# PRACTICA 1 – Algoritmos de Búsqueda en: PACMAN

**Título: Memoria Practica 1 Grupo 2301 Pareja nº 15**

**Autores: Saul Almazan de Pie & Guillermo Hoyo Bravo.**

## Sección 1 – Búsqueda Primero en Profundidad (depthFirstSearch())

* 1. **Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)**

Para implementar el algoritmo de Búsqueda primero en profundidad, nos hemos decidido por usar una pila. Introducimos el nodo inicial hacemos un bucle que termina cuando la pila esta vacías, es decir que hemos llegado al último de los nodos hoja. Para implementar la eliminación de estados repetidos guardamos el nodo en una lista llamada Visitados, y si no ha sido visitado lo introducimos a la pila.

* 1. **Lista & explicación de las funciones del framework usadas (1pt)**
* Class Stack: à util.Stack()
* def push(self,item): à Pila.push() à Pila.push((problem.getStartState(),[],[]))
* def getStartState(self): à problem.getStartState(),[],[]
* def isEmpty(self): à Pila.isEmpty()
* def pop(self): à Pila.pop()
* def isGoalState(self, state): à problema.isGoalState(nodo)
* def getSuccessors(self, state): à problema.getSuccessors(nodo)
  1. **Incluye el código añadido (0.25 pts)**

def depthFirstSearch(problem):

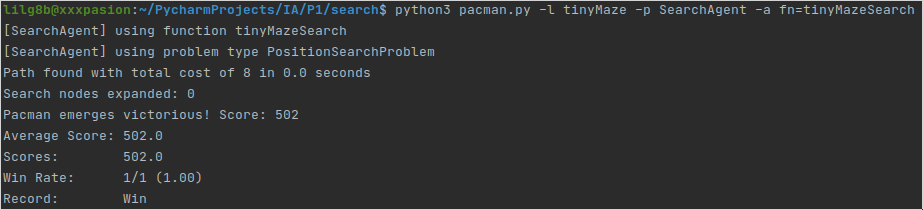
*acciones = []*  
 *aux = []*  
 *Nodos = util.Stack()*  
 *Nodos.push((problem.getStartState(),[],[]))*

while not *Nodos.isEmpty():*  
 *nodo, accion , visitado = Nodos.pop()*  
  if *problem.isGoalState(nodo):*  
  return *acciones*

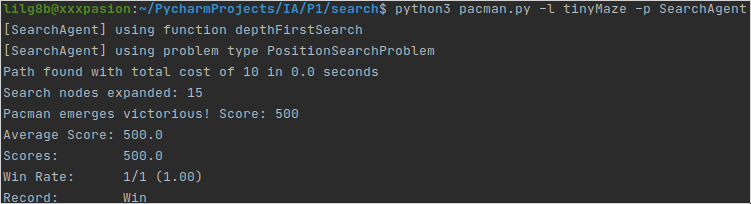
for *hijo , direcciones , pasos* in *problem.getSuccessors(nodo):*  
  if not *hijo* in *visitado:*  
   *Nodos.push((hijo,accion+[direcciones],visitado+[nodo]))*  
 *acciones = accion+[direcciones]*

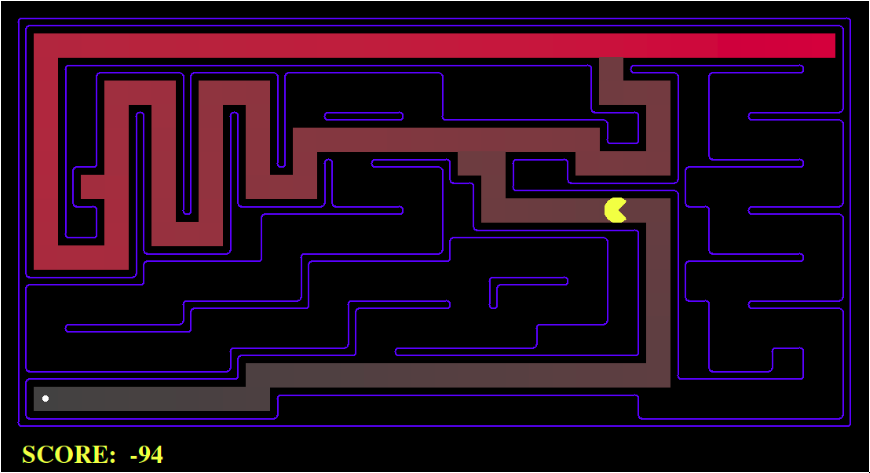
return None

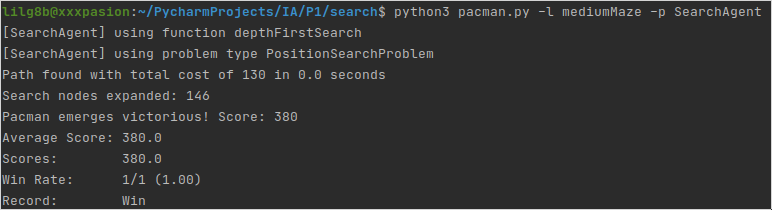
* 1. **Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)**

**Commando**: $ python3 pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch   


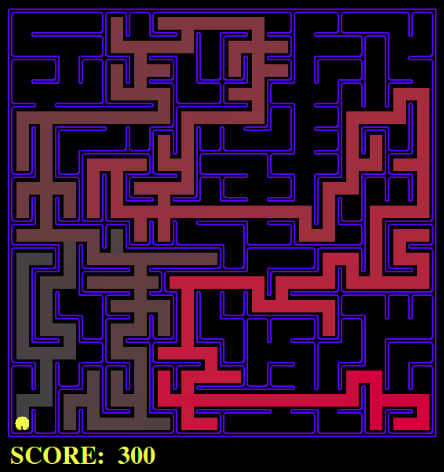
**Commando**: $ python3 pacman.py -l tinyMaze -p SearchAgent

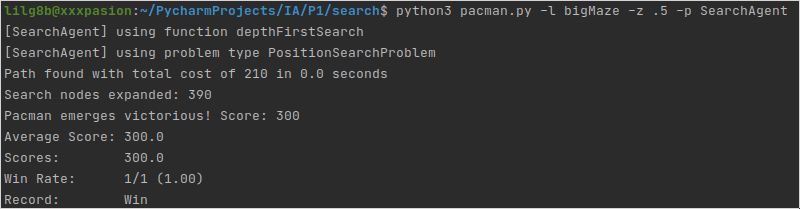


**Commando**: $ python3 pacman.py -l mediumMaze -p SearchAgent 

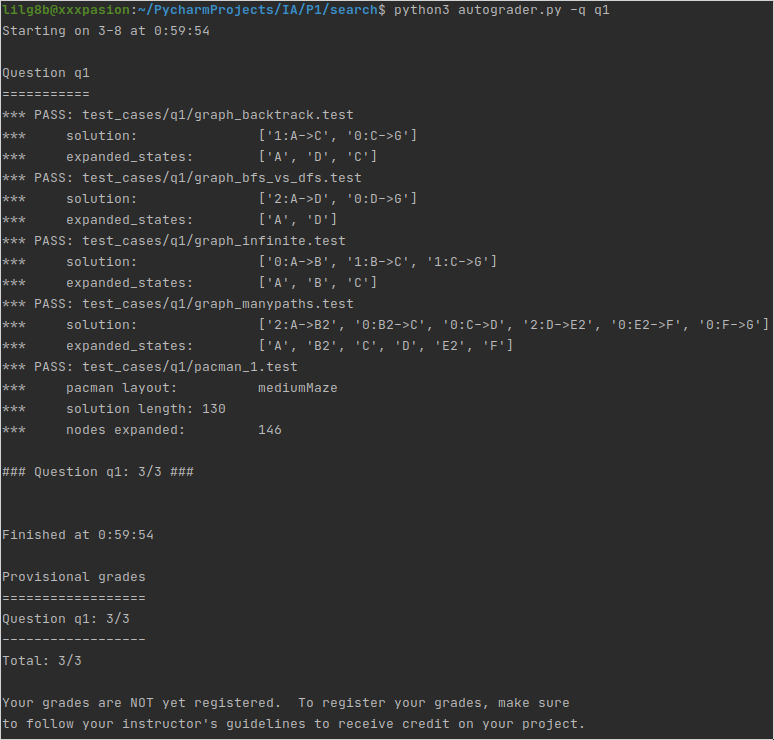


**Commando**: $ python3 pacman.py -l bigMaze -z .5 -p SearchAgent





**Commando**: $ python3 autograder.py -q q1



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

Tiny Maze:

Coste: 10

Nodos Expandidos: 15

Resultado Medio/ Resultado: 500 / 500

Llega a la solución.

Medium Maze:

Coste: 130

Nodos Expandidos: 149

Resultado Medio/ Resultado: 380 / 380

Llega a la solución.

Big Maze:

Coste: 210

Nodos Expandidos: 390

Resultado Medio/ Resultado: 300 / 300

Llega a la solución.

Longitud = 130

Nodos Expandidos = 146

Total: 3/3

**Dado que la heurística es monótona, también es admisible.**  
**Con heurística admisible, sin eliminar estados repetidos = óptima**

* 1. **Respuesta a pregunta 1.1 (1pt)**

**¿El orden de exploración es el que esperabais?**

Si, en profundidad

* 1. **Respuesta a pregunta 1.2 (1pt)**

**¿Pacman realmente va a todas las casillas exploradas en su camino hacia la meta?**

No, solo pasa por las necesarias para encontrar la solución en profundidad.

* 1. **Respuesta a pregunta 2 (1pt)**

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

El algoritmo de búsqueda en profundidad no asegura la solución óptima, aunque en este caso si lo logra.

## Sección 2 – Búsqueda en Anchura (breadthFirstSearch(problem))

* 1. **Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)**

Para este algoritmo hemos utilizado el enfoque usado en la Busqueda del primero en profundidad, ya que son muy parecidos. La diferencia aquí es que nos decidimos a usar una cola en vez de continuar con la pila. Este cambio lo propone el mismo algoritmo, que se basa en investigar los nodos por el orden en el que se descubren. De este modo, vamos investigando cada nodo en una misma profundidad, hasta llegar a la solución o al final del árbol.

* 1. **Lista & explicación de las funciones del framework usadas (1pt)**
* Class Queue: à util.Queue ()
* def push(self,item): à Queue.push() à Queue.push((problem.getStartState(),[],[]))
* def getStartState(self): à problem.getStartState(),[],[]
* def isEmpty(self): à Queue.isEmpty()
* def pop(self): à Queue.pop()
* def isGoalState(self, state): à problema.isGoalState(nodo)
* def getSuccessors(self, state): à problema.getSuccessors(nodo)
  1. **Incluye el código añadido (0.25 pts)**

def breadthFirstSearch(problem):

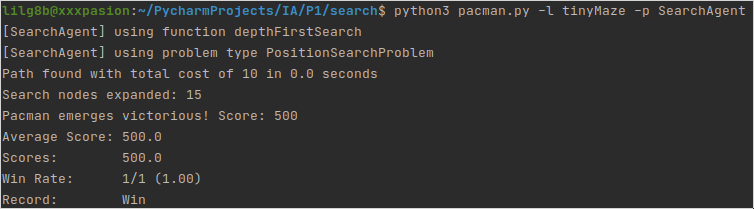
*aux = []*  
 *Nodos = util.Queue()*  
 *Nodos.push((problem.getStartState(), [], []))*

while not *Nodos.isEmpty():*  
 *nodo, accion, coste = Nodos.pop()*  
 if not *nodo* in *aux:*  
 *aux.append(nodo)*  
 if *problem.isGoalState(nodo):*  
 return *accion*

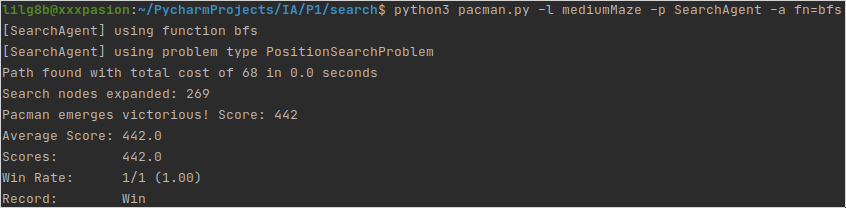
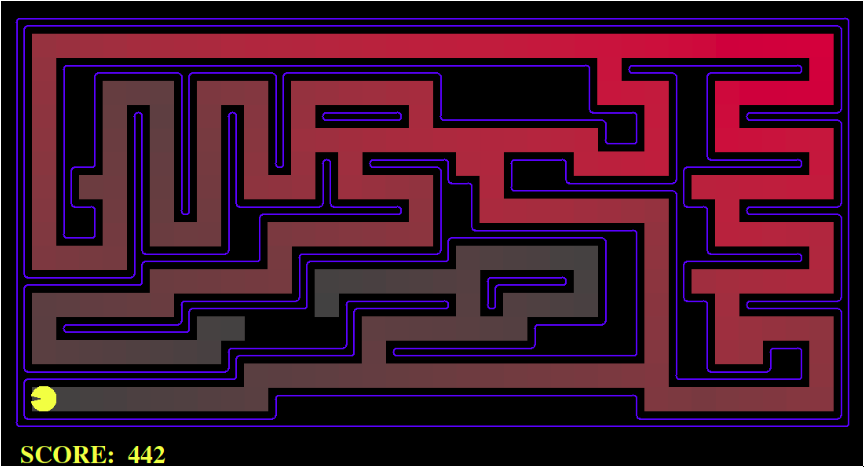
for *hijo, direcciones, pasos* in *problem.getSuccessors(nodo):*  
  *Nodos.push((hijo, accion + [direcciones], coste + [pasos]))*  
  
 return None

* 1. **Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)**

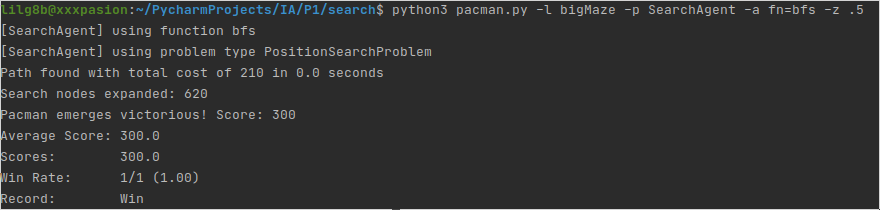
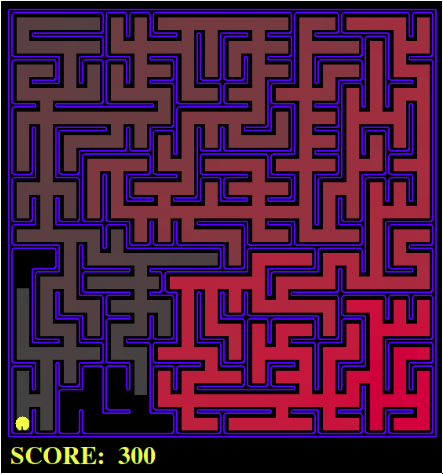
**Comando:** $ python3 pacman.py -l tinyMaze -p SearchAgent



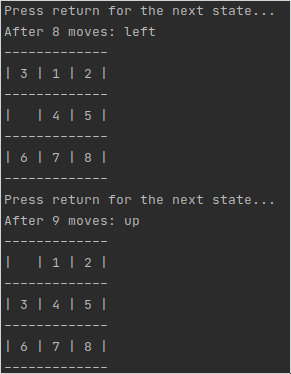
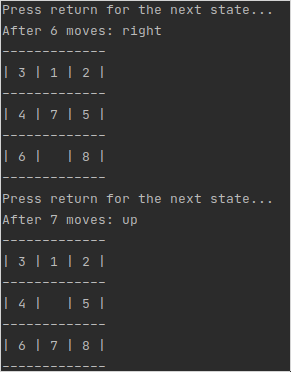
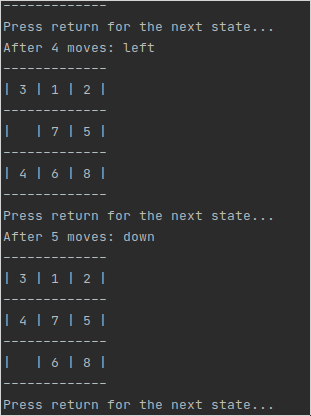
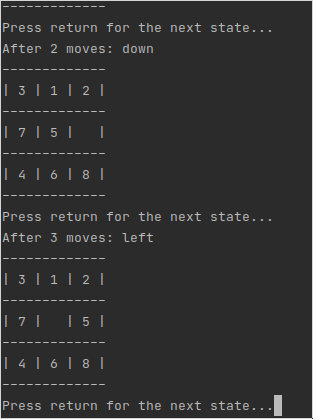
**Comando:** $ python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs



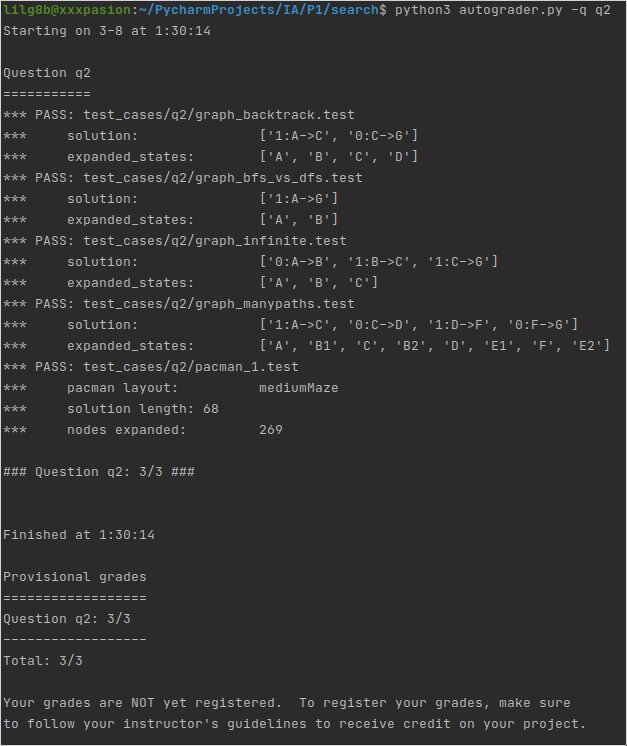
**Comando:** $ python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5



**Comando:** $ python3 eightpuzzle.py



**Comando:** $ python3 autograder.py -q q2



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

Tiny Maze:

Coste: 10

Nodos Expandidos: 15

Resultado Medio/ Resultado: 500 / 500

LLega a la solución.

Medium Maze:

Coste: 68

Nodos Expandidos: 269

Resultado Medio/ Resultado: 442 / 442

LLega a la solución.

Big Maze:

Coste: 210

Nodos Expandidos: 620

Resultado Medio/ Resultado: 300 / 300

LLega a la solución.

Longitud = 68

Nodos Expandidos = 269

Total: 3/3

**Dado que la heurística es monótona, también es admisible.**  
**Con heurística admisible, sin eliminar estados repetidos = optima**

* 1. **Respuesta a pregunta 3 (1pt)**

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

El algoritmo de búsqueda en anchura no asegura la solución óptima, aunque en este caso si lo logra.

## Sección 3 – Variar la Función de Coste (uniformCostSearch(problem))

* 1. **Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)**

Este algoritmo es muy parecido a los dos anteriores que hemos explicado. En este caso nos hemos decidido por usar la estructura de la cola de prioridad, ya que este algoritmo en vez de darle la prioridad de exploración al último nodo encontrado para investigar una rama hasta su profundidad máxima y continuar por la siguiente, o investigar cada nivel de profundidad por completo antes de pasar al siguiente, expande primero el nodo de menor coste, obteniendo la solución óptima.

* 1. **Lista & explicación de las funciones del framework usadas (1pt)**
* Class Queue: à util.PriorityQueue()
* def push(self,item): à PriorityQueue.push() à PriorityQueue.push((problem.getStartState(),[],0),0)
* def getStartState(self): à problem.getStartState(),[],[]
* def isEmpty(self): à PriorityQueue.isEmpty()
* def pop(self): à PriorityQueue.pop()
* def isGoalState(self, state): à problema.isGoalState(nodo)
* def getSuccessors(self, state): à problema.getSuccessors(nodo)
  1. **Incluye el código añadido (0.25 pts)**

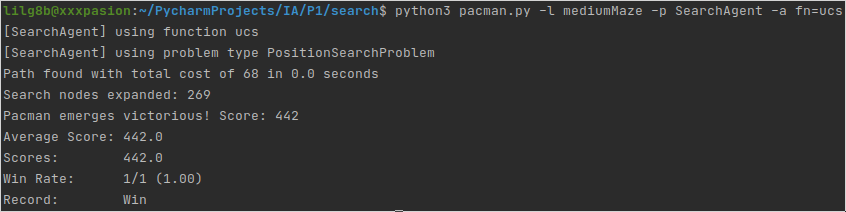
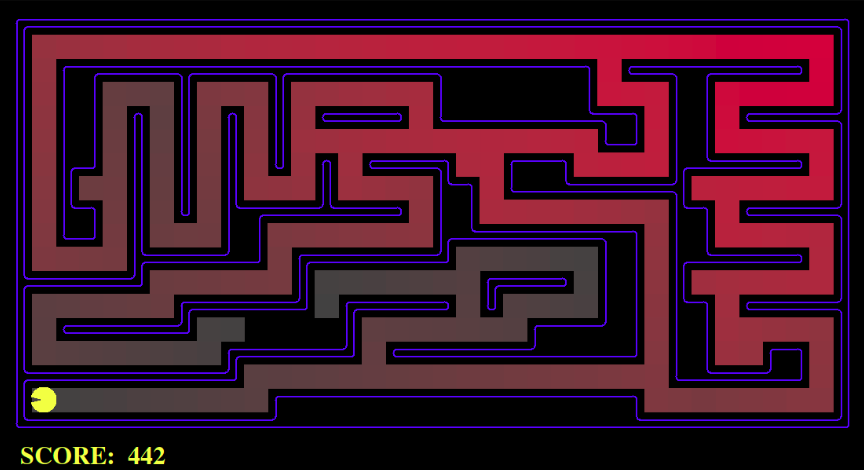
def uniformCostSearch(problem):

*aux = []*  
 *Nodos = util.PriorityQueue()*  
 *Nodos.push((problem.getStartState(), [], 0),0)*  
 while not *Nodos.isEmpty():*  
 *nodo, accion, coste = Nodos.pop()*  
 if not *nodo* in *aux:*  
  *aux.append(nodo)*  
 if *problem.isGoalState(nodo):*  
 return *accion*

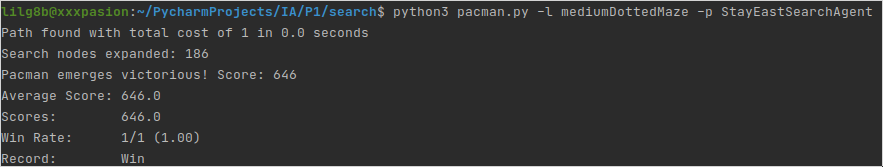
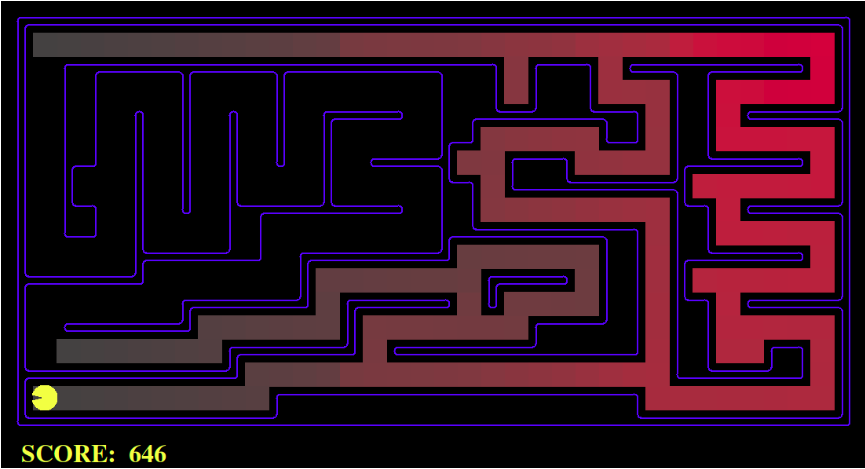
for *hijo, direcciones, pasos* in *problem.getSuccessors(nodo):*  
 *Nodos.push((hijo, accion + [direcciones], coste + pasos),coste+pasos)*  
  
 return None

* 1. **Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)**

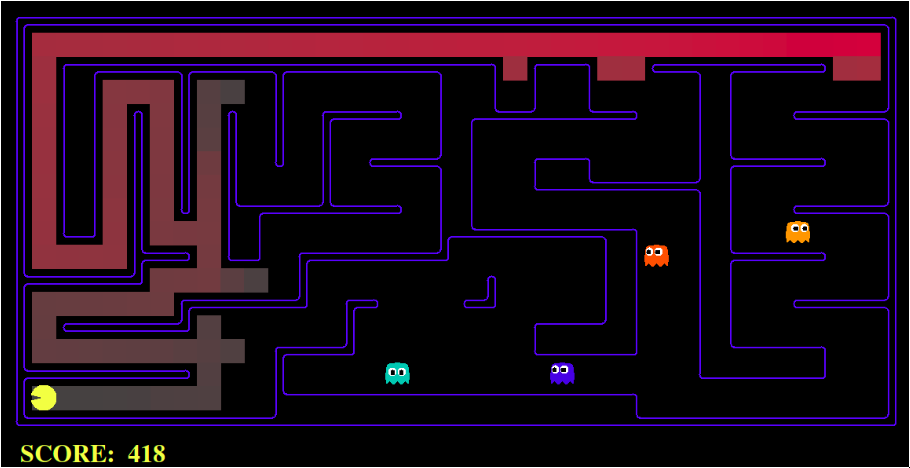
**Commando**: python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

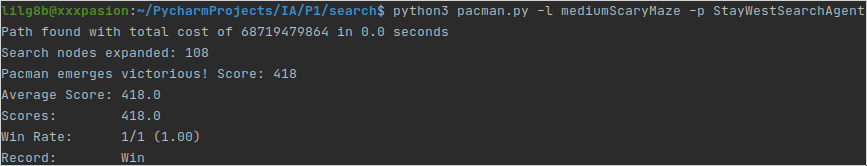


**Commando**: python3 pacman.py -l mediumDottedMaze -p StayEastSearchAgent

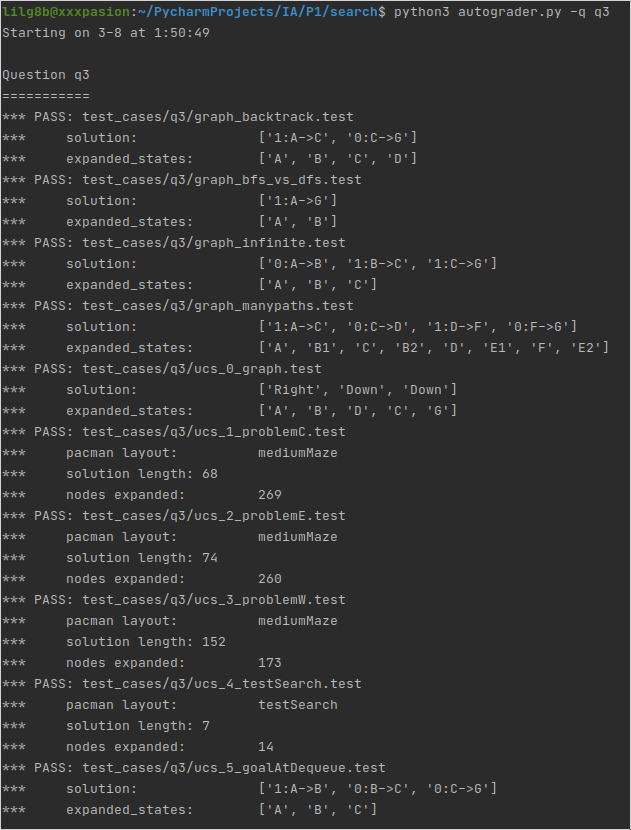


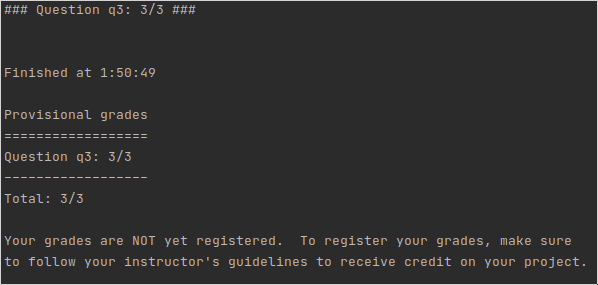
**Commando**: python3 pacman.py -l mediumScaryMaze -p StayWestSearchAgent





**Commando**: python3 autograder.py -q q3





* 1. **Conclusiones en el comportamiento de pacman, es óptimo (s/n), llega a la solución (s/n), nodos que expande, etc (1pt)**

Medium Maze SearchAgent:

Coste: 68

Nodos Expandidos: 269

Resultado Medio/ Resultado: 442 / 442

LLega a la solución.

Medium Maze East:

Coste: 1

Nodos Expandidos: 186

Resultado Medio/ Resultado: 646 / 646

LLega a la solución.

Medium Maze West:

Coste: (Infito)

Nodos Expandidos: 108

Resultado Medio/ Resultado: 418 / 418

LLega a la solución.

Total: 3/3

**Dado que la heurística es monótona, también es admisible.**  
**Con heurística admisible, sin eliminar estados repetidos = optima**

## Sección 4 – Busqueda A\* (aStarSearch(problema, heuristic=nullHeuristic))

* 1. **Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)**

Este algoritmo es la suma del algoritmo desarrollado en el punto 3 (Coste Uniforme) con una heurística. Literalmente este algoritmo es guiado por el coste mínimo de la siguiente función:

**F(n) = G(n) + H(n)**

El coste mínimo es la suma de el coste hasta el nodo n (G(n)) y el valor de la heurística para el nodo n (H(n))

Por lo que hemos usado el código implementado en el punto 3, sumando en las partes necesarias el coste adicional que proporciona la función

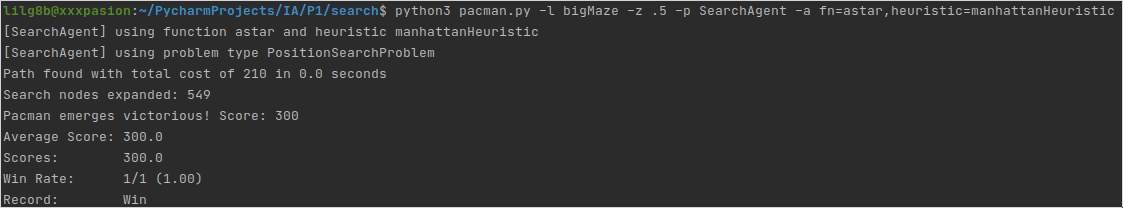
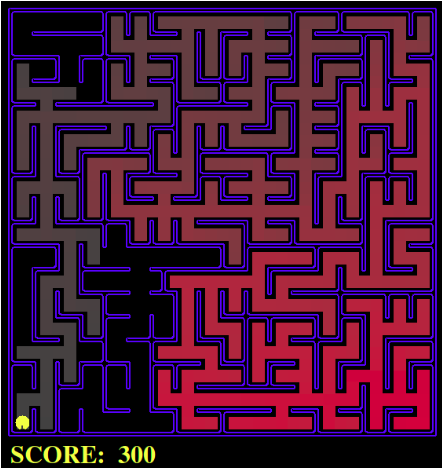
* 1. **Lista & explicación de las funciones del framework usadas (1pt)**
* Class Queue: à util.PriorityQueue()
* def push(self,item): à PriorityQueue.push() à PriorityQueue.push((problem.getStartState(),[],0), heuristic(problema.getStartState(), problem))
* def nullHeuristic(state, problema=None): à heuristic=nullHeuristic à heuristic(problema.getStartState(), problem)
* def getStartState(self): à problem.getStartState(),[],[]
* def isEmpty(self): à PriorityQueue.isEmpty()
* def pop(self): à PriorityQueue.pop()
* def isGoalState(self, state): à problema.isGoalState(nodo)
* def getSuccessors(self, state): à problema.getSuccessors(nodo)
  1. **Incluye el código añadido (0.25 pts)**

def aStarSearch(problem, heuristic=nullHeuristic):

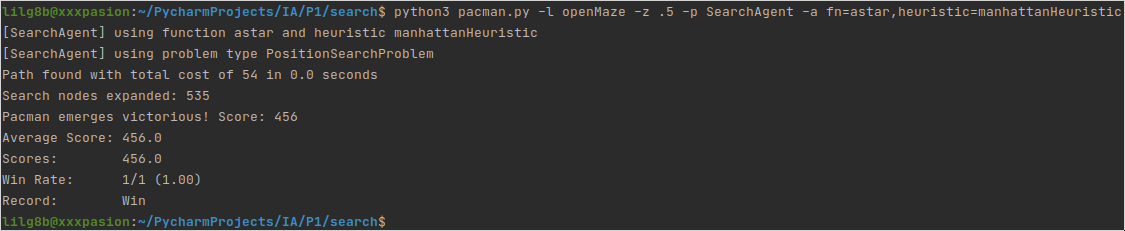
*"""Search the node that has the lowest combined cost and heuristic first."""*  
 *Abiertos = util.PriorityQueue()*  
 *Cerrados = []*  
 *"""Nodo Origen en lista abiertos"""*  
 *Abiertos.push((problem.getStartState(), [], 0), heuristic(problem.getStartState(), problem))*  
while not *Abiertos.isEmpty():*  
 *nodo, accion, coste = Abiertos.pop()*  
  
  if not *nodo* in *Cerrados:*  
 *Cerrados.append(nodo)*  
  
  if *problem.isGoalState(nodo):*  
return *accion*  
  
  for *hijo, direcciones, pasos in problem.getSuccessors(nodo):*  
 *g = coste + pasos* # f = g + h  
  *Abiertos.push((hijo, accion + [direcciones], coste + pasos), g + heuristic(hijo, problem))*  
  
return None

* 1. **Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)**

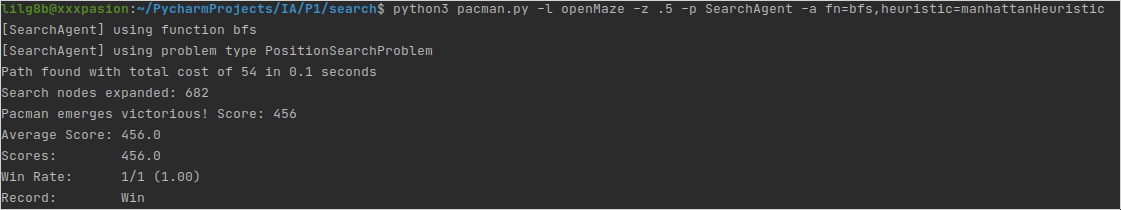
**Comando A\***: $ python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic



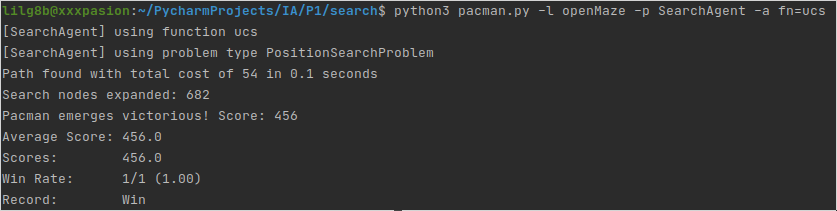
**A\* Open Maze --> Comando:** $ python3 pacman.py -l openMaze -z .5 -p SearchAgent fn=bfs fn=astar,heuristic=manhattanHeuristic



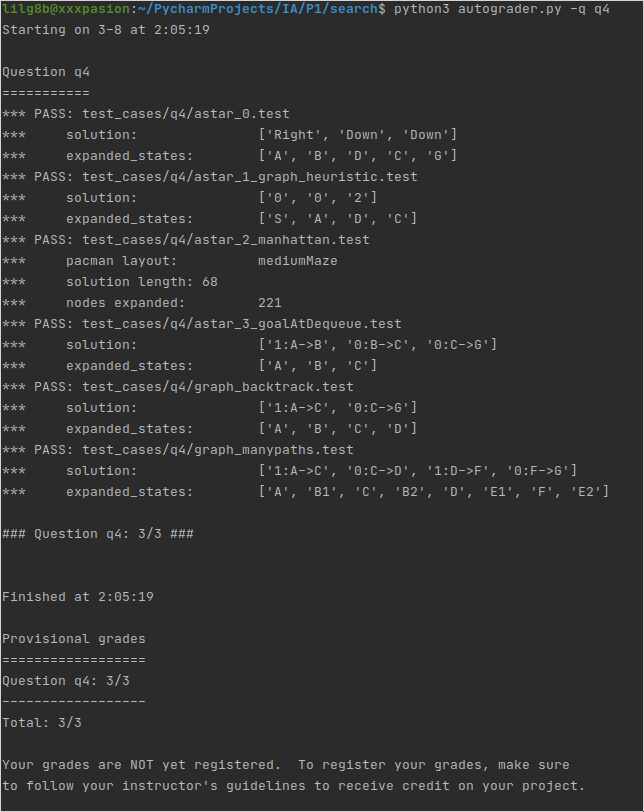
**Anchura Open Maze --> Comando:** $ python3 pacman.py -l openMaze -z .5 -p SearchAgent -a fn=bfs



CosteUniforme Open Maze --> Comando: $ python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs



**Comando**: $ python3 autograder.py -q q4



* 1. **Conclusiones en el comportamiento de pacman, es óptimo (s/n), llega a la solución (s/n), nodos que expande, etc (1pt)**

Sí, A\* sin repetición de estados siempre encuentra la solución Óptima.

Big Maze:

Coste: 210

Nodos Expandidos: 549

Resultado Medio/ Resultado: 300/ 300

Llega a la solución.

Longitud = 68

Nodos expandidos = 221

Total: 3/3

**Dado que la heurística es monótona, también es admisible.**  
 **A\* con heurística admisible, sin eliminar estados repetidos = optima**

* 1. **Respuesta a pregunta 4 (1pt)**

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

Se observa que A\* expande menos nodos y las otras 2 estrategias probadas expande mas nodos. Lo esperado.

## Sección 5 – Encontrar todas las Esquinas ()

* 1. **Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)**

Hemos completado las funciones de la clase corner problem. La funcion de isGoalState() buscamos que el número de esquinas visitadas sea igual a 4 para terminar correctamente.

* 1. **Lista & explicación de las funciones del framework usadas (1pt)**
  + def getStartState(self):
  + def isGoalState(self, state):
  + def getSuccessors(self, state):
  1. **Incluye el código añadido (0.25 pts)**

**def getStartState(self):**

return ( self.startingPosition, [])

**def isGoalState(self, state):**

*"""*

*Returns whether this search state is a goal state of the problem.*

*"""*

nodo = state[0]

esquinas = state[1]

if nodo in self.corners:

if not nodo in esquinas:

esquinas.append(nodo)

return len(esquinas) == 4

return False

**def getSuccessors(self, state):**

x,y = state[0]

esquinas = state[1]

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]*

dx, dy = Actions.directionToVector(action)

nextx, nexty = int(x+dx), int(y+dy)

pared = self.walls[nextx][nexty]

if not pared:

siguiente = list(esquinas)

nextState = (nextx, nexty)

if (not nextState in siguiente) & (nextState in self.corners):

siguiente.append(nextState)

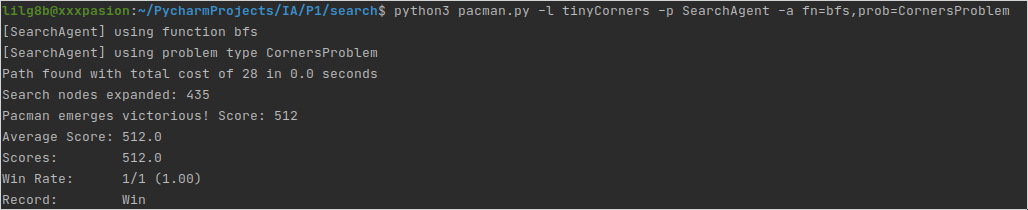
successors.append(((nextState, siguiente), action, 1))

self.\_expanded += 1 # DO NOT CHANGE

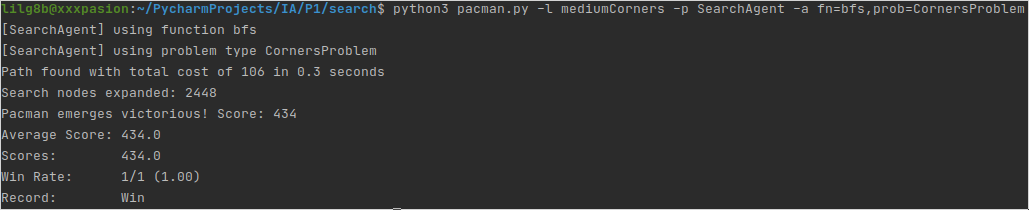
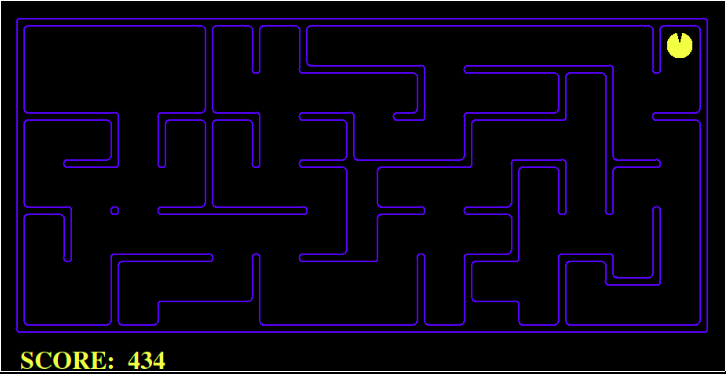
return successors

* 1. **Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)**

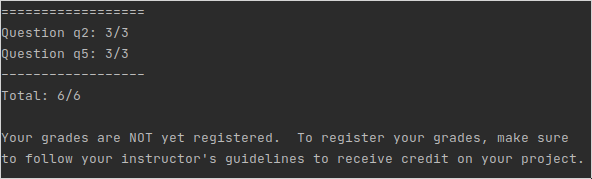
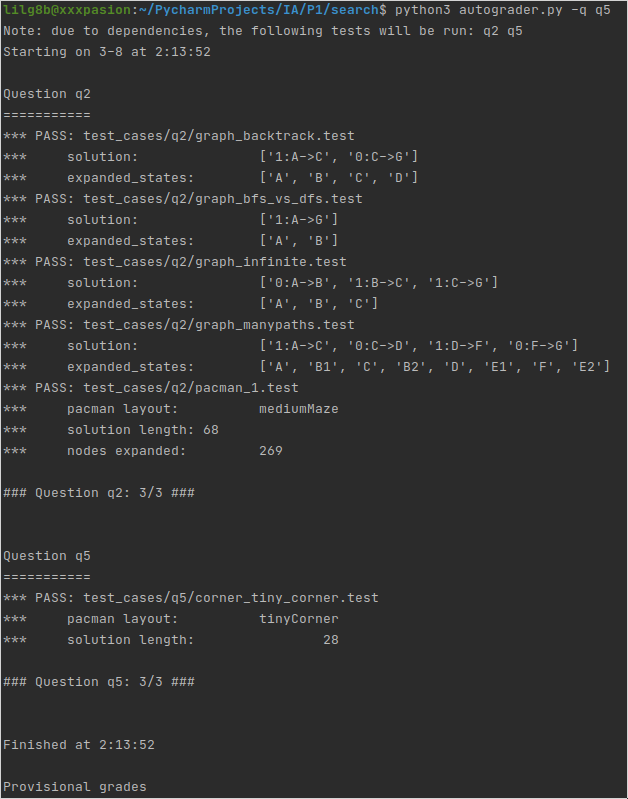
**Comando**: $ python3 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem



**Comando**: $ python3 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem



**Comando**: $ python3 autograder.py -q q5



\***Recordad** imprimir capturas de pantalla de estos tests y explicar lo que habéis observado para la memoria de la práctica.

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

Tiny Maze:

Coste: 28

Nodos Expandidos: 435

Resultado Medio/ Resultado: 512/ 512

Llega a la solución.

Medium Maze:

Coste: 106

Nodos Expandidos: 2448

Resultado Medio/ Resultado: 434/ 434

Llega a la solución.

**Dado que la heurística es monótona, también es admisible.**  
**Con heurística admisible, sin eliminar estados repetidos = optima**

## Sección 6 – Problema de las Esquinas Heurística ()

* 1. **Comentario personal en el enfoque y decisiones de la solución propuesta (1pt)**

El coste usado es el mínimo entre la heuristica de la distancia de Manhattan y de la heuristica de las esquinas, que busca un coste inferior al coste real.

* 1. **Lista & explicación de las funciones del framework usadas (1pt)**
  + def cornersHeuristic(state, problem):
  + util.manhattanDistance(nodo, corner):
  1. **Incluye el código añadido (0.25 pts)**

def cornersHeuristic(state, problem):

"""

A heuristic for the CornersProblem that you defined.

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 # These are the corner coordinates

# walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

nodo = state[0]

esquinas = state[1]

sum = 0

noVisitado = set(corners) - set(esquinas)

while len(noVisitado):

distancia, nodo = min([(util.manhattanDistance(nodo, corner), corner) for corner in noVisitado])

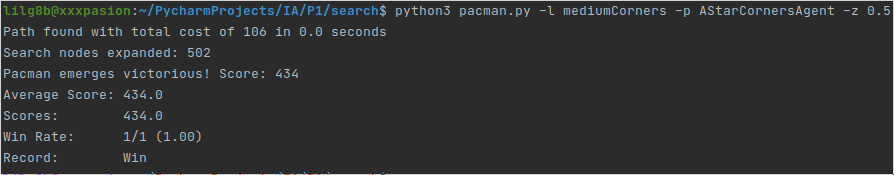
sum += distancia

noVisitado.remove(nodo)

return sum

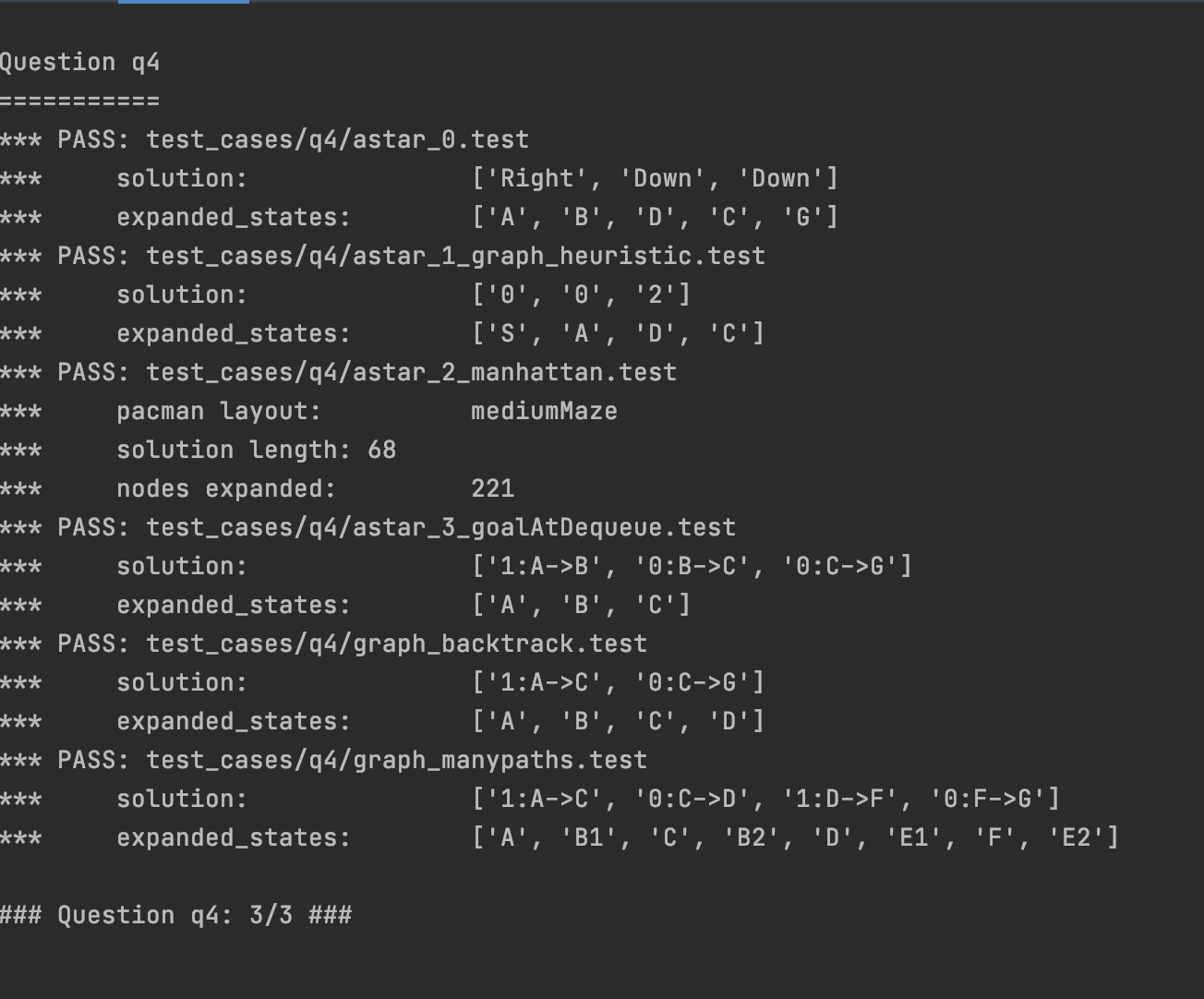
* 1. **Capturas de pantalla de los resultados de ejecución y pruebas analizando los resultados (1pt)**

**Comando**: $ python3 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5



**Comando**: $ python3 autograder.py -q q6

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





Longitud = 106

Nodos expandidos = 502

Q6: 2/3

Total: 5/6

**Dado que la heurística es monótona, también es admisible.**  
 **A\* con heurística admisible, sin eliminar estados repetidos = optima**

* 1. **Respuesta a pregunta 5: heurística (1pt) Explica la lógica de tu heurística.**

La heurística propuesta es, en sisntesis , el coste del camino mas corto que lleva al pacman desde su posición actual a traves de las esquinas no visitadas en ese momento considerando que puede atravesar paredes o la distancia Manhattan.

## Sección 7

**Comentarios personales de la realización de esta práctica**

Hemos encontrado alguna dificultad con el lenguaje de Python y la memorización o compresión del extenso código del programa de Pacman.

A demás de esto hemos hecho brainstorming para encontrar alguna heurística plausible en el punto 4 antes de pensar en la de manhattan y utilizarla, así como para la heurística del ejercicio 6.

## Nota de la memoria (40% de la práctica)

**Total de puntos (X/31.5)**