Memoria Práctica 1

Título: Búsqueda

Autores: Daniel Cerrato y David Garitagoitia

Sección 1 (1 punto)

1.1. Comentario personal en el enfoque y decisiones de la solución propuesta (0.5 puntos)

Inicialmente, decidimos atacar el problema por nuestra cuenta, usando backtracking. Pronto dimos con una solución del ejercicio, pero al intentar extrapolarlo para BFS vimos que se complicaba el código, además de que no seguía el consejo de la práctica de buscar un código que sería fácil de llevar al resto de algoritmos.

Después, decidimos seguir el algoritmo propuesto en las transparencias de clase donde se eliminan los estados repetidos para evitar bucles innecesarios. Para que se pudiera extrapolar, decidimos usar el contenedor de los estados abiertos para meter cada nodo y la lista de movimientos usados para llegar a cada nodo, de forma que cada nodo hijo aumenta por su cuenta la lista de movimientos de su nodo padre.

Sin embargo, este método no era el correcto, a pesar de que funcionaba perfectamente, pues hacía muy complicado poder generar una función general para todos los algoritmos. Así que, finalmente, nos dimos cuenta de cómo configurar el estado inicial que recibimos para que el código quede perfecto. A través de eso, desarrollamos un poco más la idea de la lista de movimientos usados para llegar a cada nodo: esta vez uniríamos la lista con los estados en su totalidad haciendo que la lista contuviese los nodos explorados hasta llegar al que se está evaluando; y así, simplemente, había que guardar la lista de nodos en la estructura correspondiente y expandir el último añadido a la lista.

Así fue posible generar la función de búsqueda general e hizo que en la función de búsqueda en profundidad solo tuviéramos que llamar a la función general, pasando por argumento la estructura usada, en este caso una pila, pues se quiere explorar primero a los hijos antes que a los hermanos.

1.1.1. Lista & explicación de las funciones del framework usadas

Para este ejercicio, aparte de las funciones básicas que usa el programa para ejecutarse, se ha codificado la función “depthFirstSearch” y la nueva generada por nosotros mismos llamada “generalSearch”.

“depthFirstSearch” no es más que una llamada a la función de búsqueda general “generalSearch”, pasándole como argumento una pila.

“generalSearch” contiene el algoritmo de búsqueda en grafo sin estado repetidos. En el set de estados cerrados (“closed”), se meterá solo la posición expandida, no el nodo entero porque esto último haría que Pac-Man volviera por donde vino.

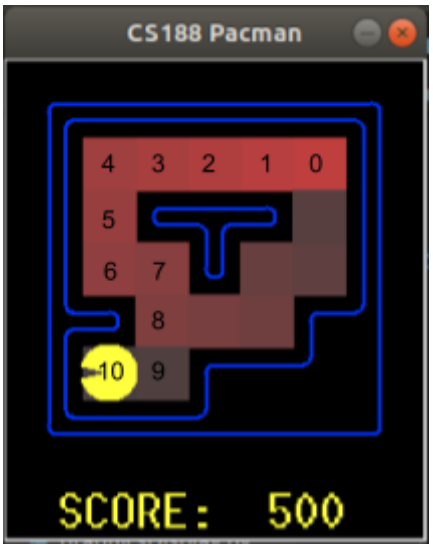
1.1.2. **Incluye el código añadido**

```python
def depthFirstSearch(problem):
    return generalSearch(problem, util.Stack())
```

```python
def generalSearch(problem, struct):
    states = []
    closed = set()
    opened = struct

    first_state = problem.getStartState()

    states.append((first_state, 'Stop', 0))
    opened.push(list(states))

    while True:
        if opened.isEmpty() is True:
            return []
        states = opened.pop()
        if problem.isGoalState(states[-1][0]):
            action_list = []
            for state in states:
                action_list.append(state[1])
            return action_list[1:]
        if states[-1][0] not in closed:
            closed.add(states[-1][0])
            for next_state in problem.getSuccessors(states[-1][0]):
                aux_list = list(states)
                aux_list.append(next_state)
                opened.push(list(aux_list))
```

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

```
eps@eps:~/Escritorio/search$ python3 pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
[SearchAgent] using function dfs
[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
```



Como se puede observar en la imagen superior, el coste total del camino escogido por el algoritmo es de 10, estos son los “pasos” que ha dado Pac-Man para llegar a la comida, como se refleja en la imagen de la izquierda.

Este no es el camino óptimo, pero como se trata de búsqueda en profundidad, el algoritmo decidió explorar primero el camino de la izquierda antes que ir por debajo.

A continuación, siguen los resultados del laberinto mediano y grande.

```
eps@eps:~/Escritorio/search$ python3 pacman.py -l mediumMaze -p SearchAgent -a fn=dfs
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.1 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
```

```
eps@eps:~/Escritorio/search$ python3 pacman.py -l bigMaze -p SearchAgent -a fn=dfs
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.2 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
```

1.2. Conclusiones en el comportamiento de pacman, es óptimo (s/n), llega a la solución (s/n), nodos que expande, etc (0.5 puntos)

Considerando óptimo como la solución de menor coste, este algoritmo no es óptimo ya que no siempre encuentra la mínima solución. Sí llega a la solución del problema.

Expansión de nodos para los distintos mapas:

tinyMaze:	Search nodes expanded: 15
mediumMaze:	Search nodes expanded: 146
bigMaze:	Search nodes expanded: 390

1.2.1. Respuesta a pregunta 1.1

El orden de exploración es el esperado y a la vez no. Lo es porque, una vez elegido el camino inicial (izquierda o abajo), explora todas las posiciones accesibles nuevas primero y, en caso de no tener más nuevas, explora el camino alternativo más "nuevo" de los que ya conoce. Pero no es el esperado totalmente porque inicialmente tenía dos caminos que coger y, por cómo está hecho el código de los sucesores, ha ido por el camino "malo".

1.2.2. Respuesta a pregunta 1.2

Pacman no va a todas las casillas exploradas, solo sigue el camino que sabe que le llevará a la meta.

1.2.3. Respuesta a pregunta 2

Esta solución no es la de menor coste, el problema de dfs es que, si encuentra una solución, puede no ser la más eficiente.

Sección 2 (1 punto)

2.1. Comentario personal en el enfoque y decisiones de la solución propuesta (0.5 puntos)

Después de acabar por primera vez el algoritmo de dfs, nos dimos cuenta de que el código se complicaba mucho si queríamos que el algoritmo fuese igual que en dfs, porque este código estaba pensado para hacer backtracking, pero en bfs no sirve.

Tras modificar el código de dfs, se consiguió que en bfs se usará el mismo algoritmo. Pero como decíamos anteriormente, no era el código correcto del todo, así que una vez encontrado el final, simplemente se dejó el código como una llamada a la función general de búsqueda, pasando por argumento la estructura de cola para la lista de estados abiertos.

2.1.1. Lista & explicación de las funciones del framework usadas

En este caso, como en el anterior, se ha codificado la función “breadthFirstSearch”, como una llamada a la función general de búsqueda (“generalSearch”) pasando como argumento una cola.

Como en la sección anterior, el código de “generalSearch” es el algoritmo de búsqueda en grafo sin repetición de estados.

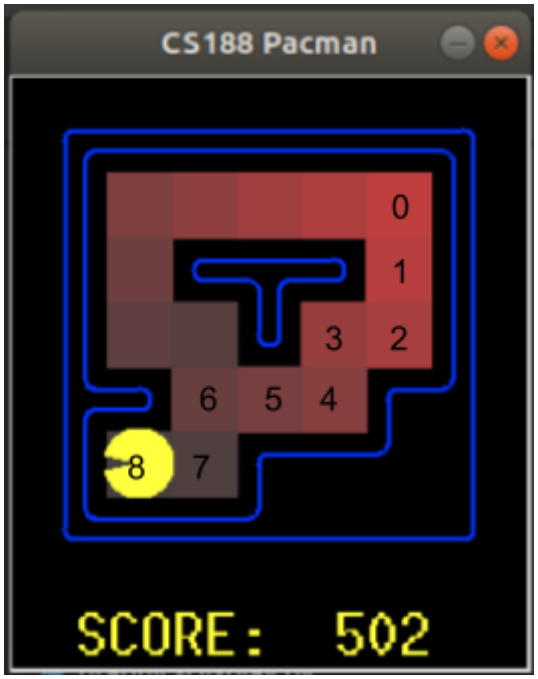
2.1.2. Incluye el código añadido

```
def breadthFirstSearch(problem):
    return generalSearch(problem, util.Queue())
```

```
def generalSearch(problem, struct):
    states = []
    closed = set()
    opened = struct

    first_state = problem.getStartState()

    states.append((first_state, 'Stop', 0))
    opened.push(list(states))

    while True:
        if opened.isEmpty() is True:
            return []
        states = opened.pop()
        if problem.isGoalState(states[-1][0]):
            action_list = []
            for state in states:
                action_list.append(state[1])
            return action_list[1:]
        if states[-1][0] not in closed:
            closed.add(states[-1][0])
            for next_state in problem.getSuccessors(states[-1][0]):
                aux_list = list(states)
                aux_list.append(next_state)
                opened.push(list(aux_list))
```

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

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



Como se puede observar en la imagen superior, el coste total del camino escogido por el algoritmo es de 8, estos son los "pasos" que ha dado Pac-Man para llegar a la comida, como se refleja en la imagen de la izquierda.

Este es el camino óptimo, pues en este caso, bfs omite la "aleatoriedad" del camino escogido y expandido primero, como pasaba en dfs.

A continuación, siguen los resultados del laberinto mediano y grande.

```
e420552@9-11-9-11:~/search$ python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
```

```
e420552@9-11-9-11:~/search$ python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
```

2.2. Conclusiones en el comportamiento de pacman, es óptimo (s/n), llega a la solución (s/n), nodos que expande, etc. (0.5 puntos)

Considerando óptimo como la solución de menor coste, este algoritmo sí que es óptimo ya que la primera solución a la que llega siempre es la de menor coste. Sí llega a la solución del problema.

Expansión de nodos para los distintos mapas:

tinyMaze:	Search nodes expanded: 15
mediumMaze:	Search nodes expanded: 269
bigMaze:	Search nodes expanded: 620

2.2.1. Respuesta a pregunta 3

Búsqueda en anchura siempre encuentra la solución de menor coste, en caso de que todos los costes internos valgan igual.

Sección 3 (1 punto)

3.1 Comentario personal en el enfoque y decisiones de la solución propuesta (0.5 puntos)

Una vez se ha logrado el código general de búsqueda simplemente basta con modificar la estructura de datos por una cola de prioridad en la que se tengan en cuenta los costes de moverse a los siguientes nodos, para que se pueda usar la función de generalSearch empleamos la PriorityQueueWithFunction
pasando la función para calcular el coste llamada nodeCost cuya función es la de calcular el coste total de llegar desde el inicio hasta el nodo a evaluar (recorre toda la lista de nodos y retorna el sumatorio de los costes

3.1.1 Lista & explicación de las funciones del framework usadas

Como se ha explicado anteriormente, se ha codificado la función "uniformCostSearch", como una llamada a la función general de búsqueda ("generalSearch") pasando como argumento una cola pero de prioridad.

Como en la sección anterior, el código de "generalSearch" es el algoritmo de búsqueda en grafo sin repetición de estados.

3.1.2 Incluye el código añadido

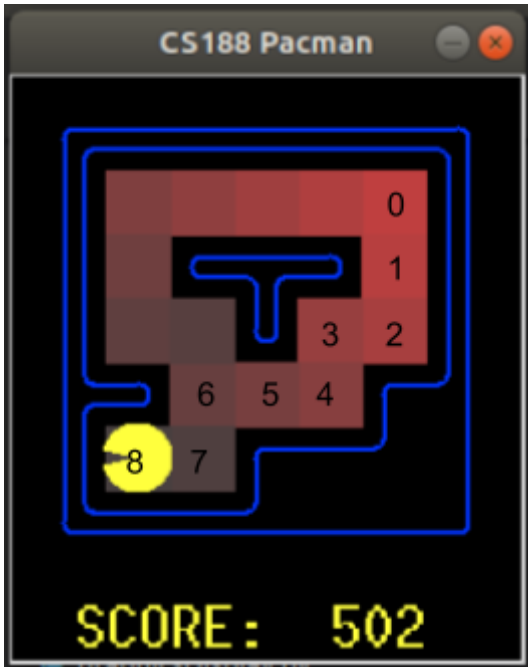
```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""

    "*** YOUR CODE HERE ***"
    from util import nodeCost
    return generalSearch(problem,
        util.PriorityQueueWithFunction(
            nodeCost
        )
    )
```

```
def nodeCost(stateList):
    cost = 0
    for state in stateList:
        cost += state[2]
    return cost
```

Literalmente se trata únicamente de codificar lo explicado en los apartados anteriores no hay mucha complejidad en estas funciones

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

```
e420552@9-11-9-11:~/search$ python3 pacman.py -l tinyMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[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
```



Como se puede observar en la imagen superior, el coste total del camino escogido por el algoritmo es de 8, estos son los “pasos” que ha dado Pac-Man para llegar a la comida, como se refleja en la imagen de la izquierda.

Este es el camino óptimo, pues en este caso, ucs omite la “aleatoriedad” del camino escogido y expandido primero, como pasaba en dfs.

A continuación, siguen los resultados del laberinto mediano y grande.

```
e420552@9-11-9-11:~/search$ python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
```

```
e420552@9-11-9-11:~/search$ python3 pacman.py -l bigMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
```

3.2 Conclusiones en el comportamiento de pacman, es optimo (s/n), llega a la solución (s/n), nodos que expande, etc (0.5 puntos)

El comportamiento es óptimo ya que encuentra la solución de menor coste pues tiene en cuenta los costes en la cola de prioridad para garantizar en todo momento la expansión de los caminos de menor coste total, similar a un algoritmo Dijkstra de búsqueda en grafos.

Llega a la solución en caso de tener pues recorre todos los caminos posibles.

Sección 4 (2 puntos)

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

Para este caso el enfoque es similar al caso anterior solo que en esta ocasión además de tener en cuenta el coste del nodo también se debe tener en cuenta la heurística del nodo a evaluar, es un código similar al anterior, únicamente modificamos la función de coste a la hora de insertar para que tenga en cuenta la heurística

4.1.1 Lista & explicación de las funciones del framework usadas

Empleamos aStarSearch que en su interior llamará a la función de búsqueda general (no es necesario la función de coste ya que empleamos una lambda para obtener la suma con la función de coste de la búsqueda anterior junto con la heurística

4.1.2 Incluye el código añadido

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

    from util import nodeCost
    return generalSearch(problem,
        util.PriorityQueueWithFunction(
            lambda a : nodeCost(a) + heuristic(a[-1][0], problem)
        )
    )
```

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

```
e420552@9-11-9-11:~/search$ python3 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
```

```
e420552@9-11-9-11:~/search$ python3 pacman.py -l openMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
```

```
e420552@9-11-9-11:~/search$ python3 pacman.py -l openMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
```

```
eps@eps:~/Escritorio/search$ python3 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 0.1 seconds
Search nodes expanded: 535
Pacman emerges victorious! Score: 456
```

Las conclusiones se pueden ver en el apartado 4.2.1

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

El comportamiento es óptimo pues llega a la solución de menor coste así como lo hacía el aStarSearch la diferencia con el anterior es que tendrá en cuenta la heurística para la expansión por lo que si la heurística es buena reducirá en gran medida el número de nodos a expandir para llegar a la menor solución.

4.2.1. Respuesta a pregunta 3

Como se puede observar, dfs (primera imagen) es el que ha obtenido una puntuación más baja, esto es porque ha recorrido más posiciones del tablero para llegar al final.

Esto se debe a que expande en profundidad, de manera que el algoritmo queda a expensas del nodo que vaya a explorar primero. Sin embargo, en el resto de casos la puntuación es la misma pues van directos a la meta.

Se puede observar también que bfs y ucs son exactamente iguales, esto se debe a que los nodos tienen coste 1 en todos los casos, de manera que ucs usa una cola de prioridad que se transforma en una cola básica como en bfs.

Por último, se puede observar que el mejor método es el de búsqueda informada con A* y la heurística de la distancia de Manhattan. Esto se debe a que la heurística permite expandir primero a los nodos que van más directos hacia la solución, expandiendo menor cantidad a lo sumo.

Sección 5 (2 puntos)

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

Puesto que este problema consiste en buscar todas las esquinas, vimos la necesidad de crear estados nuevos, donde se encuentre la información de la casilla actual y una lista de las esquinas alcanzadas. De esta forma, podemos hacer un seguimiento de las esquinas alcanzadas en cada momento y saber que, cuando la lista tenga “x” esquinas diferentes, el algoritmo ha llegado a una solución.

5.1.1. Lista & explicación de las funciones del framework usadas

En la función “getStartState” retornamos una tupla con la posición inicial en el tablero y la lista de esquinas alcanzadas vacía o, en caso de que la posición inicial sea una esquina, la lista contendrá esta esquina.

En la función “isGoalState” simplemente retornamos el resultado de la comparación entre el número de esquinas a alcanzar y la longitud de las esquinas alcanzadas en ese nodo.

En la función “getSuccessors” generamos una lista de posibles nodos accesibles, esto es quitando los nodos que caen en una posición de pared, junto con la lista de esquinas alcanzadas actualizada tras comprobar que la posición a la que se accede es una esquina o no.

5.1.2. Incluye el código añadido

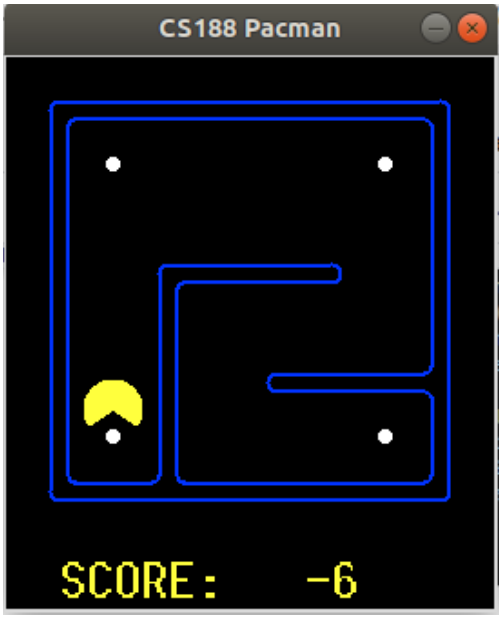
```
def getStartState(self):
    if(self.startingPosition in self.corners):
        return (self.startingPosition, frozenset(self.startingPosition))
    return (self.startingPosition, frozenset())
```

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

```
def getSuccessors(self, state):
    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
        x,y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            nextState = (nextx, nexty)
            cost = 1
            vCorners = set(state[1])
            if(nextState in self.corners and nextState not in vCorners):
                vCorners.add(nextState)
            successors.append(((nextState, frozenset(vCorners)), action, cost))

    self._expanded += 1
    return successors
```

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

```
eps@eps:~/Escritorio/search$ python3 pacman.py -l tinyCorners -p Sea
rchAgent -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
```



Como se puede observar, Pac-Man escoge la esquina inferior izquierda para empezar a recorrer las esquinas. Acto seguido simplemente va la que tiene encima, luego a la derecha y, finalmente, a por la última.

Si se intenta encontrar otro orden, se puede comprobar que este orden es el óptimo.
Esto es porque se usa bfs como algoritmo de búsqueda y, recordamos, es un algoritmo que siempre encuentra el camino óptimo hacia la solución, si esta existe.

Como se indica en la práctica, se alcanza un coste de 28 pasos.

```
eps@eps:~/Escritorio/search$ python3 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,
prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
```

En el caso del laberinto mediano, se expanden poco menos de 2000 nodos (1966), como se indica en el enunciado de la práctica.

5.2. Conclusiones en el comportamiento de pacman, es optimo (s/n), llega a la solución (s/n), nodos que expande, etc (1 punto)

Como indicábamos en el apartado anterior, al usar el algoritmo bfs para la búsqueda de esquinas, siempre obtendremos el camino óptimo en caso de existir alguno.

Sección 6 (3 puntos)

6.1. Comentario personal en el enfoque y decisiones de la solución propuesta (1.5 puntos)

Intuitiva y rápidamente, ambos llegamos a la misma conclusión: buscar recorrer las esquinas accediendo a ellas en el orden que más corto haga el trayecto, suponiendo que no hay barreras. Esto es lo que se conoce como “relajar” el problema. Como hemos visto en teoría, si relajamos el problema, encontramos una heurística monótona y, por tanto, admisible.

Primeramente, se trató de llegar a este método “hardcodeando”, es decir, escribiendo los diferentes casos directamente. Tras solucionar los pequeños problemas, se consiguió que el código funcionase y resultó ser una heurística muy buena, pero dejar la cantidad de código que había era una mala práctica.

Así que intentamos juntar todos los casos a través de bucles y recursiones, factorizando los casos buscando las repeticiones en el código. Se terminó consiguiendo con algo de arreglo, pero durante la clase el profesor nos indicó que existía una función en python que nos facilitaba parte del algoritmo, así que sin duda decidimos usarla.

6.1.1. Lista & explicación de las funciones del framework usadas

“cornersHeuristic” es la función que calcula la heurística de un estado. En nuestro caso, hemos usado la heurística comentada anteriormente. Para saber más acerca del código de la heurística, ver apartado 6.2.1.

6.1.2. Incluye el código añadido

```python
def cornersHeuristic(state, problem):
    from util import manhattanDistance
    import itertools

    if(problem.isGoalState(state)):
        return 0

    corners = problem.corners
    notVisited = list()
    for s in corners:
        if(s not in state[1]):
            notVisited.append(s)

    permutations = list(itertools.permutations(notVisited))

    minDistance = 0
    for perm in permutations:
        distance = manhattanDistance(state[0], perm[0])
        for i in range(len(perm) - 1):
            distance += manhattanDistance(perm[i], perm[i+1])
        if minDistance == 0 or distance < minDistance:
            minDistance = distance

    return minDistance
```

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

```
eps@eps:~/Escritorio/search$ python3 pacman.py -l mediumCorners -p A
StarCornersAgent
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 741
Pacman emerges victorious! Score: 434
```

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

El resultado de nuestra heurística es más que satisfactorio. El camino escogido por el algoritmo es el óptimo, llegando a la solución lo antes posible y expandiendo relativamente pocos nodos (741), bastantes menos de los que se exigen para la máxima nota (1200).

6.2.1. Respuesta a pregunta 5: heurística

Como explicamos en el apartado 6.1.1, el código intenta recorrer todas las esquinas sin contar con muros buscando la ruta más corta desde la posición actual.

Para ello consta de los siguientes pasos:

1. Comprobación de estar en un estado final o no, en caso positivo devolver 0.
2. En caso negativo, conseguir una lista con las esquinas no visitadas.
3. Crear todas las permutaciones posibles con las esquinas no visitadas.
4. Calcular la cantidad de pasos total como suma de las distancias de Manhattan desde la posición actual a una esquina y de ahí al resto de esquinas siguiendo el orden de la permutación.
5. Comparamos con la distancia más corta hasta el momento y nos quedamos con la más pequeña.
6. Finalmente, al comprobar todas las permutaciones, devolver la distancia obtenida.

Sección 7

Comentarios personales de la realización de esta práctica

Nos ha parecido una práctica muy completa y entretenida, donde hemos aprendido a manejar por completo los algoritmos de búsqueda en grafos y en árboles y a pensar en estados y heurísticas varias para diversos problemas.

Nota de la memoria (40% de la práctica)