INGENIERÍA INFORMÁTICA

Escuela Politécnica Superior

Universidad Autónoma De Madrid

|  |
| --- |
| **Assignment 1: Search.** |
|  |
|  |
| **Miguel Arnaiz Montes** |
| **Sofia Xiaofan Fernández Marín**  **Pair 4 Group 1391**  **12/03/2021** |

Index

[Section 1 3](#_Toc64805910)

[Section 2 8](#_Toc64805911)

[Section 3 13](#_Toc64805912)

[Section 4 16](#_Toc64805913)

[Section 5 16](#_Toc64805914)

[Section 6 16](#_Toc64805915)

[Section 7 16](#_Toc64805916)

[Memory grade (40% of practice) 16](#_Toc64805917)

## Section 1

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**

We had some trouble trying to approach the goal of this section because we did not really understand what the getSuccessors() function return, but we finally managed to understand it. After reviewing that we really understood the code, we started programming the assignment.

We decided that it would be better to use the Stack option from the util.py file that was given to us. This is because we knew from the theory classes that the depth-first search was LIFO algorithm. That means that the element that you add the last one, is the one that goes out the first one.

First, we made sure we had comprehended the depth-first algorithm in paper so there would not be any problem when programming it.

Finally, we started to program the function and we did not find any problem.

* 1. **List & explanation of the framework functions used (1pt)**

We used the following functions from the utils.py:

Stack(): We use a stack because we need to have first the successors of the recently expanded node because we are in depth-first search that works with LIFO.

We used the following functions from the searchAgents.py:

getStartState(): This function returns the first node of the graph. In this function we use it to add it to the openList stack. We push it with an empty list that will be the path we go over.

isGoalState(): This function returns if the element we pass as a parameter is the node that we are looking for. For this specific function, we have as parameter the last inserted element of the stack called ele. We pass ele[0] because the nodes we are adding to the stack have the node and the path.

getSuccessors(): This function returns a tuple with the nodes that are successor from the node we pass as parameter (successor, path(action), cost). The first one is the one we have as a successor, the second one the path (the action that required to get there) and the cost that has at that moment getting to that node.

The last one is not necessary in this function but in the others we have to pass (using a loop for checking that we add all the successors) the name successors[0] and the path, that it would be equal to the path of the node that we have pop form the stack ele[1] plus the path of the parent successors[1].

* 1. **Includes code written by students (0.25 pts)**

def depthFirstSearch(problem):

openList = util.Stack()

closedList = []

openList.push((problem.getStartState(), []))

while openList.isEmpty() == False:

ele = openList.pop()

if problem.isGoalState(ele[0]) == False:

closedList.append(ele[0])

for successors in problem.getSuccessors(ele[0]):

if successors[0] not in closedList:

successor = (successors[0], ele[1] + [successors[1]])

openList.push(successor)

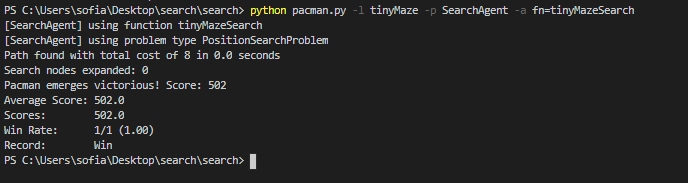
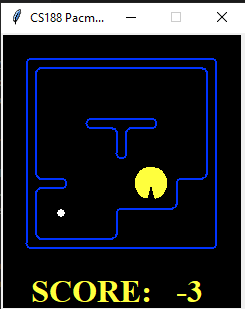
else:

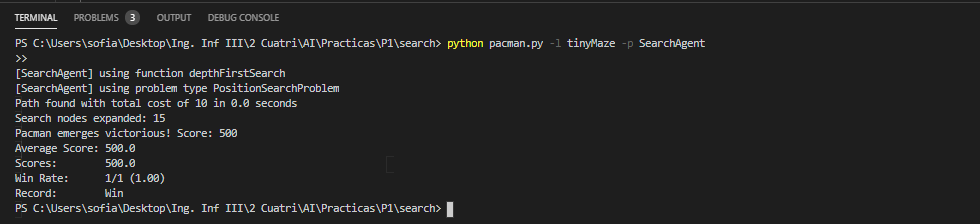
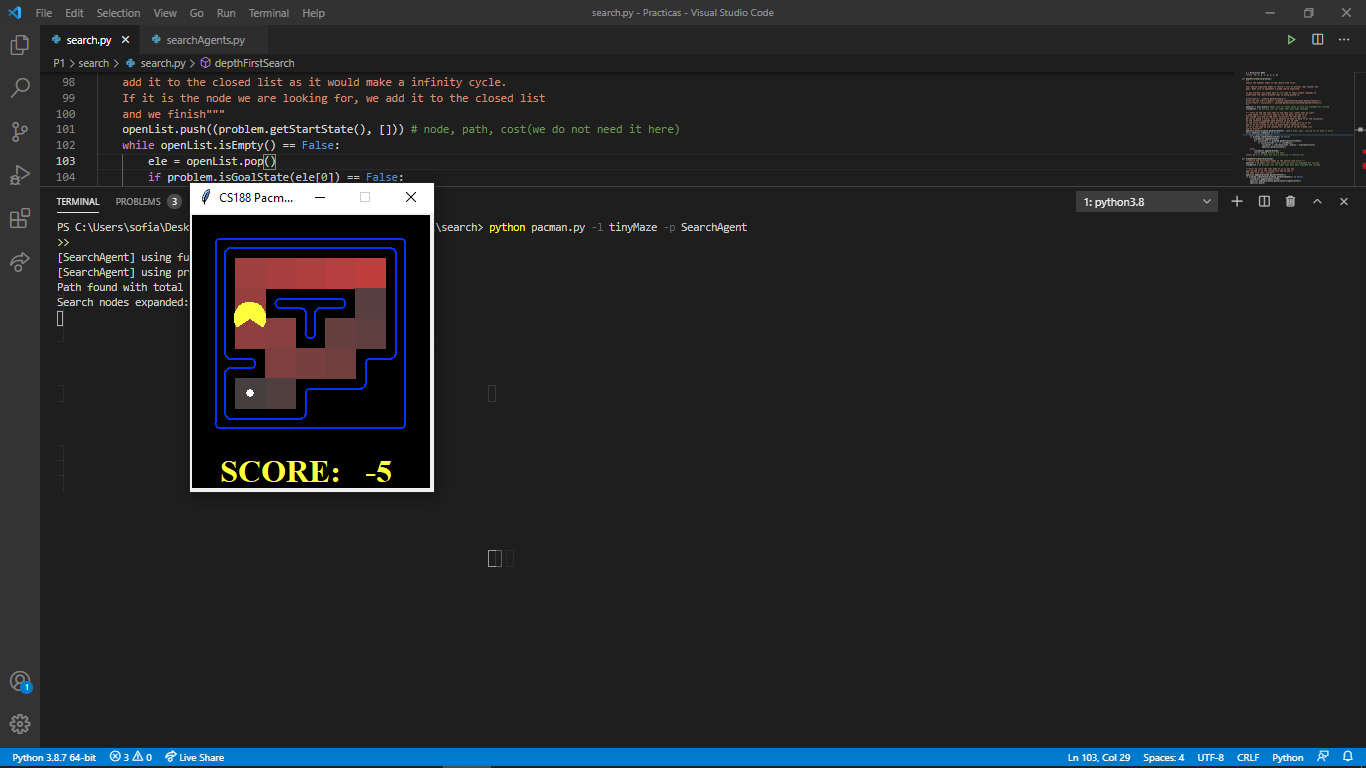
closedList.append(ele[0])

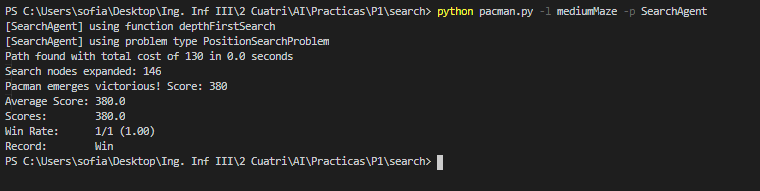
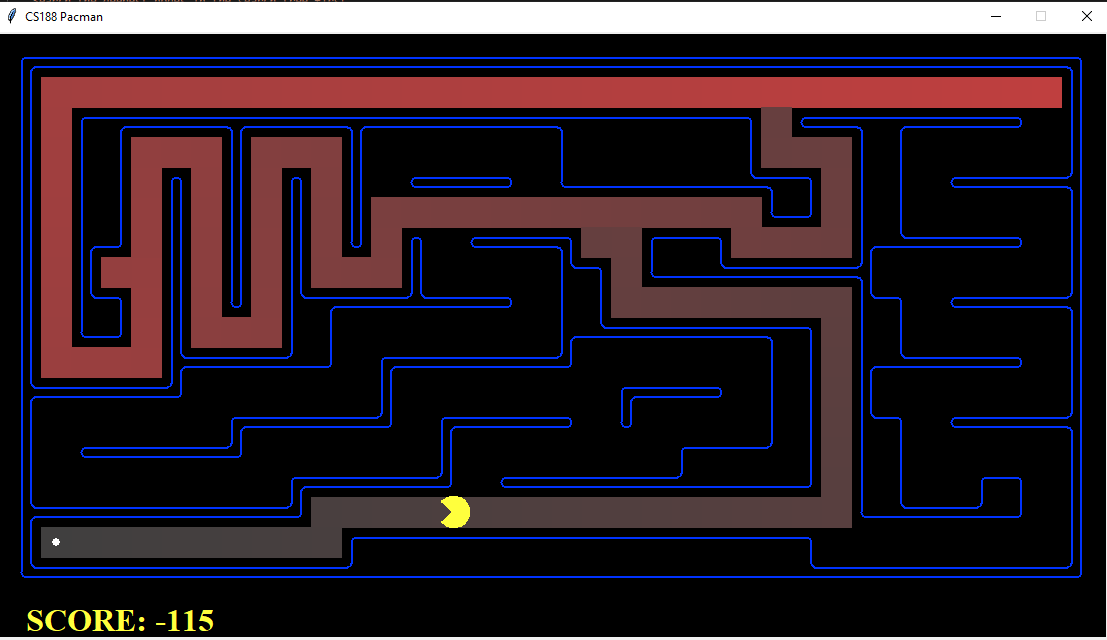
return ele[1]

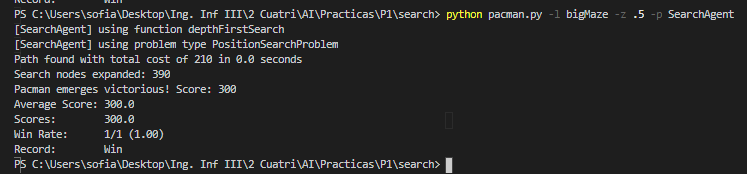
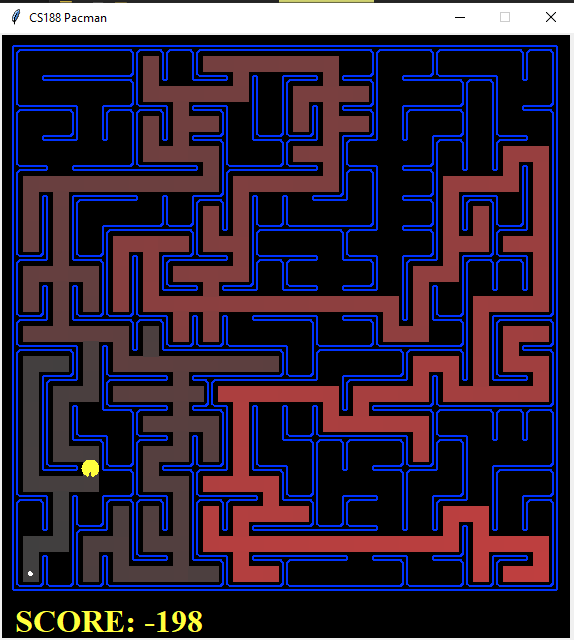
return []

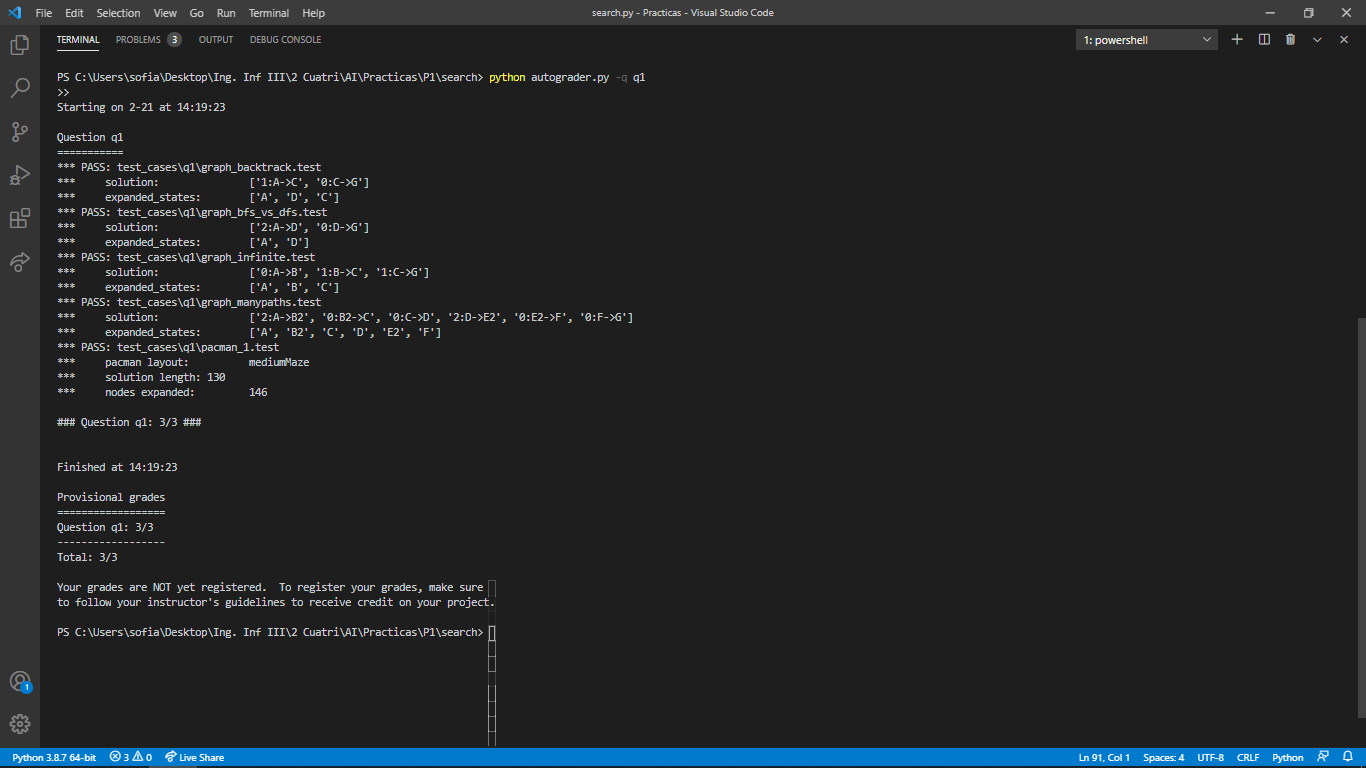
* 1. **Screenshots of executions and test carried out analyzing the results (1pt)**

python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch

python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

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

 python autograder.py -q q1

* 1. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

We get to the conclusion that the behaviour of the pacman is **optimal** because it guarantees a solution to be found. This is because we have used a closedList, a list that have nodes that have been expanded.

It is **complete**, so it guarantees to find a solution. The reason is the same as we explained earlier.

If we would not use a closedList, it would have been non-optimal and not complete.

The space we use for this algorithm is O(b\*m) and the time is O(b^m) where “b” is the branching factor and “m” is the maximum tree depth. In the worst case.

* 1. **Answer to question 1.1 (1pt)**

Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

* 1. **Answer to question 1.2 (1pt)**

Is this a least cost solution? If not, think about what depth-first search is doing wrong.

* 1. **Answer to question 2 (1pt)**

Does BFS find a least cost solution? If not, check your implementation.

## Section 2

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**

For this problem we knew the Queue option from the util.py file that was given to us was the appropriate to use because knew that the breath-first search was FIFO algorithm. That means that the element that you add the first one, is the one that goes out the first one.

This function was the most difficult to us to implement as it returned the expanded states with repeated states. We did not know why until we realize that we needed to check that the node we had pop were not in the closed list before calling to the getSuccessors() function. But finally, we managed to complete this task.

* 1. **List & explanation of the framework functions used (1pt)**

We used the following functions from the utils.py:

Queue(): We use a priority queue because we need to have first the node with least cost of the queue.

We used the following functions from the searchAgents.py:

getStartState(): This function returns the first node of the graph. In this function we use it to add it to the openList queue. We push it with an empty list that will be the path we go over.

isGoalState(): This function returns if the element we pass as a parameter is the node that we are looking for. For this function, we have as parameter the node with the least cost of the queue called ele, the one we have popped form the queue.

getSuccessors(): This function returns a tuple with the nodes that are successor from the node we pass as parameter (successor, path(action), cost). The first one is the one we have as a successor, the second one the path (the action that required to get there) and the cost that has at that moment getting to that node.

The first one is the name successors[0] and the path, that it would be equal to the path of the node that we have pop form the stack ele[1] plus the path of the parent successors[1].

* 1. **Includes code written by students (0.25 pts)**

def breadthFirstSearch(problem):

openList = util.Queue()

closedList = []

openList.push((problem.getStartState(), []))

while openList.isEmpty() == False:

ele = openList.pop()

if problem.isGoalState(ele[0]) == False:

if ele[0] not in closedList:

closedList.append(ele[0])

for successors in problem.getSuccessors(ele[0]):

successor = (successors[0], ele[1] + [successors[1]])

openList.push(successor)

else:

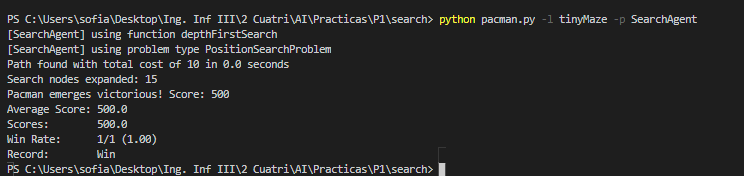
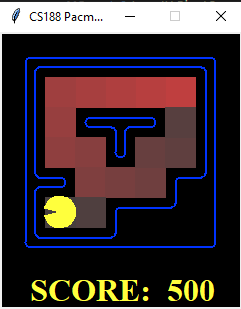
closedList.append(ele[0])

return ele[1]

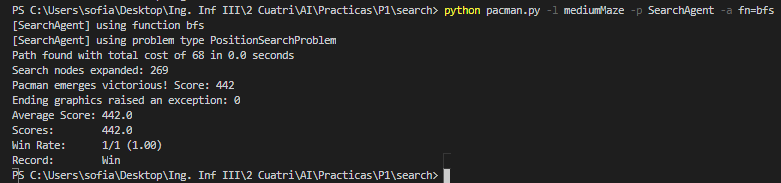
return []

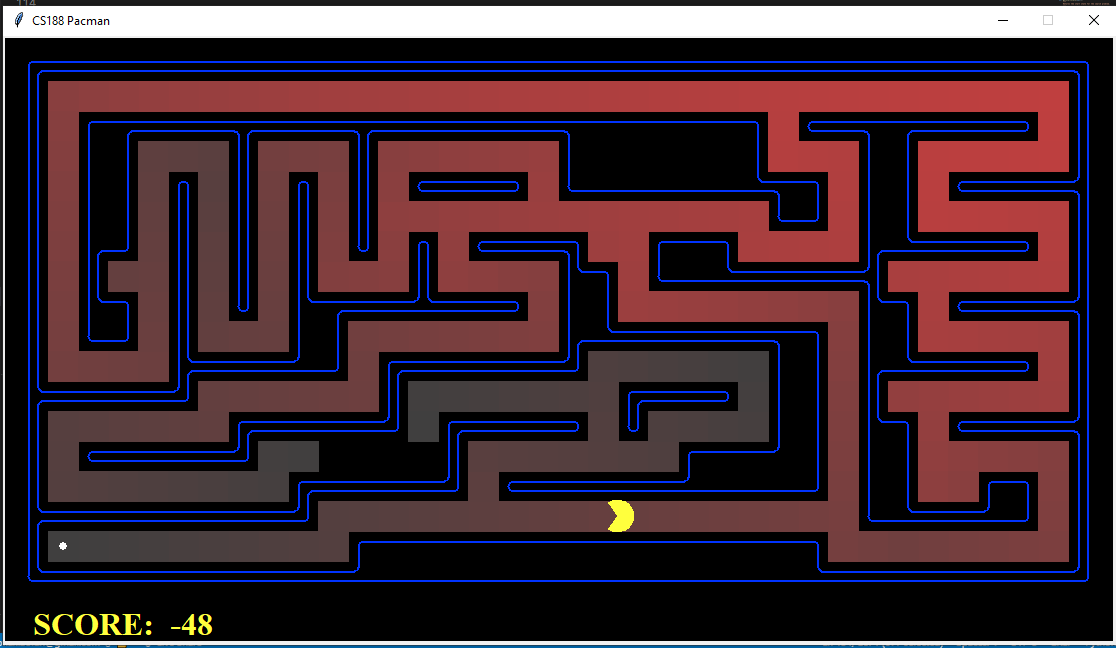
* 1. **Screenshots of executions and test carried out analyzing the results (1pt)**

python pacman.py -l tinyMaze -p SearchAgent

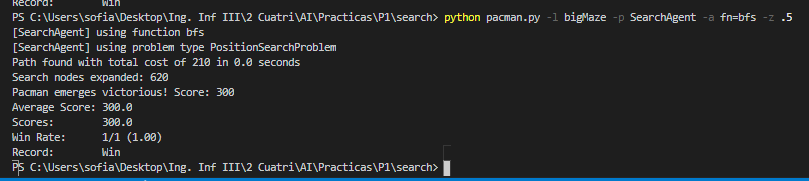


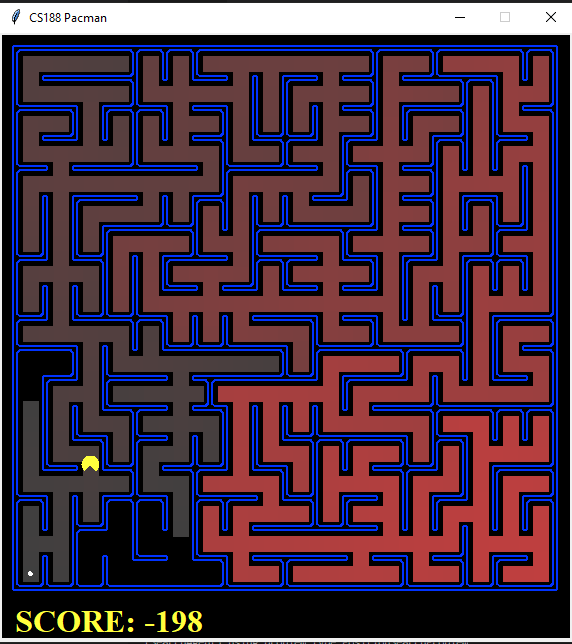
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs



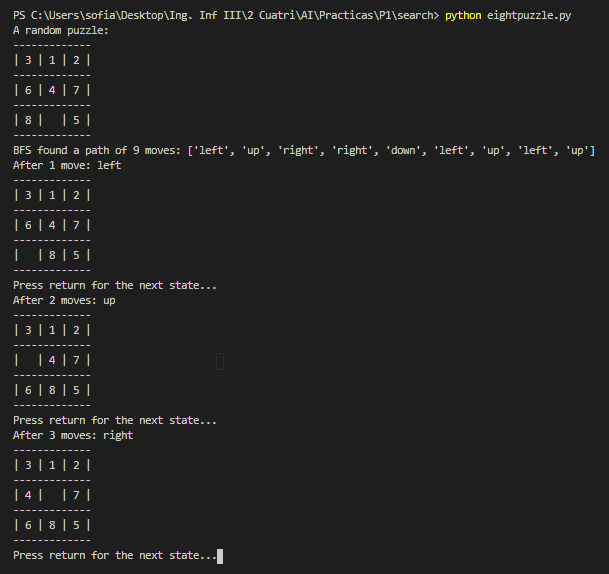


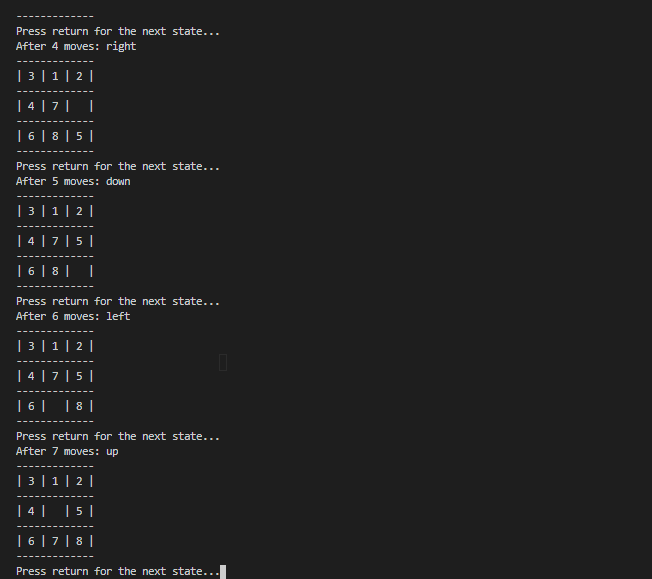
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

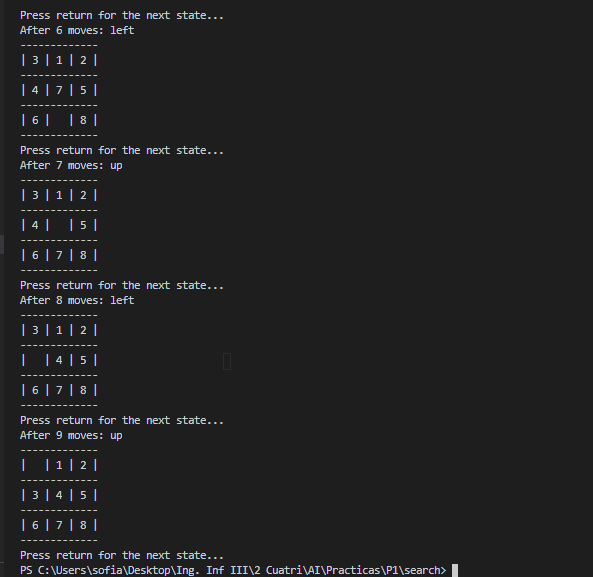


****

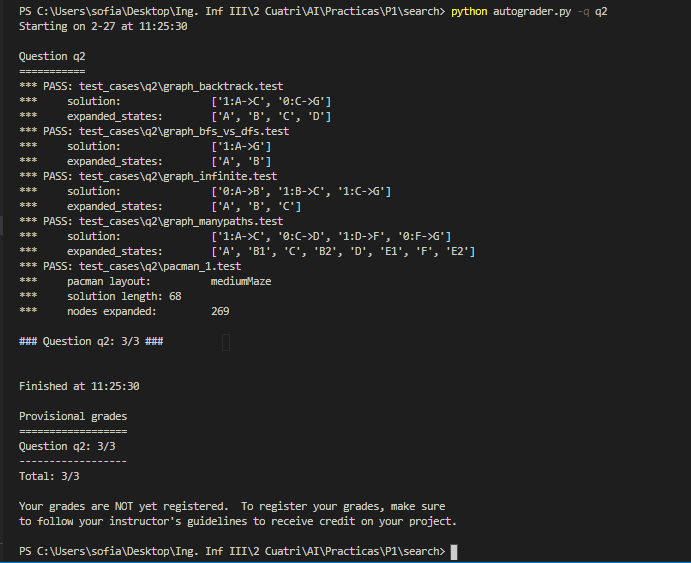
python eightpuzzle.py







python autograder.py -q q2



* 1. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

We get to the conclusion that the behaviour of the pacman is **optimal** because it guarantees a solution to be found.

It is also **complete**, so it guarantees to find a solution.

The space we use for this algorithm is O(b^d) and the time is O(b^(d+1)) where “b” is the number of children of a node and “d” is the maximum tree depth. In the worst case.

* 1. **Answer to question 3 (1pt)**

What happens on openMaze for the various search strategies?

## Section 3

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**

For this specific function we did not have any problem implementing it as we had programmed two functions before it.

Having in to account the util.py file we were provided with, we decided that it would be better to use the Priority Queue. This is because the uniform cost search was a priority queue algorithm. In other words, the node with the least cost is popped before other ones with larger cost.

* 1. **List & explanation of the framework functions used (1pt)**

We used the following functions from the utils.py:

PrioritQueue(): We use a priority queue because we need to have first the node with lest cost of the queue.

We used the following functions from the searchAgents.py:

getStartState(): This function returns the first node of the graph. In this function we use it to add it to the openList queue. We push it with an empty list that will be the path we go over. In this case we also add the cost of the nodes because we will need it as it is a uniform cost problem.

isGoalState(): This function returns if the element we pass as a parameter is the node that we are looking for. For this specific function, we have as parameter the node with the least cost of the queue called ele. We pass ele[0] because the nodes we are adding to the queue have the node, the path and the cost.

getSuccessors(): This function returns a tuple with the nodes that are successor from the node we pass as parameter (successor, path(action), cost). The first one is the one we have as a successor, the second one the path (the action that required to get there) and the cost that has at that moment getting to that node. The last value of the list that this function return is the cost of the node.

The first one is the name successors[0] and the path, that it would be equal to the path of the node that we have pop form the stack ele[1] plus the path of the parent successors[1]. We also add the cost ele[2] and successors[2].

* 1. **Includes code written by students (0.25 pts)**

def uniformCostSearch(problem):

openList = util.PriorityQueue()

closedList = []

openList.push((problem.getStartState(), [], 0), 0)

while openList.isEmpty() == False:

ele = openList.pop()

if problem.isGoalState(ele[0]) == False:

closedList.append(ele[0])

for successors in problem.getSuccessors(ele[0]):

if successors[0] not in closedList:

successor = (successors[0], ele[1] + [successors[1]])

openList.push(successor)

else:

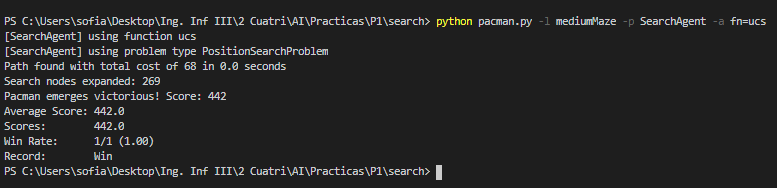
closedList.append(ele[0])

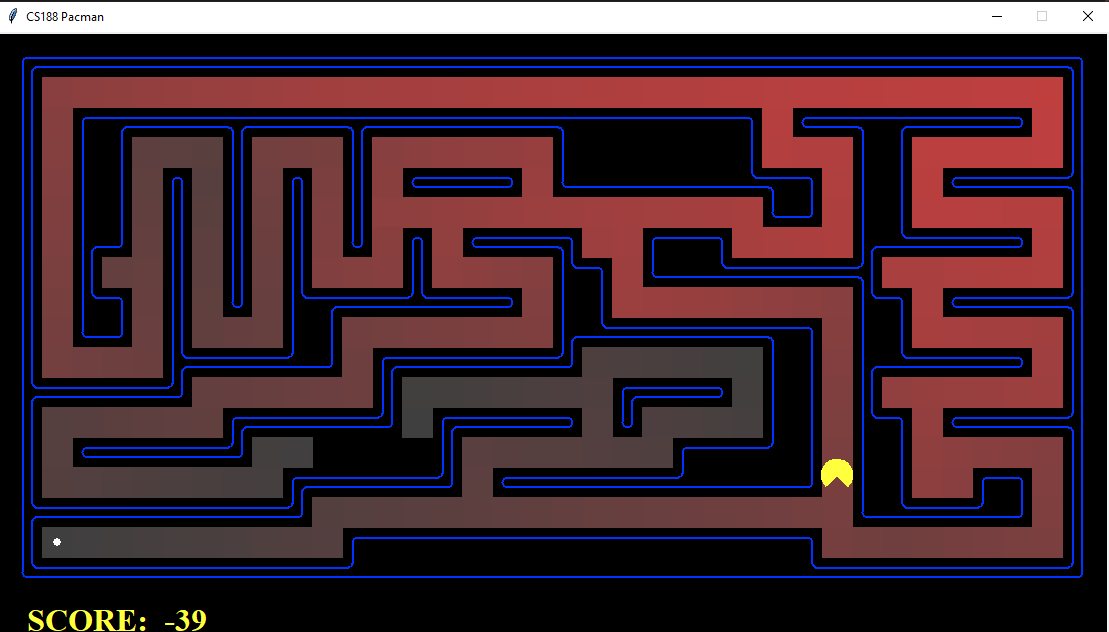
return ele[1]

return []

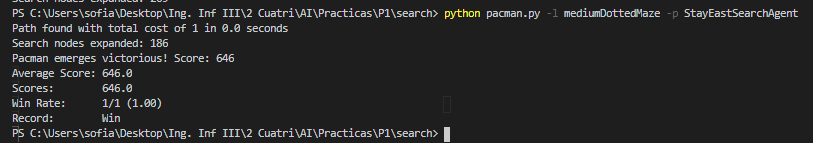
* 1. **Screenshots of executions and test carried out analyzing the results (1pt)**

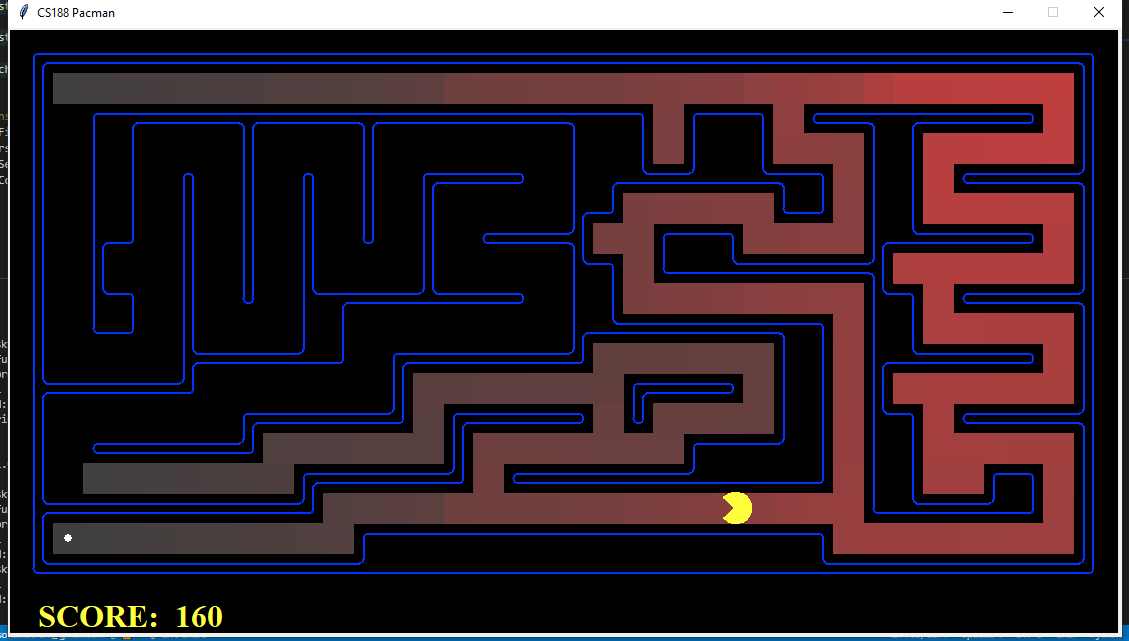
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs



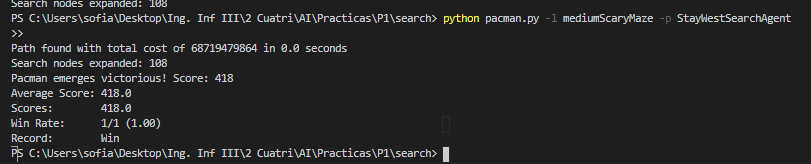


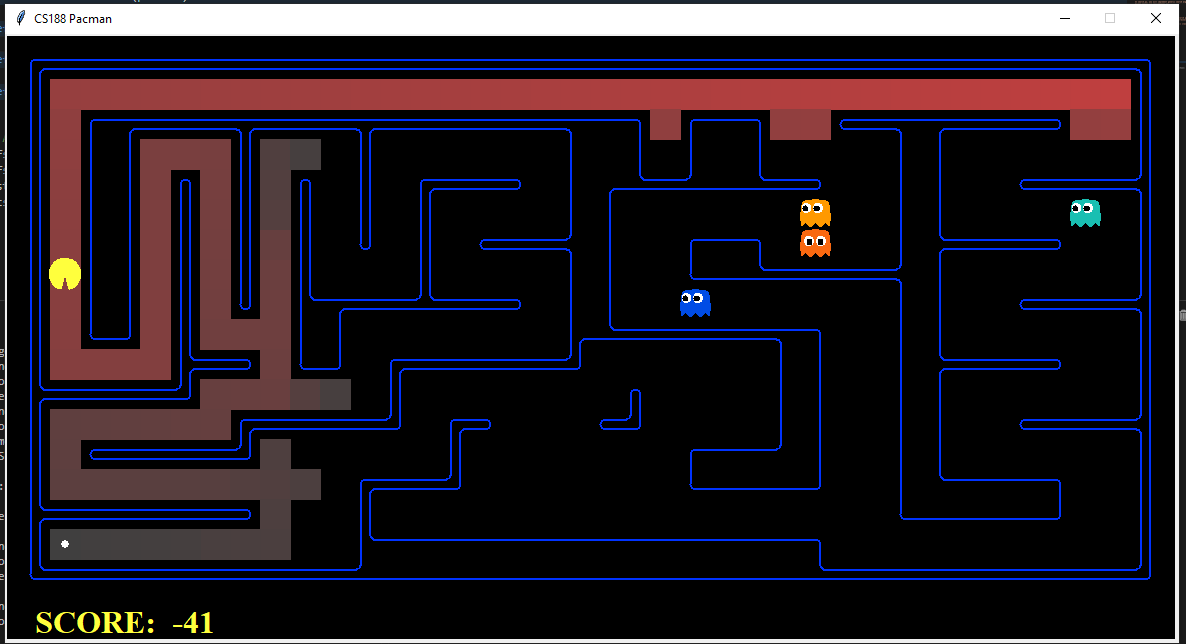
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent



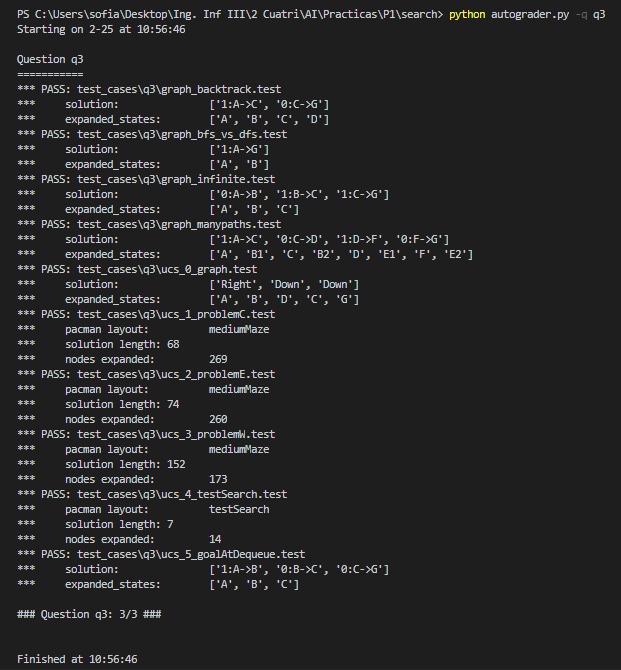


python pacman.py -l mediumScaryMaze -p StayWestSearchAgent





python autograder.py -q q3



* 1. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

We get to the conclusion that the behaviour of the pacman is **optimal** because it guarantees a solution to be found but only if cost-path (successor (n)) ≥ cost-path (n).

It is also **complete**, so it guarantees to find a solution.

The space we use for this algorithm is O(b^(c/e)) and the time is the same one where “b” is the number of children of a node, “c” is the cost path optimal solution and “e” is the minimum cost of an action.

## Section 4

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**

Most difficult part

* 1. **List & explanation of the framework functions used (1pt)**

ESTO ES COMO LOS ANTERIORES PERO CAMBIA UN POCO LAS COSAS PARA QUE NO SEA UN COPYPASE LITERAL :D

* 1. **Includes code written by students (0.25 pts)**

def aStarSearch(problem, heuristic=nullHeuristic):

openList = util.PriorityQueue()

closedList = []

openList.push((problem.getStartState(), [], 0), heuristic(problem.getStartState(), problem))

while openList.isEmpty() == False:

ele = openList.pop()

if problem.isGoalState(ele[0]) == False:

if ele[0] not in closedList:

closedList.append(ele[0])

for successors in problem.getSuccessors(ele[0]):

fSuccessor = successors[2] + heuristic(successors[0], problem)

successor = (successors[0], ele[1] + [successors[1]], successors[2] + ele[2])

openList.update(successor, fSuccessor + ele[2])

else:

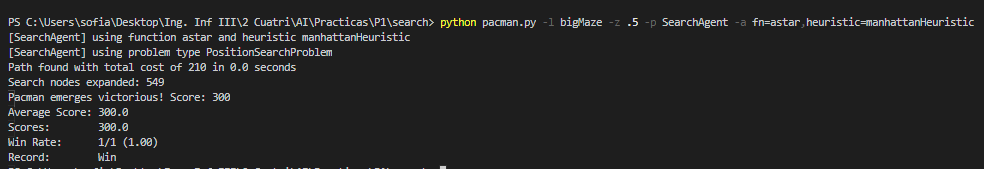
closedList.append(ele[0])

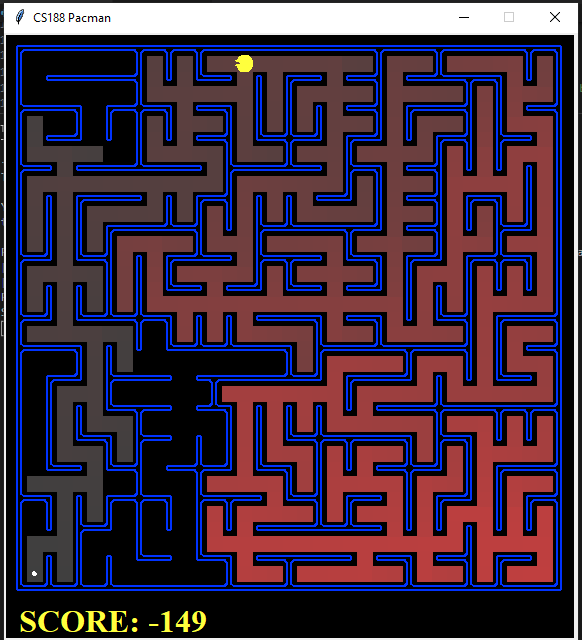
return ele[1]

return []

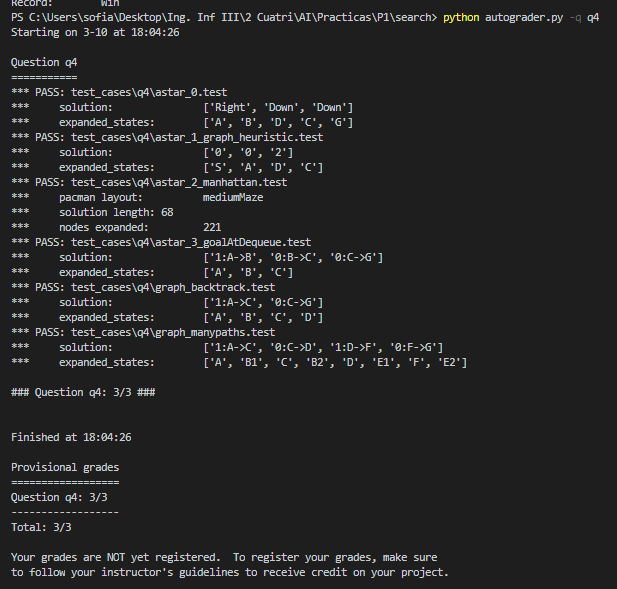
COMENTAR EL CODIGO (search.py) LO QUE HACE EL CODIGO

* 1. **Screenshots of executions and test carried out analyzing the results (1pt)**

python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic



python autograder.py -q q4



* 1. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

ESTO DE SI ES OPTIMAL Y ESAS MIERDAS TE LO PONE EN LA TEORIA PERO

A\* + WITHOUT ELIMINATION OF REPEATED STATES + H ADMISSIBLE = OPTIMAL (TREE SEARCH)

A\* + WITH ELIMINATION OF REPEATED STATES + H ADMISSIBLE = NOT NECESSARY OPTIMAL OPTIMAL

IT IS COMPLETE(REACHES THE SOLUTION)

NODES THAT IT EXPANDS= ESTÁ NE LA TEORIA

* 1. **Answer to question 4 (1pt)**

What happens on openMaze for the various search strategies?

## Section 5

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**
  2. **List & explanation of the framework functions used (1pt)**
  3. **Includes code written by students (0.25 pts)**
  4. **Screenshots of executions and test carried out analyzing the results (1pt)**
  5. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

## Section 6

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**
  2. **List & explanation of the framework functions used (1pt)**
  3. **Includes code written by students (0.25 pts)**
  4. **Screenshots of executions and test carried out analyzing the results (1pt)**
  5. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**
  6. **Answer to question 5: heuristics (1pt)**

## Section 7

Personal comments on the development of this practice

## Memory grade (40% of practice)

**Total points (X / 31.5)**