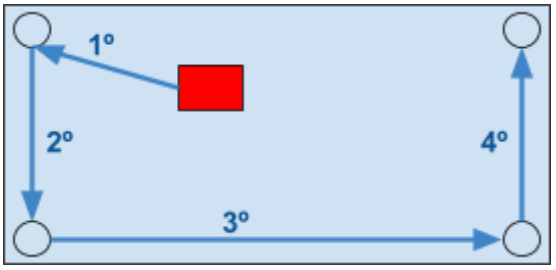
A continuación mencionaremos alguno de esos intentos previos fallidos para posteriormente analizar el resultado final obtenido (6.5.)

La primera heurística que pensamos, hacía uso del concepto de la distancia de Manhattan, ya que se trataba de acumular las distancias en línea recta a las esquinas restantes por visitar desde el punto del laberinto en el que estuviese. Al ejecutar el autograder, esta heurística era inadmisible o inconsistente para los 3 primeros ejemplos. Esto tenía sentido ya que la heurística no tenía en cuenta que si visitabas una esquina, ya no partían desde el punto anterior, lo cual nos llevó al segundo intento.

h(n) = z+y+w+n



En el segundo intento, la idea era primero encontrar la esquina más cercana en línea recta, y a partir de ahí visitar las esquinas en orden, de nuevo en línea recta teniendo en cuenta la distancia entre esquina y esquina. El valor de la heurística sería la acumulación de dicho camino. De nuevo esta heurística no era adecuada ya que era inadmisible e inconsistente para los 3 primeros ejemplos.



Finalmente, la heurística solución la explicaremos en el apartado 6.5.

6.2. Lista y explicación de las funciones del framework usadas (1pt)

Para completar las funciones anteriores, hemos cogido la estructura ya existente en otras clases que generaban otros problemas y hemos incluido el código que comprobaba todo lo mencionado anteriormente en las búsquedas, relacionado con el manejo del estado.

6.3. Incluye el código añadido (0.25 pts)

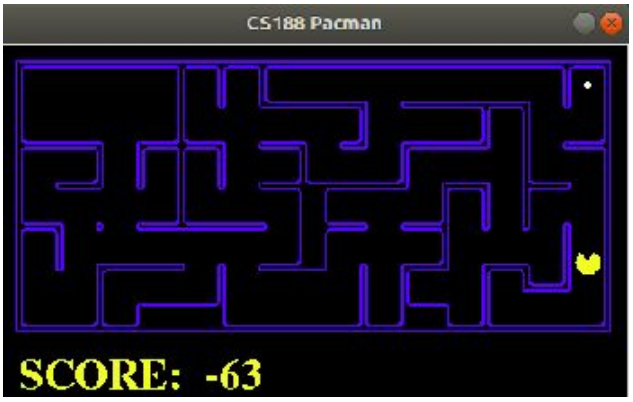
```
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      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)
    position = state[0]
    cornersLeft = state[1]
    sort = {}
    value = 0

    if len(cornersLeft) == 0:
        return value

    while len(cornersLeft) != 0:
        values = {}
        for c in cornersLeft:
            values[c] =  abs(position[0] - c[0]) + abs(position[1] - c[1])

        sort = dict(sorted(values.items(), key=lambda item: item[1]))
        value += list(sort.values())[0]
        position = list(sort.keys())[0]

        list_corners = list(cornersLeft)
        list_corners.remove(position)
        cornersLeft = tuple(list_corners)

    return value
```

6.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)

Ejecución mediumCorners Astar

```
Path found with total cost of 106 in 0.2 seconds
Search nodes expanded: 692
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

El número de nodos que se expanden = **692** y el coste es **106**.



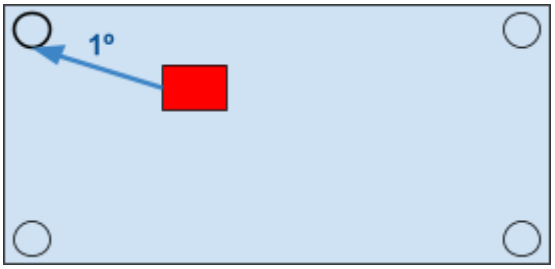
6.5. Conclusiones en el comportamiento de pacman, es óptimo (s/n), llega a la solución (s/n), nodos que expande, etc (1pt)

Como hemos podido comprobar en el laberinto de ejemplo anterior, el comportamiento de pacman **SÍ** es el óptimo y **SÍ** que llega a la solución. Además, el número de nodos es **692** que es un resultado muy aceptable de nodos expandidos. Podemos observar que encuentra una solución mucho antes que BFS ya que expande 692 hasta llegar a la solución y BFS expande 1966 (diferencia muy grande a tener en cuenta).

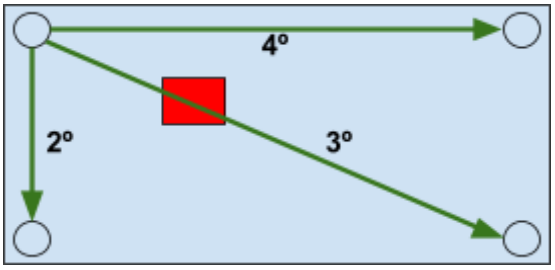
6.6. Respuesta a pregunta 5: Explica la lógica de tu heurística (1pt)

Tras los 2 primeros intentos fallidos, llegamos a una heurística similar a ambas que nos dió un resultado aceptable. La segunda idea iba bien encaminada, sólo que nos faltó tener en cuenta la posibilidad de no recorrer las esquinas en orden, después del paso 1, si no volver a valorar qué distancia era más cercana y escoger esa ruta.

1. Buscamos la esquina más cercana desde el estado actual.



2. Desde la esquina elegida en el paso nº1 elegir la siguiente esquina más cercana a esta (siempre en línea recta con distancia de Manhattan) y así sucesivamente.



El valor devuelto por la heurística será el coste acumulado de dicha ruta. Como las distancias son siempre en línea recta, se demuestra que la heurística es monótona ya que el coste estimado será siempre menor que el coste real.

Sección 7

Como conclusión, esta práctica ha sido muy interesante ya que al programar algoritmos de búsqueda sobre un juego como Pacman, que es conocido por todo el mundo, se trata de una forma bastante dinámica de no sólo aprenderlos si no de contrastar los resultados de forma visual lo cuál nos ha parecido entretenido.

La práctica nos ha ayudado a entender más en detalle los distintos tipos de búsqueda y como realmente todos son muy similares entre sí, pero a la vez pueden obtener resultados muy diferentes.

Nota de la memoria (40% de la práctica)

Total de puntos (X/31.5)