

eman ta zabal zazu



Universidad  
del País Vasco

Euskal Herriko  
Unibertsitatea

Técnicas de Inteligencia Artificial

Ingeniería Informática de Gestión y Sistemas de Información

# Practica 1

## Problemas de Búsqueda

Autor(es):

Xabier Gabiña

Diego Montoya

7 de octubre de 2024

# Índice general

<b>1. Introducción</b>	<b>3</b>
<b>2. Ejercicios</b>	<b>4</b>
2.1. DFS - Depth First Search . . . . .	4
2.2. BFS - Breadth First Search . . . . .	6
2.3. UCS - Uniform Cost Search . . . . .	7
2.4. A* Search . . . . .	8
2.5. Corners Problem: Representación . . . . .	9
2.6. Corners Problem: Heurística . . . . .	12
2.7. Eating All The Dots: Heurística . . . . .	14
2.8. Suboptimal Search . . . . .	16
<b>3. Resultados</b>	<b>18</b>
3.1. Casos de pruebas . . . . .	18
3.1.1. DFS . . . . .	18
3.1.2. BFS . . . . .	19
3.1.3. UCS . . . . .	21
3.1.4. A* . . . . .	22
3.1.5. Corners Problem: Representación . . . . .	23
3.1.6. Corners Problem: Heurística . . . . .	24
3.1.7. Eating All The Dots . . . . .	25
3.1.8. Suboptimal Search . . . . .	26
3.2. Autograder . . . . .	27

# Índice de Códigos

2.1. Implementación inicial del DFS . . . . .	4
2.2. Implementación final del DFS . . . . .	5
2.3. Implementación final del BFS . . . . .	6
2.4. Implementación final del UCS . . . . .	7
2.5. Implementación final del A* . . . . .	8
2.6. Implementación inicial del problema de las esquinas . . . . .	9
2.7. Implementación final del problema de las esquinas . . . . .	10
2.8. Implementación inicial de la heurística del problema de las esquinas . . . . .	12
2.9. Implementación final de la heurística del problema de las esquinas . . . . .	12
2.10. Implementación inicial de la heurística del problema de las esquinas . . . . .	14
2.11. Implementación final de la heurística del problema de las esquinas . . . . .	14
2.12. Implementación final del problema de las esquinas . . . . .	16
3.1. Prueba DFS . . . . .	18
3.2. Prueba BFS . . . . .	19
3.3. Prueba UCS . . . . .	21
3.4. Prueba A* . . . . .	22
3.5. Prueba Representación Corners Problem . . . . .	23
3.6. Prueba Heurística Corners Problem . . . . .	24
3.7. Prueba Eating All The Dots . . . . .	25
3.8. Prueba Suboptimal Search . . . . .	26

# 1. Introducción

En el marco de la asignatura de Técnicas de Inteligencia Artificial, se nos ha propuesto implementar y analizar diversos algoritmos de búsqueda aplicados al contexto de un proyecto académico desarrollado por la Universidad de Berkeley, basado en el clásico juego Pacman. El objetivo principal de esta práctica es profundizar en el funcionamiento de diferentes estrategias de búsqueda, estudiando su eficiencia y comportamiento en diferentes escenarios.

Los algoritmos de búsqueda son fundamentales en el campo de la inteligencia artificial, ya que permiten encontrar soluciones óptimas o satisfactorias en problemas complejos. En esta práctica, nos enfocaremos en tres tipos de algoritmos de búsqueda no informados: Depth First Search (DFS), Breadth First Search (BFS) y Uniform Cost Search (UCS). Además, exploraremos un algoritmo de búsqueda informado: A\*. Cada uno de estos algoritmos tiene sus propias características y aplicaciones, y su estudio nos permitirá comprender mejor sus ventajas y limitaciones.

A lo largo de este documento, se presentarán las implementaciones de cada uno de estos algoritmos, junto con una descripción detallada de su funcionamiento y análisis de su rendimiento. Se incluirán ejemplos prácticos y se discutirán los resultados obtenidos en diferentes escenarios de búsqueda. El objetivo es proporcionar una visión completa y comprensiva de cómo estos algoritmos pueden ser aplicados en la resolución de problemas de búsqueda en inteligencia artificial.

## 2. Ejercicios

### 2.1. DFS - Depth First Search

#### Descripción

DFS o Depth First Search es un algoritmo de búsqueda no informado que se basa en la exploración de todos los nodos de un grafo siguiendo una rama hasta llegar a un nodo hoja, para después retroceder y explorar otra rama. Este algoritmo se implementa mediante una pila, en la que se van almacenando los nodos a visitar. Su coste en tiempo es de  $O(b^m)$ , donde  $b$  es el factor de ramificación y  $m$  es la profundidad máxima del árbol. Su coste en espacio es de  $O(bm)$ , donde  $b$  es el factor de ramificación y  $m$  es la profundidad máxima del árbol.

#### Primera implementación

```
1  def depthFirstSearch(problem):
2      """
3      Implementación del algoritmo de búsqueda en profundidad.
4
5      Args:
6          problem (SearchProblem): Problema de búsqueda
7      Returns:
8          list: Lista de acciones para llegar al objetivo
9      """
10     stack = [problem] # Pila para almacenar los nodos a visitar
11     visited = set()    # Conjunto para almacenar los nodos visitados
12     path = []          # Lista para almacenar el camino al nodo objetivo
13
14     while stack:       # Mientras haya elementos en el stack
15         nodo_actual = stack.pop() # Sacar el último elemento de la pila
16         if nodo_actual in visited: # Si el nodo actual ya ha sido visitado
17             continue
18         visited.add(nodo_actual) # Marcar el nodo actual como visitado
19         path.append(nodo_actual.contenido) # Añadir el nodo actual al camino
20         if nodo_actual.isGoalState(): # Si el nodo actual es el objetivo
21             return path
22         for hijo in reversed(nodo_actual.getSucesor()): # Añadir los hijos del nodo actual a la
pila
23             stack.append(hijo)
24
```

Código 2.1: Implementación inicial del DFS

## Implementación Final

```
1  def depthFirstSearch(problem):
2      """
3      Implementación del algoritmo de búsqueda en profundidad.
4
5      Args:
6          problem (SearchProblem): Problema de búsqueda
7      Returns:
8          list: Lista de acciones para llegar al objetivo
9      """
10     stack = util.Stack() # Añadir el nodo inicial a la pila
11     stack.push([problem.getStartState(), []])
12     visited = set() # Conjunto para almacenar los nodos visitados
13
14     while not stack.isEmpty(): # Mientras haya elementos en el stack
15         nodo_actual = stack.pop() # Sacar el último elemento de la pila
16         if problem.isGoalState(nodo_actual[0]): # Si el nodo actual es el objetivo
17             return nodo_actual[1] # Devolver el camino
18         if nodo_actual[0] not in visited:
19             visited.add(nodo_actual[0])
20             for estado, accion, costo in reversed(problem.getSuccessors(nodo_actual[0])):
21                 camino = nodo_actual[1] + [accion]
22                 stack.push([estado, camino])
23
```

Código 2.2: Implementación final del DFS

## Comentarios

En la implementación inicial del algoritmo DFS, la implementación de la lista del path es incorrecta por dos motivos. El primero, es que se añade el nodo actual al path y no la acción que se ha de tomar para llegar a la meta. La segunda, es que se añaden todos los nodos visitados al path en orden de visita lo que hace que si llegamos a una hoja final se crea un salto en el path que sería imposible de realizar.

Ambos problemas se han solucionado en la implementación final del algoritmo DFS.

## 2.2. BFS - Breadth First Search

### Descripción

BFS o Breadth First Search es un algoritmo de búsqueda no informado que se basa en la exploración de todos los nodos de un grafo nivel por nivel. Este algoritmo se implementa mediante una cola en la que se van almacenando los nodos que se deben visitar, manteniendo un orden de llegada basado en los niveles.

Su coste en tiempo es de  $O(b^d)$ , donde  $b$  es el factor de ramificación y  $d$  es la profundidad del nodo más cercano del árbol y su coste en espacio es de  $O(b^d)$ , ya que debe almacenar todos los nodos del nivel actual antes de pasar al siguiente.

### Implementación Final

```
1  def breadthFirstSearch(problem):
2      """
3      Implementacion del algoritmo de busqueda en anchura.
4
5      Args:
6          problem (SearchProblem): Problema de busqueda
7      Returns:
8          list: Lista de acciones para llegar al objetivo
9      """
10     queue = util.Queue() # Añadir el nodo inicial a la cola
11     queue.push([problem.getStartState(), []])
12     visited = set()      # Conjunto para almacenar los nodos visitados
13
14     while not queue.isEmpty(): # Mientras haya elementos en la cola
15         nodo_actual = queue.pop() # Sacar el primer elemento de la cola
16         if problem.isGoalState(nodo_actual[0]): # Si el nodo actual es el objetivo
17             return nodo_actual[1] # Devolver el camino
18         if nodo_actual[0] not in visited:
19             visited.add(nodo_actual[0])
20             for estado, accion, costo in problem.getSuccessors(nodo_actual[0]): # Añadir los hijos
21                 del nodo actual a la cola
22                 camino = nodo_actual[1] + [accion]
23                 queue.push([estado, camino])
```

Código 2.3: Implementación final del BFS

### Comentarios

Este algoritmo ha tenido una sencilla implementación, ya que las dos únicas diferencias con el caso anterior es el cambio de la pila por una cola y que al añadir los nodos no es necesario hacerlo en el orden inverso ya que la cola mantiene el orden de llegada.

## 2.3. UCS - Uniform Cost Search

### Descripción

UCS o Uniform Cost Search es un algoritmo de búsqueda no informada que expande los nodos en función del costo acumulado. En contrario al BFS que prioriza la profundidad de los nodos el UCS utiliza la cola de prioridad para ordenar los nodos dependiendo su costo acumulado y expande primero el nodo con menor costo, obteniendo así una solución óptima en términos de costo.

El coste en tiempo del UCS es de  $O(b^{1+\frac{C^*}{\epsilon}})$ , donde  $b$  es el factor de ramificación,  $C^*$  es el costo de la solución óptima y  $\epsilon$  es el menor costo de transición entre nodos. Su coste de espacio es así mismo  $O(b^{1+\frac{C^*}{\epsilon}})$  ya que debe de almacenar todos los nodos en la cola de prioridad hasta encontrar la solución óptima

### Implementación Final

```
1 def uniformCostSearch(problem):
2     """
3     Implementacion del algoritmo de busqueda de coste uniforme.
4
5     Args:
6         problem (SearchProblem): Problema de busqueda
7     Returns:
8         list: Lista de acciones para llegar al objetivo
9     """
10    queue = util.PriorityQueue() # Añadir el nodo inicial a el heap
11    queue.push([problem.getStartState(), [], 0], 0)
12    visited = set() # Conjunto para almacenar los nodos visitados
13
14    while not queue.isEmpty(): # Mientras haya elementos en el stack
15        nodo_actual = queue.pop() # Sacar el último elemento de la pila
16        if problem.isGoalState(nodo_actual[0]): # Si el nodo actual es el objetivo
17            return nodo_actual[1] # Devolver el camino
18        if nodo_actual[0] not in visited:
19            visited.add(nodo_actual[0])
20            for estado, accion, costo in problem.getSuccessors(nodo_actual[0]): # Añadir los hijos
21                del nodo_actual a la pila
22                camino = nodo_actual[1] + [accion]
23                queue.push([estado, camino, nodo_actual[2] + costo], nodo_actual[2] + costo)
```

Código 2.4: Implementación final del UCS

### Comentarios

Al igual que en el caso anterior, en este, solo hemos necesitado hacer dos cambios. El primero es que la cola pasa a ser una cola de prioridad y ligado a esto el metodo push de la misma requiere de un segundo argumento que es el valor de prioridad.



## 2.4. A\* Search

### Descripción

El A\* es un algoritmo de búsqueda informada que utiliza las ventajas del UCS y Greedy Best-First Search, utilizando una función heurística para guiar la búsqueda hacia el objetivo de una manera más eficiente. A\* expande los nodos en mediante una función de costo total. Donde  $f(n) = g(n) + h(n)$ ,  $g(n)$  siendo el costo acumulado desde el nodo inicial hasta  $n$  y  $h(n)$  es una estimación heurística del costo  $n$  hasta el objetivo. Este algoritmo se implementa mediante una cola de prioridad, donde los nodos se ordenan según su valor  $f(n)$  y se expande primero el nodo con menor valor  $f(n)$ .

Su coste en tiempo es de  $O(b^d)$ , donde  $b$  es el factor de ramificación y  $d$  es la profundidad de la solución óptima, sin embargo el tiempo puede reducirse si la heurística es eficiente. Así mismo su coste de espacio es de  $O(b^d)$ , ya que se debe de almacenar todos los nodos generados en la cola de prioridad para asegurar la solución óptima.

### Implementación Final

```
1 def manhattanHeuristic(state, problem):
2     """
3     Heurística de Manhattan.
4
5     Args:
6         state (tuple): Coordenadas del estado
7         problem (SearchProblem): Problema de búsqueda
8     Returns:
9         int: Distancia de Manhattan al objetivo
10    """
11    return util.manhattanDistance(state, problem.goal)
12
13 def aStarSearch(problem, heuristic=nullHeuristic):
14    """
15    Implementación del algoritmo de búsqueda A*.
16
17    Args:
18        problem (SearchProblem): Problema de búsqueda
19        heuristic (function): Heurística para el problema
20    Returns:
21        list: Lista de acciones para llegar al objetivo
22    """
23    queue = util.PriorityQueue() # Añadir el nodo inicial a el heap
24    queue.push([problem.getStartState(), [], 0], 0)
25    visited = set() # Conjunto para almacenar los nodos visitados
26
27    while not queue.isEmpty(): # Mientras haya elementos en el stack
28        nodo_actual = queue.pop() # Sacar el último elemento de la pila
29        if problem.isGoalState(nodo_actual[0]): # Si el nodo actual es el objetivo
30            return nodo_actual[1] # Devolver el camino
31        if nodo_actual[0] not in visited:
32            visited.add(nodo_actual[0])
33            for estado, accion, costo in problem.getSuccessors(nodo_actual[0]): # Añadir los hijos
34                camino = nodo_actual[1] + [accion]
35                queue.push([estado, camino, nodo_actual[2] + costo], nodo_actual[2] + costo +
36                           heuristic(estado, problem))
```

Código 2.5: Implementación final del A\*

### Comentarios

En este caso, la implementación del A\* es muy similar a la del UCS, con la única diferencia de que se suma la heurística al valor de prioridad de la cola de prioridad.

El eurístico utilizado en este caso es la distancia de Manhattan, que es la suma de las distancias horizontales y verticales entre dos puntos. Únicamente se ha tenido que añadir la función `manhattanHeuristic` que calcula la distancia de Manhattan entre dos puntos del archivo `util.py`.

## 2.5. Corners Problem: Representación

### Descripción

En este problema, Pacman debe visitar las cuatro esquinas de un laberinto antes de que se considere que ha alcanzado el objetivo. El problema se define mediante un estado inicial, un estado objetivo y una serie de acciones que Pacman puede realizar para moverse por el laberinto.

Para representar este problema, se ha definido una clase `CornersProblem` que hereda de la clase `SearchProblem`. Esta clase almacena la información del laberinto, la posición inicial de Pacman y las esquinas que deben ser visitadas. Además, se definen los métodos necesarios para obtener el estado inicial, comprobar si un estado es objetivo y obtener los sucesores de un estado dado.

### Primera implementación

```
1 class CornersProblem(search.SearchProblem):
2     """
3     This search problem finds paths through all four corners of a layout.
4
5     You must select a suitable state space and successor function
6     """
7
8     def __init__(self, startingGameState):
9         """
10        Stores the walls, pacman's starting position and corners.
11        """
12        self.walls = startingGameState.getWalls()
13        self.startingPosition = startingGameState.getPacmanPosition()
14        top, right = self.walls.height - 2, self.walls.width - 2
15        self.corners = ((1, 1), (1, top), (right, 1), (right, top))
16        for corner in self.corners:
17            if not startingGameState.hasFood(*corner):
18                print('Warning: no food in corner ' + str(corner))
19        self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
20        # Please add any code here which you would like to use
21        # in initializing the problem
22
23    def getStartState(self):
24        return self.startingPosition
25
26    def isGoalState(self, state):
27        """
28        Returns whether this search state is a goal state of the problem.
29        """
30        if state in self.corners:
31            self.explored.add(state)
32        if len(self.explored) == 4:
33            return True
34        return False
35
36    def getSuccessors(self, state):
37        """
38        Returns successor states, the actions they require, and a cost of 1.
39
40        As noted in search.py:
41        For a given state, this should return a list of triples, (successor,
42        action, stepCost), where 'successor' is a successor to the current
43        state, 'action' is the action required to get there, and 'stepCost'
44        is the incremental cost of expanding to that successor
45        """
46
47        successors = []
48        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
49            x, y = state
50            dx, dy = Actions.directionToVector(action)
51            nextx, nexty = int(x + dx), int(y + dy)
52            if not self.walls[nextx][nexty]:
```

```

53         nextState = (nextx, nexty)
54         cost = self.costFn(nextState)
55         successors.append((nextState, action, cost))
56
57     self._expanded += 1 # DO NOT CHANGE
58     return successors
59
60 def getCostOfActions(self, actions):
61     """
62     Returns the cost of a particular sequence of actions. If those actions
63     include an illegal move, return 999999. This is implemented for you.
64     """
65     if actions is None: return 999999
66     x, y = self.startingPosition
67     for action in actions:
68         dx, dy = Actions.directionToVector(action)
69         x, y = int(x + dx), int(y + dy)
70         if self.walls[x][y]: return 999999
71     return len(actions)
72

```

Código 2.6: Implementación inicial del problema de las esquinas

## Implementación Final

```

1 class CornersProblem(search.SearchProblem):
2     """
3     This search problem finds paths through all four corners of a layout.
4
5     You must select a suitable state space and successor function
6     """
7
8     def __init__(self, startingGameState):
9         """
10        Stores the walls, pacman's starting position and corners.
11        """
12        self.walls = startingGameState.getWalls()
13        self.startingPosition = startingGameState.getPacmanPosition()
14        top, right = self.walls.height - 2, self.walls.width - 2
15        self.corners = ((1, 1), (1, top), (right, 1), (right, top))
16        for corner in self.corners:
17            if not startingGameState.hasFood(*corner):
18                print('Warning: no food in corner ' + str(corner))
19        self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
20        # Please add any code here which you would like to use
21        # in initializing the problem
22
23    def getStartState(self):
24        """
25        Returns the start state, including the initial position and the visited corners.
26        """
27        # The start state now includes an empty set of visited corners
28        return (self.startingPosition, ())
29
30    def isGoalState(self, state):
31        """
32        Returns whether this search state is a goal state of the problem.
33        """
34        # Unpack the state
35        position, visitedCorners = state
36
37        # Check if we have visited all four corners
38        return len(visitedCorners) == 4
39
40    def getSuccessors(self, state):
41        """
42        Returns successor states, the actions they require, and a cost of 1.
43        """
44        successors = []

```

```

45     # Unpack the current state
46     currentPosition, visitedCorners = state
47
48     for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
49         x, y = currentPosition
50         dx, dy = Actions.directionToVector(action)
51         nextx, nexty = int(x + dx), int(y + dy)
52
53         # Check if the next position is a wall
54         if not self.walls[nextx][nexty]:
55             nextPosition = (nextx, nexty)
56             # Check if we have reached a new corner
57             newVisitedCorners = list(visitedCorners)
58             if nextPosition in self.corners and nextPosition not in visitedCorners:
59                 newVisitedCorners.append(nextPosition)
60
61             # Create a new state with updated corner list
62             nextState = (nextPosition, tuple(newVisitedCorners))
63             cost = 1 # Step cost is always 1
64             successors.append((nextState, action, cost))
65
66     self._expanded += 1 # DO NOT CHANGE
67     return successors
68
69 def getCostOfActions(self, actions):
70     """
71     Returns the cost of a particular sequence of actions. If those actions
72     include an illegal move, return 999999. This is implemented for you.
73     """
74     if actions is None: return 999999
75     x, y = self.startingPosition
76     for action in actions:
77         dx, dy = Actions.directionToVector(action)
78         x, y = int(x + dx), int(y + dy)
79         if self.walls[x][y]: return 999999
80     return len(actions)
81

```

Código 2.7: Implementación final del problema de las esquinas

## Comentarios

El problema que hemos tenido en este ejercicio ha sido la definición del estado inicial. La primera aproximación que se nos ocurrió fue la de eliminar los corners ya visitados de la definición del problema pero esto parecía generar conflicto por lo que decidimos añadir una tupla con los corners visitados al estado inicial. De esta forma ya tenemos una representación del estado inicial que incluye la posición de pacman y las esquinas visitadas y podemos implementar el método `isGoalState` de forma efectiva.

## 2.6. Corners Problem: Heurística

### Descripción

En este problema, se nos pide implementar una heurística para el problema de las esquinas, que permita calcular una estimación del coste mínimo para visitar todas las esquinas restantes. La heurística debe ser admisible y consistente, es decir, no puede sobreestimar el coste real de llegar a la meta y debe ser siempre menor o igual al coste real.

Para este problema, hemos implementado una heurística basada en la distancia de Manhattan, que calcula la distancia mínima entre la posición actual de Pacman y la esquina más cercana que aún no ha sido visitada. Esta heurística es admisible y consistente, ya que siempre subestima el coste real de recorrer todas las esquinas y es siempre menor o igual al coste real.

### Primera implementación

```
1 def cornersHeuristic(state, problem):
2     """
3     A heuristic for the CornersProblem that you defined.
4
5     state:      The current search state
6                 (a data structure you chose in your search problem)
7
8     problem: The CornersProblem instance for this layout.
9
10    This function should always return a number that is a lower bound on the
11    shortest path from the state to a goal of the problem; i.e. it should be
12    admissible (as well as consistent).
13    """
14    currentPosition, visitedCorners = state
15    corners = problem.corners
16
17    # Identificar las esquinas que aún no han sido visitadas
18    unvisitedCorners = [corner for corner in corners if corner not in visitedCorners]
19
20    # Calcular la distancia mínima utilizando la distancia Manhattan
21    current = currentPosition
22    while unvisitedCorners:
23        distances = [(util.manhattanDistance(current, corner), corner) for corner in unvisitedCorners]
24        minDistance, closestCorner = min(distances)
25
26        heuristic = minDistance
27        current = closestCorner
28        unvisitedCorners.remove(closestCorner)
29
30    return heuristic
31
```

Código 2.8: Implementación inicial de la heurística del problema de las esquinas

### Implementación Final

```
1 def cornersHeuristic(state, problem):
2     """
3     A heuristic for the CornersProblem that you defined.
4
5     state:      The current search state (currentPosition, visitedCorners)
6     problem: The CornersProblem instance for this layout.
7     """
8     currentPosition, visitedCorners = state
9     corners = problem.corners
10
11    # Identificar las esquinas que aún no han sido visitadas
12    unvisitedCorners = [corner for corner in corners if corner not in visitedCorners]
13
```

```

14 # Calcular la distancia mínima utilizando la distancia Manhattan
15 heuristic = 0
16 current = currentPosition
17
18 while unvisitedCorners:
19     # Encontrar la esquina más cercana (distancia Manhattan)
20     distances = [(util.manhattanDistance(current, corner), corner) for corner in
unvisitedCorners]
21     minDistance, closestCorner = min(distances)
22
23     # Agregar la distancia mínima a la heurística y moverse a la siguiente esquina
24     heuristic += minDistance
25     current = closestCorner
26     unvisitedCorners.remove(closestCorner)
27
28 return heuristic
29

```

Código 2.9: Implementación final de la heurística del problema de las esquinas

## Comentarios

El principal error en la primera implementación estaba en que no se estaba calculando la heurística de forma correcta ya que se estaba usando como heurística siempre la distancia mínima entre la posición actual y la esquina más cercana cuando debería estar dando el coste mínimo de llegar a todas las esquinas restantes. Esto provocaba subestimar el costo real de recorrer todas las esquinas, ya que no se tienen en cuenta las distancias adicionales necesarias para visitar las otras esquinas.

## 2.7. Eating All The Dots: Heurística

### Descripción

En este problema, Pacman debe recoger todas las comidas del laberinto antes de que se considere que ha alcanzado el objetivo. El problema se define mediante un estado inicial, un estado objetivo y una serie de acciones que Pacman puede realizar para moverse por el laberinto.

Para representar este problema, se ha definido una clase `FoodSearchProblem` que hereda de la clase `SearchProblem`. Esta clase almacena la información del laberinto, la posición inicial de Pacman y las comidas que deben ser recogidas. Además, se definen los métodos necesarios para obtener el estado inicial, comprobar si un estado es objetivo y obtener los sucesores de un estado dado.

### Primera implementación

```
1 def foodHeuristic(state, problem):
2     """
3     A heuristic for the FoodSearchProblem.
4
5     state: (pacmanPosition, foodGrid)
6     problem: The FoodSearchProblem instance.
7     """
8     position, foodGrid = state
9
10    # Convert foodGrid to a list of food positions
11    foodList = foodGrid.asList()
12
13    # Si no hay comida restante, la heurística es 0
14    if not foodList:
15        return 0
16
17    # Calcular la distancia de laberinto a cada punto de comida desde la posición actual de Pacman
18    distances = [mazeDistance(position, food, problem.startingGameState) for food in foodList]
19
20    # La heurística es la distancia máxima a cualquier punto de comida
21    return min(distances)
22
```

Código 2.10: Implementación inicial de la heurística del problema de las esquinas

### Implementación Final

```
1 def foodHeuristic(state, problem):
2     """
3     A heuristic for the FoodSearchProblem.
4
5     state: (pacmanPosition, foodGrid)
6     problem: The FoodSearchProblem instance.
7     """
8     position, foodGrid = state
9
10    # Convert foodGrid to a list of food positions
11    foodList = foodGrid.asList()
12
13    # Si no hay comida restante, la heurística es 0
14    if not foodList:
15        return 0
16
17    # Calcular la distancia de laberinto a cada punto de comida desde la posición actual de Pacman
18    distances = [mazeDistance(position, food, problem.startingGameState) for food in foodList]
19
20    # La heurística es la distancia máxima a cualquier punto de comida
21    return max(distances)
22
```

Código 2.11: Implementación final de la heurística del problema de las esquinas

## Comentarios

El problema en la primera implementación era que se estaba calculando la heurística como la distancia mínima a cualquier punto de comida, lo que subestima el costo real de recorrer todos los puntos de comida y provoca que se elijan caminos subóptimos al no tener en cuenta las comidas más lejanas.



## 2.8. Suboptimal Search

### Descripción

En este problema, se nos pide implementar un agente de búsqueda que encuentre un camino subóptimo para recoger todas las comidas del laberinto. El agente debe utilizar una secuencia de búsquedas para encontrar el camino más corto hacia el punto de comida más cercano y repetir este proceso hasta que todas las comidas hayan sido recogidas.

Para implementar este agente, hemos definido una clase `ClosestDotSearchAgent` que hereda de la clase `SearchAgent`. Esta clase almacena las acciones que tomará Pacman para recoger todas las comidas y utiliza una secuencia de búsquedas para encontrar el camino más corto hacia el punto de comida más cercano.

### Implementación Final

```
1 class ClosestDotSearchAgent(SearchAgent):
2     """Search for all food using a sequence of searches"""
3
4     def registerInitialState(self, state):
5         """
6         El agente registra el estado inicial y sigue buscando el punto de comida más cercano
7         hasta que todos los puntos de comida hayan sido consumidos.
8         """
9         self.actions = [] # Almacena las acciones que tomará Pacman
10        currentState = state
11
12        # Repite mientras haya puntos de comida restantes
13        while currentState.getFood().count() > 0:
14            # Encuentra el camino hacia el punto de comida más cercano
15            nextPathSegment = self.findPathToClosestDot(currentState)
16
17            # Añade estas acciones al plan del agente
18            self.actions += nextPathSegment
19
20            # Asegura que todas las acciones sean válidas
21            for action in nextPathSegment:
22                legal = currentState.getLegalActions()
23                if action not in legal:
24                    raise Exception(f'findPathToClosestDot returned an illegal move: {action}!\n{
currentState}')
25
26            # Genera el siguiente estado sucesor para Pacman
27            currentState = currentState.generateSuccessor(0, action)
28
29        self.actionIndex = 0
30        print(f'Path found with cost {len(self.actions)}.')
31
32    def findPathToClosestDot(self, gameState):
33        """
34        Devuelve una lista de acciones que llevan al punto de comida más cercano, comenzando desde
35        el estado actual del juego.
36        """
37        # Define un problema de búsqueda para cualquier alimento disponible
38        problem = AnyFoodSearchProblem(gameState)
39
40        # Utiliza el algoritmo de búsqueda (BFS en este caso) para encontrar el camino más corto
41        # hacia el punto de comida más cercano
42        from search import breadthFirstSearch
43        return breadthFirstSearch(problem)
44
45    class AnyFoodSearchProblem(PositionSearchProblem):
46        """
47        Un problema de búsqueda para encontrar un camino a cualquier punto de comida.
48        Este problema es similar a PositionSearchProblem, pero con un objetivo distinto: cualquier punto
49        de comida.
50        """
```

```

49
50 def __init__(self, gameState):
51     """Almacena la información del estado de juego para su uso posterior."""
52     # Almacena los alimentos del estado de juego actual
53     self.food = gameState.getFood()
54
55     # Llama al constructor de PositionSearchProblem
56     super().__init__(gameState)
57     self.walls = gameState.getWalls()
58     self.startState = gameState.getPacmanPosition()
59     self.costFn = lambda x: 1 # El costo de cada acción es 1
60
61     # Variables de seguimiento (no es necesario modificar)
62     self._visited, self._visitedlist, self._expanded = {}, [], 0
63
64 def isGoalState(self, state):
65     """
66     Devuelve si el estado actual (la posición de Pacman) es un estado objetivo.
67     El estado es un objetivo si contiene un punto de comida.
68     """
69     x, y = state
70     # El estado es objetivo si hay comida en la posición actual de Pacman
71     return self.food[x][y]
72

```

Código 2.12: Implementación final del problema de las esquinas

## Comentarios

En este caso, la implementación del agente `ClosestDotSearchAgent` es muy sencilla, ya que simplemente se encarga de buscar el camino más corto hacia el punto de comida más cercano y añadirlo a la lista de acciones que tomará Pacman. Para ello reciclamos la implementación del algoritmo BFS que ya teníamos implementado ya que es el algoritmo más adecuado para encontrar el camino más corto en este caso.

# 3. Resultados

## 3.1. Casos de pruebas

### 3.1.1. DFS

```
1 python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
2 [SearchAgent] using function tinyMazeSearch
3 [SearchAgent] using problem type PositionSearchProblem
4 Path found with total cost of 8 in 0.0 seconds
5 Search nodes expanded: 0
6 Pacman emerges victorious! Score: 502
7 Average Score: 502.0
8 Scores: 502.0
9 Win Rate: 1/1 (1.00)
10 Record: Win
11
12 python pacman.py -l tinyMaze -p SearchAgent
13 [SearchAgent] using function depthFirstSearch
14 [SearchAgent] using problem type PositionSearchProblem
15 Path found with total cost of 8 in 0.0 seconds
16 Search nodes expanded: 15
17 Pacman emerges victorious! Score: 502
18 Average Score: 502.0
19 Scores: 502.0
20 Win Rate: 1/1 (1.00)
21 Record: Win
22
23 python pacman.py -l mediumMaze -p SearchAgent
24 [SearchAgent] using function depthFirstSearch
25 [SearchAgent] using problem type PositionSearchProblem
26 Path found with total cost of 246 in 0.0 seconds
27 Search nodes expanded: 269
28 Pacman emerges victorious! Score: 264
29 Average Score: 264.0
30 Scores: 264.0
31 Win Rate: 1/1 (1.00)
32 Record: Win
33
34 python pacman.py -l bigMaze -z .5 -p SearchAgent
35 [SearchAgent] using function depthFirstSearch
36 [SearchAgent] using problem type PositionSearchProblem
37 Path found with total cost of 210 in 0.0 seconds
38 Search nodes expanded: 466
39 Pacman emerges victorious! Score: 300
40 Average Score: 300.0
41 Scores: 300.0
42 Win Rate: 1/1 (1.00)
43 Record: Win
44
```

Código 3.1: Prueba DFS

### 3.1.2. BFS

```
1 python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
2 [SearchAgent] using function bfs
3 [SearchAgent] using problem type PositionSearchProblem
4 Path found with total cost of 68 in 0.0 seconds
5 Search nodes expanded: 269
6 Pacman emerges victorious! Score: 442
7 Average Score: 442.0
8 Scores: 442.0
9 Win Rate: 1/1 (1.00)
10 Record: Win
11
12 python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=bfs
13 [SearchAgent] using function bfs
14 [SearchAgent] using problem type PositionSearchProblem
15 Path found with total cost of 210 in 0.0 seconds
16 Search nodes expanded: 620
17 Pacman emerges victorious! Score: 300
18 Average Score: 300.0
19 Scores: 300.0
20 Win Rate: 1/1 (1.00)
21 Record: Win
22
23 python eightpuzzle.py
24 A random puzzle:
25 -----
26 | 2 |   | 5 |
27 -----
28 | 3 | 4 | 1 |
29 -----
30 | 6 | 7 | 8 |
31 -----
32 BFS found a path of 9 moves: ['left', 'down', 'right', 'right', 'up', 'left', 'down', 'left', 'up']
33 ]
34 After 1 move: left
35 -----
36 |   | 2 | 5 |
37 -----
38 | 3 | 4 | 1 |
39 -----
40 | 6 | 7 | 8 |
41 -----
42 Press return for the next state...
43 After 2 moves: down
44 -----
45 | 3 | 2 | 5 |
46 -----
47 |   | 4 | 1 |
48 -----
49 | 6 | 7 | 8 |
50 -----
51 Press return for the next state...
52 After 3 moves: right
53 -----
54 | 3 | 2 | 5 |
55 -----
56 | 4 |   | 1 |
57 -----
58 | 6 | 7 | 8 |
59 -----
60 Press return for the next state...
61 After 4 moves: right
62 -----
63 | 3 | 2 | 5 |
64 -----
65 | 4 | 1 |   |
66 -----
67 | 6 | 7 | 8 |
68 -----
69 Press return for the next state...
```

```

69 After 5 moves: up
70 -----
71 | 3 | 2 |   |
72 -----
73 | 4 | 1 | 5 |
74 -----
75 | 6 | 7 | 8 |
76 -----
77 Press return for the next state...
78 After 6 moves: left
79 -----
80 | 3 |   | 2 |
81 -----
82 | 4 | 1 | 5 |
83 -----
84 | 6 | 7 | 8 |
85 -----
86 Press return for the next state...
87 After 7 moves: down
88 -----
89 | 3 | 1 | 2 |
90 -----
91 | 4 |   | 5 |
92 -----
93 | 6 | 7 | 8 |
94 -----
95 Press return for the next state...
96 After 8 moves: left
97 -----
98 | 3 | 1 | 2 |
99 -----
100 |   | 4 | 5 |
101 -----
102 | 6 | 7 | 8 |
103 -----
104 Press return for the next state...
105 After 9 moves: up
106 -----
107 |   | 1 | 2 |
108 -----
109 | 3 | 4 | 5 |
110 -----
111 | 6 | 7 | 8 |
112 -----
113 Press return for the next state...
114

```

Código 3.2: Prueba BFS

### 3.1.3. UCS

```
1 python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
2 [SearchAgent] using function ucs
3 [SearchAgent] using problem type PositionSearchProblem
4 Path found with total cost of 68 in 0.0 seconds
5 Search nodes expanded: 269
6 Pacman emerges victorious! Score: 442
7 Average Score: 442.0
8 Scores: 442.0
9 Win Rate: 1/1 (1.00)
10 Record: Win
11
12 python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
13 [SearchAgent] using function depthFirstSearch
14 [SearchAgent] using problem type PositionSearchProblem
15 Path found with total cost of 1.000976583804004 in 0.0 seconds
16 Search nodes expanded: 186
17 Pacman emerges victorious! Score: 646
18 Average Score: 646.0
19 Scores: 646.0
20 Win Rate: 1/1 (1.00)
21 Record: Win
22
23 python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
24 [SearchAgent] using function depthFirstSearch
25 [SearchAgent] using problem type PositionSearchProblem
26 Path found with total cost of 68719479864 in 0.0 seconds
27 Search nodes expanded: 108
28 Pacman emerges victorious! Score: 418
29 Average Score: 418.0
30 Scores: 418.0
31 Win Rate: 1/1 (1.00)
32 Record: Win
33
```

Código 3.3: Prueba UCS

### 3.1.4. A\*

```
1 python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
2 [SearchAgent] using function astar and heuristic manhattanHeuristic
3 [SearchAgent] using problem type PositionSearchProblem
4 Path found with total cost of 210 in 0.0 seconds
5 Search nodes expanded: 549
6 Pacman emerges victorious! Score: 300
7 Average Score: 300.0
8 Scores:          300.0
9 Win Rate:        1/1 (1.00)
10 Record:          Win
11
```

Código 3.4: Prueba A\*

### 3.1.5. Corners Problem: Representación

```
1 python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
2 [SearchAgent] using function bfs
3 [SearchAgent] using problem type CornersProblem
4 Path found with total cost of 28 in 0.0 seconds
5 Search nodes expanded: 435
6 Pacman emerges victorious! Score: 512
7 Average Score: 512.0
8 Scores:          512.0
9 Win Rate:        1/1 (1.00)
10 Record:         Win
11
12 python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
13 [SearchAgent] using function bfs
14 [SearchAgent] using problem type CornersProblem
15 Path found with total cost of 106 in 0.0 seconds
16 Search nodes expanded: 2448
17 Pacman emerges victorious! Score: 434
18 Average Score: 434.0
19 Scores:          434.0
20 Win Rate:        1/1 (1.00)
21 Record:         Win
22
```

Código 3.5: Prueba Representación Corners Problem



### 3.1.6. Corners Problem: Heurística

```
1 python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
2 [SearchAgent] using function depthFirstSearch
3 [SearchAgent] using problem type PositionSearchProblem
4 Path found with total cost of 106 in 0.0 seconds
5 Search nodes expanded: 901
6 Pacman emerges victorious! Score: 434
7 Average Score: 434.0
8 Scores:      434.0
9 Win Rate:    1/1 (1.00)
10 Record:     Win
11
```

Código 3.6: Prueba Heurística Corners Problem

### 3.1.7. Eating All The Dots

```
1 python pacman.py -l testSearch -p AStarFoodSearchAgent
2 [SearchAgent] using function depthFirstSearch
3 [SearchAgent] using problem type PositionSearchProblem
4 Path found with total cost of 7 in 0.0 seconds
5 Search nodes expanded: 10
6 Pacman emerges victorious! Score: 513
7 Average Score: 513.0
8 Scores:          513.0
9 Win Rate:        1/1 (1.00)
10 Record:         Win
11
12 python pacman.py -l trickySearch -p AStarFoodSearchAgent
13 [SearchAgent] using function depthFirstSearch
14 [SearchAgent] using problem type PositionSearchProblem
15 Path found with total cost of 60 in 17.1 seconds
16 Search nodes expanded: 4137
17 Pacman emerges victorious! Score: 570
18 Average Score: 570.0
19 Scores:          570.0
20 Win Rate:        1/1 (1.00)
21 Record:         Win
22
```

Código 3.7: Prueba Eating All The Dots

### 3.1.8. Suboptimal Search

```
1 python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
2 [SearchAgent] using function depthFirstSearch
3 [SearchAgent] using problem type PositionSearchProblem
4 Warning: this does not look like a regular search maze
5 Path found with cost 350.
6 Pacman emerges victorious! Score: 2360
7 Average Score: 2360.0
8 Scores:      2360.0
9 Win Rate:    1/1 (1.00)
10 Record:     Win
11
```

Código 3.8: Prueba Suboptimal Search

## 3.2. Autograder

```
1 Starting on 10-6 at 13:02:51
2
3 Question q1
4 =====
5
6 *** PASS: test_cases/q1/graph_backtrack.test
7 ***   solution:   ['1:A->C', '0:C->G']
8 ***   expanded_states: ['A', 'B', 'C']
9 *** PASS: test_cases/q1/graph_bfs_vs_dfs.test
10 ***   solution:   ['0:A->B', '0:B->D', '0:D->G']
11 ***   expanded_states: ['A', 'B', 'D']
12 *** PASS: test_cases/q1/graph_infinite.test
13 ***   solution:   ['0:A->B', '1:B->C', '1:C->G']
14 ***   expanded_states: ['A', 'B', 'C']
15 *** PASS: test_cases/q1/graph_manypaths.test
16 ***   solution:   ['0:A->B1', '0:B1->C', '0:C->D', '0:D->E1', '0:E1->F', '0:F->G']
17 ***   expanded_states: ['A', 'B1', 'C', 'D', 'E1', 'F']
18 *** PASS: test_cases/q1/pacman_1.test
19 ***   pacman layout:   mediumMaze
20 ***   solution length: 246
21 ***   nodes expanded: 269
22
23 ### Question q1: 3/3 ###
24
25
26 Question q2
27 =====
28
29 *** PASS: test_cases/q2/graph_backtrack.test
30 ***   solution:   ['1:A->C', '0:C->G']
31 ***   expanded_states: ['A', 'B', 'C', 'D']
32 *** PASS: test_cases/q2/graph_bfs_vs_dfs.test
33 ***   solution:   ['1:A->G']
34 ***   expanded_states: ['A', 'B']
35 *** PASS: test_cases/q2/graph_infinite.test
36 ***   solution:   ['0:A->B', '1:B->C', '1:C->G']
37 ***   expanded_states: ['A', 'B', 'C']
38 *** PASS: test_cases/q2/graph_manypaths.test
39 ***   solution:   ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
40 ***   expanded_states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
41 *** PASS: test_cases/q2/pacman_1.test
42 ***   pacman layout:   mediumMaze
43 ***   solution length: 68
44 ***   nodes expanded: 269
45
46 ### Question q2: 3/3 ###
47
48
49 Question q3
50 =====
51
52 *** PASS: test_cases/q3/graph_backtrack.test
53 ***   solution:   ['1:A->C', '0:C->G']
54 ***   expanded_states: ['A', 'B', 'C', 'D']
55 *** PASS: test_cases/q3/graph_bfs_vs_dfs.test
56 ***   solution:   ['1:A->G']
57 ***   expanded_states: ['A', 'B']
58 *** PASS: test_cases/q3/graph_infinite.test
59 ***   solution:   ['0:A->B', '1:B->C', '1:C->G']
60 ***   expanded_states: ['A', 'B', 'C']
61 *** PASS: test_cases/q3/graph_manypaths.test
62 ***   solution:   ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
63 ***   expanded_states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
64 *** PASS: test_cases/q3/ucs_0_graph.test
65 ***   solution:   ['Right', 'Down', 'Down']
66 ***   expanded_states: ['A', 'B', 'D', 'C', 'G']
67 *** PASS: test_cases/q3/ucs_1_problemC.test
68 ***   pacman layout:   mediumMaze
```

```

69 *** solution length: 68
70 *** nodes expanded: 269
71 *** PASS: test_cases/q3/ucs_2_problemE.test
72 *** pacman layout: mediumMaze
73 *** solution length: 74
74 *** nodes expanded: 260
75 *** PASS: test_cases/q3/ucs_3_problemW.test
76 *** pacman layout: mediumMaze
77 *** solution length: 152
78 *** nodes expanded: 173
79 *** PASS: test_cases/q3/ucs_4_testSearch.test
80 *** pacman layout: testSearch
81 *** solution length: 7
82 *** nodes expanded: 14
83 *** PASS: test_cases/q3/ucs_5_goalAtDequeue.test
84 *** solution: ['1:A->B', '0:B->C', '0:C->G']
85 *** expanded_states: ['A', 'B', 'C']
86
87 ### Question q3: 3/3 ###
88
89
90 Question q4
91 =====
92
93 *** PASS: test_cases/q4/astar_0.test
94 *** solution: ['Right', 'Down', 'Down']
95 *** expanded_states: ['A', 'B', 'D', 'C', 'G']
96 *** PASS: test_cases/q4/astar_1_graph_heuristic.test
97 *** solution: ['0', '0', '2']
98 *** expanded_states: ['S', 'A', 'D', 'C']
99 *** PASS: test_cases/q4/astar_2_manhattan.test
100 *** pacman layout: mediumMaze
101 *** solution length: 68
102 *** nodes expanded: 221
103 *** PASS: test_cases/q4/astar_3_goalAtDequeue.test
104 *** solution: ['1:A->B', '0:B->C', '0:C->G']
105 *** expanded_states: ['A', 'B', 'C']
106 *** PASS: test_cases/q4/graph_backtrack.test
107 *** solution: ['1:A->C', '0:C->G']
108 *** expanded_states: ['A', 'B', 'C', 'D']
109 *** PASS: test_cases/q4/graph_manypaths.test
110 *** solution: ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
111 *** expanded_states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
112
113 ### Question q4: 3/3 ###
114
115
116 Question q5
117 =====
118
119 *** PASS: test_cases/q5/corner_tiny_corner.test
120 *** pacman layout: tinyCorner
121 *** solution length: 28
122
123 ### Question q5: 3/3 ###
124
125
126 Question q6
127 =====
128
129 *** PASS: heuristic value less than true cost at start state
130 *** PASS: heuristic value less than true cost at start state
131 *** PASS: heuristic value less than true cost at start state
132 path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West',
'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'West', 'West',
', 'West', 'South', 'South', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South',
', 'South', 'South', 'West', 'West', 'South', 'South', 'South', 'West', 'West', 'East', 'East',
', 'North', 'North', 'North', 'East', 'East', 'East', 'East', 'East', 'East', 'East', 'East',
', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East',
', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'East', 'South',
', 'South', 'South', 'South', 'East', 'East', 'North', 'North', 'East', 'East', 'South', 'South',

```



```

202 *** PASS: test_cases/q8/closest_dot_2.test
203 ***   pacman layout:   Test 2
204 ***   solution length:   1
205 [SearchAgent] using function depthFirstSearch
206 [SearchAgent] using problem type PositionSearchProblem
207 Warning: this does not look like a regular search maze
208 *** PASS: test_cases/q8/closest_dot_3.test
209 ***   pacman layout:   Test 3
210 ***   solution length:   1
211 [SearchAgent] using function depthFirstSearch
212 [SearchAgent] using problem type PositionSearchProblem
213 Warning: this does not look like a regular search maze
214 *** PASS: test_cases/q8/closest_dot_4.test
215 ***   pacman layout:   Test 4
216 ***   solution length:   3
217 [SearchAgent] using function depthFirstSearch
218 [SearchAgent] using problem type PositionSearchProblem
219 Warning: this does not look like a regular search maze
220 *** PASS: test_cases/q8/closest_dot_5.test
221 ***   pacman layout:   Test 5
222 ***   solution length:   1
223 [SearchAgent] using function depthFirstSearch
224 [SearchAgent] using problem type PositionSearchProblem
225 Warning: this does not look like a regular search maze
226 *** PASS: test_cases/q8/closest_dot_6.test
227 ***   pacman layout:   Test 6
228 ***   solution length:   2
229 [SearchAgent] using function depthFirstSearch
230 [SearchAgent] using problem type PositionSearchProblem
231 Warning: this does not look like a regular search maze
232 *** PASS: test_cases/q8/closest_dot_7.test
233 ***   pacman layout:   Test 7
234 ***   solution length:   1
235 [SearchAgent] using function depthFirstSearch
236 [SearchAgent] using problem type PositionSearchProblem
237 Warning: this does not look like a regular search maze
238 *** PASS: test_cases/q8/closest_dot_8.test
239 ***   pacman layout:   Test 8
240 ***   solution length:   1
241 [SearchAgent] using function depthFirstSearch
242 [SearchAgent] using problem type PositionSearchProblem
243 Warning: this does not look like a regular search maze
244 *** PASS: test_cases/q8/closest_dot_9.test
245 ***   pacman layout:   Test 9
246 ***   solution length:   1
247
248 ### Question q8: 3/3 ###
249
250
251 Finished at 13:03:06
252
253 Provisional grades
254 =====
255 Question q1: 3/3
256 Question q2: 3/3
257 Question q3: 3/3
258 Question q4: 3/3
259 Question q5: 3/3
260 Question q6: 3/3
261 Question q7: 5/4
262 Question q8: 3/3
263 -----
264 Total: 26/25
265
266 Your grades are NOT yet registered. To register your grades, make sure
267 to follow your instructor's guidelines to receive credit on your project.
268

```