

INTELIGENCIA ARTIFICIAL

Práctica - 1

G.1311 - P2

Joaquín Abad Díaz - Carlos García Santa

RESUMEN

En la práctica se tiene en consideración los siguientes elementos para cada sección: un mapa cualquiera, un problema a resolver en ese mapa y un algoritmo utilizado para resolver dicho problema. Por ejemplo se puede resolver el problema de encontrar un punto de comida fijo (PositionSearchProblem) en un mapa mediano (mediumMaze) utilizando búsqueda primero en anchura (breadthFirstSearch). En algun caso que otro como en la sección 3 se cambia también el agente utilizado para forzar una estrategia en concreto basada en costes.

En ningún momento ha sido necesario modificar la estructura de datos utilizada para representar el nodo de búsqueda.

Para las secciones 1 a 4 se utiliza PositionSearchProblem, sobre el cual es importante definir que hace isGoalState y getStartState y getSuccesors.

-getStartState: Se utiliza para obtener el estado desde el cual el algoritmo de búsqueda debe comenzar, este está dado por la información del layout .

-isGoalState: Se utiliza para verificar si el estado actual es la meta. Si state es igual a self.goal, la función devolverá True, y False en caso contrario. Esto también está determinado por la información del layout.

-getSuccesors: Se utiliza para expandir un nodo durante la búsqueda, identificando todos los estados a los que se puede llegar desde el estado actual y las acciones necesarias para llegar a ellos. También lleva a cabo un seguimiento de la visualización y la contabilidad de los nodos expandidos y visitados durante la búsqueda.

Estas funciones permiten navegar y explorar un espacio de estados en un algoritmo de búsqueda, proporcionando la lógica necesaria para identificar el estado inicial, reconocer el estado objetivo, y expandir un estado en los estados sucesores, respectivamente.

Sección 1: Encontrar un punto de comida fijo usando la búsqueda primero en profundidad

Utilizamos una pila (Stack) para almacenar los nodos que aún no se han explorado y un conjunto (set) para almacenar los nodos que ya hemos visitado para evitar su reexploración. Tras eso inicializamos la pila con el estado inicial y una lista vacía de acciones.

Ahora iteramos mientras la pila no esté vacía:

1. Obtenemos el nodo y las acciones desde el top de la pila.
2. Verificamos si el estado actual es un estado objetivo. Si es el estado objetivo, retornamos las acciones que nos llevaron a este estado.
3. Verificamos si el estado actual ya ha sido visitado. En caso negativo añadimos el estado al conjunto de nodos visitados e iteramos a través de todos los sucesores del estado actual añadiendo cada sucesor y la lista de acciones anterior más la acción a realizar a la pila.

Si la pila está vacía y no se ha encontrado una solución, retornamos None.

```
def depthFirstSearch(search_problem):  
    stack = util.Stack()  
    visited_list = set()  
    stack.push((search_problem.getStartState(), []))  
  
    while not stack.isEmpty():  
        state, actions = stack.pop()  
  
        if search_problem.isGoalState(state):  
            return actions  
  
        if state not in visited_list:  
            visited_list.add(state)  
            for successor, action, _ in  
search_problem.getSuccessors(state):  
                stack.push((successor, actions + [action]))  
  
    return None
```

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?

Sí, el orden de exploración es el esperado. Pacman, utilizando DFS, irá tan profundo como pueda en una dirección antes de retroceder y probar otras direcciones. La dirección específica en la que Pacman decide moverse primero depende del orden en que se devuelvan los sucesores.

En cuanto a la exploración, pacman se mueve a todas las casillas visitadas pero no a todas las expandidas.

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

No porque DFS explora tan profundamente como sea posible a lo largo de una rama antes de retroceder, lo que podría llevarlo por un camino significativamente más largo al objetivo cuando podría haber un camino más corto o de menos coste disponible en un nodo anterior.

*Se puede apreciar la diferencia de coste con el resto de algoritmos en el **Anexo**.

Sección 2: Búsqueda en anchura

El siguiente código es idéntico al usado para dfs; sin embargo, en este caso en vez de usarse una pila, se utiliza una cola. Al utilizar un mecanismo FIFO en vez de uno LIFO el siguiente nodo sucesor que se explora es el primero que entra a la cola, cambiando el orden en el que se exploran los nodos: se exploran primero los nodos hermanos y se profundiza tras haberlos explorado.

```
def breadthFirstSearch(search_problem):
    queue = util.Queue()
    visited_list = set()
    queue.push((search_problem.getStartState(), []))

    while not queue.isEmpty():
        state, actions = queue.pop()

        if search_problem.isGoalState(state):
            return actions

        if state not in visited_list:
            visited_list.add(state)
```

```
        for successor, action, _ in
search_problem.getSuccessors(state):
            queue.push((successor, actions + [action]))

return None
```

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

Sí, ya que si todos los costos de acción son iguales, BFS garantiza encontrar el camino más corto a la solución ya que explora los nodos en orden de su distancia desde el punto de partida.

*Se puede apreciar la diferencia de coste con el resto de algoritmos en el **Anexo**.

Sección 3: Variar la función de coste

La estructura de datos clave aquí es una cola de prioridad que asegura que siempre estamos explorando el camino de menor costo hasta ahora. Si el costo para llegar a los estados sucesores es siempre positivo, este método garantiza encontrar una solución óptima.

Es importante mencionar que el estado inicial se introduce en la cola de prioridad con coste 0 y que esta se encargará de ordenar los elementos por su coste asegurándose de que los de menor coste se procesan primero.

```
def uniformCostSearch(search_problem):
    prio_queue = util.PriorityQueue()
    visited_list = set()
    prio_queue.push((search_problem.getStartState(), [], 0), 0)

    while not prio_queue.isEmpty():
        state, actions, cost = prio_queue.pop()

        if search_problem.isGoalState(state):
            return actions

        if state not in visited_list:
            visited_list.add(state)
```

```

        for successor, action, step_cost in
search_problem.getSuccessors(state):
            new_cost = cost + step_cost
            prio_queue.push((successor, actions + [action],
new_cost), new_cost)

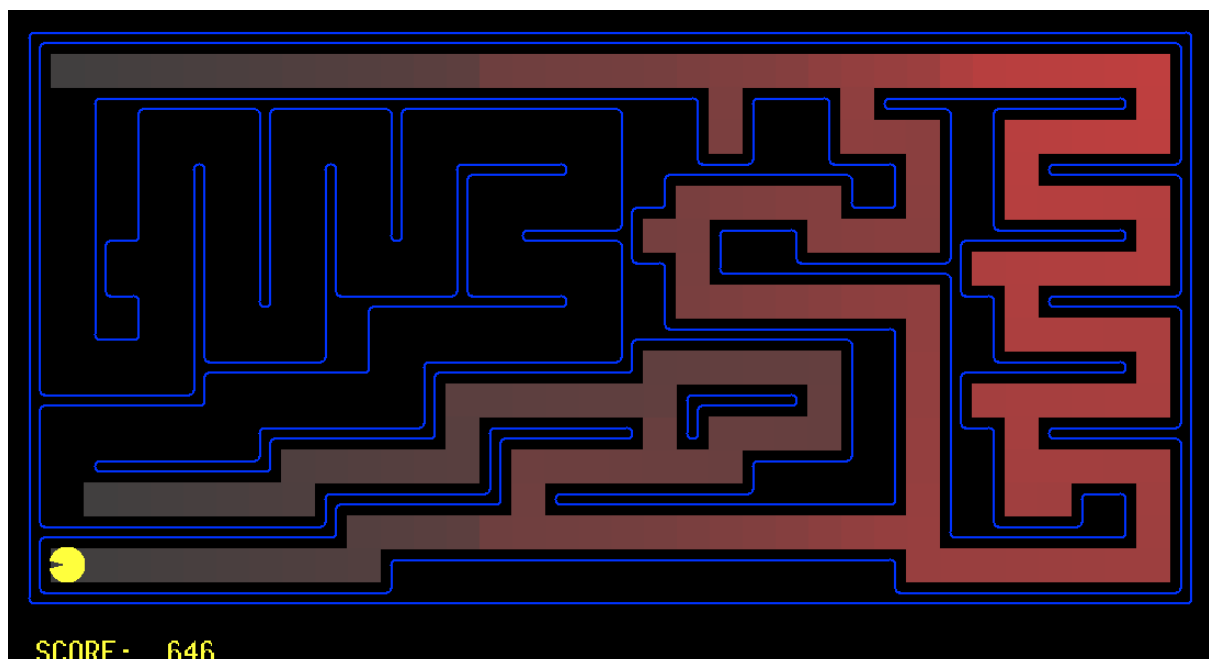
return None

```

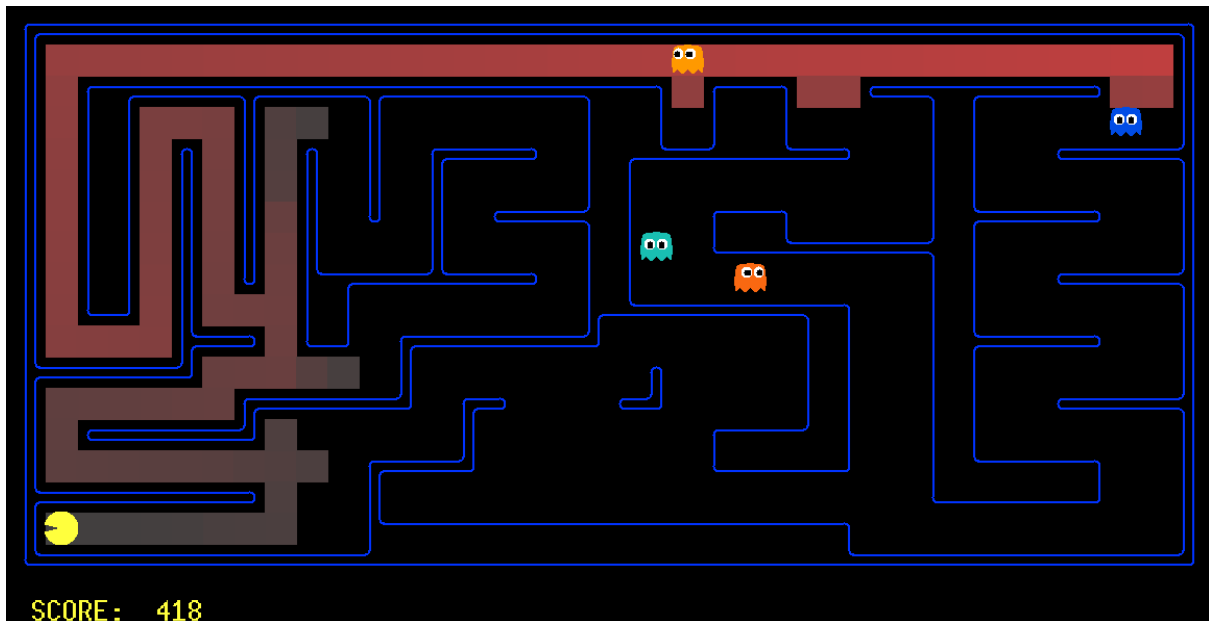
Los datos esperados son costes de ruta muy bajos y muy altos para StayEastSearchAgent y StayWestSearchAgent, respectivamente, debido a sus funciones de coste exponencial.

Los datos resultantes son los siguientes, que coinciden con lo esperado.

		mediumDottedMaze
UCS StayEast	Score	646
	Cost	1
	Expanded nodes	186
		mediumScaryMaze
UCS StayWest	Score	418
	Cost	68719479864
	Expanded nodes	108



python pacman.py -l mediumDottedMaze -p StayEastSearchAgent



```
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Sección 4: Búsqueda A*

A* funciona como el la función de costo uniforme usando una cola de prioridad; sin embargo, en este caso también recibe una heurística la cual se calculará por cada nodo para sumarse al costo acumulado en ese nodo. Se puede observar [ANEXO] que en bigMaze se encuentra la solución óptima un poco más rápido que la búsqueda de coste uniforme (aproximadamente 549 frente a 620 nodos expandido).

```
def aStarSearch(search_problem, heuristic=nullHeuristic):  
  
    prio_queue = util.PriorityQueue()  
  
    # Iniciar la lista con el estado inicial y un coste de camino de 0  
    start_state = search_problem.getStartState()  
    prio_queue.push((start_state, [], 0), 0 + heuristic(start_state,  
search_problem))  
  
    visited_list = set()  
  
    while not prio_queue.isEmpty():  
        state, actions, cost_g = prio_queue.pop()
```

```

    if search_problem.isGoalState(state):
        return actions

    if state not in visited_list:
        visited_list.add(state)

    for successor, action, step_cost in
search_problem.getSuccessors(state):
        new_cost_g = cost_g + step_cost
        cost_h = heuristic(successor, search_problem)
        cost_f = new_cost_g + cost_h
        prio_queue.push((successor, actions + [action],
new_cost_g), cost_f)


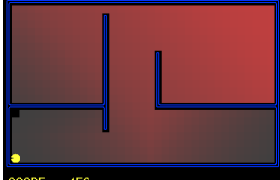
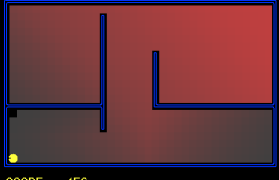

    return None

```

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

Dfs genera un coste mucho mayor al resto de los algoritmos; sin embargo, expande menos nodos que bfs y ucs. Estos dos comparten una puntuación mucho mayor y un coste mucho menor que dfs con A*, pero este último alcanza el número más bajo de nodos expandidos.

openMaze	DFS	BFS	UCS	A* MANHATTAN
Score	212	456	456	456
Cost	298	54	54	54
Expanded nodes	576	682	682	535

DFS	BFS	UCS	A* MANHATTAN
 SCORE: 212	 SCORE: 456	 SCORE: 456	 SCORE: 456

Sección 5: Encontrar todas las esquinas

Para lograr una abstracción del estado del sistema que permita detectar si se han alcanzado las cuatro esquinas o no, hemos decidido retornar como estado inicial una tupla compuesta por un primer elemento que corresponde a la posición inicial (x, y) del agente y un segundo elemento el cual se trata de una tupla de cuatro booleanos en la que cada valor representa si la esquina correspondiente ha sido visitada o no. Esta representación ha sido elegida principalmente dada su sencillez y fácil comprensión, lo que nos ha permitido definir posteriormente con facilidad el estado meta, el cual se comprueba que se ha alcanzado pasando la tupla de booleanos (c1, c2, c3, c4) por una puerta lógica AND de 4 entradas, que devolverá TRUE únicamente en el caso de que las cuatro esquinas tengan el valor TRUE; y también nos ha facilitado la implementación de la función que obtiene los sucesores de un nodo, donde utilizamos una tupla (nextCorners) con nuevamente cuatro valores que representan si las esquinas han sido visitadas, en este caso empleamos la puerta lógica OR con dos entradas, una correspondiendo al booleano cx donde x es el número de esquina y otra entrada ((nextx, nexty) == self.corners[x]) que comprueba si la siguiente posición a la que se puede mover Pacman (nextx, nexty) es igual a las coordenadas de la esquina x, si la igualdad se cumple se obtiene TRUE y se actualiza el estado de esa esquina. Se utiliza una puerta OR ya que si cx ya es TRUE no importa que valor tome la igualdad y esa esquina seguirá habiendo sido visitada, lo mismo ocurre al contrario, si cx no ha sido visitado y tiene valor FALSE pero la posición del nodo sucesor corresponde con la de una esquina la esquina se actualizará a TRUE. Luego en la lista de nodos sucesores introducimos la tupla nextCorners que sería el nuevo estado de la tupla de esquinas si Pacman se trasladará a ese sucesor. De esta manera usamos una tupla como registro que nos indica el estado de acceso a las esquinas del laberinto.

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

def isGoalState(self, state):
```

```

    """
    Returns whether this search state is a goal state of the
    problem.
    """
    """
    *** YOUR CODE HERE ***
    _, (c1, c2, c3, c4) = state
    return c1 and c2 and c3 and c4

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 = []
    currentPosition, (c1, c2, c3, c4) = received_state

    for action in [Directions.NORTH, Directions.SOUTH,
Directions.EAST, Directions.WEST]:
        x, y = currentPosition
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)

        if not self.walls[nextx][nexty]:
            nextCorners = (c1 or (nextx, nexty) == self.corners[0],
                           c2 or (nextx, nexty) == self.corners[1],
                           c3 or (nextx, nexty) == self.corners[2],
                           c4 or (nextx, nexty) == self.corners[3])
            successors.append(((nextx, nexty), nextCorners),
action, 1))

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

```

(usando bfs)	
tinyCorners	mediumCorners
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	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

Sección 6: Problema de las esquinas: heurística

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

Primero comenzamos analizando el objetivo a alcanzar que es que Pacman alcance las cuatro esquinas, siendo claves para esta meta la posición del agente, la posición de las esquinas, el registro de esquinas visitadas y los posibles obstáculos del laberinto. A partir de ello la primera estrategia que ideamos fue la de emplear como estimación la distancia desde la posición en la que se encuentre el agente hasta la esquina más cercana no visitada como estimación, para ello observamos la utilidad de la distancia Manhattan la cual es particularmente conveniente ya que calcula la distancia entre dos puntos sin tener en cuenta los obstáculos, de este modo nuestra estimación heurística siempre sería más optimista que el coste mínimo real, dado que los laberintos contienen paredes que funcionan como obstáculos. Esta heurística siempre sería admisible y consistente, sin embargo no aporta la suficiente información para que el algoritmo A* sea mucho más óptimo y no expanda muchos más nodos de lo necesario, obteniendo 1475 nodos expandidos y no superando todos los test del autograder. De este modo pensamos en introducir en la heurística algo que aportase mayor cantidad de información para resolver el problema sin ser excesivamente costosa computacionalmente, así decidimos que una vez el agente alcanzase la esquina no visitada más cercana a él calculase la siguiente esquina no visitada más cercana de tal manera que la heurística se acerca más a la solución completa y el algoritmo expande menos nodos, expandiendo en concreto 919 nodos y pasando todos los test automáticos. Esta heurística es siempre consistente ya que si tomamos la fórmula $h(n) \leq c(n, a, n') + h(n')$ vemos que la heurística desde un nodo n siempre va a ser menor o igual que el coste de acción de llegar al siguiente nodo n' que siempre es 1 más la estimación de llegar a la meta desde el nodo n' que al utilizar la suma de las distancias mínimas se reduce siempre en 1.

```
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

    currentPosition, visitedCorners = cur_state
    unvisitedCorners = [corners[i] for i in range(4) if not
visitedCorners[i]]

    if not unvisitedCorners:
        return 0

    # Compute the Manhattan distance from the current position to the
nearest unvisited corner
    distancesToCorners = [util.manhattanDistance(currentPosition,
corner) for corner in unvisitedCorners]
    min_distance = min(distancesToCorners)

    # Estimate the cost to travel between all the remaining unvisited
corners
    total_corner_distance = 0
    for i in range(len(unvisitedCorners)):
        for j in range(i+1, len(unvisitedCorners)):
            total_corner_distance = max(total_corner_distance,
util.manhattanDistance(unvisitedCorners[i], unvisitedCorners[j]))
    print("min_distance: " + str(min_distance))
    print("total_corner_distance: " + str(total_corner_distance))
    return min_distance
```

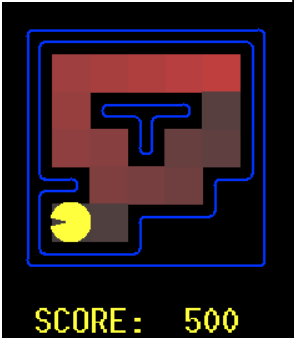
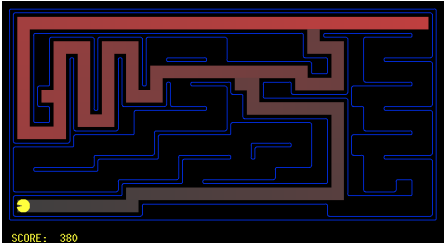
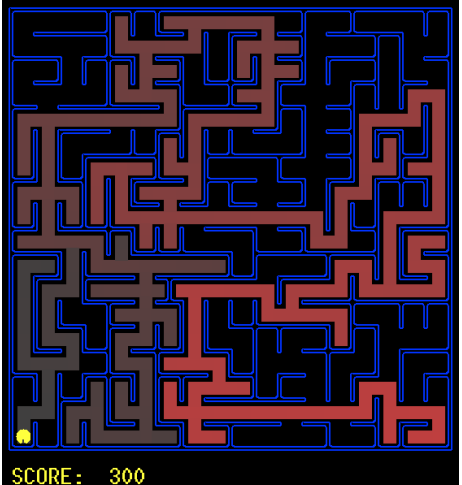
Inteligencia Artificial, 2023-2024
Práctica-1

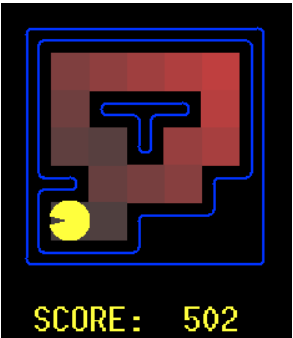
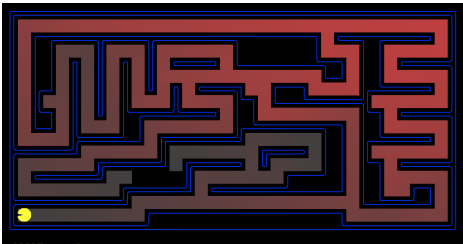
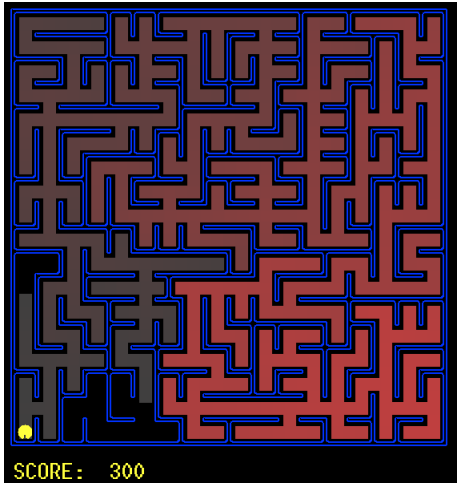
Comparación en mediumCorners	
nullHeuristic	cornersHeuristic
Path found with total cost of 106 in 0.2 seconds Search nodes expanded: 1966 Pacman emerges victorious! Score: 434 Average Score: 434.0 Scores: 434.0 Win Rate: 1/1 (1.00) Record: Win	Path found with total cost of 106 in 0.0 seconds Search nodes expanded: 919 Pacman emerges victorious! Score: 434 Average Score: 434.0 Scores: 434.0 Win Rate: 1/1 (1.00) Record: Win

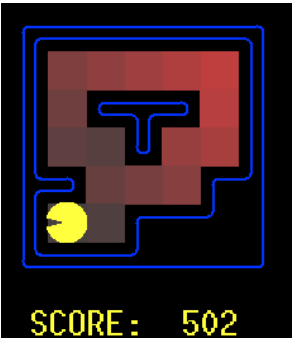
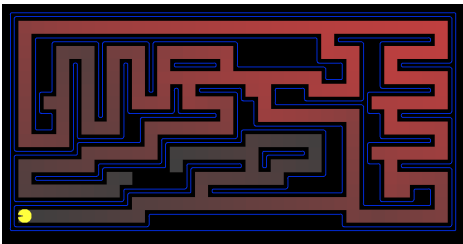
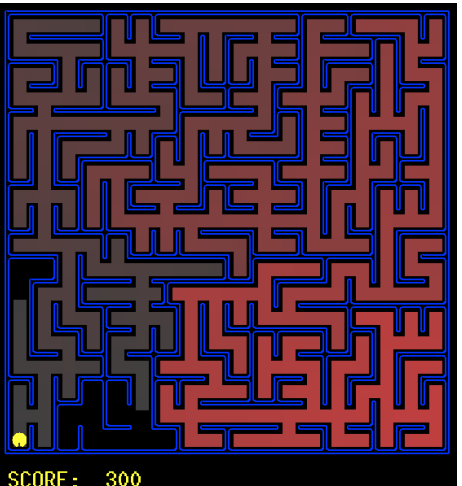
CornersHeuristic expande únicamente 774 nodos.

Anexo:

		tinyMaze	mediumMaze	bigMaze
DFS	Score	500	380	300
	Cost	10	130	210
	Expanded nodes	15	146	390
BFS	Score	502	442	300
	Cost	8	68	210
	Expanded nodes	15	269	620
UCS	Score	502	442	300
	Cost	8	68	210
	Expanded nodes	15	269	620
A* ManhattanH	Score	502	442	300
	Cost	8	68	210
	Expanded nodes	14	221	549

	tinyMaze	mediumMaze	bigMaze
DFS	 A small maze with a yellow Pac-Man character at the bottom left. The solution path is highlighted in blue, and the explored area is shaded in red. The score is 500.	 A medium-sized maze with a yellow Pac-Man character at the bottom left. The solution path is highlighted in blue, and the explored area is shaded in red. The score is 380.	 A large, complex maze with a yellow Pac-Man character at the bottom left. The solution path is highlighted in blue, and the explored area is shaded in red. The score is 300.

	tinyMaze	mediumMaze	bigMaze
BFS	 A small maze with a yellow Pac-Man character at the bottom left. The solution path is highlighted in blue, and the explored area is shaded in red. The score is 502.	 A medium-sized maze with a yellow Pac-Man character at the bottom left. The solution path is highlighted in blue, and the explored area is shaded in red. The score is 442.	 A large, complex maze with a yellow Pac-Man character at the bottom left. The solution path is highlighted in blue, and the explored area is shaded in red. The score is 300.

	tinyMaze	mediumMaze	bigMaze
UCS	 A small maze with a yellow Pac-Man character at the bottom left. The solution path is highlighted in blue, and the explored area is shaded in red. The score is 502.	 A medium-sized maze with a yellow Pac-Man character at the bottom left. The solution path is highlighted in blue, and the explored area is shaded in red. The score is 442.	 A large, complex maze with a yellow Pac-Man character at the bottom left. The solution path is highlighted in blue, and the explored area is shaded in red. The score is 300.

	tinyMaze	mediumMaze	bigMaze
A* MANHATTAN	