GLOBAL CAMPUS
NEBRIJA

Practica 1 Inteligencia Artificial

José María Fernádez Gómez

Pregunta 1 (3 puntos): Encontrar un punto fijo de comida mediante la primera búsqueda de profundidad:

----------CODIGO----------

```
actual = problem.getStartState() #Definimos el primer nodo raiz
visited = [] #Array que contendra los nodos ya visitados
stripe = util.Stack() #Defino la pila que contendrá todos los nodos de la franja actual, estos nodos son los que iremos desplegando en cada iteracion
stripe.push((actual, [])) #Incluyo el nodo raiz en la pila

while not stripe.isEmpty(): #Mientras que la franja de trabajo no este vacia
        actual, path = stripe.pop()
        visited.append(actual)#Marco como visitado el nodo actual

        if problem.isGoalState(actual): #Si el nodo actual es el estado meta
        return path# Terminamos y devuelvo el camino que hemos obtenido

        for succ in problem.getSuccessors(actual): #Desplegamos el nodo anterior y miramos sus sucesores
                if succ[0] not in visited: #Comprobamos que el nodo mas a la izquierda no este visitado
                        stripe.push((succ[0], path + [succ[1]])) #Si no está visitado lo añadimos a la franja de trabajo e incluimos el siguiente en el camino
util.raiseNotDefined()
```

----------END CODIGO----------

Al ejecutar “python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch” el programa coje por predeterminado el algoritmo de busqueda en profundidad y resuelve el problema siempre yendo por la rama mas a la izquierda posible hasta llegar al final.

Pregunta 2 (3 puntos): Primera búsqueda de amplitud

----------CODIGO----------

```
actual=problem.getStartState()#Definimos el primer nodo raiz
    stripe = util.Queue()#Definimos una pila que contendrá los nodos de la franaj que se irán
                                desplegando
    visited = []#Array con estados ya visitados
    path = []#Array con el camino que se ha ido siguiendo
    stripe.push((actual, []))#Encolamos el estado actual(inicial)
    visited.append(actual)#Marcamos como visitado el nodo raiz

    while not stripe.isEmpty():#Mientras que franja de trabajo no este vacia
        actual, path = stripe.pop()

        if problem.isGoalState(actual):#Si el nodo actual es el nodo meta
            return path #Terminamos y devolvemos el camino hasta l nodo actual

        else:

            for succ, direction,cost in problem.getSuccessors(actual):#Desplegamos los sucesores de mi
                                                                        nodo acatual

                if not succ in visited:#Para sucesor no visitado(En este caso miramos todos los sucesores y
                                            nos solo el de mas a alq izquierda)
                    visited.append(succ)#Marcamos cada sucesor como visitado
                    stripe.push((succ, path + [direction]))#Lo incluimos en la franja junto al camino previo y la
                                                            direccion que hemos tomado
```

----------END CODIGO----------

Al ejecutar “python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs” le indicamos al programa que queremos resolver el mismo problema que el ejercicio anterior pero en esta ocasión empleando el algoritmo de busqueda en anchura, esto quiere deccir que en vez de ir rama por rama hasta el final, empezará a ir por niveles, completando la busqueda en cada nivel antes de pasar al siguiente .

Si comparamos los resultados entre estos dos algoritmos vemos que el primero expande menos nodos que el segundo, por lo que es más eficiente para este problema y este laberinto en concreto

Pregunta 3 (3 puntos): Variación de la función de coste

----------CODIGO----------

```
stripe = util.PriorityQueue() #DEfinimos una cola con prioridad como nuestra estructura de datos
    stripe.push( (problem.getStartState(), [], 0), 0 ) #Introducimos en la cola el nodo inicial, las
                                                       #direcciones a seguir y el coste
                                                       #inicial como primer argumento, y la prioridad 0
                                                       #como segundo
    visited = [] #Declaramos un array que contendrá los nodos visitados

    while not stripe.isEmpty(): # Mientras que la franja de trabajo no esté vacia
        node, actions, curCost = stripe.pop() # Asignaremos el primer elemento de la fila a las variables
                                              #node actions y coste

        if(not node in visited): #Si el primer nodo de la cola no esta visitadp
            visited.append(node) #Lo marcamos como visitado

            if problem.isGoalState(node): #Comprobamos si hemos encontrado el nodo destino
                return actions # Si es asi devolvemos las acciones que hemos tomado hasta encontrar ese
                               #nodo

            for child, direction, cost in problem.getSuccessors(node): #Si no es el destino expandimos el
                                                                       #nodo y sus sucesores
                stripe.push((child, actions+[direction], curCost + cost), curCost + cost) #Metemos en la
                                                                                          #franja de trabajo
                                                                                          #cada sucesor, las
                                                                                          #acciones
                                                                    #previas y la nueva que hay que tomar,el
                                                                     #coste acumulado,
                                                                    #y por ultimo le indicamos la prioridad con el
                                                                     #coste acumulado,
                                                                    # asi siempre tomara el camino con menos
                                                                     #coste
```

----------END CODIGO----------

Al ejecutar “python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs” el programa resuelve el problema a través del algorítmo de busqueda por coste uniforme, este se consigue por medio de una cola de prioridad, ya que esta cola sacará primero el argumento con la prioridad mas baja, por eso al comienzo le asignamos 0 como prioridad al nodo raiz, para forzar que siempre empiece por ahi.
Este algoritmo tambien guarda en la cola las direcciones tomadas y el costo acumulado total hasta llegar al nodo en cuestión.

Pregunta 4 (3 puntos): Búsqueda A*

----------CODIGO----------

```
stripe = util.PriorityQueue() #Definimos una cola con prioridad como estructura de datos
    stripe.push((problem.getStartState(),[],0),heuristic(problem.getStartState(), problem)) #Metemos el
            #nodo raiz con el conjunto vacio de acciones tomadas y coste 0 por ser el primero en la cola
    visited = [] #Declaramos un array vacio que sera el conjunto de nodos visitados

    while not stripe.isEmpty(): #Mientras que la franja de trabajo no este vacia
        node, actions, accCost = stripe.pop()  #Sacamos el nodo actual,las acciones tomadas hasta el
                                               #momento y el coste accumulado total

        if(not node in visited): #Si no esta visitado ya el nodo
            visited.append(node) #LO marcamos como visitado

            if problem.isGoalState(node): #Si es el nodo destino
                return actions #Devolvemos las acciones tomadas hasta ese punto

            for child, direction, cost in problem.getSuccessors(node): #Expandimos el nodo actual y sus
                                                                       #sucesores
                g = accCost + cost #Calculamos el coste real hasta ese punto
                stripe.push((child, actions+[direction], accCost + cost), g + heuristic(child, problem))
                #Metemos lo mismo que en el caso del ucs pero sumandole la
                #heurisitca para ver que camino es mejor
```

----------END CODIGO----------

Al ejecutar “python pacman.py -l bigMaze -z .5 -p SearchAgent -afn=astar, heuristic = manhattanHeuristic” El programa resuelve el problema de una manera similar a como lo hacia en el UCS con la unica diferencia de que aquí le estamos sumando al coste acumulado total el coste que hemos definido en la heuristica, por lo que la prioridad de cada nodo se verá afectada por este dato y si la heurisitica es acertada y optimista sera mejor que el UCS. Con el caso de la heurisitica de Manhattan en este laberinto funciona bien y nos da un rendimiento mayor que el UCS

¿Qué sucede en openMaze para las diversas estrategias de búsqueda?

DFS: Expande muchisimos nodos y el algoritmo toma un camino muy largo

BFS: El algoritmo funciona bien pero expande muchsimos nodos, como el UCS

UCS: El algoritmo se comporta bien pero expande muchisimos nodos para encontrar la solucion

ASTAR: El algoritmo se comporta bien y expande muchos nodos menos que UCS

Pregunta 6 (3 puntos): Problemas de esquinas: Heurística

----------CODIGO----------

```
corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
    "*** YOUR CODE HERE ***"
    node = state[0] #Nodo actual
    visitedCorners = state[1]
    unvisitedCorners = [] #Array con los nodos que aun no he visitado
    sum = 0 # Coste en distancia  acumulado hasta llegar a una esquina
    for corner in corners: # Para cada esquina
        if not corner in visitedCorners: # Si no he visitado esa esquina
            unvisitedCorners.append(corner) #La meto en el array de nos visitados
    currentPoint = node #Mi nodo actual es el primer nodo de mi estructura de datos
    while len(unvisitedCorners) > 0: # Mientras que haya alguna esquina que no he visitado
        distance, corner = min([(util.manhattanDistance(currentPoint, corner), corner) for corner in
unvisitedCorners]) #Me quedo con la esquina y la distancia que sea minima para
                                                                                    #todas las esquinas que
                                                                                    #no he visitado
        sum += distance #Actualizo la distancia total recorrida con la distancia a la esquina mas cercana
        currentPoint = corner #Me voy a mover hasta la esquina mas cercana
        unvisitedCorners.remove(corner) # Como me he movido a la esquina mas cercana la saco de las
                                        #no visitadas
    return sum # La heurisitca a devolver es la distancia total a la esquina mas cercana
```

----------END CODIGO----------

En esta heurisitica se devuelve siempre la distancia minima a la esquina mas cercana, por lo que siempre el valor va a ser menor o igual al costo real, asi que es una heuristica optimista por lo tanto una heuristica aceptable

Pregunta 7 (4 puntos): Comer todos los puntos

No se ha realizado, no he conseguido hacer que funcione ni entender como tengo que hacerlo

Pregunta 8 (3 puntos): Búsqueda subóptima

----------CODIGO----------

----------findPathToClosestDot----------

```
def findPathToClosestDot(self, gameState):
    """
    Returns a path (a list of actions) to the closest dot, starting from
    gameState.
    """
    # Here are some useful elements of the startState
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)

    "*** YOUR CODE HERE ***"
    actions = search.ucs(problem) #Busco el punto mas cercano con el algoritmos a estrella
    return actions# Devuelvo las acciones encontradas para cada punto encontradas arriba
```

----------findPathToClosestDot----------

----------isGoalState----------

```
"""
    The state is Pacman's position. Fill this in with a goal test that will
    complete the problem definition.
    """
    x,y = state

    "*** YOUR CODE HERE ***"
    foodList = self.food.asList() #Declaro todos los puntos de comida que hay
    distance, food = min([(util.manhattanDistance(state, food), food) for food in foodList])
        #La distancia y el punto de comida a buscar es el que tenga menor distancia segun la
        #heuristica de  manhhatan
    isGoal = state == food #Le deciomos que el objetivo es el estado actual si este tiene un punto de
                           #comida
    return isGoal
```

----------isGoalState----------

----------END CODIGO----------

Si ejecutamos “python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5” El programa resolverá el proble de buscar el punto de comida mas cercano pero usando el algoritmo de busqueda UCS que no es optimo como el a estrella pero se ejecuta mucho mas rapido ya que tiene que hacer muchos menos calculos