# MEMORIA P1: Búsqueda

Daniel Aquino Santiago

Jorge Paniagua Moreno

Grupo 1312

Pareja 02

## EJERCICIO 1

## DFS ALGORITHM

```
def depthFirstSearch(search_problem):
    stack = util.Stack()
    path = []
    visited = []
    state = search_problem.getStartState()
    visited.append(state)

    parent = {}
    from_dir = {}
    parent[state] = state
    from_dir[state] = None

    while not search_problem.isGoalState(state):
        for successor in search_problem.getSuccessors(state):

            son_st = successor[0]
            if son_st not in visited:
                parent[son_st] = state
                from_dir[son_st] = successor[1]
                stack.push(son_st)

        while state in visited:
            if stack.isEmpty():
                return None

            state = stack.pop()
        visited.append(state)
    while parent[state] != state:
        path.insert(0, from_dir[state])
        state = parent[state]

    return path
```

### 1- Inicialización de variables:

Se crea una pila que se utilizara para mantener los estados a explorar.

Se inicializan listas ‘path’ y ‘visitited’ para realizar un seguimiento del camino y los estados visitados.

Se obtiene el estado inicial del problema de búsqueda y se agrega a la lista ‘visited’.

Se inicializan los diccionarios ‘parent’ y ’from_dir’ para llevar un registro de los padres y de las direcciones desde las cuales se llegó a cada estado.

## 2 - Bucle Principal

Mientras el estado actual no sea el estado objetivo se realizarán las siguientes acciones:

Se recorren los sucesores del estado actual ‘successor’.

Para cada sucesor, se verifica si ya ha sido visitado, se actualizan los diccionarios con la información del sucesor y se agrega el sucesor a la pila.

Luego se sale del bucle cuando se alcanza el estado objetivo o no hay más estados en la pila que explorar.

## 3 - Reconstruir el camino

Se utiliza un bucle para reconstruir el camino desde el estado objetivo hasta el estado inicial utilizando la información almacenada en los diccionarios. El camino se almacena en la lista ‘path’.

## En Resumen:

Esta función implementa un algoritmo de búsqueda que comienza desde un estado inicial y explora sucesivamente los sucesores hasta encontrar el estado objetivo, luego reconstruye el camino desde el estado objetivo hasta el estado inicial y lo devuelve como una lista de direcciones. Si no se encuentra un camino se devuelve None.

Ejecución python pacman.py -l tinyMaze -p SearchAgent

```
[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
```

Se ha encontrado un camino con un coste de 10 unidades, se expandieron un total de 15 nodos durante la búsqueda.

Ejecución python 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
```

Se ha encontrado un camino con un coste de 130 unidades, se expandieron un total de 146 nodos durante la búsqueda.

Ejecución python pacman.py -l bigMaze -z .5 -p SearchAgent

```
[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
```

Se ha encontrado el resultado para un camino con un coste de 210 unidades, se expandieron un total de 390 nodos durante la búsqueda.

Pregunta 1.1 ¿El orden de exploración es el que esperabais? ¿Pacman realmente va a todas las casillas exploradas en su camino hacia la meta?

Si es el que esperábamos, ya que Pacman por cada nivel accede hasta el estado más profundo posible, realmente no va a todas las casillas exploradas, ya que algunas de ellas no conllevan hacia la meta o no forman parte de casillas con solución o no sea adecuada para la correcta funcionalidad del algoritmo.

Pregunta 1.2 ¿Es esta la función de menor coste? Si no es así, pensad qué está haciendo mal la búsqueda en profundidad.

No este algoritmo no asegura encontrar la función de menor coste, solo de encontrar una solución al problema, para ello deberíamos considerar algoritmos de búsqueda informada como A* o búsqueda de costo uniforme, los cuales están diseñados específicamente para encontrar una solución óptima en término de coste, el cual es el tema a tratar en la actual pregunta.

EJERCICIO 2

BFS ALGORITHM

```python
def breadthFirstSearch(search_problem):
    queue = util.Queue()
    path = []
    visited = set()
    state = search_problem.getStartState()
    parent = {}
    from_dir = {}
    parent[state] = state
    from_dir[state] = None
    queue.push((state, path))
    while not queue.isEmpty():
        state, path = queue.pop()
        if search_problem.isGoalState(state):
            return path
        if state not in visited:
            visited.add(state)
            for successor in search_problem.getSuccessors(state):
                son_st = successor[0]
                if son_st not in visited:
                    parent[son_st] = state
                    from_dir[son_st] = successor[1]
                    queue.push((son_st, path + [successor[1]]))
    return None
```

## 1- Inicialización de variables

Se crea una cola que se utilizara para mantener estados explorados, la cola funciona en un principio FIFO.

Se inicializará una lista ‘path’ vacía y un conjunto ‘visited’ vacío para llevar un registro de los estados visitados.

Se obtiene el estado inicial del problema de búsqueda y se agrega a la cola justo con su correspondiente path.

Se inicializan los diccionarios para llevar un registro de los padres y las direcciones desde las cuales se llegó a cada estado.

## 2 - Bucle Principal

Mientras la cola no esté vacía, se realizarán las siguientes acciones:

Se extrae el estado actual y el camino asociado de frente de la cola.

Se verifica si el estado actual es el objetivo, si es el objetivo se devuelve el camino encontrado.

Si el estado actual no ha sido visitado previamente, se marca como visitado y se recorren los sucesores del estado actual.

Para cada sucesor que no ha sido visitado se actualizan los diccionarios con la información del sucesor, y se agrega el sucesor a la cola junto con su camino actualizado.

## 3 - Devolución de resultados

Si se ha completado la exploración de todos los estados posibles sin encontrar el objetivo, se devuelve None, lo que indica que no se ha encontrado un camino desde el estado inicial al estado objetivo.

## En resumen:

Esta función implementa un algoritmo de búsqueda en anchura utilizando una cola para explorar los estados de manera exhaustiva. Encuentra un camino desde el estado inicial hasta el estado objetivo y lo devuelve como una lista de direcciones. Si no se encuentra un camino, devuelve None.

Ejecución python pacman.py -l tinyMaze -p SearchAgent

```
[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
```

Se ha encontrado el resultado para un camino con un coste de 10 unidades, se expandieron un total de 15 nodos durante la búsqueda.

Ejecución python 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
```

Se ha encontrado el resultado para un camino con un coste de 68 unidades, se expandieron un total de 269 nodos durante la búsqueda.

Ejecución python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

```
[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
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Se ha encontrado el resultado para un camino con un coste de 210 unidades, se expandieron un total de 620 nodos durante la búsqueda.

Pregunta 2.1 ¿BA encuentra una solución de menor coste? Si no es así, verificad vuestra implementación.

Si es correcto, ya que BA expande todos los nodos en un nivel dado antes de avanzar al siguiente nivel, lo que garantiza que encuentre la solución con el camino más corto primero.

Ejecucion python eightpuzzle.py

```
Press return for the next state...
After 13 moves: left
-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
```

La realización completa del puzzle se genera en 13 movimientos como se indica en la salida final del programa, al encontrar el resultado.

EJERCICIO 3

UFS ALGORITHM

```python
def uniformCostSearch(search_problem):
    priority_queue = util.PriorityQueue()
    path = []
    visited = set()
    state = search_problem.getStartState()
    cost = 0

    parent = {}
    from_dir = {}
    parent[state] = state
    from_dir[state] = None

    priority_queue.push((state, path, cost), cost)

    while not priority_queue.isEmpty():
        state, path, cost = priority_queue.pop()

        if search_problem.isGoalState(state):
            return path

        if state not in visited:
            visited.add(state)
            for successor in search_problem.getSuccessors(state):
                son_st, action, step_cost = successor
                if son_st not in visited:
                    parent[son_st] = state
                    from_dir[son_st] = action
                    priority_queue.push((son_st, path + [action], cost + step_cost), cost + step_cost)

    return None
```

## 1 - Inicialización de variables:

Se crea una cola de prioridad que se utilizará para mantener los estados a explorar, los elementos en esta cola se ordenan según el costo acumulado hasta el momento, lo que significa que se explorarán primero los estados menos costosos.

Se inicializan una lista ‘path’ y un conjunto ‘visited’ vacío para llevar un registro de los estados visitados.

Se obtiene el estado inicial del problema de búsqueda y se agrega a la cola de prioridad junto con su correspondiente ‘path’ y costo acumulado.

Se inicializan los diccionarios para llevar un registro de los padres y las direcciones desde las cuales se llegó a cada estado.

## 2 - Bucle Principal

Mientras la cola de prioridad no esté vacía se realizarán las siguientes acciones:

Se extrae el estado, el camino asociado y el costo acumulado desde el frente de la cola de prioridad.

Se verifica si el estado actual es el estado objetivo, si es el objetivo se devuelve el camino encontrado.

Si el estado no ha sido visitado previamente se marca como visitado y se recorren los sucesores del estado actual.

Para cada sucesor que no ha sido visitado, se actualizan los diccionarios con la información del sucesor y se calcula el nuevo costo acumulado, luego se agrega el sucesor a la cola de prioridad, junto con su camino actualizado y el nuevo costo acumulado.

## 3 - Devolución de resultado:

Si se ha completado la exploración de todos los estados posibles sin encontrar el objetivo, se devuelve None lo que indica que no se ha encontrado un camino desde el estado inicial al estado objetivo.

## En Resumen:

Esta función tiene un coste heurístico igual a cero, encuentra un camino desde el estado inicial hasta el estado objetivo y lo devuelve como una lista de direcciones, si no se encuentra un camino, devuelve None. El algoritmo garantiza que se explore primero el camino con el menor costo acumulado.

Ejecución python 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
```

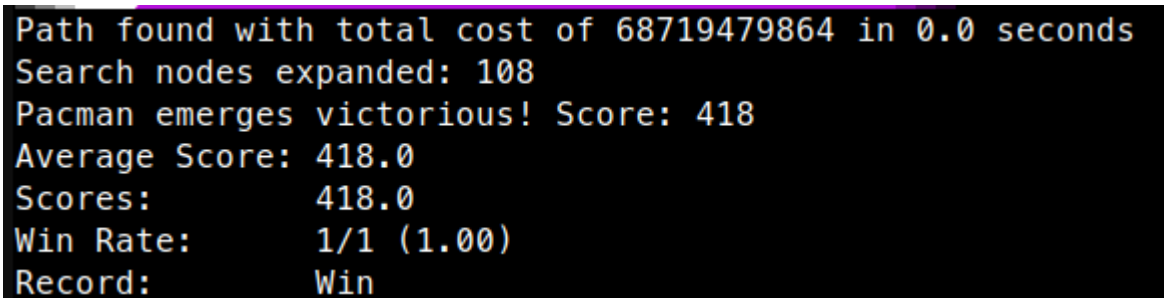
Se ha encontrado el resultado para un camino con un coste de 68 unidades, se expandieron un total de 269 nodos durante la búsqueda.

Ejecución python pacman.py -l mediumDottedMaze -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
```

Se ha encontrado el resultado para un camino con un coste de 186 unidades, se expandieron un total de 646 nodos durante la búsqueda.

Ejecución python pacman.py -l mediumScaryMaze -p StayWestSearchAgent




Se ha encontrado el resultado para un camino con un coste de 108 unidades, se expandieron un total de 418 nodos durante la búsqueda.

EJERCICIO 4

A* ALGORITHM

```python
def aStarSearch(search_problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    queue = util.PriorityQueue()
    ret = []
    expanded = []
    state = search_problem.getStartState()
    expanded.append(state)
    parent = {}
    direction = {}
    parent[state] = state
    direction[state] = None
    accumulatedCost = {}
    accumulatedCost[state] = 0
    while not search_problem.isGoalState(state):
        for successor in search_problem.getSuccessors(state):
            childState = successor[0]
            if childState not in expanded:
                newCost = accumulatedCost[state] + successor[2]
                if childState not in parent.keys() or accumulatedCost[childState] > newCost:
                    accumulatedCost[childState] = newCost
                    parent[childState] = state
                    direction[childState] = successor[1]
                queue.push(childState, accumulatedCost[childState] + heuristic(childState, search_problem))
        while state in expanded:
            if queue.isEmpty():
                return None
            state = queue.pop()
        expanded.append(state)
    while parent[state] != state:
        ret.insert(0, direction[state])
        state = parent[state]
    return ret
```

## 1- Input:

La función toma dos argumentos: search_problem y heuristic. search_problem es un objeto que representa el problema de búsqueda y debe implementar ciertos métodos. heuristic es una función que estima el costo restante desde un estado dado hasta el objetivo. Si no se proporciona, se utiliza una función de heurística nula (nullHeuristic) por defecto.

## 2- Inicialización de estructuras de datos:

La función inicializa varias estructuras de datos, como una cola de prioridad (queue) para mantener los estados a explorar, una lista ret para almacenar el camino encontrado, una lista expanded para realizar un seguimiento de los estados que ya se han expandido, y varios diccionarios (parent, direction, accumulatedCost) para mantener información sobre los nodos y el costo acumulado.

## 3- Bucle principal:

La función entra en un bucle mientras el estado actual (state) no es el estado objetivo (según isGoalState). En cada iteración de este bucle, se realizan los siguientes pasos:

a. Se itera a través de los sucesores del estado actual utilizando getSuccessors(state).

Cada sucesor tiene tres componentes: el estado del sucesor (childState), la acción para llegar al sucesor (successor[1]), y el costo desde el estado actual hasta el sucesor (successor[2]).

b. Para cada sucesor, se calcula el nuevo costo acumulado (newCost) sumando el costo acumulado del estado actual y el costo del sucesor.

c. Si el sucesor no ha sido expandido o si el nuevo costo es menor que el costo acumulado previamente registrado para el sucesor, se actualiza la información relacionada con el sucesor en los diccionarios parent, direction, y accumulatedCost.

d. El sucesor se agrega a la cola de prioridad queue con una prioridad que es la suma del costo acumulado y el valor de la heurística (accumulatedCost[childState] + heuristic(childState, search_problem)).

e. Se extrae de la cola de prioridad el siguiente estado a explorar (state) y se lo agrega a la lista expanded.

## 4- Construcción del camino:

Una vez que se encuentra el estado objetivo, la función construye el camino desde el estado objetivo hasta el estado inicial siguiendo las referencias en el diccionario parent. El camino se almacena en la lista ret.

## 5- Resultado:

Finalmente, la función devuelve el camino encontrado en forma de una lista de acciones que llevan desde el estado inicial al estado objetivo.

En resumen, esta función implementa el algoritmo A* para buscar el camino más corto desde un estado inicial a un estado objetivo en un grafo ponderado. Utiliza una heurística (o una heurística nula si no se proporciona) para guiar la búsqueda y asegurarse de explorar los estados más prometedores primero.

Ejecución python 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 0.0 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
```

Se ha encontrado un camino con un coste de 210 unidades, se expandieron un total de 549 nodos durante la búsqueda.

Pregunta 4.1 ¿Qué sucede en openMaze para las diversas estrategias de búsqueda?

A la hora de usar openMaze para las diversas estrategias de búsqueda, encontramos que la ruta encontrada tiene un menor coste en el A* y al variar la función de coste. Mientras que en BP y BA usar el openMaze no nos hace tener un menor coste, sino que al contrario.

# EJERCICIO 5

# CORNERS PROBLEM

```python
def __init__(self, startingGameState):
    """
    Stores the walls, pacman's starting position and corners.
    """
    self.walls = startingGameState.getWalls()
    self.startingPosition = startingGameState.getPacmanPosition()
    top, right = self.walls.height-2, self.walls.width-2
    self.corners = ((1, 1), (1, top), (right, 1), (right, top))
    for corner in self.corners:
        if not startingGameState.hasFood(*corner):
            print('Warning: no food in corner ' + str(corner))
    self._expanded = 0  # DO NOT CHANGE; Number of search nodes expanded
    # Please add any code here which you would like to use
    # in initializing the problem
    self.startingGameState = startingGameState
    cornersVisited = [False, False, False, False]
    if self.startingPosition == self.corners[0]:
        cornersVisited[0] = True
    if self.startingPosition == self.corners[1]:
        cornersVisited[1] = True
    if self.startingPosition == self.corners[2]:
        cornersVisited[2] = True
    if self.startingPosition == self.corners[3]:
        cornersVisited[3] = True
    self.startingState = (self.startingPosition, tuple(cornersVisited))

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

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    cornersVisited = state[1]
    if cornersVisited[0] and cornersVisited[1] and cornersVisited[2] and cornersVisited[3]:
        return True
    else:
        return False
```

```python
def getSuccessors(self, received_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 = received_state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]
        if not hitsWall:
            nextState = (nextx, nexty)
            cornersVisited = list(received_state[1])
            if nextState == self.corners[0]:
                cornersVisited[0] = True
            if nextState == self.corners[1]:
                cornersVisited[1] = True
            if nextState == self.corners[2]:
                cornersVisited[2] = True
            if nextState == self.corners[3]:
                cornersVisited[3] = True
            cost = 1
            successors.append( ( (nextState, tuple(cornersVisited)), action, cost, ) )
    self._expanded += 1  # DO NOT CHANGE
    return successors
```

## __init__(self, startingGameState):

El constructor de la clase recibe un objeto startingGameState, que representa el estado inicial del juego de PacMan. En este constructor, se inicializan varios atributos importantes:

self.walls: Representa las paredes en el diseño del juego.

self.startingPosition: Representa la posición inicial de PacMan.

self.corners: Almacena las coordenadas de las cuatro esquinas del diseño.

Se verifica si hay comida en cada esquina, y se muestra un mensaje de advertencia si alguna de las esquinas no tiene comida.

self._expanded: Un contador para llevar un registro del número de nodos de búsqueda expandidos.

self.startingGameState: Almacena el estado inicial del juego.

self.startingState: Representa el estado inicial del problema de búsqueda, que incluye la posición inicial de PacMan y una tupla que indica qué esquinas ha visitado PacMan.

## getStartState(self):

Devuelve el estado de inicio del problema de búsqueda. El estado incluye la posición inicial de PacMan y una tupla que registra qué esquinas ha visitado PacMan.

## isGoalState(self, state):

Verifica si el estado dado es un estado objetivo, es decir, si PacMan ha visitado todas las esquinas. Comprueba la tupla que registra las esquinas visitadas en el estado.

## getSuccessors(self, received_state):

Esta función genera sucesores para el estado dado. Para cada una de las cuatro direcciones posibles (norte, sur, este y oeste), verifica si el movimiento en esa dirección es legal (no choca con una pared). Si el movimiento es legal, se calcula el nuevo estado resultante después del movimiento. Luego, se actualiza la tupla de esquinas visitadas en el nuevo estado si Pac-Man visita una esquina. Cada sucesor tiene un costo de 1, ya que cada movimiento tiene un costo uniforme.
Los sucesores generados son triples que consisten en el nuevo estado, la acción requerida para llegar a ese estado y el costo de la transición.
Finalmente, se actualiza el contador _expanded para rastrear el número de nodos de búsqueda expandidos.

Esta clase CornersProblem se utiliza para definir un problema de búsqueda en el contexto del juego de Pac-Man, donde el objetivo es encontrar un camino que permita a PacMan visitar todas las esquinas del diseño. La función getSuccessors se encarga de generar los sucesores para un estado dado, teniendo en cuenta las restricciones del juego.

Ejecución python pacman.py -l tinyCorners -p SearchAgent -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
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Se ha encontrado un camino con un coste de 28 unidades, se expandieron un total de 252 nodos durante la búsqueda.

Ejecución python 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 0.0 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Se ha encontrado un camino con un coste de 106 unidades, se expandieron un total de 1966 nodos durante la búsqueda.

# EJERCICIO 6

# CORNERS HEURISTIC

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

      cur_state:   The current search state
                   (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    corners = problem.corners
    walls = problem.walls

    dist = -1
    it = 0
    pos, goal = cur_state

    while (it < 4):
        if(goal[it] == False):
            tdist = util.manhattanDistance(problem.corners[it], pos)
            if (dist == -1):
                dist = tdist
            elif (dist < tdist):
                dist = tdist
        it += 1
    if(dist == -1):
        return 0
    return dist
```

Se trata de una heurística implementada para el problema de las esquina en un algoritmo de búsqueda. El objetivo principal de esta función es proporcionar una estimación del costo mínimo para llegar desde el estado actual a un estado objetivo del problema. Esta heurística debe ser admisible y consistente, lo que significa que no debe de sobreestimar el costo real y debe seguir una regla de monotonía

## 1 - PARÁMETRO DE ENTRADA

cur_state: representa el estado actual en la búsqueda.

problem: instance de CornersProblem que contiene información sobre la disposición del laberinto, incluyendo la ubicación de las esquinas y de las paredes.

## 2 - INICIALIZACIÓN DE VARIABLES

corners y walls se asignan a las esquinas y las paredes del problema.

pos y goals se desempaqueta de la tupla cur_state.

heuristic_dist se inicializa en 0 y se usará para acumular la estimación heurística.

## 3 - CÁLCULO DE ESQUINAS NO VISITADAS

Se crea una lista llamada unvisited_conrners, que contiene las coordenadas de las esquinas que aún no han sido visitadas. Esto se hace mediante una lista por comprensión que recorre las cuatro esquinas y verifica si la correspondiente entrada en la lista goal es False.

## 4 - CÁLCULO DE LA ESTIMACIÓN HEURÍSTICA

Si hay esquinas no visitadas se procede a calcular una estimación heurística.

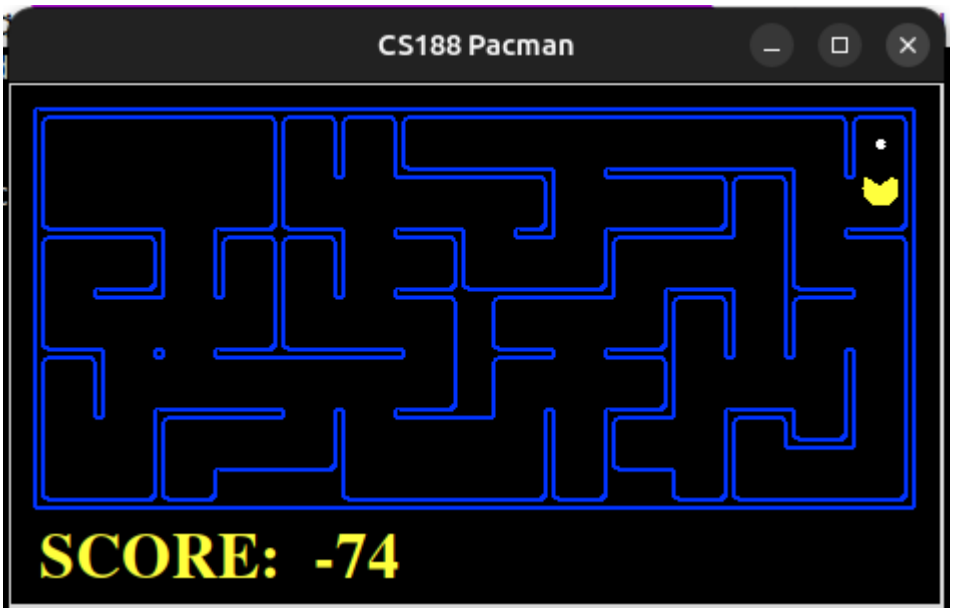
Se utiliza una función útil.manhattan_distance para calcular la distancia Manhattan desde la posición actual a cada una de las esquinas no visitadas.

La suma de estas distancias se acumulan en heursitic_distance como la estimación heurística del costo mínimo para alcanzar todas las esquinas no visitadas asumiendo que se puede llegar directamente a cada una de ellas sin ningún tipo de obstáculo.

## 5 - RETORNO DE LA ESTIMACIÓN HEURÍSTICA

La función devuelve Heuristic_distance como la estimación heurística del costo mínimo para alcanzar el estado objetivo. Si no hay esquinas no visitadas la estimación heurística es 0, lo que indica que no hay que recorrer ninguna distancia adicional para alcanzar el objetivo.

Ejecucion python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5




```
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 1136
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Los nodos explorados han sido 1136, con un coste total de 106 en 0.1 segundos.

Pregunta 6.1: Describe el proceso que se ha seguido para diseñar la heurística y explica la lógica de la misma

1-Comprensión del problema: el problema implica navegar por un laberinto y visitar todas las esquinas antes de llegar al objetivo

2-Identificar la información relevante: determinamos cual es la información relevante para calcular la distancia, que en este caso es conocer la ubicación de las esquinas y las paredes del laberinto.

3-Elegir una métrica de distancia: En esta caso la distancia de Manhattan es la elección más natural ya que el movimiento solo puede ser horizontal o vertical en una cuadrícula

4-Pensar en la estrategia de estimación:  Pensamos en estimar la distancia mínima que hace falta para visitar todas las esquinas no visitadas.

5-Diseñar la heurística específica:

-Identificar las esquinas no visitadas

-Calcular la distancia de Manhattan desde la posición actual a cada esquina no visitada

-Mantener un seguimiento de la distancia máxima encontrada hasta ahora

6-Garantizar que la heurística sea admisible:

Encontramos una heurística que garantiza que la búsqueda sea admisible y nunca sobreestima el costo

Y tras estos pasos conseguimos dicha heurística.