Estructura de la memoria

Título Inteligencia Artificial. Práctica 1: Búsqueda

Autores Luis Miguel Nucifora y Bernardo Andrés Zambrano

Sección 1 (1 punto)

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

Para realizar la búsqueda en busqueda en profundidad planteamos la opción de utilizar la estructura de datos de la pila, la cual utilizariamos para guardar los nodos sucesores del nodo que estuviesemos explorando y de esta forma expandir siempre el nodo introducido más recientementente introducido en la lista de nodos.

1.1.1 Lista & explicación de las funciones del framework usadas

Utilizamos la clase *Stack* del archivo utils junto a las funciones de *push* y *pop* para controlar la expansión y revisión de los estados.
También hemos usado la función *getStartState* para guardarlo en una variable, *isGoalState* para comprobar si se ha llegado a la meta al explorar un nodo y *getSuccessors* para introducirlos en la pila.

1.1.2 Incluye el código añadido

```
start_state = problem.getStartState()
start_node = (start_state, [])

open_list = util.Stack()

visited_list = []

open_list.push(start_node)

while not open_list.isEmpty():
    current_state, moves = open_list.pop()
    if current_state not in visited_list:
        visited_list.append(current_state)

        if problem.isGoalState(current_state):
            return moves
        else :
            successors = problem.getSuccessors(current_state)

        for successor_state, successor_move, successor_cost in successors:
            new_moves = moves + [successor_move]
            new_node = (successor_state, new_moves)
            open_list.push(new_node)

empty_moves = []
return empty_moves
```

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

```
shadeeon@Shadeeon:~/Universidad/ia/p1/search$ python3 pacman.py -l tinyMaze -p S
earchAgent
[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
```

```
shadeeon@Shadeeon:~/Universidad/ia/p1/search$ python3 pacman.py -l mediumMaze -p
 SearchAgent
[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
```

```
shadeeon@Shadeeon:~/Universidad/ia/p1/search$ python3 pacman.py -l bigMaze -p Se
archAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 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
```

1.2 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 de pacman no es en absoluto óptimo. En las capturas podemos observar que si llega a la solución para todos los casos propuestos.

1.2.1 Respuesta a pregunta 1.1

El orden de exploración es el que esperábamos para la utilización de este algoritmo.

1.2.2 Respuesta a pregunta 1.2

Pacman no va a todas las casillas exploradas, ya que en ocasiones se exploran caminos sin salida y estos no los explora.

1.2.3 Respuesta a pregunta 2

La búsqueda de profundidad no es la búsqueda de menor coste, ya que esta al usarse puede encontrar una solución que esté a una gran profundidad antes de explorar otra rama que llegue a la solución con menos pasos.

Sección 2 (1 punto)

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

Para realizar la búsqueda en busqueda en profundidad planteamos la opción de utilizar la estructura de datos de la cola, la cual utilizariamos para guardar los nodos sucesores del nodo que estuviesemos explorando y de esta forma expandir siempre el nodo que fue introducido primero en la lista de nodos y explorar así todas las ramas.

2.1.1 Lista & explicación de las funciones del framework usadas

Utilizamos la clase *Queue* del archivo utils junto a las funciones de *push* y *pop* para controlar la expansión y revisión de los estados.
También hemos usado la función *getStartState* para guardarlo en una variable, *isGoalState* para comprobar si se ha llegado a la meta al explorar un nodo y *getSuccessors* para introducirlos en la cola.

2.1.2 Incluye el código añadido

```python
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    start_state = problem.getStartState()
    start_node = (start_state, [])

    open_list = util.Queue()

    visited_list = []

    open_list.push(start_node)

    while not open_list.isEmpty():
        current_state, moves = open_list.pop()
        if current_state not in visited_list:
            visited_list.append(current_state)

            if problem.isGoalState(current_state):
                return moves
            else :
                successors = problem.getSuccessors(current_state)

            for successor_state, successor_move, successor_cost in successors:
                new_moves = moves + [successor_move]
                new_node = (successor_state, new_moves)
                open_list.push(new_node)
    empty_moves = []
    return empty_moves
```

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

```
shadeeon@Shadeeon:~/Universidad/ia/p1/search$ python3 pacman.py -l tinyMaze -p S
earchAgent -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
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

2.2 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 de pacman usando la búsqueda en anchura es óptimo, pues siempre va a encontrar la solución que está a la distancia mínima. Sin embargo expande muchos más nodos que la búsqueda en profundidad.

```
shadeeon@Shadeeon:~/Universidad/ia/p1/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
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
shadeeon@Shadeeon:~/Universidad/ia/p1/search$ python3 pacman.py -l bigMaze -p Se
archAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 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
```

2.2.1 Respuesta a pregunta 3

BA encuentra siempre una solución de menor coste al ser óptimo.

Sección 3 (1 punto)

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

Para realizar la búsqueda en busqueda de coste uniforme planteamos la opción de utilizar la estructura de datos de la cola de prioridad, la cual utilizariamos para guardar los nodos sucesores del nodo que estuviesemos explorando.

A diferencia del apartado anterior en este caso debemos considerar el coste de las acciones a la hora de introducir los nodos en la cola. Se explorarán primero pues las ramas con acciones de menor coste.

3.1.1 Lista & explicación de las funciones del framework usadas

Utilizamos la clase *PriorityQueue* del archivo utils junto a las funciones de *push* y *pop* para controlar la expansión y revisión de los estados.

También hemos usado la función *getStartState* para guardarlo en una variable, *isGoalState* para comprobar si se ha llegado a la meta al explorar un nodo y *getSuccessors* para introducirlos en la cola.

3.1.2 Incluye el código añadido

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    start_state = problem.getStartState()
    start_node = (start_state, [], 0)

    open_list = util.PriorityQueue()

    visited_list = []

    open_list.push(start_node, 0)

    while not open_list.isEmpty():
        current_state, moves, cost= open_list.pop()
        if current_state not in visited_list:
            visited_list.append(current_state)

            if problem.isGoalState(current_state):
                return moves
            else :
                successors = problem.getSuccessors(current_state)

            for successor_state, successor_move, successor_cost in successors:
                new_moves = moves + [successor_move]
                new_cost = cost + successor_cost
                new_node = (successor_state, new_moves, new_cost)
                open_list.push(new_node, new_cost)
    empty_moves = []
    return empty_moves
```

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

```
shadeeon@Shadeeon:~/Universidad/ia/p1/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
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
shadeeon@Shadeeon:~/Universidad/ia/p1/search$ python3 pacman.py -l mediumDottedM
aze -p StayEastSearchAgent
Path found with total cost of 1 in 0.0 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
```

```
shadeeon@Shadeeon:~/Universidad/ia/p1/search$ python3 pacman.py -l mediumScaryMa
ze -p StayWestSearchAgent
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
```

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)

Podemos observar que en todos los casos probados se llega a la solución. Se expanden menos nodos que cuando usamos BA. Este algoritmo no tiene por que llegar siempre a la meta en el menor número de pasos ya que busca las rutas con menor coste.

Sección 4 (2 puntos)

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

Para realizar la búsqueda a estrella planteamos la opción de utilizar la estructura de datos de la cola de prioridad, la cual utilizariamos para guardar los nodos sucesores del nodo que estuviesemos explorando.

En este apartado debemos considerar tanto el coste de las acciones como el resultado de la función heurística que utilicemos, en este caso la de la distancia de Manhattan. Combinaremos estos aspectos para calcular la función de coste a utilizar.

4.1.1 Lista & explicación de las funciones del framework usadas

Utilizamos la clase *PriorityQueue* del archivo utils junto a las funciones de *push* y *pop* para controlar la expansión y revisión de los estados.

También hemos usado la función *getStartState* para guardarlo en una variable, *isGoalState* para comprobar si se ha llegado a la meta al explorar un nodo, *getCostOfActions* para calcular el coste de las acciones del algoritmo, heuristic para calcular el valor de la función heurística en función de la posición y getSuccessors para introducirlos en la cola.

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 ***"
    open_list = util.PriorityQueue()

    visited_list = []

    start_state = problem.getStartState()
    start_node= (start_state, [], 0)

    open_list.push(start_node, 0)

    while not open_list.isEmpty():
        current_state, moves, current_cost = open_list.pop()

        current_node = (current_state, current_cost)
        visited_list.append(current_node)

        if problem.isGoalState(current_state):
            return moves
        else :
            successors = problem.getSuccessors(current_state)

            for successor_state, successor_moves, successor_cost in successors:
                new_moves = moves + [successor_moves]
                new_cost = problem.getCostOfActions(new_moves)
                new_node = (successor_state, new_moves, new_cost)

                visited_flag = False
                for visited in visited_list:
                    visited_state, visited_cost = visited

                    if (successor_state == visited_state) and (new_cost >= visited_cost):
                        visited_flag = True
                if not visited_flag:
                    open_list.push(new_node, new_cost + heuristic(successor_state, problem))
                    visited_list.append((successor_state, new_cost))

    return []
```

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

```
shadeeon@Shadeeon:~/Universidad/ia/p1/search$ python3 pacman.py -l bigMaze -z .5
 -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 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
```

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

Podemos comprobar que llega a la solución en el caso propuesto y su coste es menor que el de la búsqueda de coste uniforme así como los nodos expandidos, demostrando así que la heurística ayuda a pacman a ser más eficiente en su recorrido.

4.2.1 Respuesta a pregunta 4

Al ejecutar openMaze el coste de la búsqueda de coste uniforme y de A* es el mismo, pero este último expande menos nodos.

Sección 5 (2 puntos)

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

El objetivo principal a la hora de hacer el ejercicio era conseguir que se visitasen todas las esquinas utilizando el algoritmo A* anteriormente implementado. Hemos buscado también como implementar la búsqueda de sucesores de manera correcta.

5.1.1 Lista & explicación de las funciones del framework usadas

Se han usado las funciones *getWalls* y getPacmanPosition con el objetivo de identificar la posición y obstáculos de Pacman.

Y Usamos la función *directionToVector* para saber en que dirección nos estamos moviendo.

5.1.2 Incluye el código añadido

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor function
    """

    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 expande
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"
        self.startingGameState = startingGameState
```

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"
    return self.startingPosition, []

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    "*** YOUR CODE HERE ***"
    position = state[0]
    corners_visited = state[1]
    print(str(corners_visited))

    print(self.startingGameState)

    # comprobamos si el estado actual es unna de las esquinas
    if position in self.corners:
        #si la esuqina no esta visitda actualizamos las esquinas visitadas
        if not position in corners_visited:
            corners_visited.append(position)
        # devolveremos true cuando todas las esquinas han sido visitadas
        return len(corners_visited) == len(self.corners)
    #Como para estar en un estado final debes estar en alguna esquina se devuelve false
    return False
```

```
        #comprobamos si este movimiento se puede hacer
        hitsWall = self.walls[nextx][nexty]
        if not hitsWall:
            #obtenemos la lista de esquinas visitadas antes del movimiento
            successors_corners_visited  = list(corners_visited)
            #obtenemos la nueva posicion
            next_state = (nextx, nexty)

            #comprobamos si la nueva coordenadas es una esquina
            if next_state in self.corners:
                #si es una esquina comprobamos si ha sido visitada
                if not next_state in successors_corners_visited:
                    #si no ha sida visitada actualizamos las esquinas visitadas
                    successors_corners_visited.append(next_state)

            cost = 1
            successor = (
                (next_state, successors_corners_visited), action, cost)
            successors.append(successor)

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

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

     As noted in search.py:
        For a given state, this should return a list of triples, (successor,
        action, stepCost), where 'successor' is a successor to the current
        state, 'action' is the action required to get there, and 'stepCost'
        is the incremental cost of expanding to that successor
    """

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

        "*** YOUR CODE HERE ***"
        #obtenemos las cordenadas actuales
        x, y = state[0]
        #obtenemos las esquinas que han sido visitadas
        corners_visited = state[1]
        #obtenemos la direccion en la que nos movemos en esta accion
        dx, dy = Actions.directionToVector(action)
        #obtenemos las nuevas coordenadas del movimiento
        nextx, nexty = int(x + dx), int(y +dy)
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)
```

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

```
Path found with total cost of 28 in 0.1 seconds
Search nodes expanded: 435
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
Path found with total cost of 106 in 1.2 seconds
Search nodes expanded: 2448
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

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

Pacman llega a la solución en ambos casos, sin embargo necesita expandir demasiados nodos para llegar a la solución, esto de cara a problemas más complejos nos daría bastantes complicaciones por lo que se hace necesario recurrir a alguna heurística.

Sección 6 (3 puntos)

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

Hemos decidido comenzar basándonos en la heurística de la distancia de Manhattan para resolver este problema, sin embargo nos topamos con el problema de la existencia de los muros, lo que hacía que la heurística no reducía demasiado el número de nodos expandidos. De esta forma nos enfocamos en darle más precisión al problema.

Debido a esto optamos por utilizar la función de MazeDistance para tener más precisión.

6.1.1 Lista & explicación de las funciones del framework usadas

Usamos la función *mazeDistance* para calcular la distancia a las esquinas usando el algoritmo de búsqueda en profundidad y la función *isGoalState* para comprobar si estamos ya en la meta.

6.1.2 Incluye el código añadido

```python
"*** YOUR CODE HERE ***"
gameState = problem.startingGameState
pos = state[0]
visited = state[1]
heuristic = 0

# Comprobamos las esquinas que faltan por visitar
remaining = [x for x in corners if x not in visited]

dists = []

if len(remaining) > 0:
    for c in remaining:

        # Calculamos la distancia minima entre nuestra posicion y la esquina
        # hacemos esto con todas las esquinas restantes
        dis = mazeDistance(pos, c, gameState)
        dists.append(dis)

# Elegimos el valor maximo entre las distancias a las esquinas
# como valor de la funcion heuristica
if(len(dists) > 0):
    heuristic = max(dists)

if problem.isGoalState(state):
    return 0

return heuristic  # Default to trivial solution
```

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

```
Path found with total cost of 106 in 5.2 seconds
Search nodes expanded: 980
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

resultados

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)

Pacman llega a la solución expandiendo en este caso menos de 1000 nodos. Lo cual es una gran mejoría respecto al apartado anterios, en el cual se expandían más del doble de nodos.

6.2.1 Respuesta a pregunta 5: heurística

Lo primero que hace nuestra heurística es comprobar las esquinas que nos quedan por visitar. Para cada una de ellas calculamos la distancia a la que nos encontramos de ella usando la función mazeDistance y las guardamos en una lista de distancias. Por último elegimos el valor más alto de esa lista y utilizamos ese valor como retorno de la función heurística.

Sección 7

Comentarios personales de la realización de esta práctica

Al ser los cuatro primeros ejercicios bastante similares en el apartado de código nos hemos centrado en realizar correctamente el primer apartado para después modificarlo de cara a realizar el resto de apartados.

Utilizando los conocimientos que hemos obtenido de los apartados anteriores hemos podido realizar el problema de las esquinas.

Para la heurística hemos partido de la distancia de Manhattan y hemos ido acotando desde ahí hasta llegar a la que hemos obtenido.

Nota de la memoria (40% de la práctica)