# **Report**

**Assignment 1: Search**

**Chiara Mocetti, Ebad Malik – P10**

## 

## ***Search algorithms (search.py)***

1. **General design (0.5 points):**
   1. **Describe the data structure used to represent the search node and whether it was necessary to modify it at any point of the assignment to facilitate the implementation of any of the search algorithms.**

In order to represent the search node we used a tuple storing the following information: the position tuple (x, y), the action (direction) and the cost of the action. This means that no modifications were done. We eventually decided to keep the code simple for explainability and readability. We explain in (ii), the alternative strategies that we thought of and why we decided against them.

* 1. **Explain the alternatives considered and the reasons why this representation was chosen.**

Another alternative considered was a search node containing the path from the start node as well, however, keeping the path separate helped us proceed in our algorithm with ease maintaining readability and generality of the algorithm. Intead, we kept the path inside our queue strategy, as a tuple with the state. Therefore, our queue algorithms, such as stack, queue etc kept a tuple of (state,path) which were pushed in and popped from them.

## ***Section 1 (2.0 points)***

1. **Design:**
   1. **Explain the approach taken to design the search algorithm.**

In order to design the search algorithm we first thought about the implementation of all the different search methods (Depth First Search, Breadth First Search, Uniform Cost Search and A\* Search). While we designed them we noticed shared lines of code within their implementation and we then decided to write a unique search algorithm following the pseudocode of graph-search. We used an open-list (whose type was the queue strategy i.e stack queue etc) and a closed-list to store visited nodes.

Regarding the Depth First Search we run the Search Algorithm using the open-list as a stack.

* 1. **List and explain the functions you used, among those provided to implement the assignment.**

For the implementation of the Section 1 we used the following functions: isGoalState() to check whether the analyzed note is the goal state; getSuccessors() to access to the visited node’s successors to be expanded and getStartState() to access to the start state. For our graph search, we have utilized the SearchAlgorithm(), which is used in all the search algorithms as general graph-search.

* 1. **Include and explain the code you implemented.**

**def SearchAlgorithm(problem, open\_list):**

**"""A general Graph-Search algorithm implementation. which takes in a problem configuration**

**And returns a path to the goal state using the algorithm-specific strategy provided through open\_list.**

**Args:**

**problem (\*\*\*SearchProblem): Different Layouts of the grid, position etc**

**Also contains menthods as the StartState, GoalState, Successors of any node, Cost of Action etc.**

**open\_list (queuing strategy): It is algorithm-specific queue structure, Stack, Queue, PriorityQueue etc**

**contains the current state and the path traversed**

**Returns:**

**Returns the path taken by the graph search algorithm with specific queuing strategy**

**Example:**

**>>SearchAlgorithm(PositionSearchProblem, open\_list):**

**['West', 'West', 'West', 'West', 'South', 'South', 'East', 'South', 'South', 'West']**

**"""**

**closed\_list = [] # List of states that have been visited**

**(current\_state, path) = open\_list.pop() #Get the initial state and path**

**while not problem.isGoalState(current\_state):**

**if current\_state not in closed\_list:**

**closed\_list.append(current\_state)**

**successors = problem.getSuccessors(current\_state)**

**#Add Each successor node to the open\_list**

**for (successor, direction, cost) in successors:**

**current\_path = path + [direction]**

**open\_list.push((successor, current\_path))**

**if open\_list.isEmpty() == True:**

**return None**

**(current\_state, path) = open\_list.pop()**

**return path**

**def depthFirstSearch(problem):**

**"""**

**Search the deepest nodes in the search tree first. (Using Stack)**

**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()))**

**"""**

**open\_list = util.Stack()**

**open\_list.push((problem.getStartState(), []))**

**return SearchAlgorithm(problem, open\_list)**

1. **Efficiency of the search algorithm:**
   1. **How many nodes are expanded?**
   2. **Does it reach the optimal solution?**
   3. **Is it optimal?**

We show the run of Tiny Maze with DFS:

15 nodes are expanded

Total Cost of Path = 10

Optimal Cost = 8

It reaches the goal but it is not optimal.

1. **Answer question 1.1 of the assignment statement.**

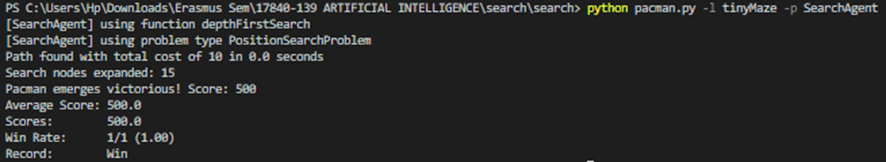
The exploration order is as expected since we are using depth first search with a stack and we see in our implementation that pacman does not go to all the explored states, instead it expands the deepest node first by going to the darker red route.

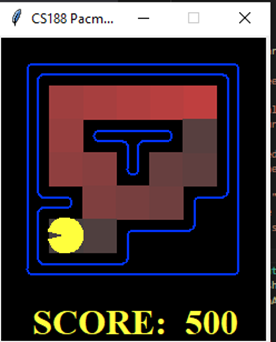
1. **Answer question 1.2 of the assignment statement.**

It is not a least costly solution since DFS is not optimal or complete. It requires backtracking when it reaches a dead end while it is expanding the deepest node.

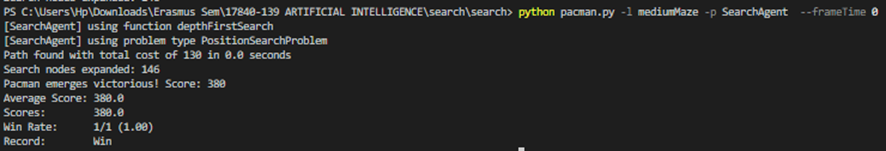
1. **Tests: Include the tests performed to illustrate the search results.**

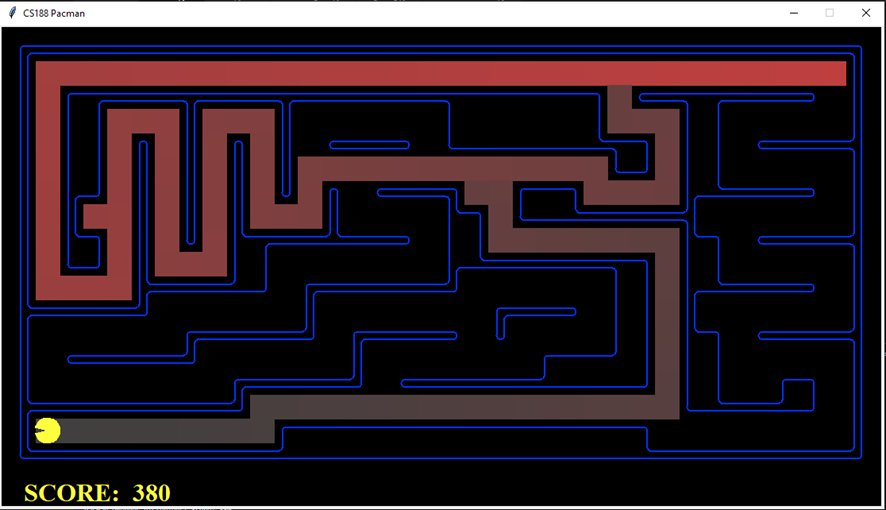
*Tiny Maze*:

****

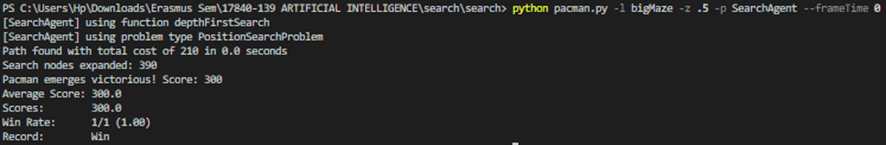
****

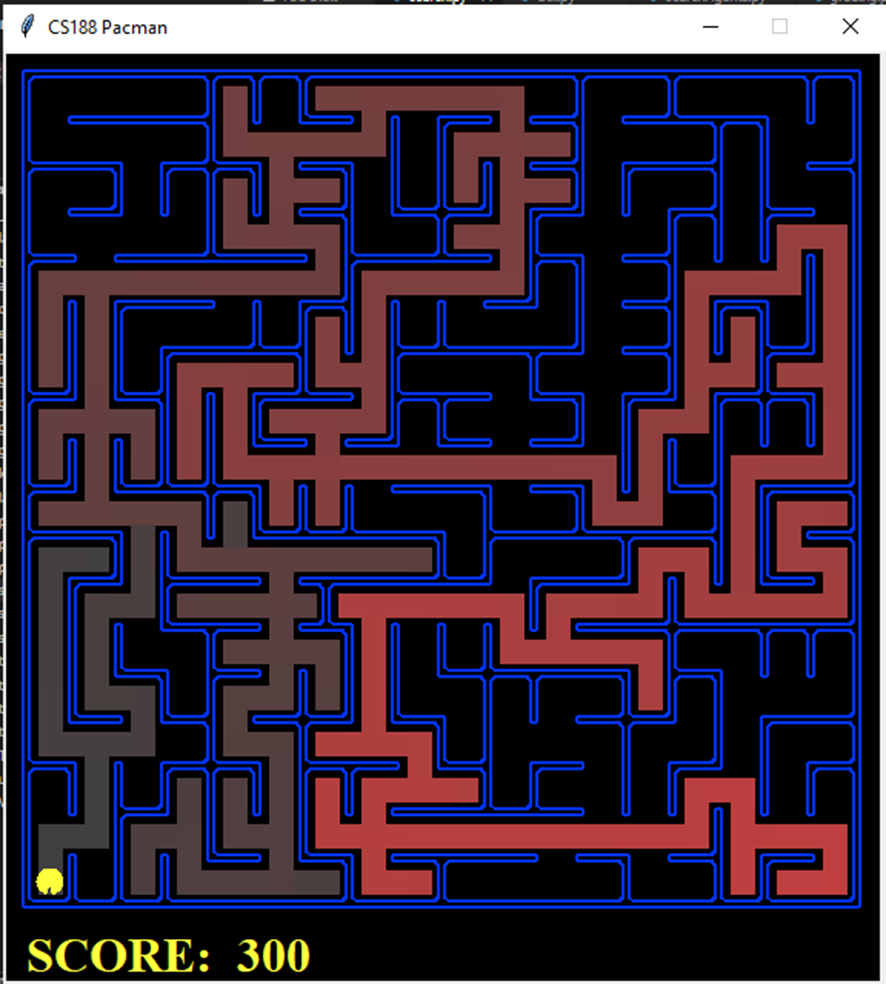
*Medium Maze:*

****

****

*Big Maze:*

****

****

## ***Section 2 (0.75 points)***

1. **Design:**
   1. **Explain the approach taken to design the search algorithm.**

Regarding the Breadth First Search we run the Search Algorithm using an open-list queue.

* 1. **List and explain the functions you used, among those provided to implement the assignment.**

For the implementation of the Section 2 we used the following functions: isGoalState() to check whether the analyzed note is the goal state; getSuccessors() to access to the visited node’s successors to be expanded and getStartState() to access to the start state. For our graph search, we have utilized the SearchAlgorithm(), which is used in all the search algorithms as general graph-search.

* 1. **Include and explain the code you implemented.**

**def breadthFirstSearch(problem):**

**"""**

**Search the shallowest nodes in the search tree first (Using Queue).**

**"""**

**open\_list = util.Queue()**

**open\_list.push((problem.getStartState(), []))**

**return SearchAlgorithm(problem, open\_list)**

1. **Efficiency of the search algorithm:**
   1. **How many nodes are expanded?**
   2. **Does it reach the optimal solution?**
   3. **Is it optimal?**

We show the run of Medium Maze with BFS:

269 nodes are expanded

Path found with total cost of 68

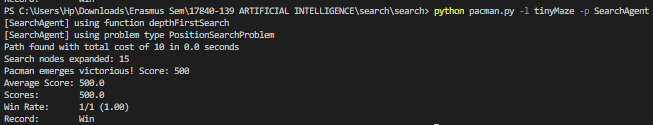
It reaches the goal with optimal cost. Therefore, it is optimal.

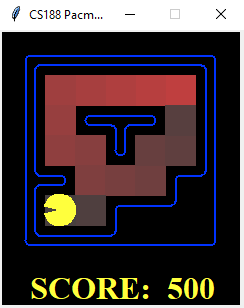
1. **Answer question 2.1 of the assignment statement.**

Yes, BFS finds the optimal solution.

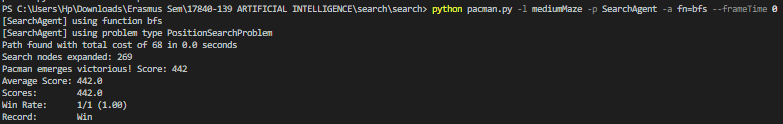
1. **Tests: Include the tests performed to illustrate the search results.**

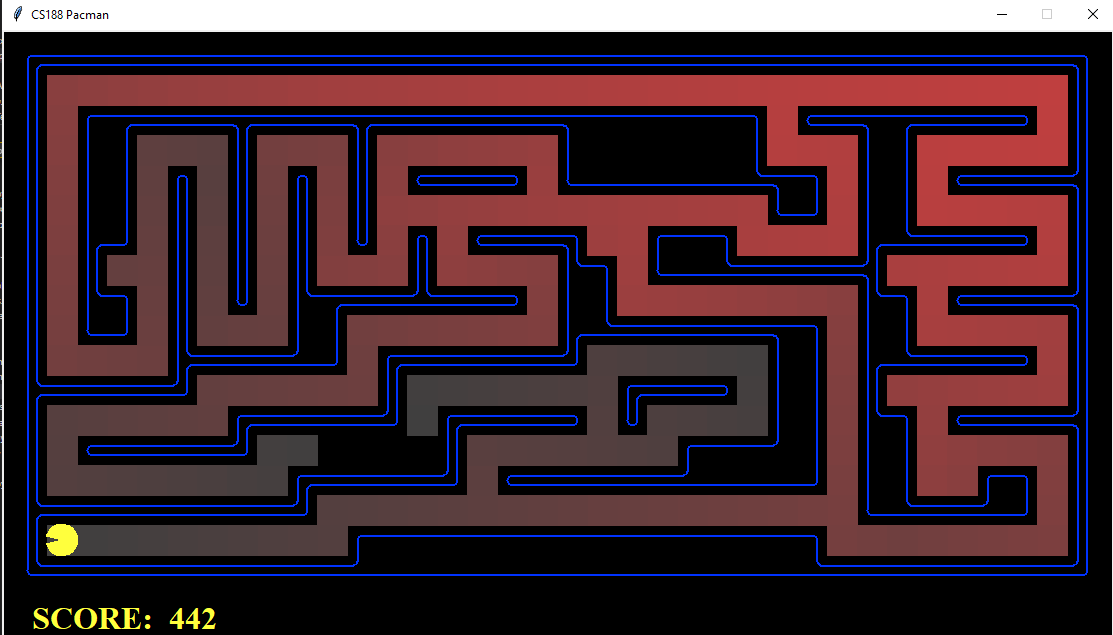
*Tiny Maze:*

****

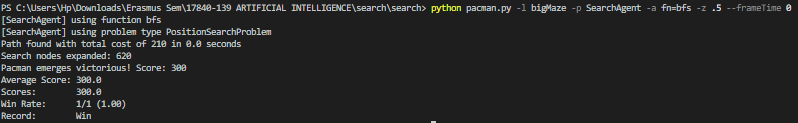
****

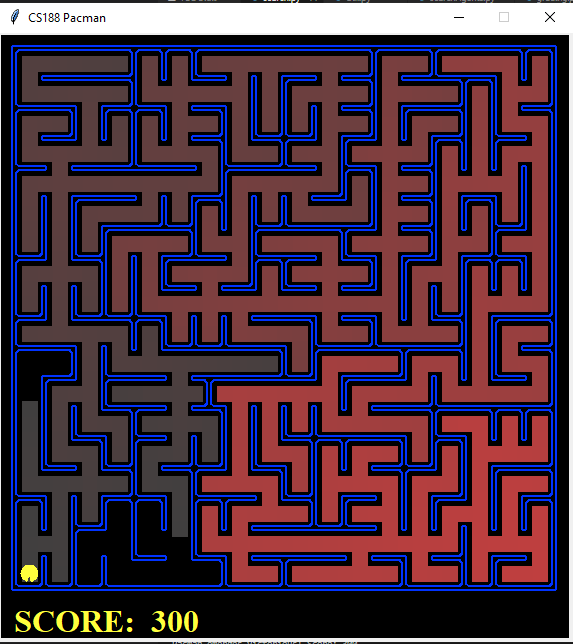
*Medium Maze:*

****

****

*Big Maze:*

****

****

## ***Section 3 (0.75 puntos)***

1. **Design:**
   1. **Explain the approach taken to design the search algorithm.**

Regarding the Uniform Cost Search we run the Search Algorithm using as open-list a priority queue, giving as priority function the cost of the path retrieved using getCostofActions() function with our new class implementation:

class PriorityQueueWithPathFunction(PriorityQueue):

"""

Implements a priority queue with the same push/pop signature of the

Queue and the Stack classes. This is designed for drop-in replacement for

those two classes. The caller has to provide a priority function, which

extracts each item's priority using it's path and/or heuristic.

Design is similar to PriorityQueueWithFunction, however, we utilise the

the Cost function on the Path variable and if needed the heuristic of

the State.

If heuristic is not provided, works as a priority queue for for UCS.

Else (heuristic is provided), works as a priority queue for A\* search.

"""

def \_\_init\_\_(self, priorityFunction, problem = None, heuristic = None):

"priorityFunction (item) -> priority"

self.priorityFunction = priorityFunction # store the priority function

self.heuristic = heuristic

self.problem = problem

PriorityQueue.\_\_init\_\_(self) # super-class initializer

def push(self, item):

"Adds an item to the queue with priority from the priority function"

if(self.heuristic is not None):

PriorityQueue.push(self, item, self.priorityFunction(item[1]) + self.heuristic(item[0],self.problem))

else:

PriorityQueue.push(self, item, self.priorityFunction(item[1]))

* 1. **List and explain the functions you used, among those provided to implement the assignment.**

For the implementation of the Section 3 we used the following functions: isGoalState() to check whether the analyzed note is the goal state; getSuccessors() to access to the visited node’s successors to be expanded and getStartState() to access to the start state and getCostofActions() to save the cost of the path. For our graph search, we have utilized the SearchAlgorithm(), which is used in all the search algorithms as general graph-search.

* 1. **Include and explain the code you implemented.**

**def uniformCostSearch(problem):**

**"""**

**Search the node of least total cost first.**

**(Using Priority Queue with Path Function)**

**"""**

**open\_list = \**

**util.PriorityQueueWithPathFunction(problem.getCostOfActions)**

**open\_list.push((problem.getStartState(), []))**

**return SearchAlgorithm(problem, open\_list)**

1. **Efficiency of the search algorithm:**
   1. **How many nodes are expanded?**
   2. **Does it reach the optimal solution?**
   3. **Is it optimal?**

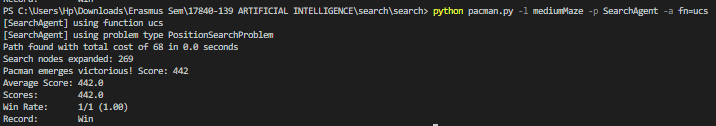
We show the run of Medium Scary Maze with UCS:

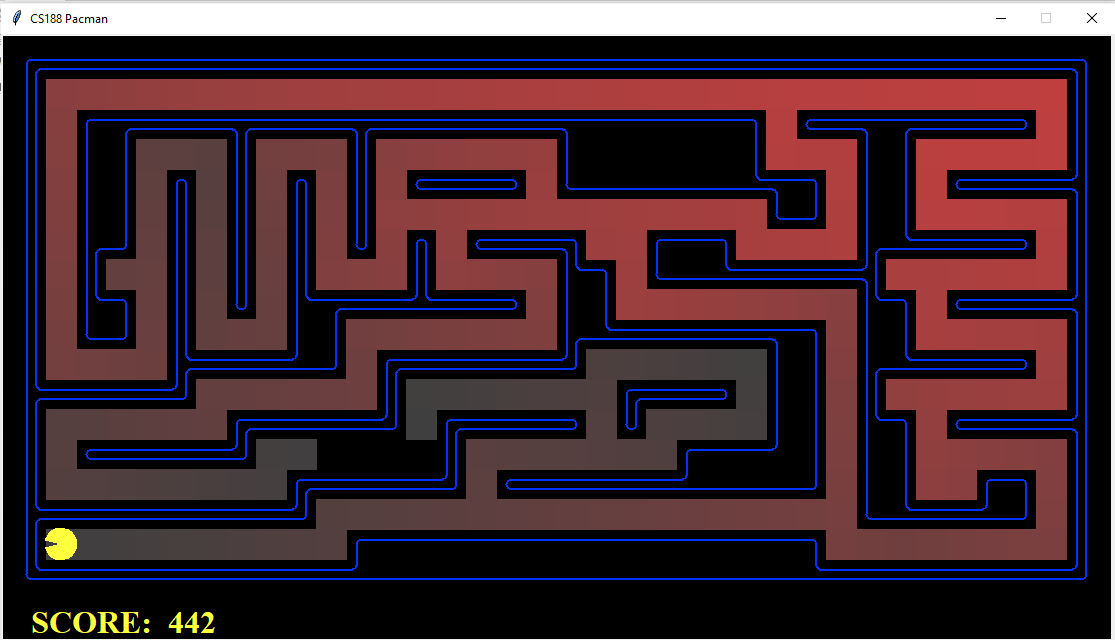
108 nodes are expanded.

It reaches the goal with optimal cost. Therefore, it is optimal. UCS is optimal.

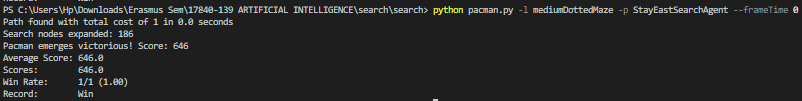
1. **Tests: Include the tests performed to illustrate the search results.**

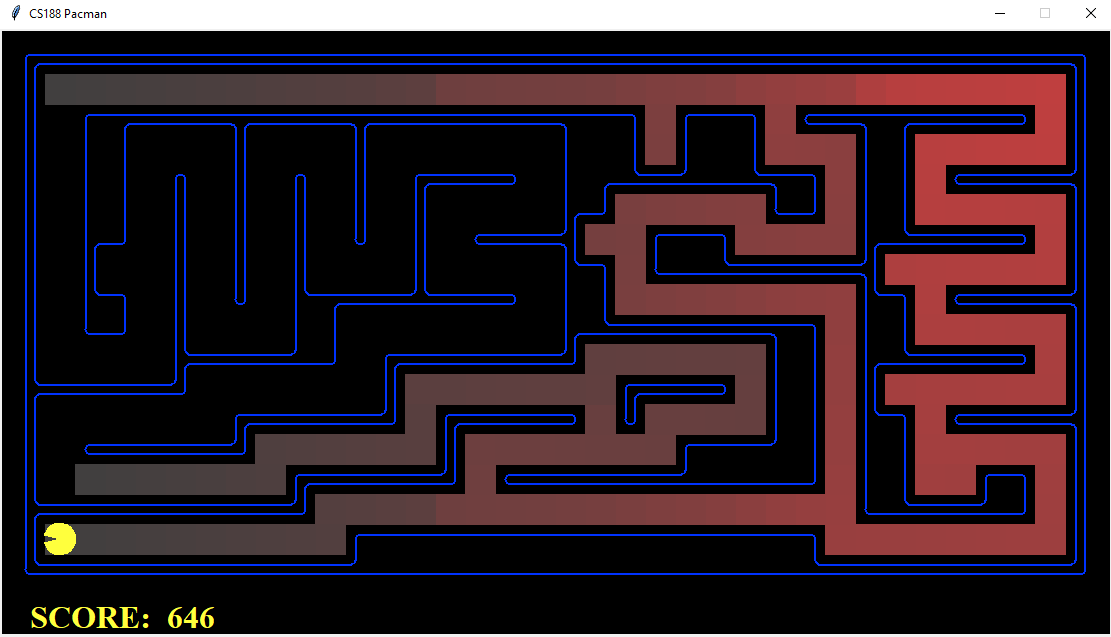
*Medium Maze:*

****

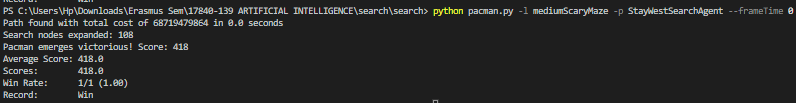
****

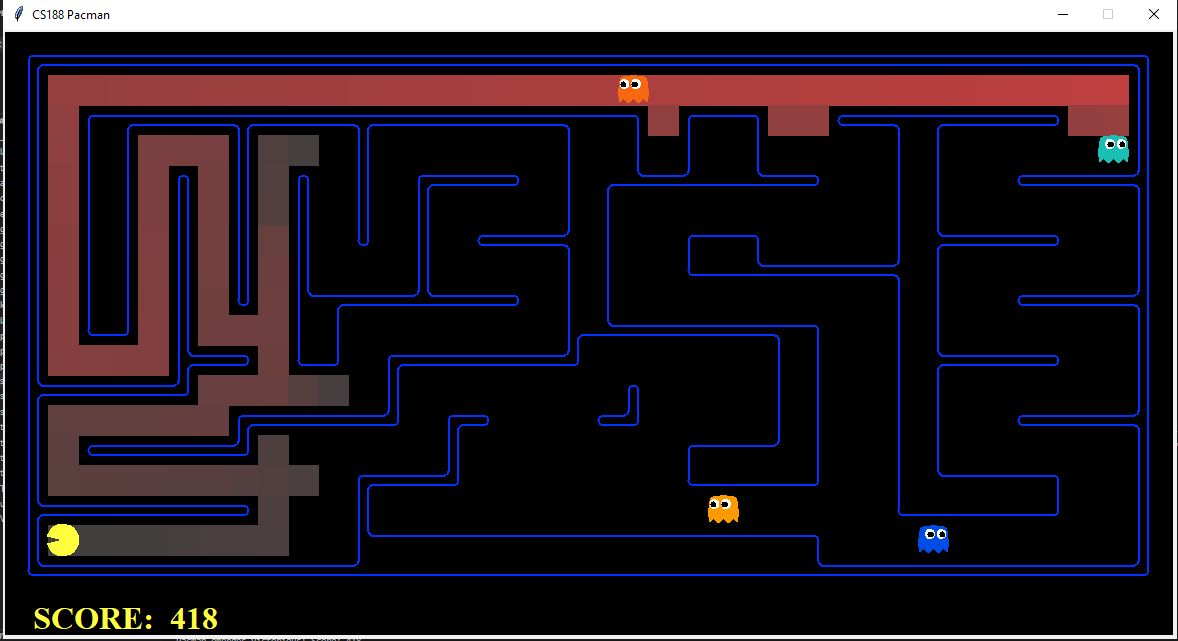
*Medium Dotted Maze:*

****

****

*Medium Scary Maze:*

****

****

***Section 4 (2 points)***

1. **Design:**
   1. **Explain the approach taken to design the search algorithm.**

Regarding the A\* Search we run the Search Algorithm using as open-list as a PriorityQueuewithPathFunction(). It is a class we have made for easy and generalization of PriorityQueue. Code is mentioned above.

Implements a priority queue with the same push/pop signature of the

Queue and the Stack classes. This is designed for drop-in replacement for

those two classes. The caller has to provide a priority function, which

extracts each item's priority using it's path and/or heuristic.

* 1. **List and explain the functions you have used, among those provided to implement the assignment.**

For the implementation of the Section 4 we used the following functions: isGoalState() to check whether the analyzed note is the goal state; getSuccessors() to access to the visited node’s successors to be expanded and getStartState() to access to the start state and getCostofActions() to save the cost of the path.

* 1. **Include and explain the code you implemented.**

**def aStarSearch(problem, heuristic=nullHeuristic):**

**"""Search the node that has the lowest combined cost and heuristic first.**

**(Using Priority Queue with Path Function) """**

**open\_list = \**

**util.PriorityQueueWithPathFunction(problem.getCostOfActions,**

**problem, heuristic)**

**open\_list.push((problem.getStartState(), []))**

**return SearchAlgorithm(problem, open\_list)**

1. **Efficiency of the search algorithm:**
   1. **How many nodes are expanded?**
   2. **Does it reach the optimal solution?**
   3. **Is it optimal?**

We show the run of Medium Scary Maze with UCS:

Path found with total cost of 210 in 0.1 seconds.

Search nodes expanded: 549.

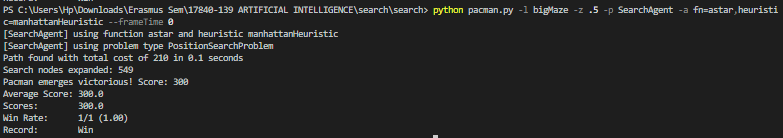
It reaches the optimal solution. A\* is Optimal!

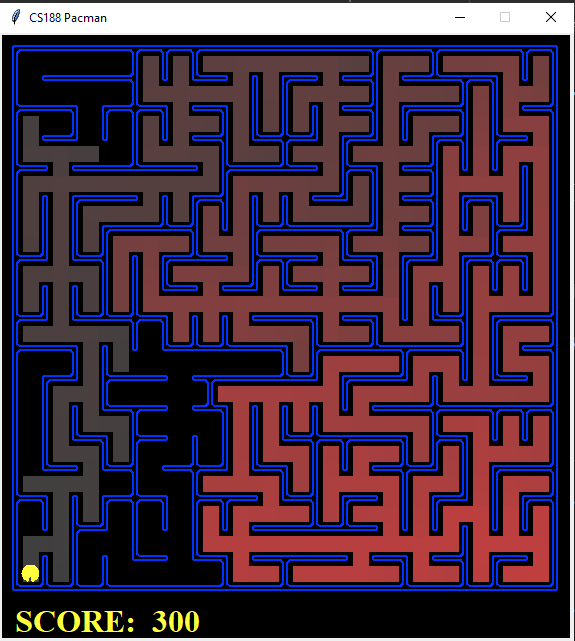
1. **Answer question 4.1 of the assignment statement.**

Different search strategies result in different exploration methods of the maze, As each of them follow a different strategy. We implemented DFS,BFS,UCS and A\*. DFS is done with a stack, BFS with a queue, UCS and A\* with a priority Queue. They all have a different strategy that can result in different exploration paths towards the goal.

1. **Tests: Include the tests performed to illustrate the search results.**

*Big Maze:*

******



## 

## ***Search-based agents (searchAgents.py)***

## ***Section 5 (2 points)***

1. **System state:**
   1. **Describe how the system state (not to be confused with "search node") is characterized to solve this search problem.**

Our System state is a named tuple that contains the position of the node as well as the list of visited corners. This makes it easier to check with each state the unvisited corners and the visited corners and how many corners are left to visit (whether goal is complete or not).

**node\_tuple = namedtuple('State', ['position', 'visited\_corners'])**

* 1. **Explain the alternatives that have been considered and the reasons why this representation has been chosen.**

Alternate strategies that we considered were without visited corner information in the state and as a class variable. However, This is not a smart solution since it can easily be solved with the named tuple solution that we implemented. This makes it easier for code readability as well.

* 1. **Include examples of specific states.**

STATE Example 1 : State(position=(2, 5), visited\_corners=[(6, 1), (6, 6)])

STATE Example 2 : State(position=(1, 6), visited\_corners=[(6, 1), (6, 6), (1, 6)])

STATE Example 3 : State(position=(6, 1), visited\_corners=[(1, 1), (1, 6), (6, 6), (6, 1)])

STATE Example 4 : State(position=(3, 2), visited\_corners=[(1, 1)])

1. **Implementation:**
   1. **List and explain the functions you have used, among those provided to implement the assignment.**

For the implementation of the Section 5 we used the following function: Actions.directionToVector(action)

* 1. **Include and explain the code you implemented.**

**class CornersProblem(search.SearchProblem):**

**"""**

**This search problem finds paths through all four corners of a layout.**

**You must select a suitable state space and successor function**

**"""**

**def \_\_init\_\_(self, startingGameState):**

**"""**

**Stores the walls, pacman's starting position and corners.**

**"""**

**self.walls = startingGameState.getWalls()**

**self.startingPosition = startingGameState.getPacmanPosition()**

**top, right = self.walls.height-2, self.walls.width-2**

**self.corners = ((1,1), (1,top), (right, 1), (right, top))**

**for corner in self.corners:**

**if not startingGameState.hasFood(\*corner):**

**print('Warning: no food in corner ' + str(corner))**

**self.\_expanded = 0 # DO NOT CHANGE; Number of search nodes expanded**

**# Please add any code here which you would like to use**

**# in initializing the problem**

**"\*\*\* YOUR CODE HERE \*\*\*"**

**def getStartState(self):**

**"""getStartState**

**Initializes and returns the start state as a named tuple,**

**containing the starting position (x,y) and the corners that**

**have been visited.**

**Returns:**

**named\_tuple: Start State containing position & visited corners**

**"""**

**# Declaring namedtuple()**

**node\_tuple = namedtuple('State', ['position', 'visited\_corners'**

**])**

**# Initializing named tuple with the start state**

**start\_state = node\_tuple(self.startingPosition, [])**

**return start\_state**

* 1. **Explain what the function 'getSuccessors' returns in this problem.**

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

**"""getSuccessors: Return sucessor states of a state**

**Calculates the true successors of a state. True successors are**

**the ones that are not a wall position and can access a corner**

**that is unvisited.**

**Args:**

**state: Contains position (x,y) on the grid & the visited corners.**

**Returns:**

**list: valid successor states, the actions they require, and a cost of 1.**

**For a given state, this returns a list of triples, (successor,**

**action, stepCost), where 'successor' is a valid successor to the current**

**state, 'action' is the action required to get there, and 'stepCost'**

**is the incremental cost of expanding to that successor**

**Example:**

**>>> print(self.walls[4][4])**

**1**

**Arbitrary problem with a wall at (4,4)**

**>>> State = State(position=(4, 5), visited\_corners=[])**

**>>> problem.getSuccessors(state)**

**[**

**(State(position=(4, 6), visited\_corners=[]), 'North', 1),**

**(State(position=(5, 5), visited\_corners=[]), 'East', 1),**

**(State(position=(3, 5), visited\_corners=[]), 'West', 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]**

**"\*\*\* YOUR CODE HERE \*\*\*"**

**node\_tuple = namedtuple('State', ['position',**

**'visited\_corners']) # Declaring namedtuple()**

**visited\_corners = state.visited\_corners # Corners already visited**

**(x, y) = state.position # Current position**

**(dx, dy) = Actions.directionToVector(action) # Get Cost of Successor**

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

**hits\_wall = self.walls[nextx][nexty]**

**# If it is not a wall**

**if not hits\_wall:**

**visited\_corners\_successor = list(visited\_corners) # initialise with the corner's already visited by Pacman**

**next\_node = (nextx, nexty)**

**# Append to Successor's visited if he can access a corner that we have NOT visited**

**if next\_node in self.corners:**

**if next\_node not in visited\_corners\_successor:**

**visited\_corners\_successor.append(next\_node)**

**# Populate complete states as (( next\_node, successor's visited corners), Direction, Cost)**

**successor = node\_tuple(next\_node,**

**visited\_corners\_successor) # Create the named tuple from the successor**

**next\_state = (successor, action, 1) # Create the tuple of the state from the successor**

**successors.append(next\_state)**

**self.\_expanded += 1 # DO NOT CHANGE**

**return successors**

* 1. **Describe the implementation of 'isGoalState' for this problem.**

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

**"""isGoalState**

**Checks if we have visited another corner or not,**

**If we have, Add it to the visited corners list in the state tuple.**

**Return True if all the corners are successfully visited.**

**Args:**

**state: Contains position (x,y) on the grid & the visited corners.**

**Returns:**

**bool: Returns True if all four corners are visited, False otherwise**

**"""**

**# Append to visited if current position is a corner that we have NOT visited**

**if state.position in self.corners:**

**if not state.position in state.visited\_corners:**

**state.visited\_corners.append(state.position)**

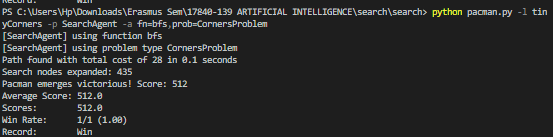
**# If all 4 corners are visited, return true, false otherwise**

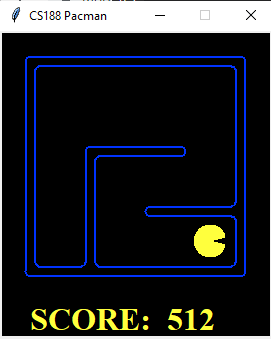
**return len(state.visited\_corners) == len(self.corners)**

**return False**

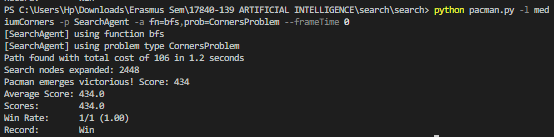
1. **Tests: Include the tests performed to illustrate the implementation.**

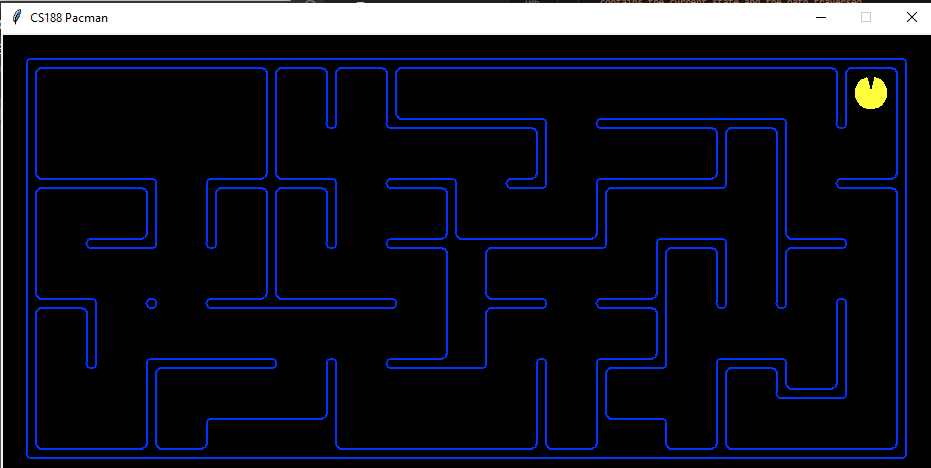
*Tiny Corners*

****

****

*Medium Corners*

******



## 

## ***Section 6 (2 points)***

1. **Heuristic:**
   1. **Describe the process followed to design the heuristic.**

In order to design the heuristic we used the relaxation method. In this specific context, we defined the relaxed problem removing the restrictions represented by the walls. Afterwards, we implemented the algorithm using as heuristic the optimal Manhattan distance of visiting all the unvisited corners.

Considering all the different configurations of Pacman with food in 1, 2, 3 and 4 corners we realized there was a configuration for 3, for which the so defined heuristic was not consistent. To solve this issue, we implemented a special case. It is explained in the doc string.

* 1. **Explain the heuristic logic.**

Shown below in the code implemented

1. **Implementation:**
   1. **List and explain the functions you have used, among those provided to implement the assignment.**

For the implementation of Section 6 we used the function manhattanDistance() to compute the distance between the corners for the heuristic function. We have also used getOppositeCorner() which we have explained as a docstring.

* 1. **Include and explain the code you implemented.**

**def getOppositeCorner(visited\_corner, corners\_to\_visit):**

**"""getOppositeCorner:**

**Calculates the furthest corner to the visited\_corner using**

**manhattan distance from all the unvisited corners,**

**This is the corner opposite to the visited\_corner.**

**This helper function is used in cornersHeuristic.**

**Args:**

**visited\_corner (tuple): Position of the corner to find opposite corner from**

**corners\_to\_visit (list) : List of unvisited corners**

**Returns:**

**tuple: the corner's position which is opposite to the visited\_corner.**

**Example:**

**>>> corners\_to\_visit**

**[(1, 1), (1, 6), (6, 1)]**

**>>> visited\_corner**

**(6,6)**

**>>> getOppositeCorner(visited\_corner, corners\_to\_visit)**

**(1,1)**

**"""**

**maxDist = 0**

**for corner in corners\_to\_visit:**

**if util.manhattanDistance(visited\_corner, corner) > maxDist:**

**maxDist = util.manhattanDistance(visited\_corner, corner)**

**opposite\_corner = corner**

**return opposite\_corner**

**def cornersHeuristic(state, problem):**

**"""cornersHeuristic:**

**Finds a consistent/monotonic heuristic to visit all the unvisited corners.**

**This is a lower bound on the shortest path from the state to a goal**

**of the problem. It makes use of Manhattan Distance to calculate distance**

**and chooses the nearest corner until all corners are visited.**

**However, This is not consistent for the case of 3 unvisited corners,**

**Which can take heuristic distance with a diagonal.**

**For example:**

**if bottom-right(6,6) is already visited.**

**Pacman at (3,1) goes (1,1) -> (1,6) --> (6,1) = 17**

**Pacman at (4,1) goes (6,1) -> (1,1) --> (1,6) = 12**

**Hence, Not consistent.**

**We fix it by making sure that the opposite corner of the**

**visited corner is not visited first.**

**After fix:**

**Pacman at (3,1) goes (6,1) -> (1,1) --> (1,6) = 13**

**Pacman at (4,1) goes (6,1) -> (1,1) --> (1,6) = 12**

**This is Consistent!**

**Args:**

**problem (\*\*\*\*\*\*Problem): Different Layouts of the grid, position etc**

**Also contains menthods as the StartState, GoalState, Successors of any node, Cost of Action etc.**

**state: Contains position (x,y) on the grid & the visited corners.**

**Returns:**

**tuple: A admissible and consistent heuristic number.**

**"""**

**corners = problem.corners # These are the corner coordinates**

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

**visited\_corners = state[1]**

**corners\_to\_visit = []**

**for corner in corners:**

**if corner not in visited\_corners:**

**corners\_to\_visit.append(corner)**

**# While not all corners are visited find via manhattanDistance**

**# the most efficient path for each corner**

**total\_cost = 0**

**current\_point = state[0]**

**# With 3 corners to visit, we remove the opposite corner of the already visited corner,**

**# Since visiting this opposite corner first will result in inconsistent heuristic.**

**# Therefore, with 3 corners, we always visit corners in an L shape.**

**removed\_corners = []**

**if len(corners\_to\_visit) == 3:**

**opposite\_corner = getOppositeCorner(visited\_corners[0],**

**corners\_to\_visit)**

**corners\_to\_visit.remove(opposite\_corner)**

**removed\_corners.append(opposite\_corner)**

**while corners\_to\_visit:**

**# For each corner in corners\_to\_visit, get the manhattanDistance, get the closest corner and choose that one to visit.**

**distances = []**

**for corner in corners\_to\_visit:**

**distances.append(util.manhattanDistance(current\_point,**

**corner))**

**heuristic\_cost = min(distances)**

**min\_index = distances.index(heuristic\_cost)**

**corner = corners\_to\_visit[min\_index]**

**# Remove the chosen corner as it is visited, change pacman position, add the cost**

**corners\_to\_visit.remove(corner)**

**current\_point = corner**

**total\_cost += heuristic\_cost**

**# Add back the removed corners into corners to visit list, since now they are a viable option as next corner to be visited.**

**for corner in removed\_corners:**

**corners\_to\_visit.append(corner)**

**distances.append(util.manhattanDistance(current\_point,**

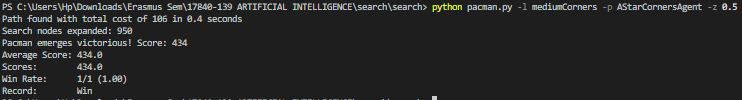
**corner))**

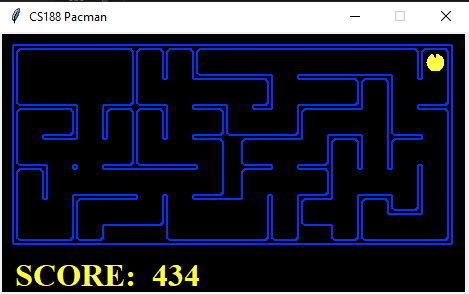
**removed\_corners.remove(corner)**

**return total\_cost**

* 1. **Tests: Include the results of the tests performed to analyze the heuristic.**

*Medium Corners:*

****

****