Escuela Politécnica Superior, Ingeniería Informática

Inteligencia Artificial

Memoria Práctica 1

Búsqueda

Alumnos: Ángel Bernal García y Elena Diego Bernabeu

Grupo 2313 Pareja 1

10 de marzo de 2021

Índice

1. Sección 1

1.1. Comentario personal en el enfoque y decisiones de la solución propuesta

En general en la estructura del código nos hemos basado en el pseudocódigo proporcionado, contando con listas de estados visitados y movimientos usados para cada estado. Para poder llevar a cabo esto hemos almacenado todos los pasos en cada estado y los estados visitados en la pila para poder acceder a ellos fácilmente. De esta forma, por ejemplo, al devolver la lista de movimientos realizados, devolvemos la correspondiente del estado final.

1.2. Lista & explicación de las funciones del framework usadas

- "getStartState()": Devuelve el estado inicial del problema.
- "getSuccessors()": Devuelve los sucesores del estado proporcionado.
- "Stack()": Inicializa la pila.
- "push()": Introduce un elemento en la pila correspondiente, dejándolo arriba en ella.
- "pop()": Saca el elemento de arriba de la pila correspondiente y lo devuelve.

1.3. Incluye el código añadido

Hemos completado el código de la función "depthFirstSearch(problem)"

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    startState = problem.getStartState()
    listaEstadosVisitados = []
    listaEstadosVisitados.append(startState)
    listaCerrados = []
    listaCerrados.append(startState)
    listaAbiertos = util.Stack()
    listaMovimientos = []
    listaAbiertos.push([(startState, 'Start', 0), listaMovimientos,listaEstadosVisitados])

    sucesores = problem.getSuccessors(startState)
    for sucesor in sucesores:
        if sucesor not in listaEstadosVisitados:
            listaAbiertos.push([sucesor, listaMovimientos, listaEstadosVisitados])

    while (listaAbiertos.isEmpty() == False):
        nextState = listaAbiertos.pop()
        if(problem.isGoalState(nextState[0][0]) is True):
            print("Objetivo alcanzado")
            nextState[1].append(nextState[0][1])
            return nextState[1]
        if nextState[0][0] not in listaCerrados:
            listaCerrados.append(nextState[0][0])
            listaMovimientos = nextState[1].copy()
            listaMovimientos.append(nextState[0][1])
            listaEstadosVisitados = nextState[2].copy()
            listaEstadosVisitados.append(nextState)
            sucesores = problem.getSuccessors(nextState[0][0])
            for sucesor in sucesores:
                if sucesor not in listaEstadosVisitados:
                    listaAbiertos.push([sucesor, listaMovimientos, listaEstadosVisitados])
```

Figura 1: Función de búsqueda en profundidad

1.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

Como podemos ver los resultados que hemos obtenidos han sido exitosos, el *Pacman* siempre llegaba a donde estaba la comida.

El problema que hemos visto con el algoritmo que hemos usado ha sido que el *Pacman* podría haber sido más eficiente yendo por otro camino y, por tanto, ahorrándose tiempo. De hecho, se puede ver a simple vista en el laberinto pequeño (Figura 2) que si nada más empezar baja en vez de ir a la izquierda, se podría haber ahorrado unas casillas.

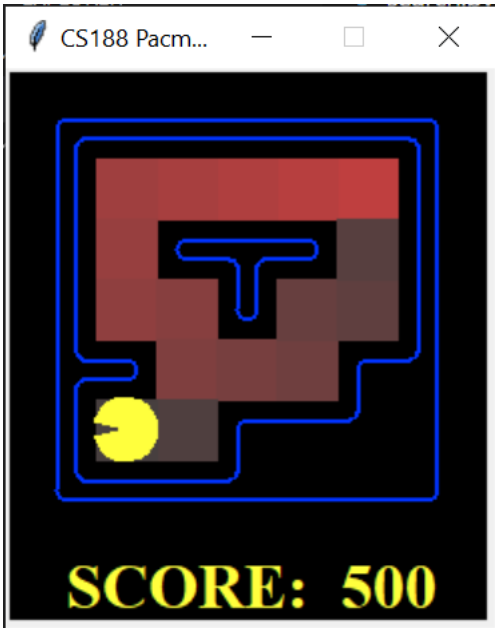

Figura 2: Laberinto pequeño

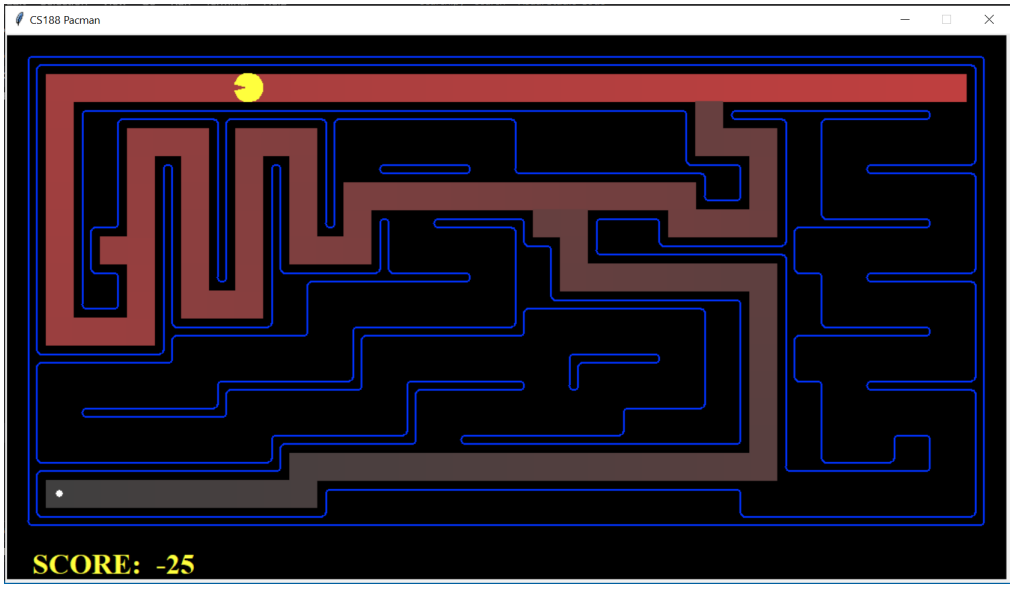

Figura 3: Laberinto mediano

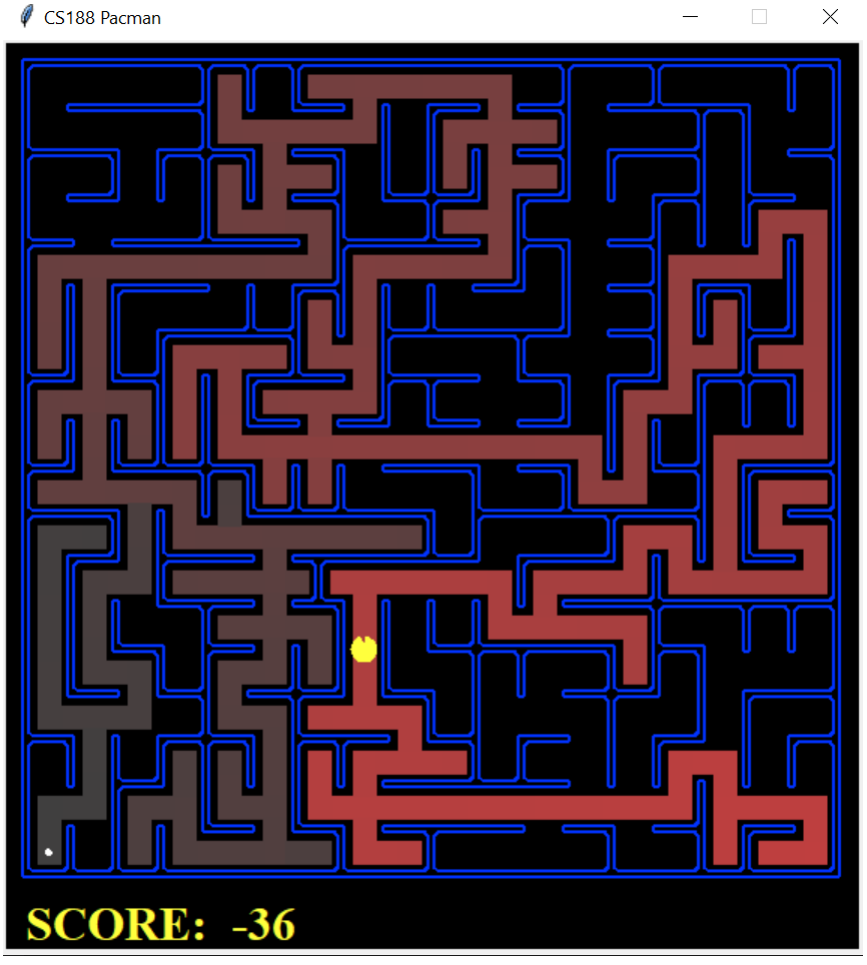

Figura 4: Laberinto grande

```
PS C:\Users\base9\Documents\Universidad\Tercero\ia\practica1\IA_P1\search> python autograder.py -q q1
Starting on 3-9 at 12:18:02

Question q1
==========
Objetivo alcanzado
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'D', 'C']
Objetivo alcanzado
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***     solution:               ['2:A->D', '0:D->G']
***     expanded_states:        ['A', 'D']
Objetivo alcanzado
*** PASS: test_cases\q1\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
Objetivo alcanzado
*** PASS: test_cases\q1\graph_manypaths.test
***     solution:               ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']
Objetivo alcanzado
*** PASS: test_cases\q1\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 130
***     nodes expanded:         146

### Question q1: 3/3 ###


Finished at 12:18:02

Provisional grades
==================
Question q1: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Figura 5: Autograder

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

Pacman llega a la solución, aunque no por el camino más óptimo. En todos los laberintos se deja caminos más cortos, debido a que como está hecho por profundidad no prioriza el menor coste, sino el primero que encuentra.

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

El orden de exploración es el que tiene sentido usando el algoritmo de profundidad. Pacman no va a todas las casillas exploradas, pues en multitud de casos llevan a caminos sin salida o directamente encuentra antes un camino que llegue al destino.

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

Consideramos que no es la solución de menor coste, ya que, como hemos mencionado anteriormente, podemos ver que hay otros caminos que serían más cortos para llegar al final. Por lo tanto podemos concluir que no es el algoritmo más eficiente, ya que puede llegar a perder tiempo y recursos del ordenador en continuar la ejecución en vez de usar otro algoritmo que vaya por un camino más adecuado.

2. Sección 2

2.1. Comentario personal en el enfoque y decisiones de la solución propuesta

El algoritmo en anchura está implementado exactamente igual que el algoritmo en profundidad, salvo que en este caso usamos una cola en vez de una pila.

2.2. Lista & explicación de las funciones del framework usadas

- "getStartState()": Devuelve el estado inicial del problema.
- "getSuccessors()": Devuelve los sucesores del estado proporcionado.
- "Queue()": Inicializa la cola.
- "push()": Introduce un elemento en la cola correspondiente, dejándolo arriba en ella.
- "pop()": Saca el elemento de arriba de la cola correspondiente y lo devuelve.

2.3. Incluye el código añadido

Hemos completado el código de la función "breadthFirstSearch(problem)"

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    startState = problem.getStartState()
    listaEstadosVisitados = []
    listaEstadosVisitados.append(startState)
    listaCerrados = []
    listaCerrados.append(startState)
    listaAbiertos = util.Queue()
    listaMovimientos = []
    listaAbiertos.push([(startState, 'Start', 0), listaMovimientos,listaEstadosVisitados])

    sucesores = problem.getSuccessors(startState)
    for sucesor in sucesores:
        if sucesor not in listaEstadosVisitados:
            listaAbiertos.push([sucesor, listaMovimientos, listaEstadosVisitados])

    while (listaAbiertos.isEmpty() == False):
        nextState = listaAbiertos.pop()
        if(problem.isGoalState(nextState[0][0]) is True):
            print("Objetivo alcanzado")
            nextState[1].append(nextState[0][1])
            return nextState[1]
        if nextState[0][0] not in listaCerrados:
            listaCerrados.append(nextState[0][0])
            listaMovimientos = nextState[1].copy()
            listaMovimientos.append(nextState[0][1])
            listaEstadosVisitados = nextState[2].copy()
            listaEstadosVisitados.append(nextState)
            sucesores = problem.getSuccessors(nextState[0][0])
            for sucesor in sucesores:
                if sucesor not in listaEstadosVisitados:
                    listaAbiertos.push([sucesor, listaMovimientos, listaEstadosVisitados])

    lista_vacia = []
    return lista_vacia
```

Figura 6: Función de búsqueda en anchura

2.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

Podemos observar que con este algoritmo se van a valorar más nodos que con el algoritmo anterior. Por tanto, los caminos por los que va son más eficientes que la búsqueda en profundidad.

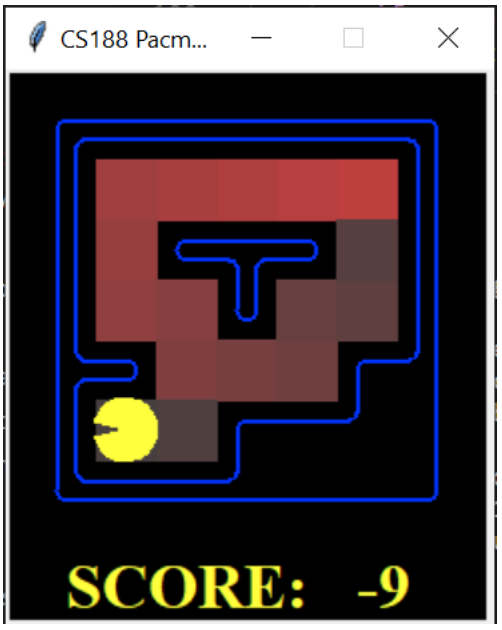

Figura 7: Laberinto pequeño

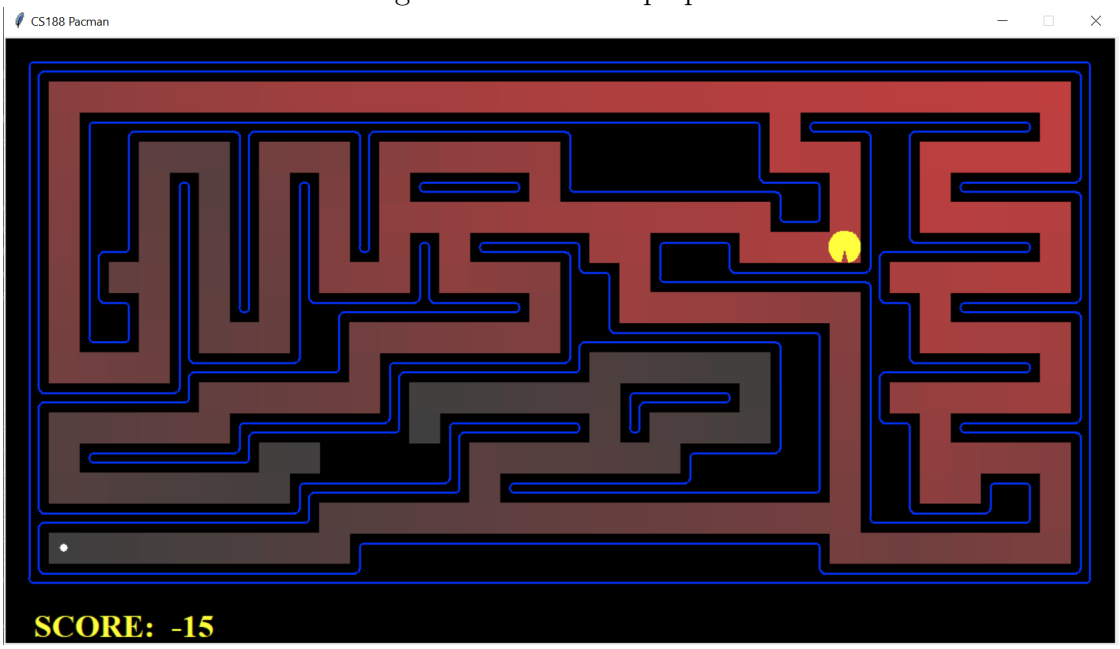

Figura 8: Laberinto mediano

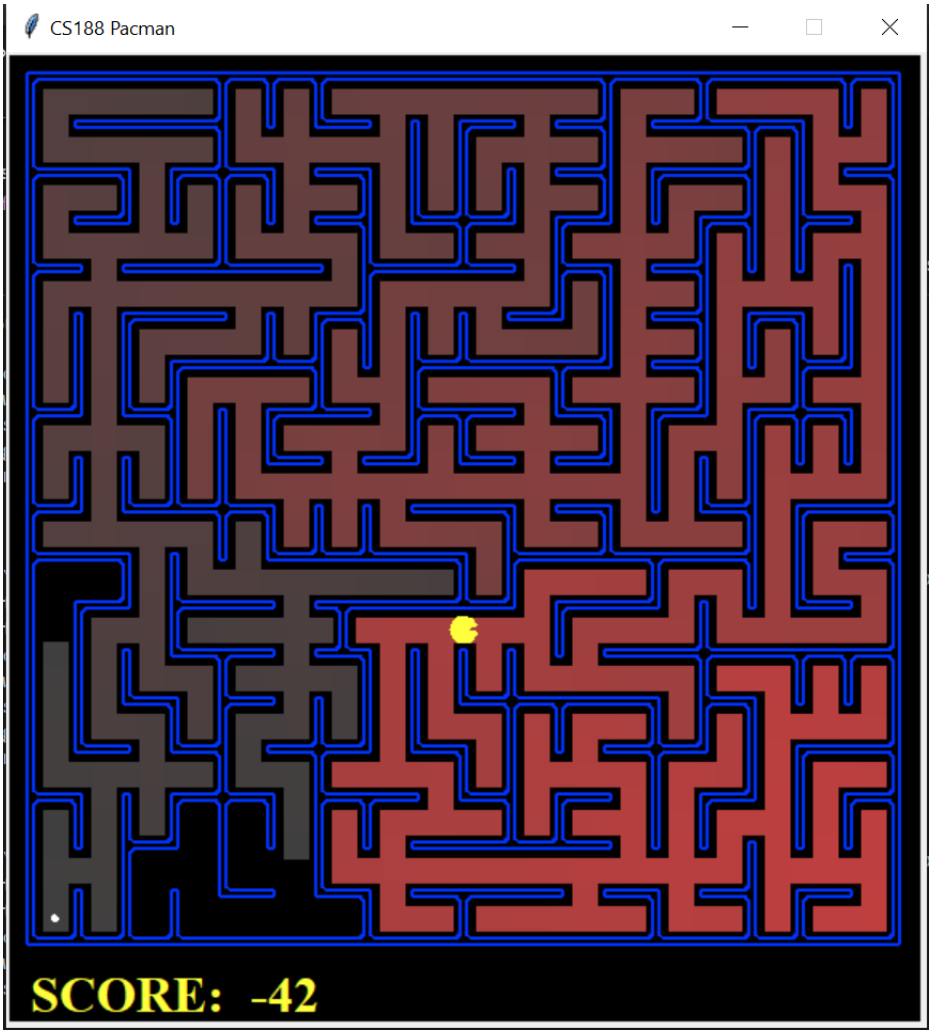

Figura 9: Laberinto grande

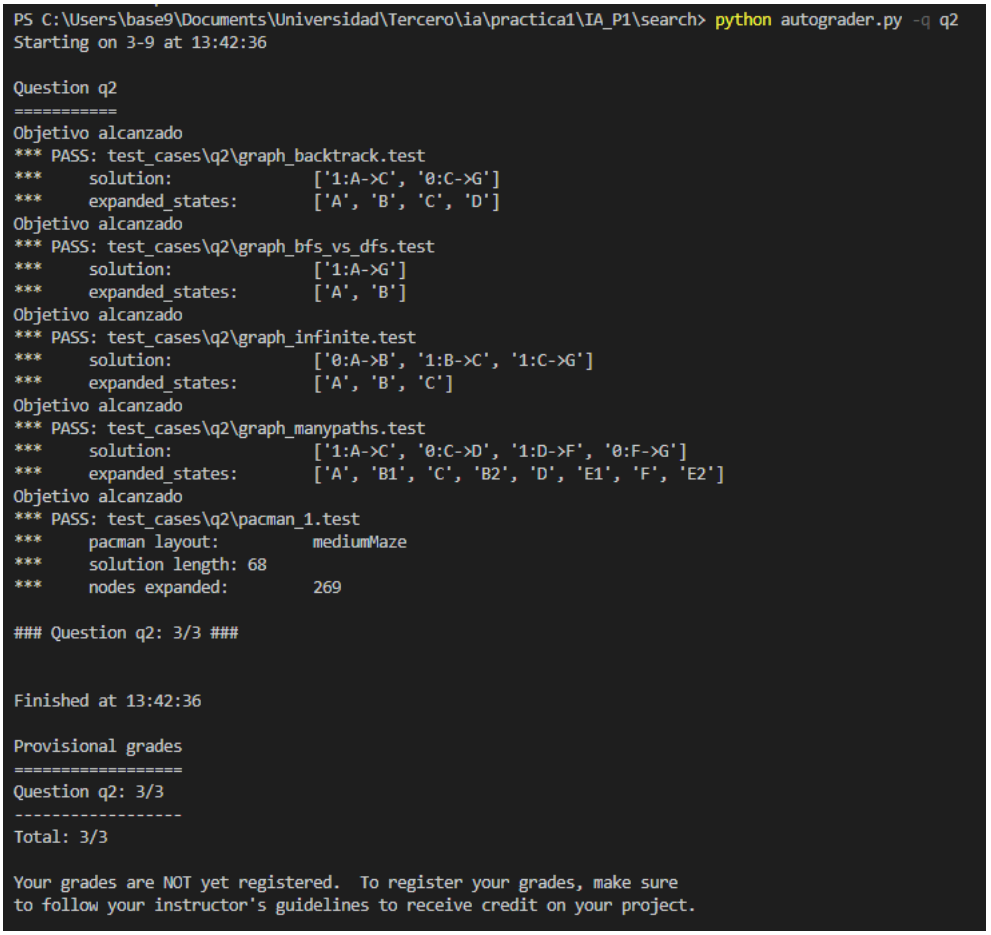

Figura 10: Autograder

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

En este caso, Pacman encuentra y llega al destino de forma mejor que en el algoritmo de profundidad. Además, el camino que encuentra es el camino óptimo.

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

BA encuentra una solución de menor coste que el algoritmo de profundidad gracias a que verifica más nodos y continúa el camino por el más apropiado.

3. Sección 3

3.1. Comentario personal en el enfoque y decisiones de la solución propuesta

Este algoritmo está hecho de la misma forma que los anteriores, pero cambiando la cola o la pila por la cola con prioridad, la cual se ordena automáticamente por prioridades. En la cola introducimos el coste de cada paso junto con el coste total que lleva el estado en el que está actualmente, de forma que se ordena por coste acumulado.

3.2. Lista & explicación de las funciones del framework usadas

- "getStartState()": Devuelve el estado inicial del problema.
- "getSuccessors()": Devuelve los sucesores del estado proporcionado.
- "PriorityQueue()": Inicializa la cola de prioridad.
- "push()": Introduce un elemento en la cola correspondiente, dejándolo arriba en ella.
- "pop()": Saca el elemento de arriba de la cola correspondiente y lo devuelve.

3.3. Incluye el código añadido

Hemos completado el código de la función "uniformCostSearch(problem)"

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    startState = problem.getStartState()
    listaEstadosVisitados = []
    listaEstadosVisitados.append(startState)
    listaCerrados = []
    listaCerrados.append(startState)
    listaAbiertos = util.PriorityQueue()
    listaMovimientos = []
    listaAbiertos.push([(startState, 'Start', 0), listaMovimientos, listaEstadosVisitados, 0], 0.0)

    sucesores = problem.getSuccessors(startState)
    for sucesor in sucesores:
        if sucesor not in listaEstadosVisitados:
            listaAbiertos.push([sucesor, listaMovimientos, listaEstadosVisitados, sucesor[2]], sucesor[2])

    while (listaAbiertos.isEmpty() == False):
        nextState = listaAbiertos.pop()
        if(problem.isGoalState(nextState[0][0]) is True):
            print("Objetivo alcanzado")
            nextState[1].append(nextState[0][1])
            print(nextState[3])
            return nextState[1]
        if nextState[0][0] not in listaCerrados:
            listaCerrados.append(nextState[0][0])
            listaMovimientos = nextState[1].copy()
            listaMovimientos.append(nextState[0][1])
            listaEstadosVisitados = nextState[2].copy()
            listaEstadosVisitados.append(nextState)
            sucesores = problem.getSuccessors(nextState[0][0])
            for sucesor in sucesores:
                if sucesor not in listaEstadosVisitados:
                    prioridad = nextState[3] + sucesor[2]
                    listaAbiertos.push([sucesor, listaMovimientos, listaEstadosVisitados, prioridad], prioridad)

    lista_vacia = []
    return lista_vacia
```

Figura 11: Función de búsqueda de coste uniforme

3.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

Podemos observar que con este algoritmo se van a valorar casi todos los nodos y se va a elegir el mejor camino posible en base a la suma de los costes, para elegir el que sea menor.

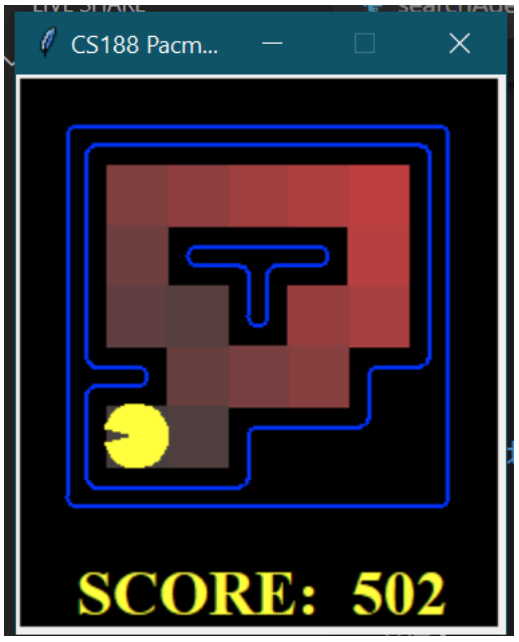

Figura 12: Laberinto pequeño

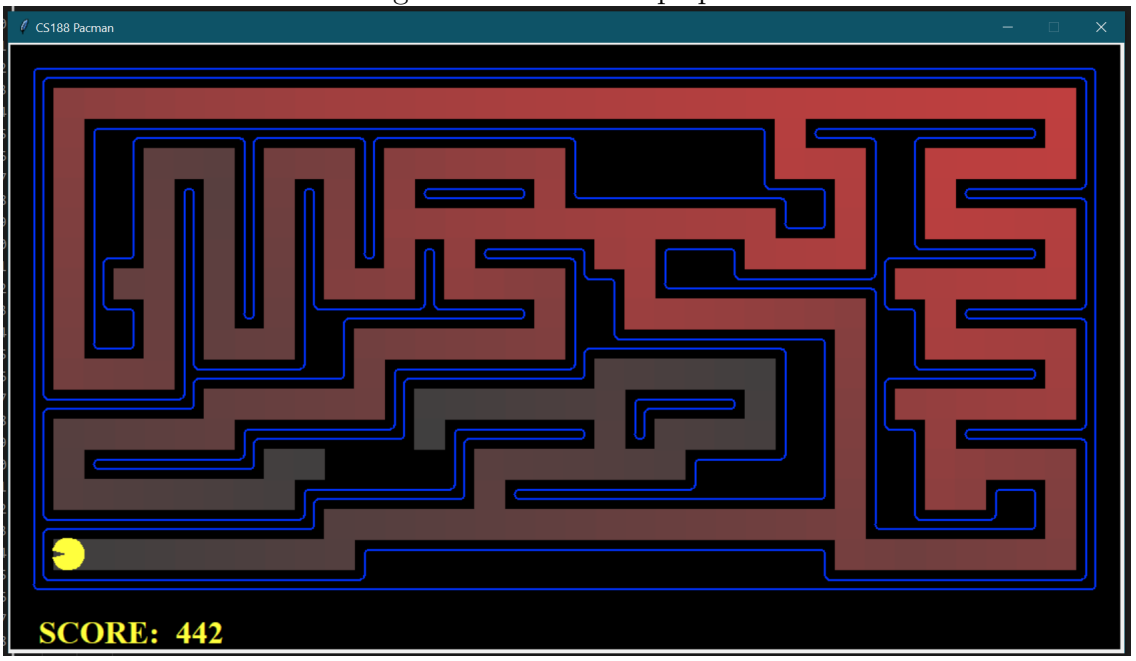

Figura 13: Laberinto mediano

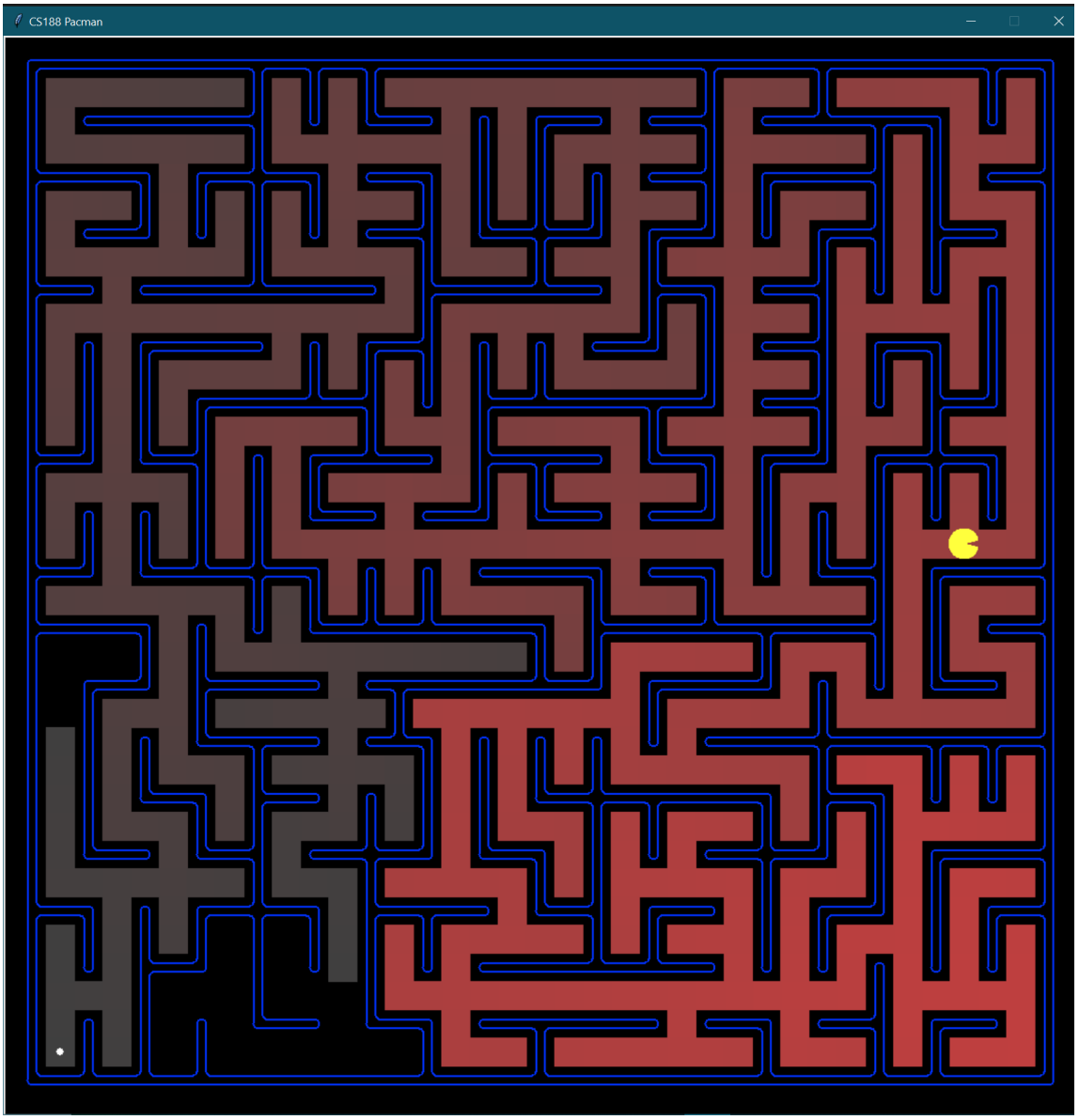

Figura 14: Laberinto grande

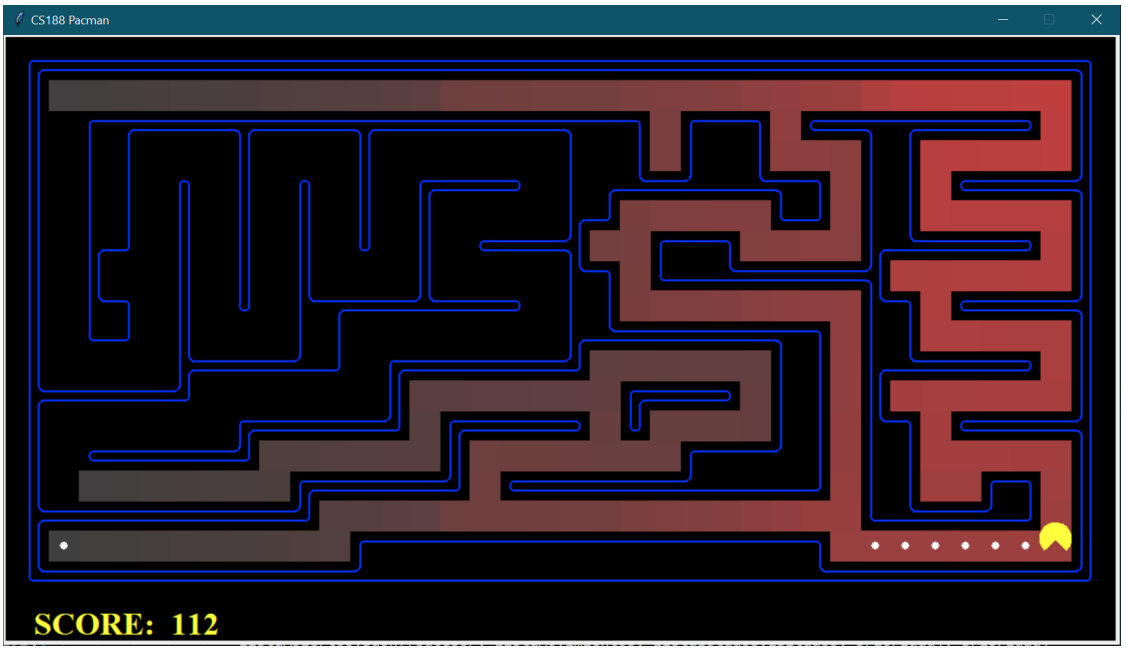

Figura 15: Laberinto con comida

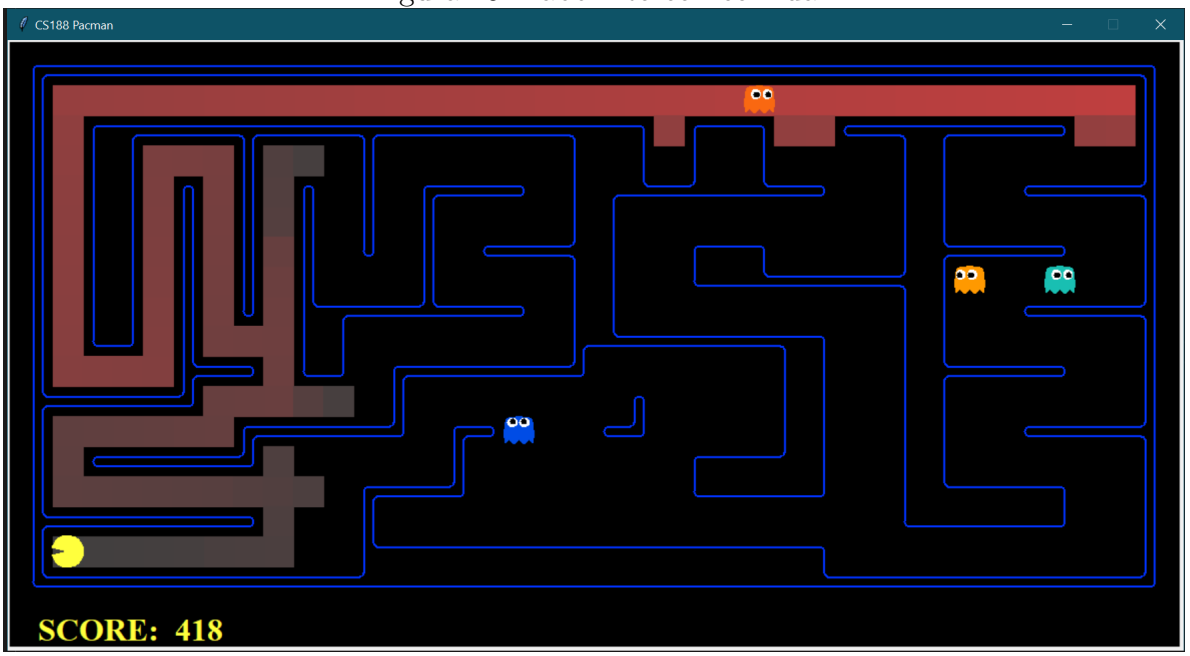

Figura 16: Laberinto con fantasmas

```
Question q3
===========
Objetivo alcanzado
10.0
*** PASS: test_cases\q3\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
Objetivo alcanzado
2.0
*** PASS: test_cases\q3\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
Objetivo alcanzado
21.0
*** PASS: test_cases\q3\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
Objetivo alcanzado
2210.0
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
Objetivo alcanzado
5.0
*** PASS: test_cases\q3\ucs_0_graph.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
Objetivo alcanzado
68
*** PASS: test_cases\q3\ucs_1_problemC.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269
Objetivo alcanzado
1.000976583804004
*** PASS: test_cases\q3\ucs_2_problemE.test
***     pacman layout:          mediumMaze
***     solution length: 74
***     nodes expanded:         260
Objetivo alcanzado
17183280440
*** PASS: test_cases\q3\ucs_3_problemW.test
***     pacman layout:          mediumMaze
***     solution length: 152
***     nodes expanded:         173
Objetivo alcanzado
7
*** PASS: test_cases\q3\ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
Objetivo alcanzado
3.0
*** PASS: test_cases\q3\ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###


Finished at 16:55:19

Provisional grades
==================
Question q3: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

PS C:\Users\Tyxy\Desktop\Uni\IA\Practica1\IA_P1\search>
```

Figura 17: Autograder

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

Pacman llega a la solución sin problema tomando el mismo camino que el algoritmo de búsqueda por anchura. En comparación con los otros algoritmos, ejecuta correctamente los laberintos con fantasmas y con más comida de forma óptima y satisfactoria.

4. Sección 4

4.1. Comentario personal en el enfoque y decisiones de la solución propuesta

Para este ejercicio hemos cogido el código de la función de coste uniforme y hemos cambiado la prioridad de forma que sumamos el coste total del camino más la heurística elegida para la prueba, la cual se pasa por comando en terminal.

4.2. Lista & explicación de las funciones del framework usadas

- "getStartState()": Devuelve el estado inicial del problema.
- "getSuccessors()": Devuelve los sucesores del estado proporcionado.
- "PriorityQueue()": Inicializa la cola de prioridad.
- "push()": Introduce un elemento en la cola correspondiente, dejándolo arriba en ella.
- "pop()": Saca el elemento de arriba de la cola correspondiente y lo devuelve.
- "manhattanHeuristic()": Función que proporciona la distancia Manhattan entre dos puntos.
- "euclideanHeuristic()": Función que proporciona la distancia euclidea entre dos puntos.

4.3. Incluye el código añadido

Hemos completado el código de la función "aStarSearch(problem)"

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node of least total cost first."""
    startState = problem.getStartState()
    listaEstadosVisitados = []
    listaEstadosVisitados.append(startState)
    listaCerrados = []
    listaCerrados.append(startState)
    listaAbiertos = util.PriorityQueue()
    listaMovimientos = []
    heur = heuristic(startState, problem)
    listaAbiertos.push([(startState, 'Start', 0), listaMovimientos, listaEstadosVisitados, heur], heur)

    sucesores = problem.getSuccessors(startState)
    for sucesor in sucesores:
        if sucesor not in listaEstadosVisitados:
            heur = sucesor[2] + heuristic(sucesor[0], problem)
            listaAbiertos.push([sucesor, listaMovimientos, listaEstadosVisitados, heur], heur)

    while (listaAbiertos.isEmpty() == False):
        nextState = listaAbiertos.pop()
        if(problem.isGoalState(nextState[0][0]) is True):
            print("Objetivo alcanzado")
            nextState[1].append(nextState[0][1])
            return nextState[1]
        if nextState[0][0] not in listaCerrados:
            listaCerrados.append(nextState[0][0])
            listaMovimientos = nextState[1].copy()
            listaMovimientos.append(nextState[0][1])
            listaEstadosVisitados = nextState[2].copy()
            listaEstadosVisitados.append(nextState)
            sucesores = problem.getSuccessors(nextState[0][0])
            for sucesor in sucesores:
                if sucesor not in listaEstadosVisitados:
                    camino = problem.getCostOfActions(listaMovimientos) + sucesor[2]
                    heur = camino + heuristic(sucesor[0], problem)
                    listaAbiertos.push([sucesor, listaMovimientos, listaEstadosVisitados, heur], heur)

    lista_vacia = []
    return lista_vacia
```

Figura 18: Función de búsqueda A*

4.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

Probando con las distintas heurísticas hemos llegado a la conclusión de que es más eficaz la heurística de la distancia Manhattan con respecto a la otra en este ejercicio. También es una solución más óptima con respecto a la búsqueda uniforme.

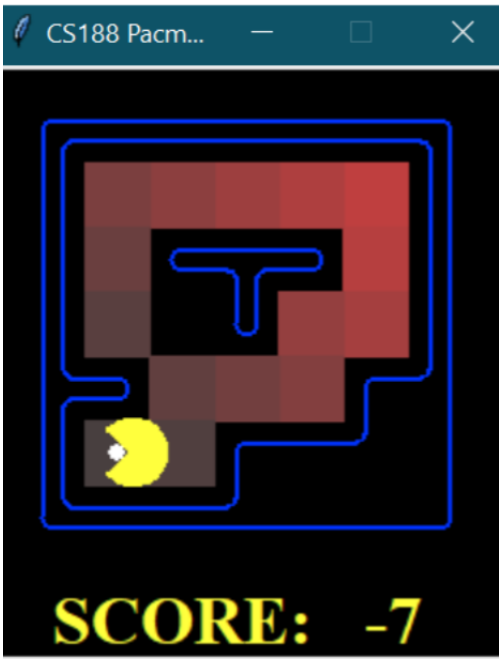

Figura 19: Laberinto pequeño

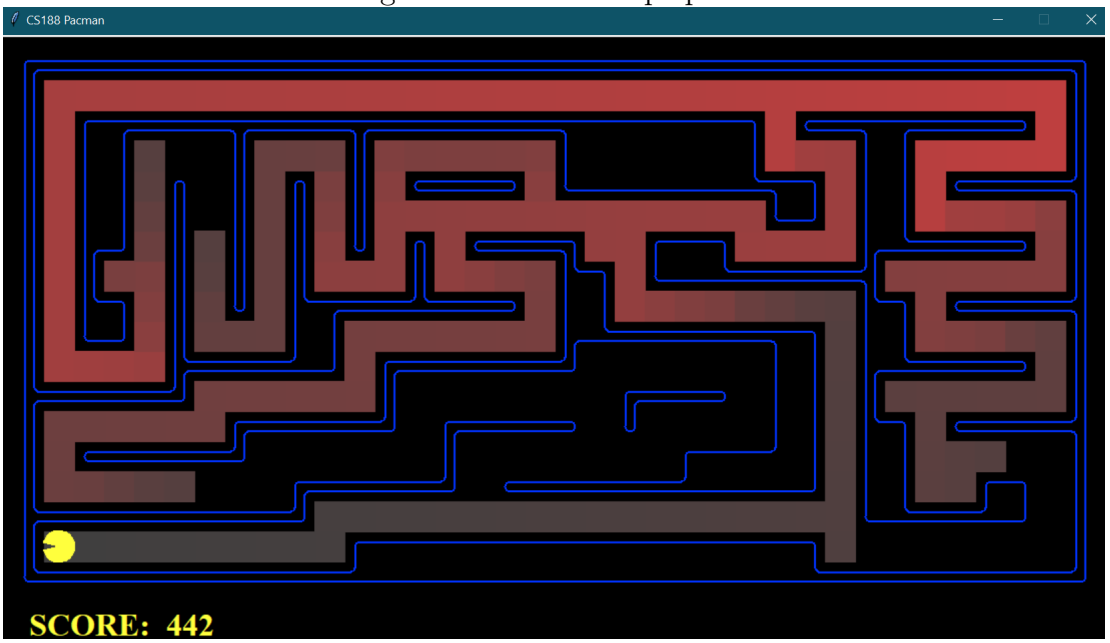

Figura 20: Laberinto mediano

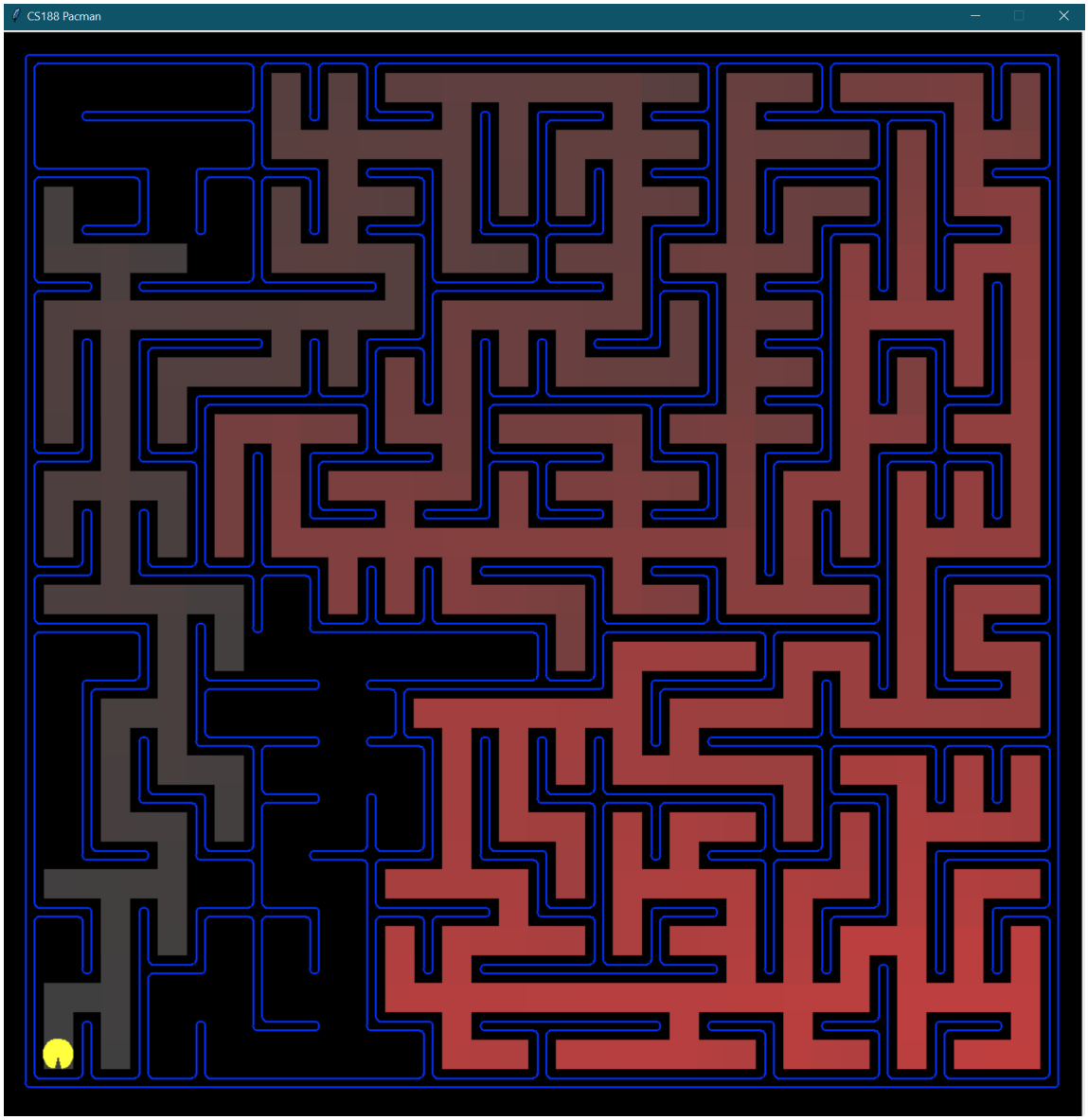

Figura 21: Laberinto grande

```
Question q4
===========
Objetivo alcanzado
*** PASS: test_cases\q4\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
Objetivo alcanzado
*** PASS: test_cases\q4\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
Objetivo alcanzado
*** PASS: test_cases\q4\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
Objetivo alcanzado
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
Objetivo alcanzado
*** PASS: test_cases\q4\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
Objetivo alcanzado
*** PASS: test_cases\q4\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2

### Question q4: 3/3 ###


Finished at 13:52:30

Provisional grades
==================
Question q4: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

PS C:\Users\Tyxy\Desktop\Uni\IA\Practica1\IA_P1\search>
```

Figura 22: Autograder

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

Pacman llega a la solución de forma satisfactoria, además usando el camino más óptimo para ello gracias al algoritmo A* cuando se usa la heurística de la distancia Manhattan, con la euclidea es más costoso.

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

Todas las estrategias salvo DepthFirst van por el mismo camino, siendo el más óptimo. En cambio, DepthFirst pasa primero por varias filas innecesarias en el camino.

5. Sección 5

5.1. Comentario personal en el enfoque y decisiones de la solución propuesta

Hemos usado la estrategia BreadthFirstSearch como método de búsqueda y hemos modificado el state

5.2. Lista & explicación de las funciones del framework usadas

- "directionToVector()": Obtiene las coordenadas del siguiente punto.

5.3. Incluye el código añadido

Hemos completado el código de la función "aStarSearch(problem)"

```python
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    esquinas = []
    state = (self.startingPosition[0], self.startingPosition[1], esquinas)
    return state

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    if len(state[2]) == 4:
        return True
    return False
```

Figura 23: Primeras funciones de la clase CornersProblem

```python
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]
        x = state[0]
        y = state[1]
        z = state[2].copy()
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            if (nextx, nexty) in self.corners and (nextx, nexty) not in z:
                z.append((nextx, nexty))
            successors.append( ( (nextx, nexty, z), action, 1) )

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

Figura 24: Segundas funciones de la clase CornersProblem

5.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

Los resultados son óptimos con respecto al enunciado propuesto, aunque no es lo ideal para este tipo de ejercicio de laberintos.

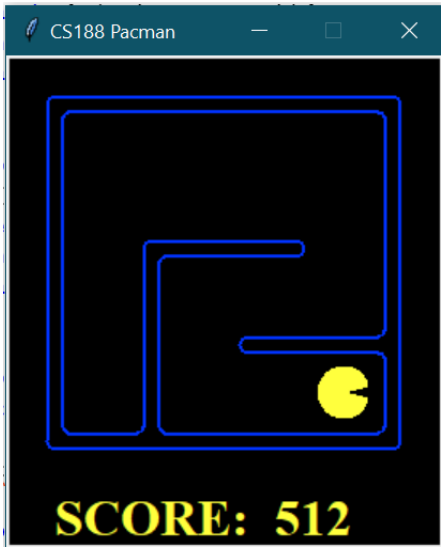

Figura 25: Laberinto pequeño

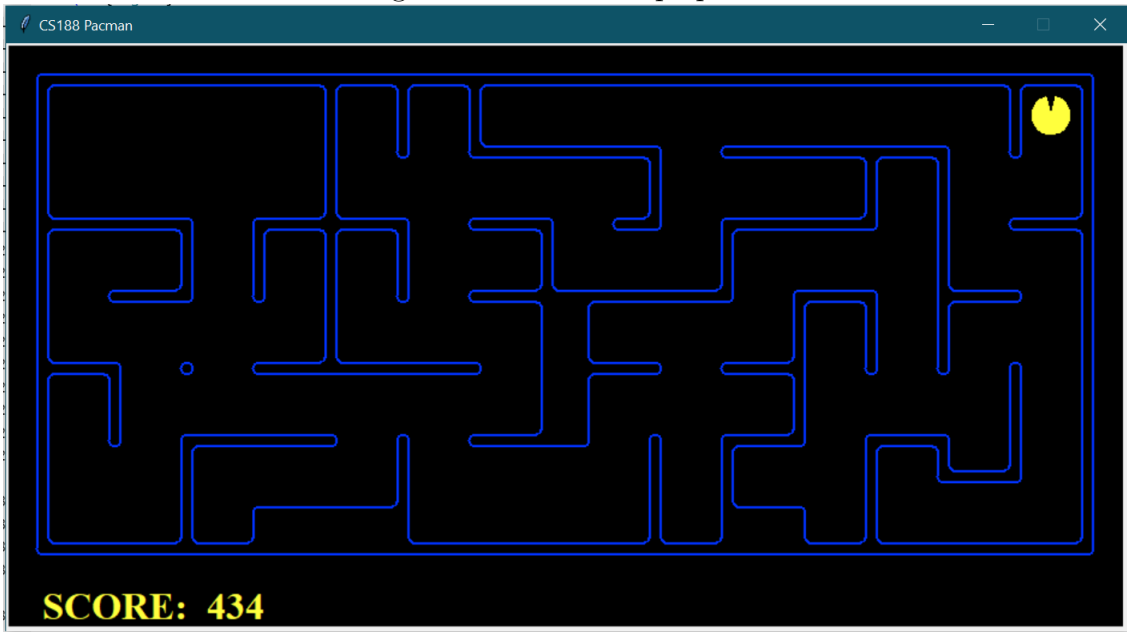

Figura 26: Laberinto mediano

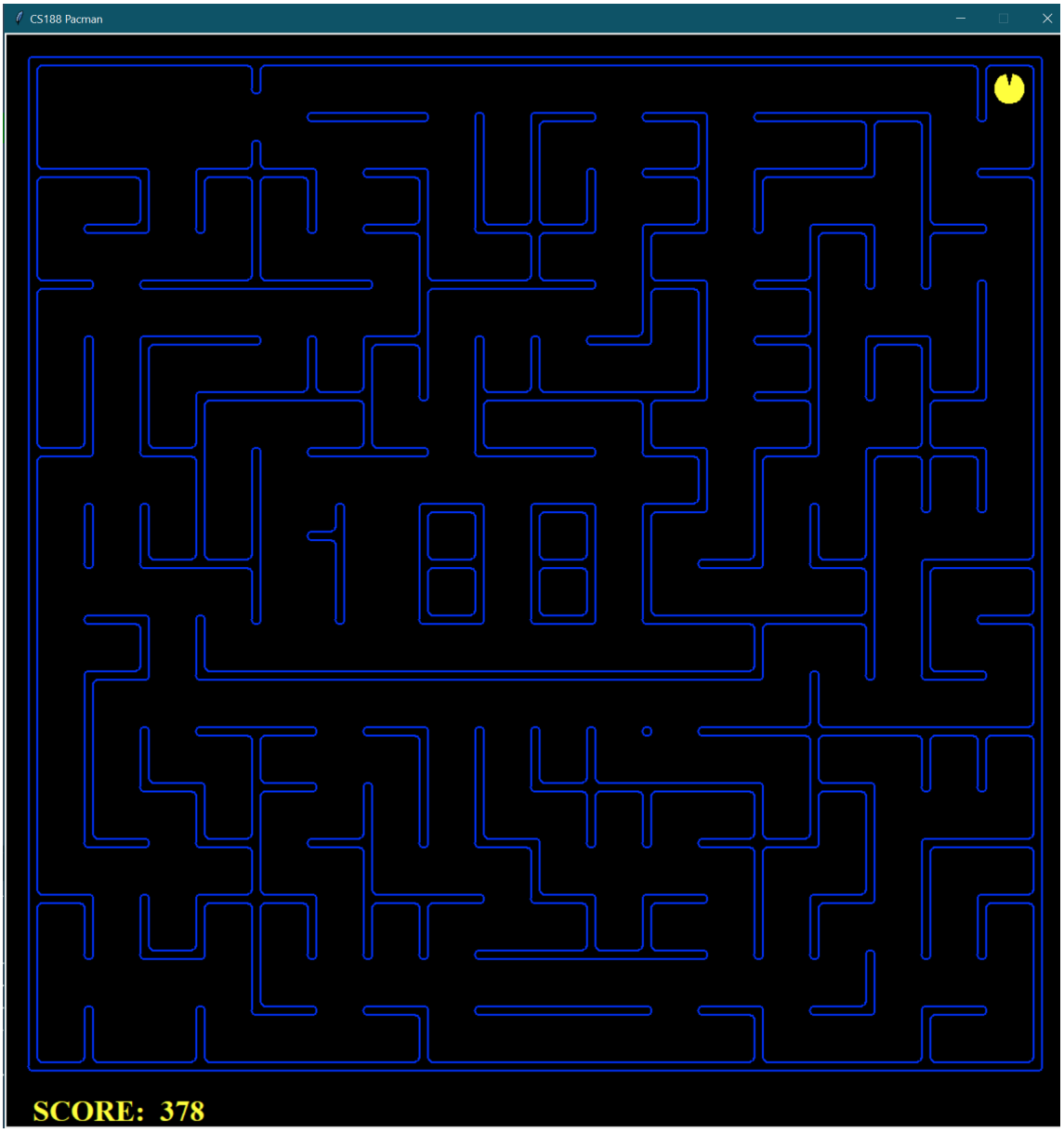

Figura 27: Laberinto grande

```
Question q2
===========
Objetivo alcanzado
*** PASS: test_cases\q2\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
Objetivo alcanzado
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
Objetivo alcanzado
*** PASS: test_cases\q2\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
Objetivo alcanzado
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
Objetivo alcanzado
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###


Question q5
===========
Objetivo alcanzado
*** PASS: test_cases\q5\corner_tiny_corner.test
***     pacman layout:          tinyCorner
***     solution length:                28

### Question q5: 3/3 ###


Finished at 17:01:36

Provisional grades
==================
Question q2: 3/3
Question q5: 3/3
------------------
Total: 6/6

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Figura 28: Autograder

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

Pacman llega a todas las esquinas de forma óptima. Suele ir en orden sin cruzarse en diagonal por el laberinto. Además, expande los nodos indicados en el enunciado de la práctica.

6. Sección 6

6.1. Comentario personal en el enfoque y decisiones de la solución propuesta

6.2. Lista explicación de las funciones del framework usadas

6.3. Incluye el código añadido

6.4. Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados

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

6.6. Explica la lógica de tu heurística.

Tras investigar distintas heurísticas (ya que necesitábamos una con la que poder ir a múltiples objetivos) y distintos métodos, llegamos a la conclusión de que necesitábamos calcular la distancia absoluta (o distancia Mahattan) del punto en el que se encontraba el *Pacman* y las esquinas del mapa

Para eso, calculamos la distancia Manhattan desde la posición en la que nos encontramos hasta las distintas esquinas, y eso lo hacemos por cada una no visitada del mapa. Mientras tanto, cogemos la mínima distancia de cada iteración para cada esquina y sustituimos la posición del *Pacman* por la siguiente esquina visitada.

Se van sumando esas cantidades y obtenemos la mínima distancia que se necesita para pasar por cada esquina no visitada desde el punto inicial, pasado por argumento.

7. Sección 7

Comentarios personales de la realización de esta práctica Consideramos que hemos aprendido mucho sobre la implementación de distintos algoritmos en programación que ya habíamos visto previamente, tanto en otros años y asignaturas como en la propia asignatura de Inteligencia Artificial. Confíamos en que nos ha proporcionado una buena base para seguir

aprendiendo algoritmos dedicados al aprendizaje y otros aspectos de la IA.