# ARTIFICIAL INTELLIGENCE MEMORY P1

# Done by: José Manuel López-Serrano Tapia,

## Section 0

Summary of the concepts reviewed in the practice

## Section 1

* 1. **Personal comment on the approach and decisions of the proposed solution** 
     1. **List & explanation of the framework functions used**
     2. **Includes code written by students**
     3. **Screenshots of executions and test carried out analyzing the results**
  2. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc** 
     1. **Answer to question 1.1**
     2. **Answer to question 1.2**
     3. **Answer to question 2**

## Section 2

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

## Section 3

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

## Section 4

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

## Section 5

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

## Section 6

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

## Section 7

Personal comments on the development of this practice

## Section 0: Summary

In the first four sections we will review the three uninformed search algorithms, depth first, breadth first and uniform cost, and the a\* algorithm. We review graph search (elimination of repeated states) and test all algorithms in various mazes so that we can analyse each one’s ability to find an optimal path, under what circumstances, and how efficiently it is performed.

In section 5 we focus on defining a problem itself: the state space, initial state, goal state and successor function. The 4 algorithms already coded should hopefully work with a well-coded problem, since the code is modulated enough.

In section 6 we find a consistent heuristic for the problem in section 5, for the purpose of applying a\* and witnessing its true power with more complex problems.

## Section 1: Depth First Search

Code:

Texto

Descripción generada automáticamente

For the first four exercises we created the GraphSearch function: given an open list structure as an argument, it carries out the search algorithm with elimination of repeated states (using a list as the closed list). We found it very useful to create a separate Node class to facilitate working with the state space. The class works as follows:

Texto

Descripción generada automáticamente

A node object only stores three pieces of information: the game state itself it refers to, a complete list of operators from the root node to the current node, and the total cost required to get from the root node to the current node. The last two pieces are assembled using the parent node’s information (the parent is passed as an argument in the node constructor) plus the operator and cost of the current node. The root node is the only one without a parent. In GraphSearch, we can see how useful the node structure is, since each node carries the list of operators already in case they contain the goal state.

Texto

Descripción generada automáticamente

All four search algorithms are almost as simple as this one: we use a stack as the open list and execute the GraphSearch algorithm. A stack for depth search is appropriate since stacks have a LIFO (Last In First Out) policy. This means the nodes generated from the latest expanded node will be expanded first, thus prioritising depth.

Testing:

Imagen que contiene Icono

Descripción generada automáticamentePatrón de fondo, Escala de tiempo

Descripción generada automáticamenteCódigo QR

Descripción generada automáticamenteTexto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

Pacman does reach the goal on all three attempts since elimination of repeated states guarantees it. However, if we were to implement tree search with a depth first search, Pacman might follow a path of infinite length and never finish the search.

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

As we can see, more than half the state space in medium and big mazes was not ever generated. This is to be expected from a depth first search, since the “current path” followed is not ever changed unless there are no more successors to generate. Of course, not all expanded nodes will be on the final solution: that is the nature of search algorithm.

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

Depth first search rarely offers an optimal solution, by virtue of not considering all possibilities in the state space and committing to a path until it must rectify. This case is no different.

## Section 2: Breadth First Search

Code:

Texto

Descripción generada automáticamente

We used a queue as the open list, which has a FIFO (First In First Out) policy, which is appropriate because we ensure a node of higher depth never gets expanded before a lower depth node.

Testing:

Imagen que contiene Icono

Descripción generada automáticamentePatrón de fondo

Descripción generada automáticamenteImagen que contiene Código QR

Descripción generada automáticamenteTexto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

**Question to answer: Does BFS find a least cost solution?**

It is not surprising that Pacman finds a solution on all experiments, since breadth first search guarantees to find the shallowest solution that exists. Since in the current problem all actions have the same cost of 1, the shallowest path is the most optimal by default. Thus, Pacman’s solution will always be optimal under these circumstances.

What is not optimal, though, is the number of search nodes expanded: almost all the game states were generated while finding the least-cost path (since, as we established, a lower depth path will always be explored earlier than another path of higher depth). This is an inefficient use of time and space.

## Section 3: Uniform Cost Search

Code:

Texto

Descripción generada automáticamente

We will use the priority queue with function as the open list: this queue pops the element with the lowest priority score. Uniform cost search expands the node with the lowest cost path from root node to current node, so our priority function will return exactly that: the cumulative cost from root to the current node that every node has already stored (as explained in the Node class in Section 1). We defined an internal function that did as described and used it as argument for the queue.

Testing:

Patrón de fondo

Descripción generada automáticamenteTexto

Descripción generada automáticamente

In this first test we can notice that the algorithm behaves the same as in the breadth first search algorithm. This is because all actions cost the same, so all nodes of the same depth have the same priority; the priority queue then behaves like a regular queue. Thus, Pacman does find a solution and it is optimal.

Next two tests are done with two different agents, respectively: the first agent heavily favours staying on the East side of the map, while the second agent heavily favours the West side. This is done by varying the cost function.

Imagen que contiene Patrón de fondo

Descripción generada automáticamenteImagen de la pantalla de un video juego

Descripción generada automáticamente con confianza bajaTexto

Descripción generada automáticamente

In these cases, uniform cost search is very different from breadth first search since the cost function does no longer increase monotonously with depth. The behaviour is definitely optimal since successors of nodes will always have a higher cost path than the node itself, while breadth first search would not consider path costs and end up with a highly inefficient solution, even though it is the lowest-depth solution.

## Section 4: A\* Search

Code:

Texto

Descripción generada automáticamente

The structure of this function is almost identical to the uniform cost search function, with a difference being the priority function: in a\* the function returns the sum of the cumulative path cost and the result of the heuristic given as an argument.

Testing:

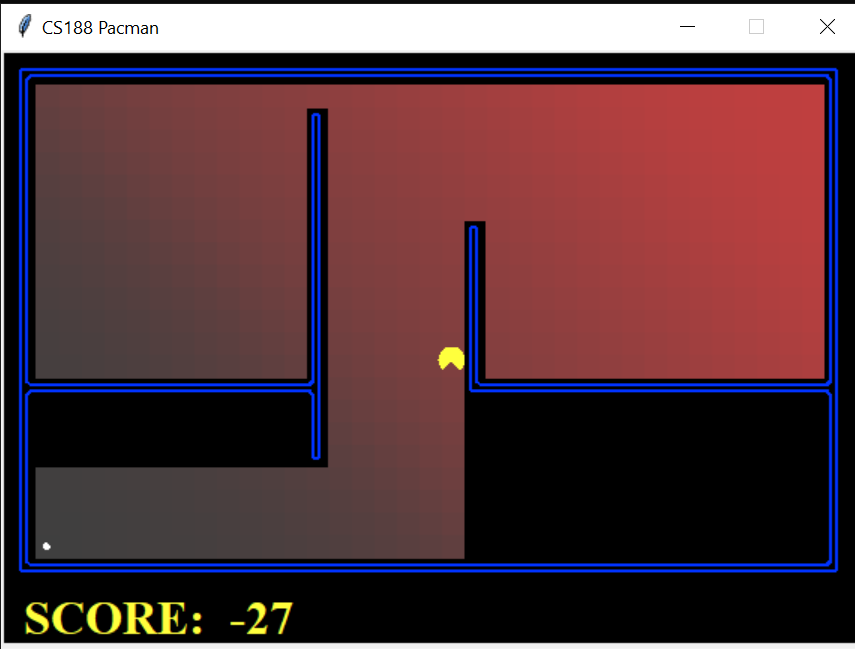
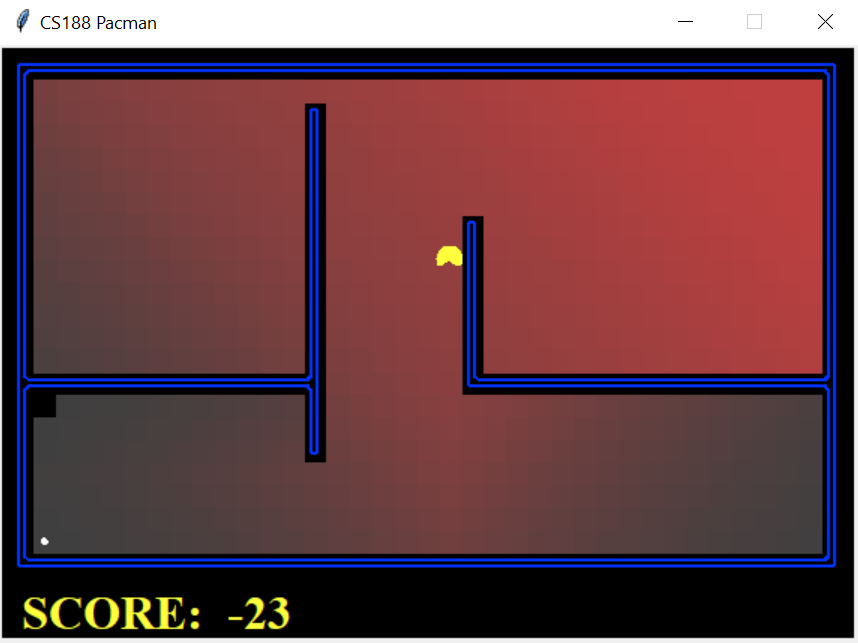
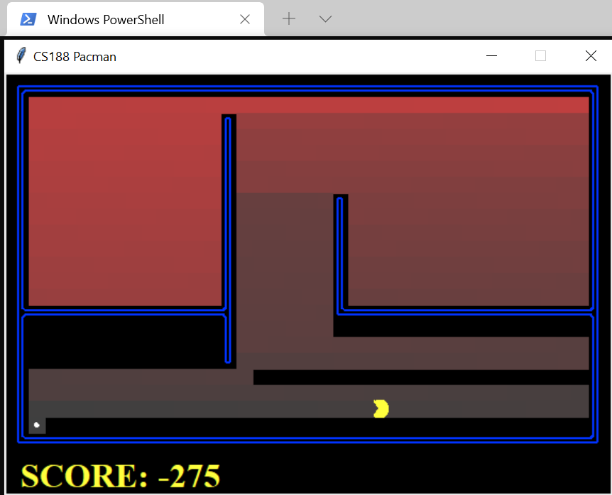
Imagen que contiene Código QR

Descripción generada automáticamenteTexto

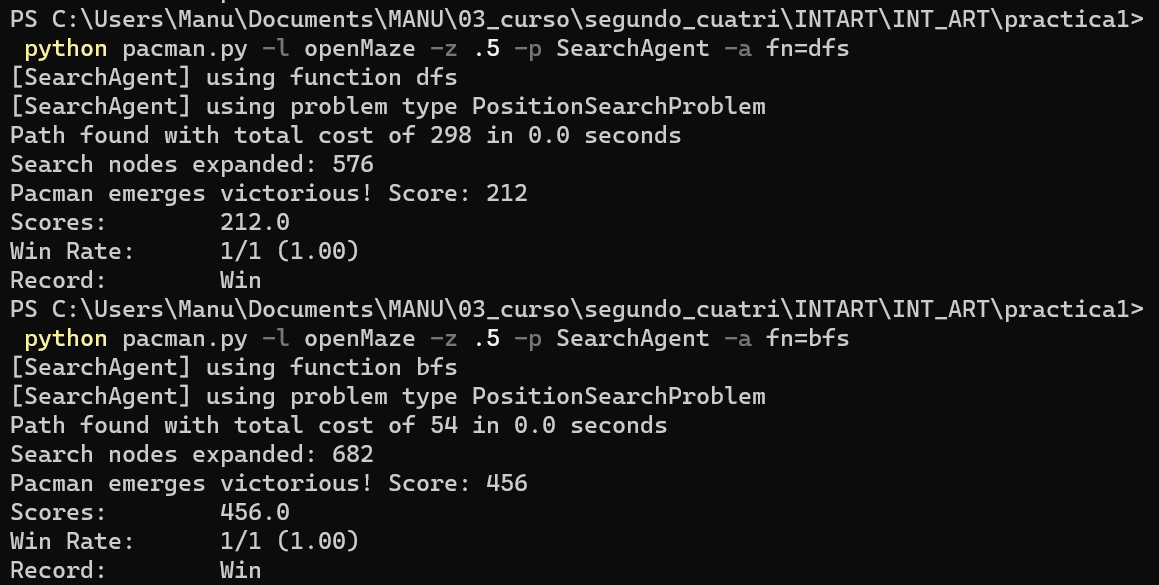
Descripción generada automáticamente

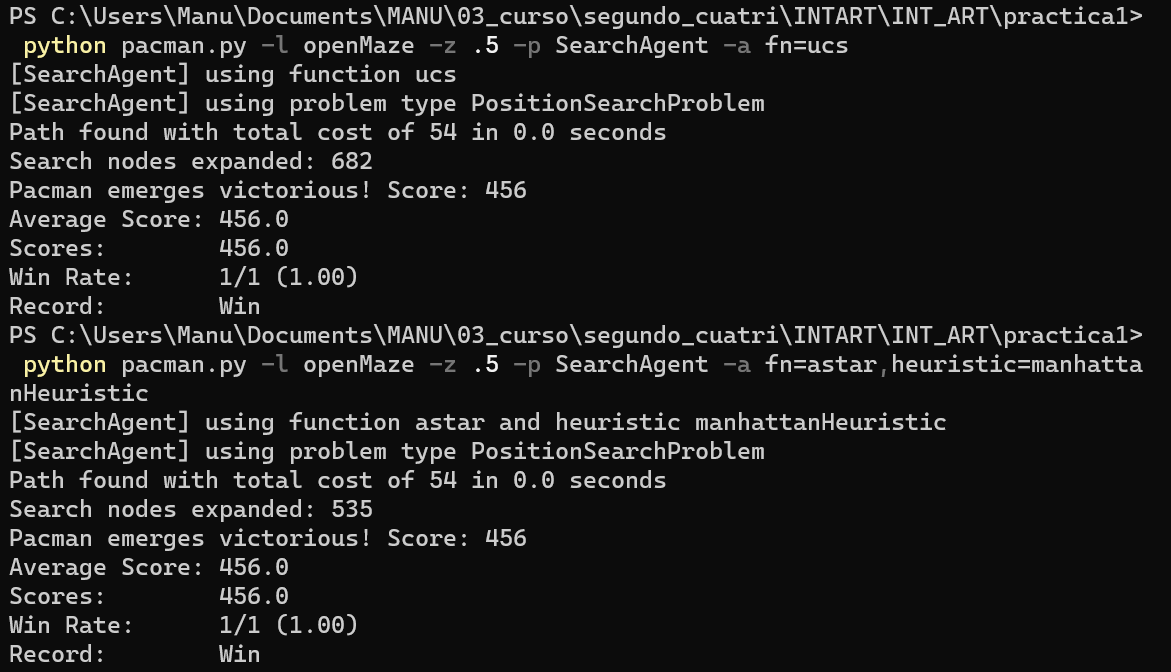
Pacman does find the solution, which is optimal. Since the Manhattan heuristic is admissible and monotonic, it will always find the optimal solution for graph search. It is noticeable that the a\* solution needed less node expansion than the uniform cost search algorithm: the a\* algorithm is also optimally efficient.

**Question to answer: What happens on openMaze for the various search strategies?**

****

Top left: dfs. Top right: bfs. Bottom left: ucs. Bottom right: a\*.

****

****

In order from top to bottom: dfs, bfs, ucs, a\*.

Depth first search is the only algorithm that did not generate the lowest cost path, in fact having a cost almost 6 times over the optimal cost. It is worth noting that successors to a state are generated following this order: North, South, East, West. Since depth first uses a stack as open list, the reverse order is the one followed when picking the next node to expand. Thus, it explains why Pacman directly goes West and stays there in the starting moments of the algorithm. It also explains Pacman’s movements: it moves left until the limit, one tile South, then right until the limit, another tile South, etc. Since we’ve covered, the succession order naturally favours West and East, so since the maze is so open, it makes sense why Pacman’s movement is so inefficient.

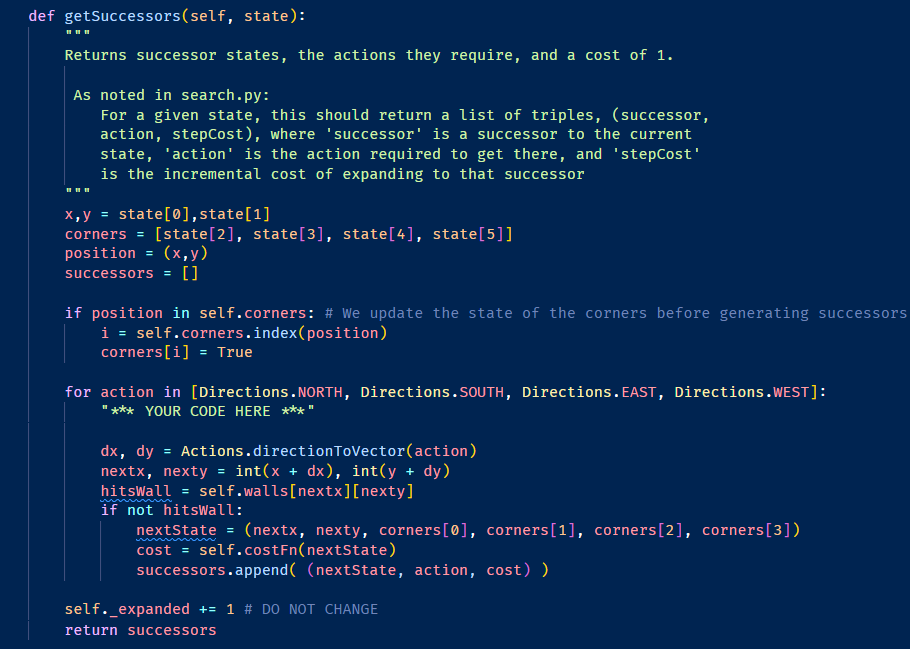
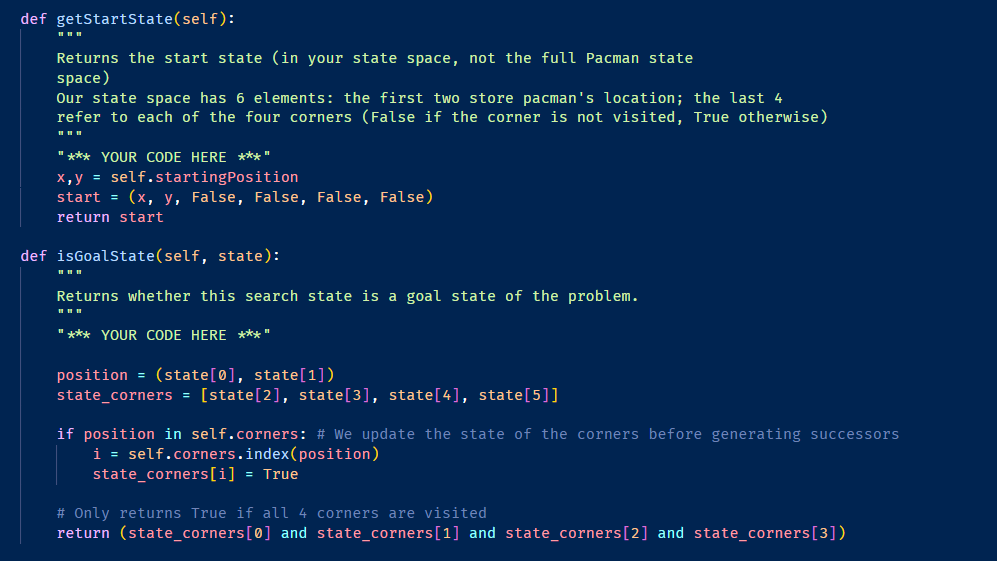
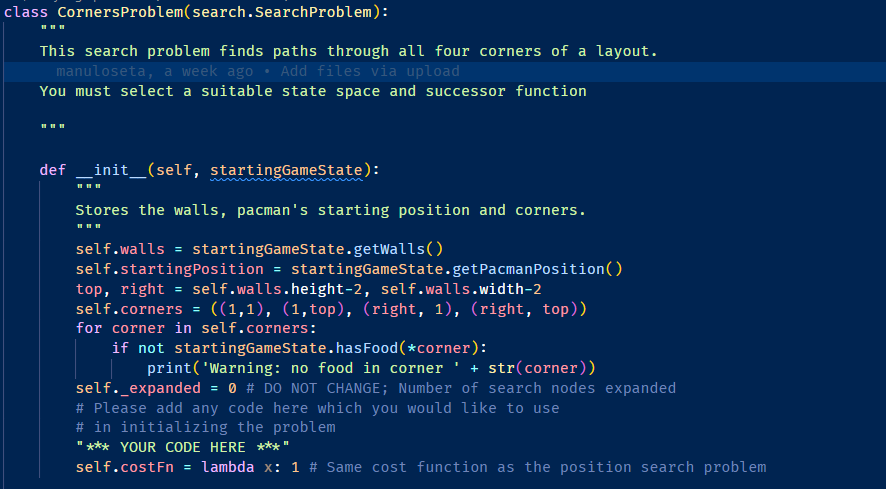
As we have already covered in Section 3, uniform cost search is identical to breadth first search given that every action costs the same, which is clearly visible in both tests. The path found is optimal, but the number of expanded nodes is highly inefficient (all but one node was expanded, since the goal state was so deep in the search graph). Contrary to depth first, since both breadth first and uniform cost worked with queues, we can see the East side of the maze being explored first. Also, it is worth nothing that bfs and ucs have a smoother gradient from red to grey, since lower depth states were always expanded first, as opposed to dfs, in which you can basically figure out the node expansion path the algorithm took. Bfs and ucs do have more expanded nodes than dfs.

Last algorithm is a\*, which is both optimal cost and optimally efficient in finding the solution. The gradient of its maze is rather like the bfs and ucs ones, but with a stark contrast at the bottom part: an inefficient route was never taken. This is because the only thing the state space does not take into accounts are the walls; you encounter them once you generate successors. This explains why a\* does not struggle in the end: with no more walls in place, the consistent heuristic effectively guides the graph expansion to the goal (since the Manhattan heuristic treats the maze like a simplified version of the problem, with no walls). This contrasts with the two algorithms before, which perform uniformed searches.

## Section 5: Corners Problem

Code:

As opposed to the last 4 sections, which were based on a search problem (finding your way from one initial state to the end state), the corners problem is concerned to find the lowest cost path to guide Pacman to all 4 corners of the rectangular maze. In this section we needed to define the state space itself, as well as the functions to define the start space, recognise a goal state and to generate successors.



The entirety of the CornersProblem class in three screenshots

First is the constructor method, in which we just added an extra line: the same cost function as in the search problem from before (so all actions still cost 1).

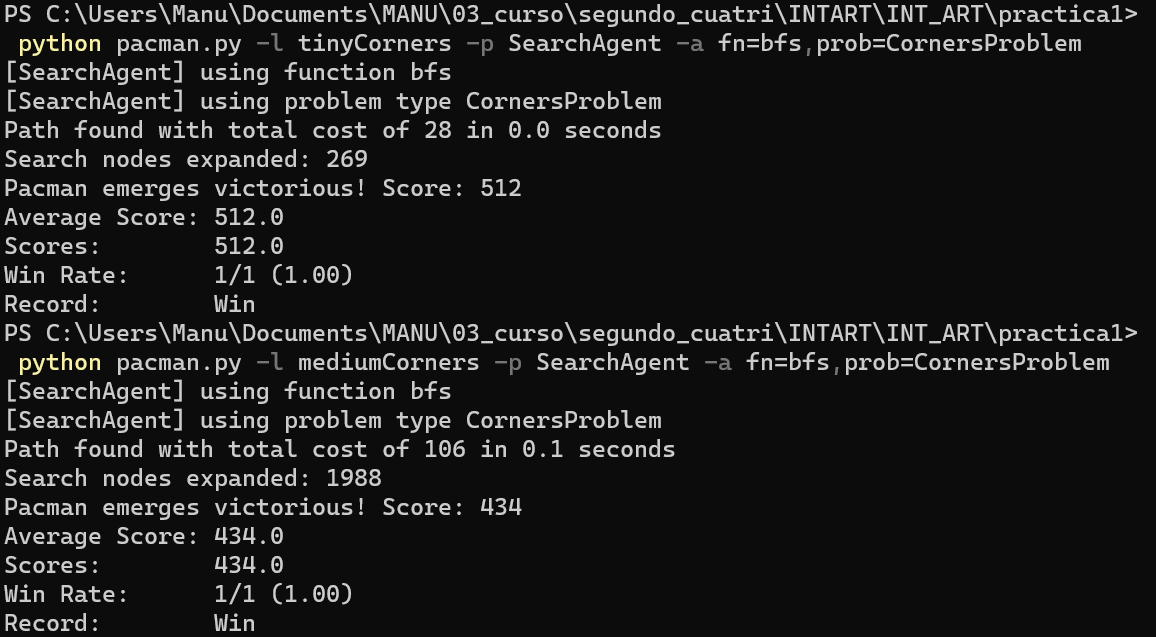
The next function is to define the start state, and we decided to go with a 6-element tuple: the two first elements reference Pacman’s position, and the other four elements are Booleans that reference the state of each corner (False if it hasn’t been visited, True otherwise). The start state, then, is the starting position for Pacman followed by four False Booleans.

The next function identifies the goal state, which is a state with all Booleans being True. Note that the actual goal state will only have three True Booleans, and the function checks whether Pacman’s current position is the remaining corner to be visited.

Last function generates the successors. It behaves very similarly to the one in the search problem in terms of updating Pacman’s position. In our case, we also check whether the current position is a corner and change the Booleans accordingly. We then apply the updated Booleans to all the valid successor positions.

Although our chosen state space is not abstract enough to support certain changes (for example, changing the number of corners), it was chosen due to the relative simplicity and elegance of a 6-element tuple.

Testing:



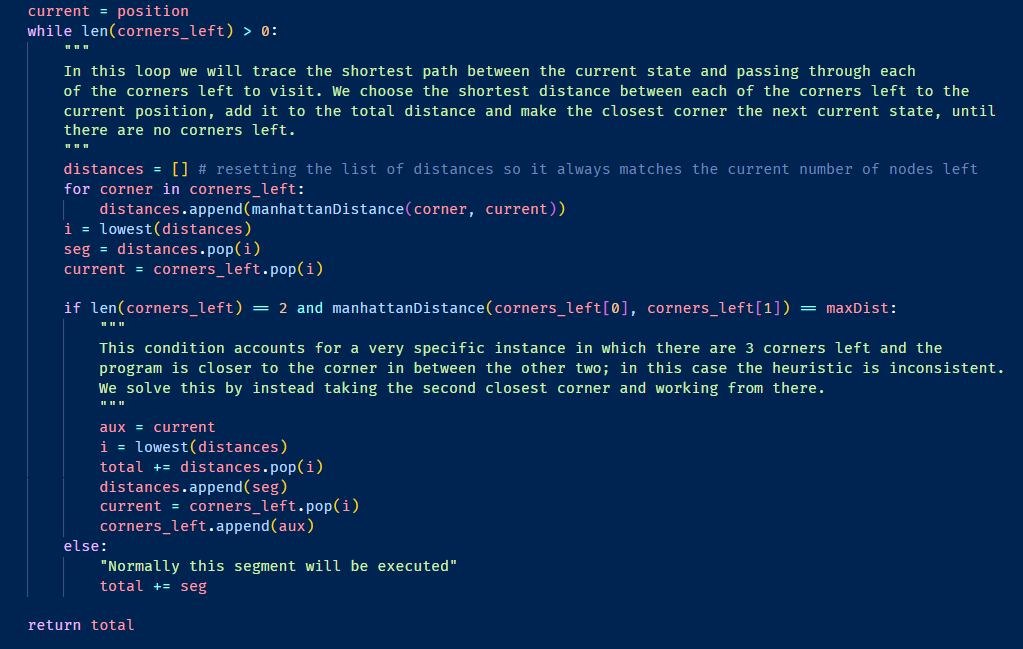
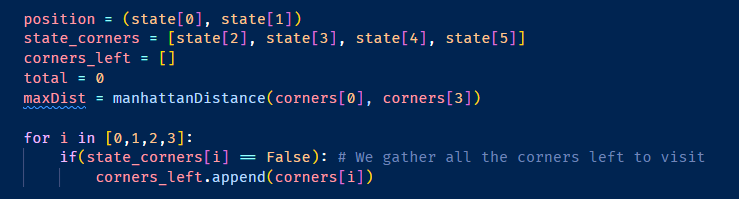
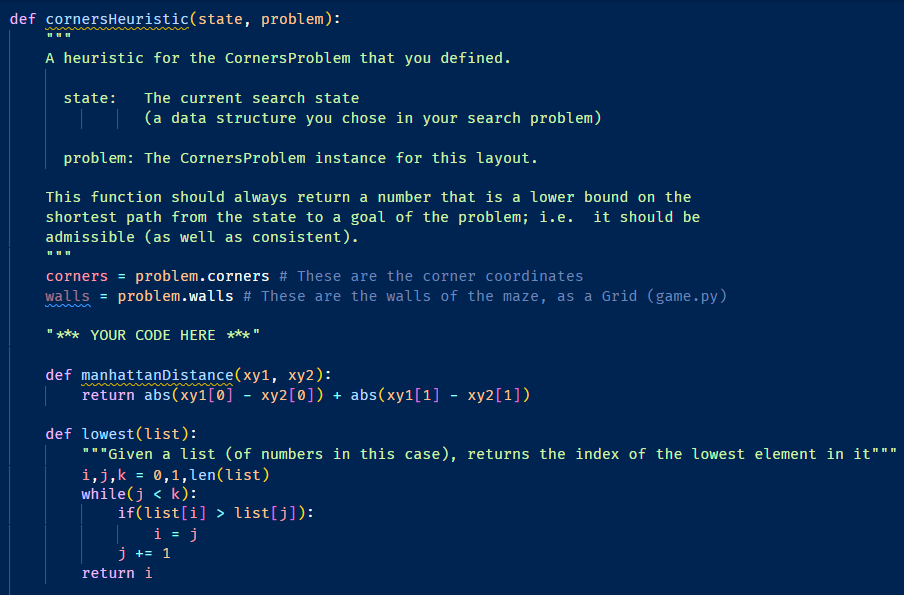
Note: we did not consider necessary to include screenshots of Pacman navigating the maze since the red gradient information is disabled, which was the only useful information it gave.

Due to this problem being much more complex than the search problem, it is not surprising to see the number of nodes expanded to be much greater than the total cost of the path found; the number of possible states quadrupled with the new state space, which would make the graph much larger to navigate. Since we are working with breadth first search and graph search, we already know Pacman is guaranteed to reach the goal, and with an optimal path.

## Section 6: Corners Problem: Heuristic

Code:

The goal in this last section is to come up with a consistent heuristic for the corners problem, so that the a\* algorithm is guaranteed to find an optimal path for graph search.



Note: this is the entirety of the heuristic function, split up in three screenshots; each one ends immediately where the next one starts.

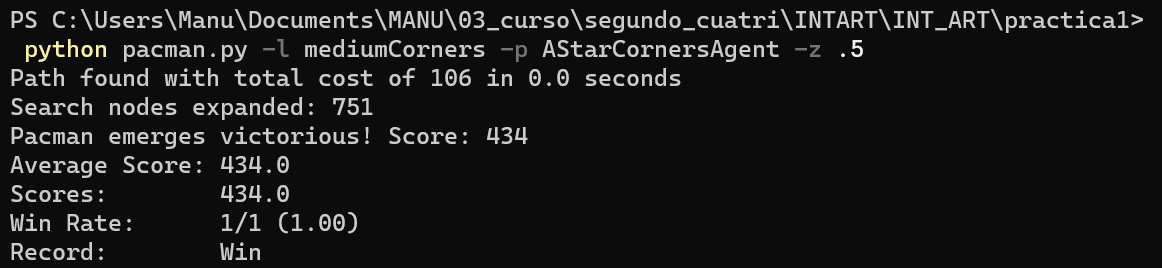
Our heuristic will rely on the Manhattan distance, since we created an internal function to calculate it.

**Question to answer: explain the logic behind your heuristic**

Our heuristic applies the principle of relaxation: we calculate the hypothetical lowest Manhattan distance from our current state to the corners left to visit of the maze. We first gather all the remaining corners in a list. We then enter the loop (third screenshot), in which we will find the lowest distance between the “current” position and the remaining corners. We compute all the distances into a list, calculate the lowest (by using the internal function of the same name) and the corner corresponding to the lowest distance becomes the new “current” position, it is removed from the list of remaining corners, and said distance is added to the total distance. This process is repeated until there are no corners left in the list, and we return the total distance.

The heuristic ideally constructs the shortest path to get to the goal by adding intermediate paths. However, there is a specific case in which the heuristic becomes inconsistent: there are three corners left to visit and Pacman is closer to the one corner in the middle of the other two. In this case, the current described heuristic would return a value resembling one where there are four corners left. Our if loop inside the while loop intends to resolve this issue: instead of taking the shortest distance, we would take the second shortest, so that we visit one of the exterior corners, and the remaining path aligns how we need it to.

Testing:



We can observe the a\* algorithm finding the lower cost path with dramatically less expanded nodes than the tests in Section 5. We can see the true power of the a\* algorithm now that we are working with a more complex problem.