P1: Search

Kevin de la Coba and Juan Luis Sanz

Section 1

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

Our main struggle in the first section was how to make the code generic. First, we created a class to store a search-tree, but we realised that a search-tree is not really needed in the algorithm, instead of that, we could use the structure that the **problem class** gave us. When we were getting the starting node, we realised that this one was given without any additional information (cost and action to get to it), our solution to this problem was to add this information in the node, but before going into any detail let's see the structure we used:

[state, action, cost, [route to this state]]

We have a **list** in which we include the **current state**, the **action** to get there, the **accumulated cost** and the **route** from the starting state to the current node. Each node is stored in the closed and opened lists when it is specified by the algorithm. We save the route to the current state, so when we are in the goal state, we must return the route to this node.

DFS (depth first search) uses a stack. The stack is the one who give us the correct node from the opened-list. We made a generic function that receives a problem and the **opened-list**, so in the DFS function we call this generic function sending as an argument the **stack** data structure.

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

The exercise was developed using the following framework functions:

- We used from *util.py* the class **stack**. The methods we used from the structure were **isEmpty**, **push** and **pop**.
- As we were receiving as an argument a *SearchProblem*, we used from this class the functions:
  - getStartState: To get the initial state in the problem.
  - isGoalState: To check if a state is a goal state.
  - getSuccessors: To get the successors of one state.

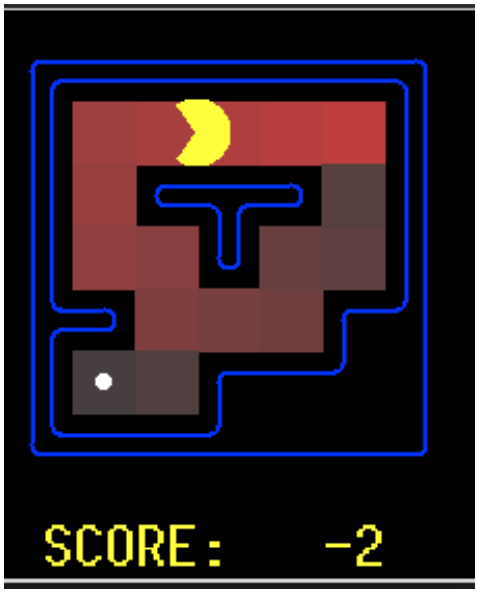
1.3. Includes code written by students (0.25 pts)

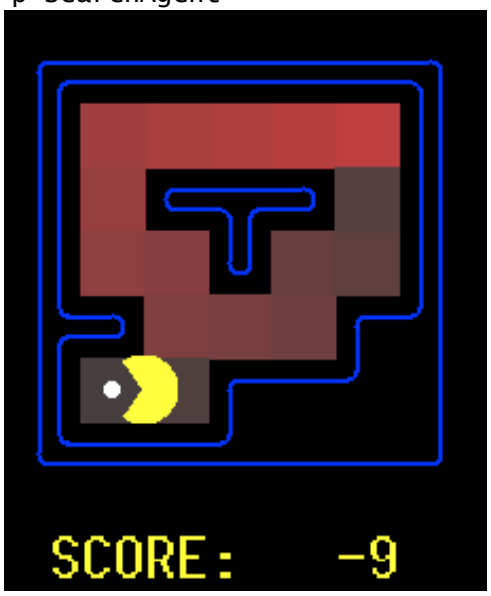
```
def depthFirstSearch(problem):
    return solveSimpleSearch(problem, util.Stack())

def solveSimpleSearch(problem, opened_list):
    start_state = problem.getStartState()

    # Initialize the opened-list with root-node
    opened_list.push([start_state, None, 0, []]) # [Starting state, Last action, Cost,
Path]

    # Initialize the closed-list as an empty list
    closed_list = []

    # Iterating
    while True:
        # If the open list is empty error
        if opened_list.isEmpty():
            return None
```

```
# Getting the node from the opened list
current_node = opened_list.pop()

# Checking if this is the goal
if problem.isGoalState(current_node[0]):
    return current_node[-1]

# If the node is not in the closed list we add it and
# we expand it and we add it to the closed list
if not any(current_node[0] == node[0] for node in closed_list):
    closed_list.append(current_node)
    # Iterating through the successors and adding them to the open list
    for child in problem.getSuccessors(current_node[0]):
        child_node = list(child)

        # Adding to the child the route to get there
        # child path = current node path + action to get to the node
        child_node.append(current_node[-1])
        child_node[-1] = current_node[-1] + [child[1]]

        # Adding to the child the accumulated cost
        child_node[2] += current_node[2]

        # Adding the node to the closed list
        opened_list.push(child_node)
```

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

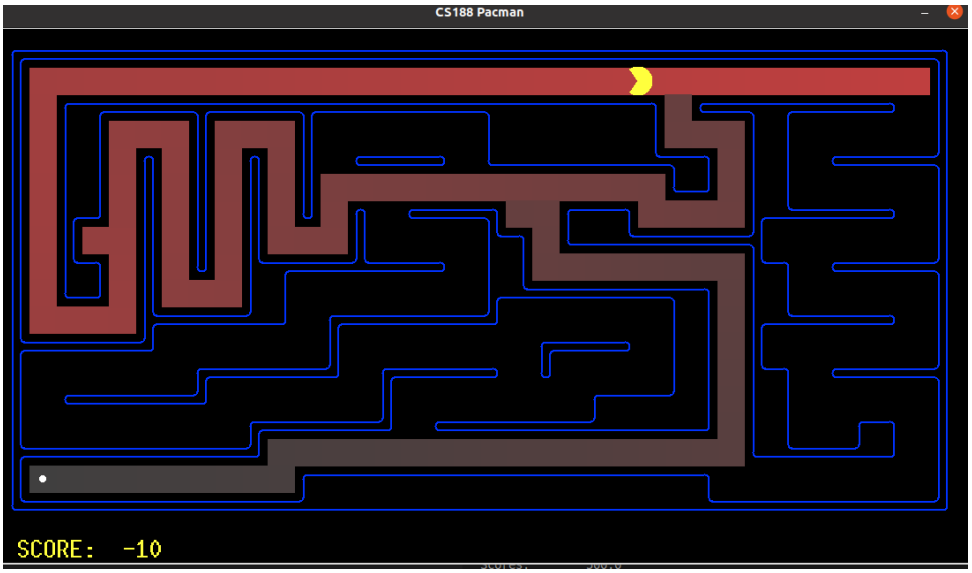
`python pacman.py -l tinyMaze -p SearchAgent`

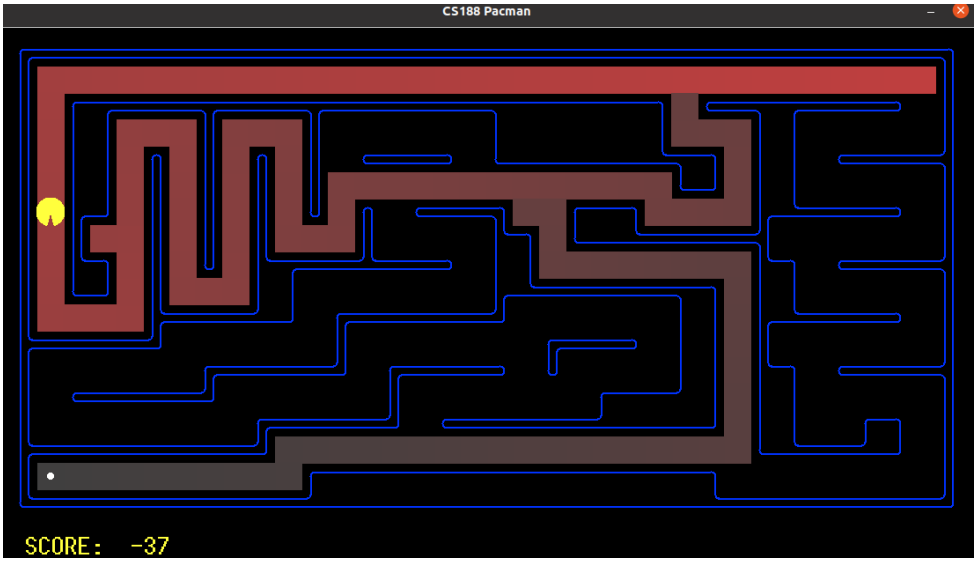


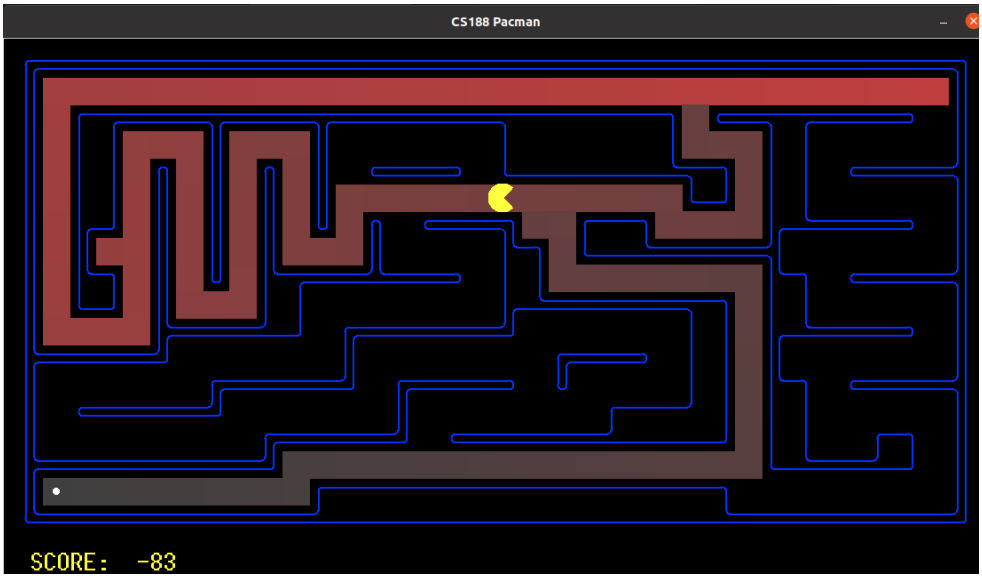


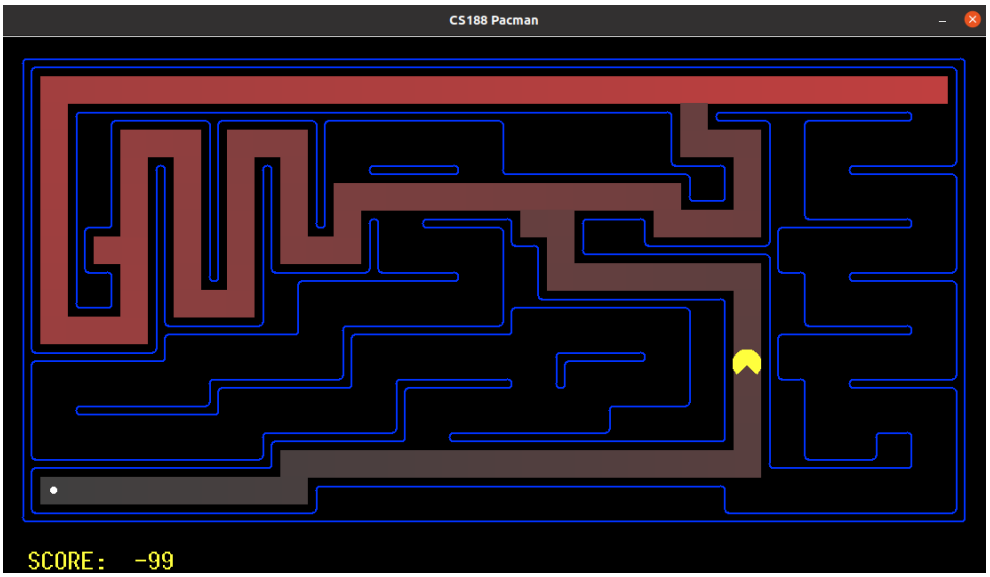
When running the game, a Pacman board is shown, and the states explored are marked in red. The results given seem to be correct as they are the same than the ones given in the hint of the statement and autograder is satisfied.
We can see that the path is not the shortest one.

`python pacman.py -l mediumMaze -p SearchAgent`


CS188 Pacman
SCORE: -10

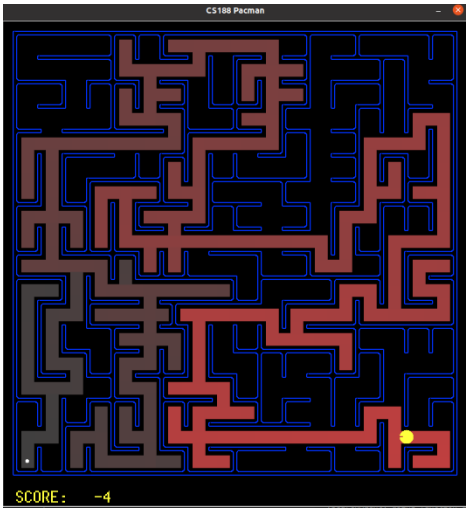

CS188 Pacman
SCORE: -37

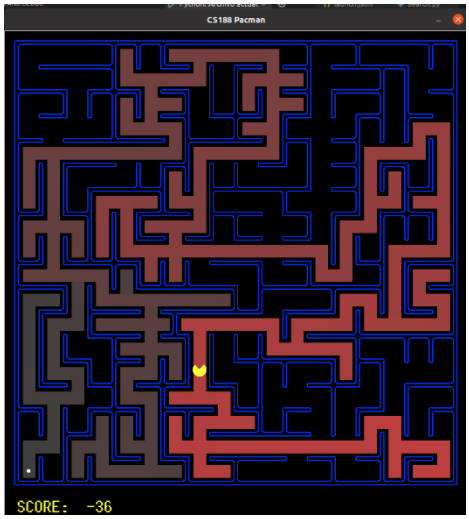

CS188 Pacman
SCORE: -83

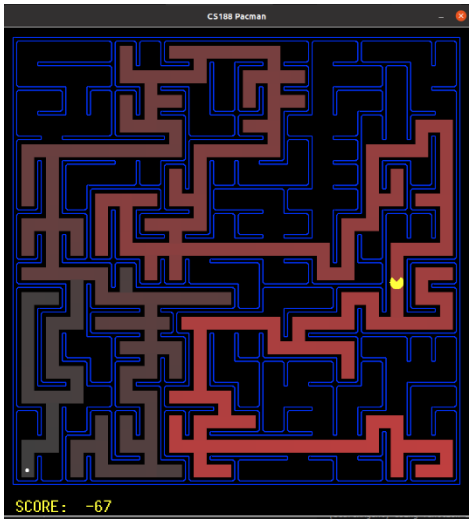


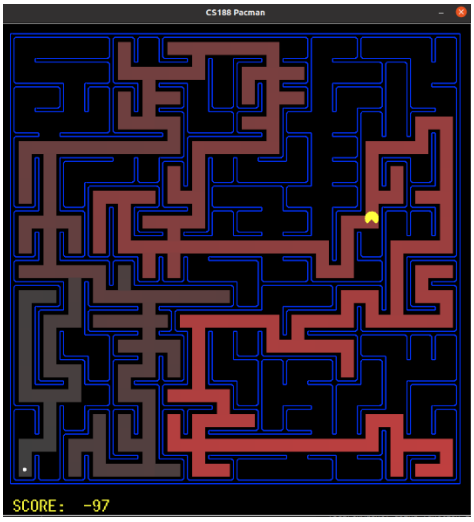
Again, Pacman is not following the shortest path. We can see that DFS unless it finds a "wall" it looks for the first path he finds, we can see that at some point DFS explored a path that was going up, it was going to an already explored state (a "wall"), as he could not expand that node he went back and continued with a path which ended in the goal state.

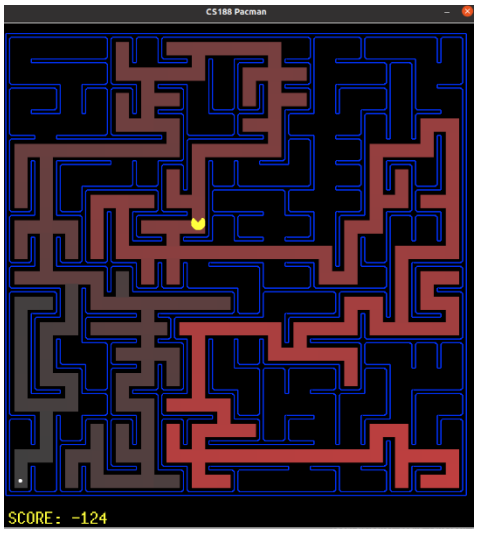
```
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

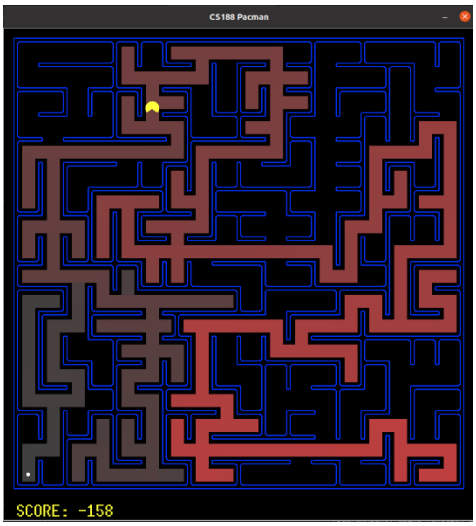


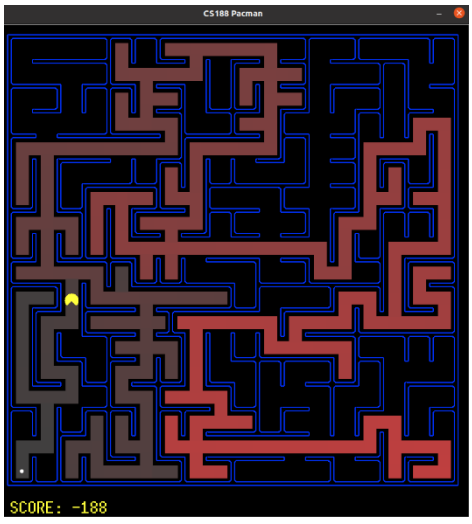


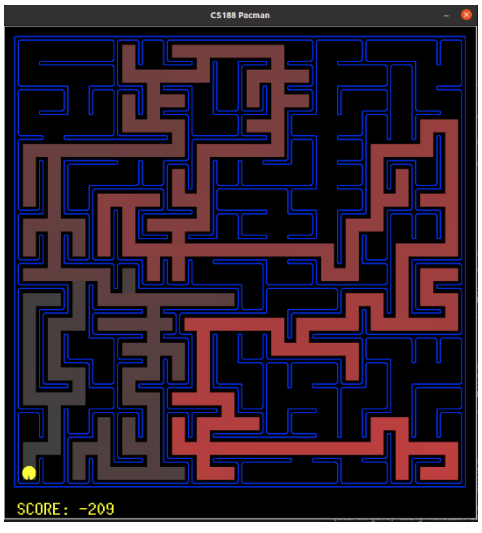












In the big maze we can see that not many nodes have been explored but, as before the path is not the optimal.

1.2. Conclusions on the behavior of pacman, it is optimal (y / n), reaches the

DFS **is not optimal**, in our case the **solution is reached** but the DFS algorithm does not guarantee a solution.

- **15 nodes** were expanded having a final **cost of 10** using the tinyMaze.
- In the mediumMaze **146 nodes** were expanded having a final **cost of 130**.
- In the bigMaze **390 nodes** were expanded having a final **cost of 210**.

1.3. Answer to question 1.1 (1pt) ***Is the exploration order what you would have expected?***

Well, we were not expecting anything because we do not know the mazes by heart, but as we are not allowing cycles in the algorithm, we were expecting that the pacman could find the solution.

1.4. Answer to question 1.2 (1pt) ***Does Pacman actually go to all the explored squares on his way to the goal?***

No, pacman does not go to all the explored squares.

1.5. Answer to question 2 (1pt) ***Is this a least cost solution? If not, think about what depth-first search is doing wrong.***

No, it is not, and the reason is because DFS does not choose the nodes from the opened-list efficiently. DFS is always expanding the last nodes that have been introduced in the opened-list until the path built by choosing this nodes ends with a wall (that expands a node without childs). If the first node contains the most efficient route to a goal-state then DFS is efficient, otherwise it is not.

Section 2

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

As we already built the generic function in the *section 1*, we just send to the generic function a **queue** as the opened-list, so BFS (breadth first search) uses a queue as a data structure.

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

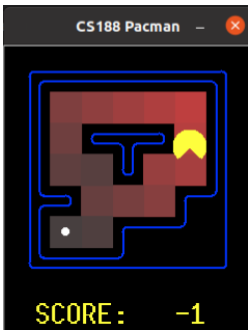
In this section we used the same generic code as in the *section 1*. The main difference is that we used a different *util.py* class, in this case a **queue**. The methods used were **isEmpty**, **pop** and **push**.

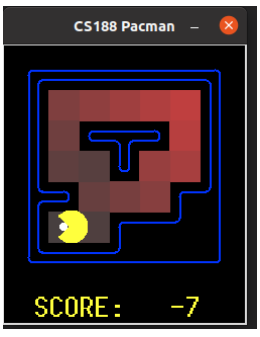
2.3. Includes code written by students (0.25 pts)

```
def breadthFirstSearch(problem):
    return solveSimpleSearch(problem, util.Queue())
```

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

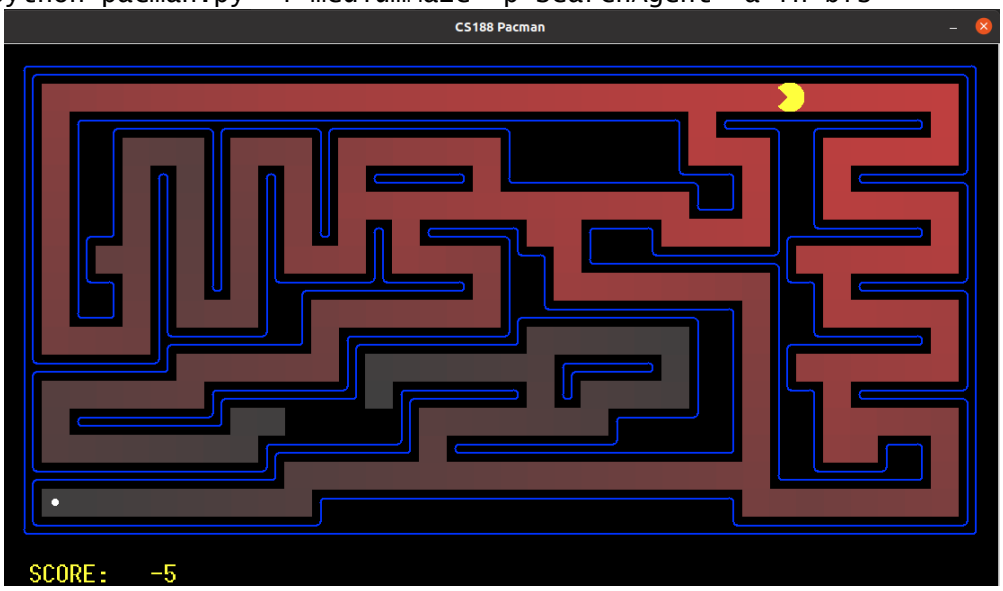
```
python pacman.py -l tinyMaze -p SearchAgent
```

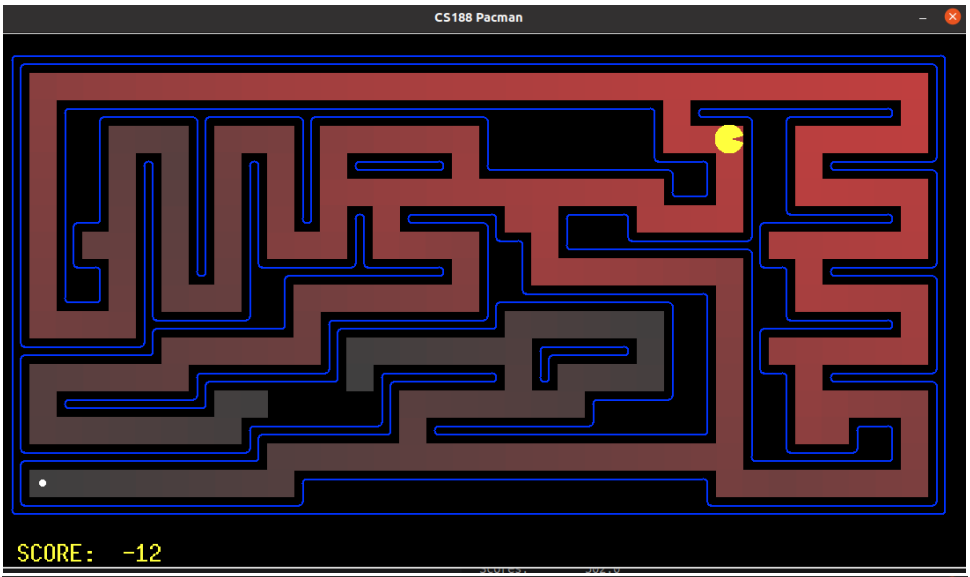


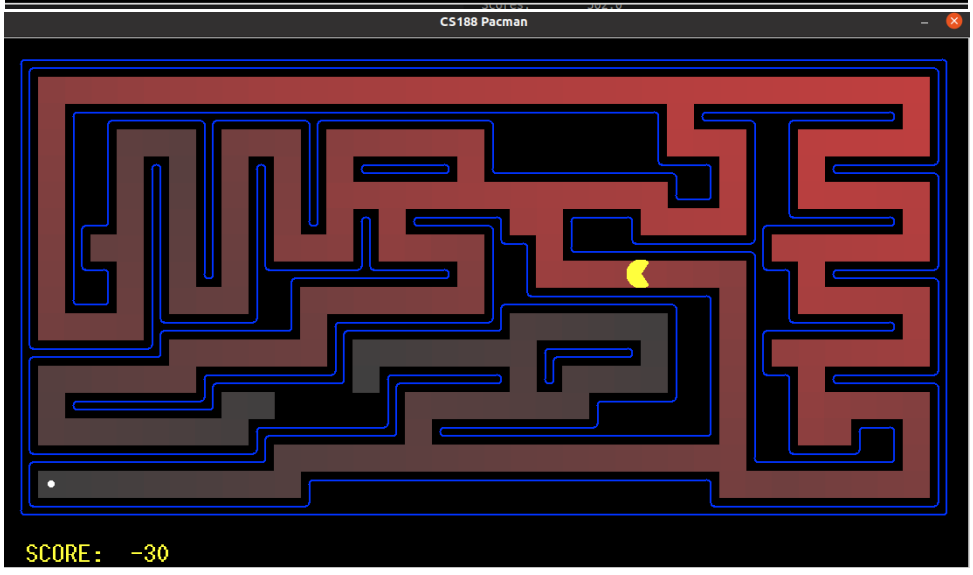


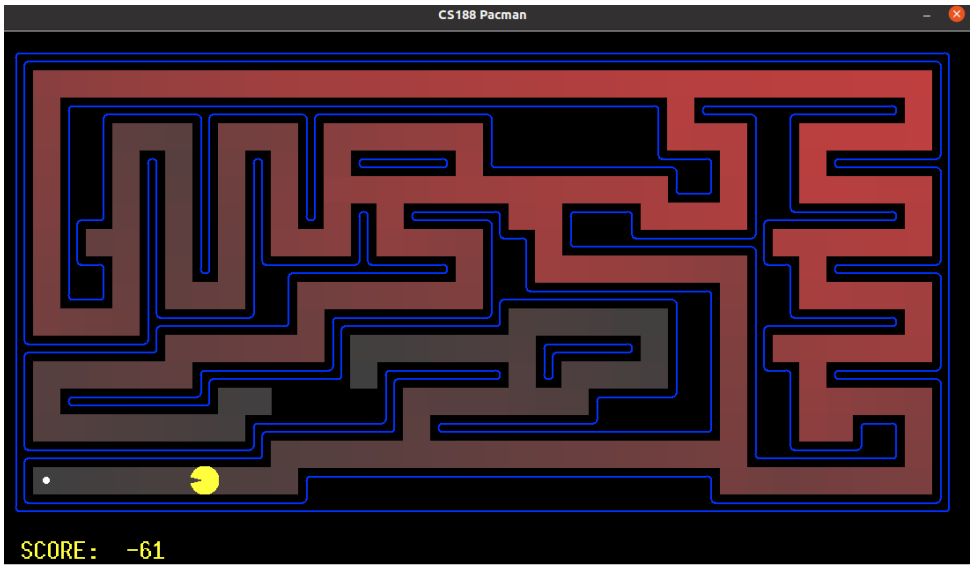
We can see that Pacman now chooses the shortest path.

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



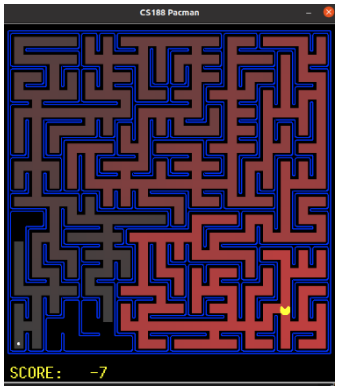


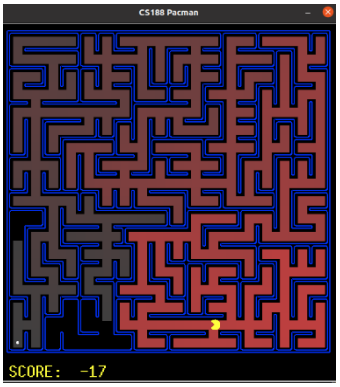


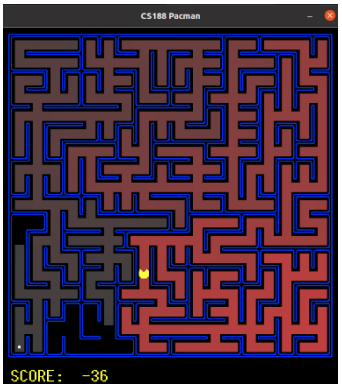


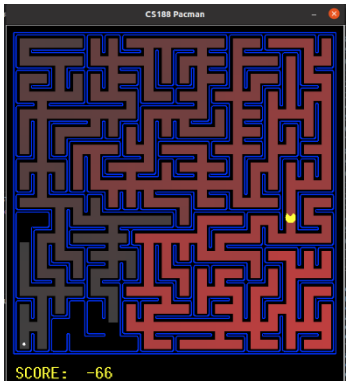
We can see that it still chooses the shortest path but almost all positions have been generated.

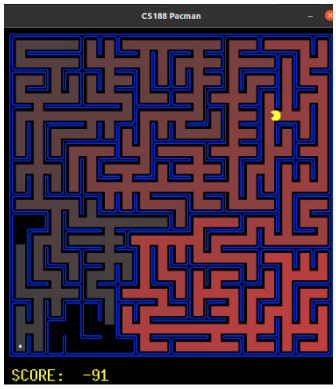
```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

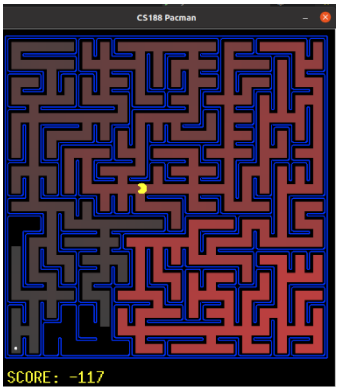


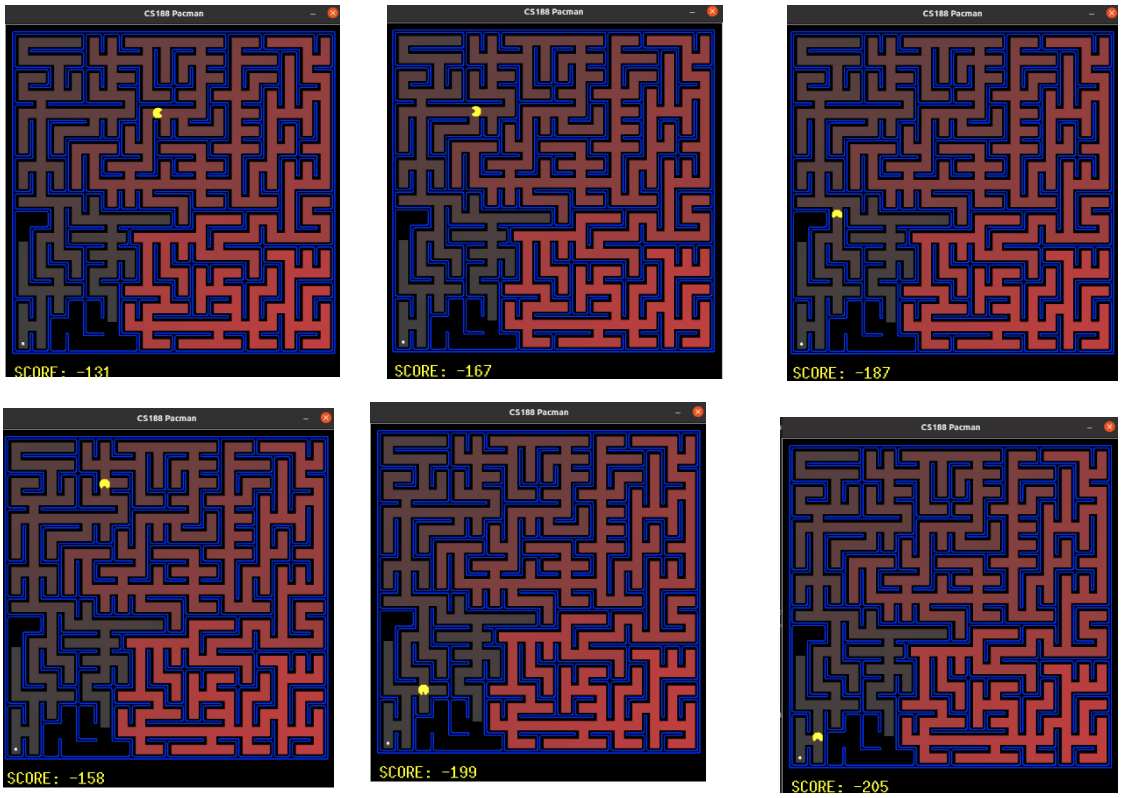












As before the route is the shortest one but, we can see that almost all the positions were generated.

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

Yes, it is optimal and **yes**, it reaches the solution.

- In the tinyMaze it **expands 15 nodes** having a **final cost of 8**.
- In the mediumMaze it **expands 269 nodes** having a final **cost of 68**
- In the bigMaze it **expands 620 nodes** having a final **cost of 210**.

If we compare it with DFS we can see that DFS expands less nodes, but it does not find the shortest path, BFS does but it expands more nodes than DFS.

2.6. Answer to question 3 (1pt) *Does BFS find a least cost solution? If not, check your implementation.*

Yes, it does.

Section 3

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

For this section we were still using the generic function of the *section 1* and *2*. In this section we need to take into account the cost of each state, so for that the algorithm needs a priority queue.

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

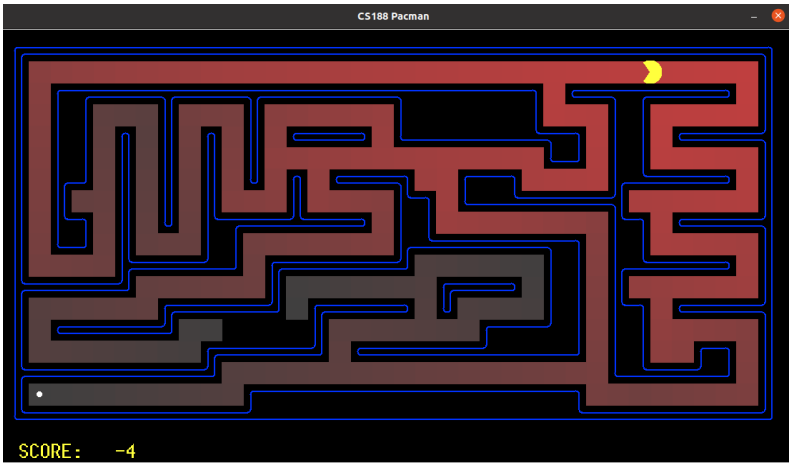
As in *section 1* and *2*, we are still using the generic function to solve this kind of problems. This time we still used a class from *util.py*, we used **priorityQueueWithFunction**. This class allowed us to, whenever we push an element, take the cost, and save it as the priority. The methods we used were **isEmpty**, **pop** and **push**.

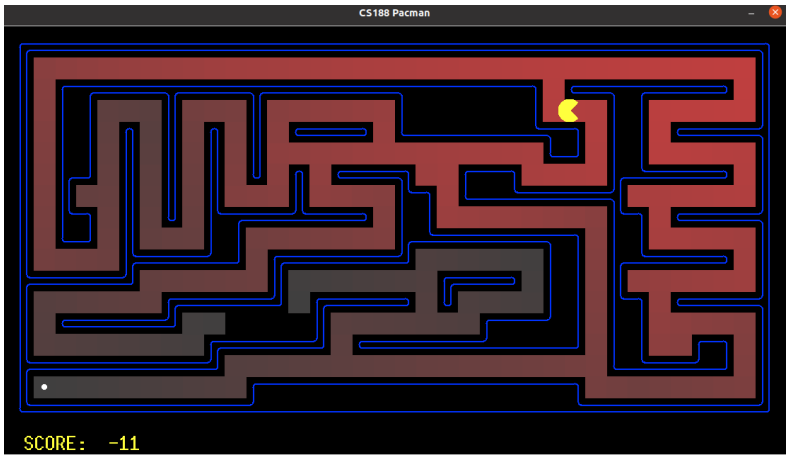
3.3. Includes code written by students (0.25 pts)

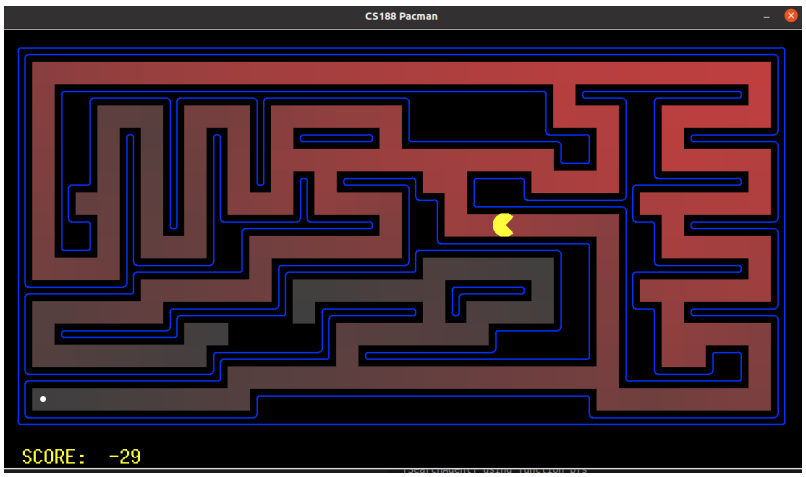
```python
def uniformCostSearch(problem):
    # priorityFunction = lambda state: state[2]
    def priorityFunction(state):
        return state[2]
    openlist = util.PriorityQueueWithFunction(priorityFunction)
    return solveSimpleSearch(problem, openlist)
```

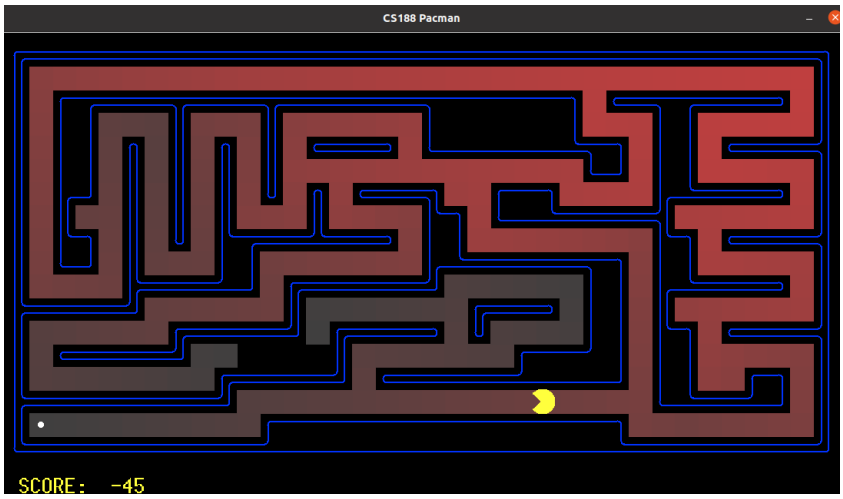
3.4. Screenshots of executions and test carried out analyzing the results (1pt)

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



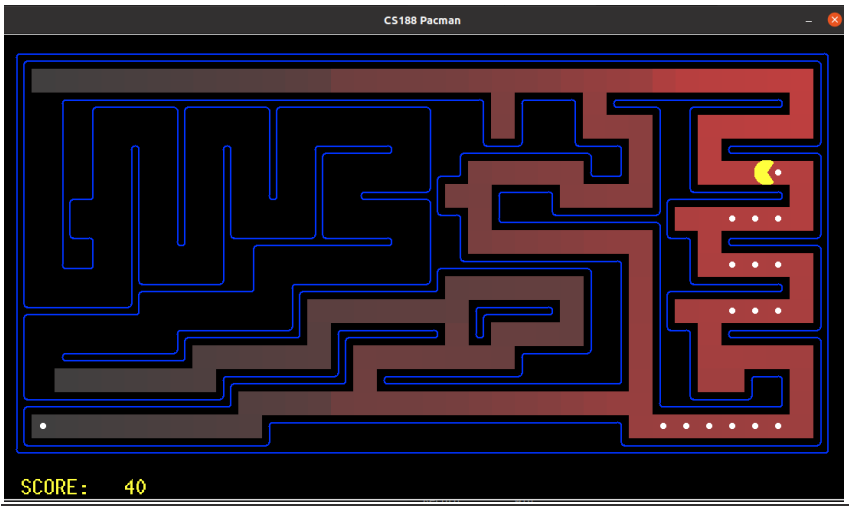


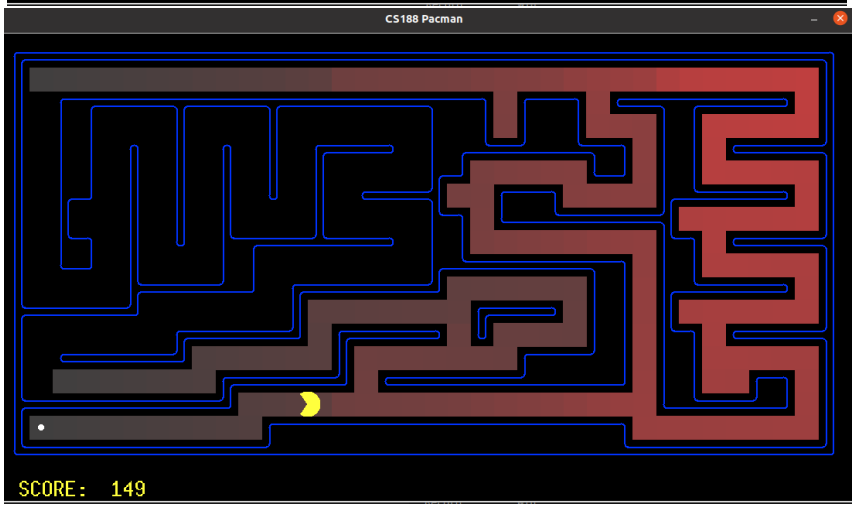




We can see that the generated nodes are almost the same as it was with the breadth first search algorithm, it also follows the same path as breadth first search, the shortest one. This is because all the positions have a cost equal to 1.

```
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
```

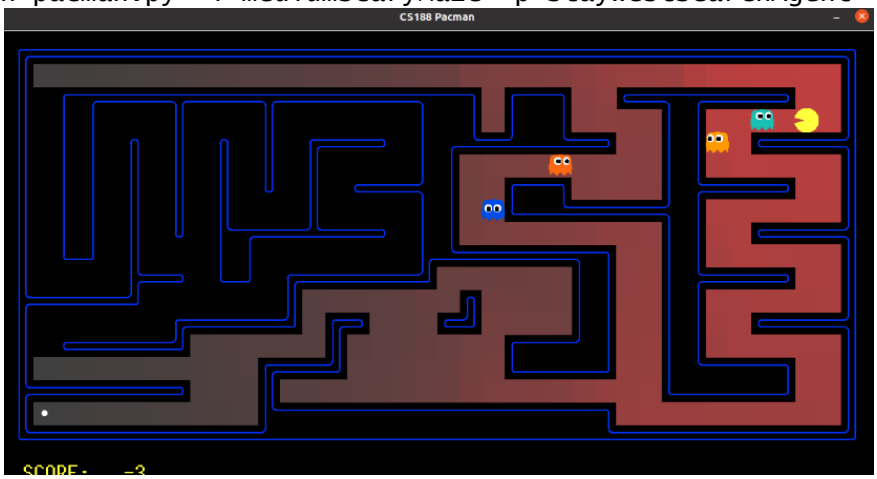




In this maze Pacman followed the path where the food was, also we can see that the generated states were not the same as in the first medium maze, we can conclude that

the positions with food have a smaller cost than the others. We could see that the accumulated cost at the end is 1.

```
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```



In this maze the execution was short, Pacman was eaten by the ghost just after it started, also after few runs, we could see that Pacman was following the same path, so the ghosts were not modifying the path.

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

UCS (uniform cost search) it is only optimal if the accumulated cost of the successor is bigger or equal than the path of the current node. In pacman that condition is fulfilled, so we can say that **yes**, **it is optimal**. As we do not allow cycles by eliminating repeated states, pacman **will reach a solution always**, unless we are in a problem with more agents that can kill the pacman before reaching the goal.

- Using the mediumMaze there were **269 expanded nodes** with a **final cost 68.**
- Using the mediumDottedMaze there were **620 expanded nodes** with a **final cost 1**.
- Using the mediumScaryMaze there were **230 expanded nodes** with a **final cost 1**.

The behaviour of UCS is similar to the behaviour of BFS, the problem is that in the mediumMaze the cost for each state is 1, so we cannot take advantage of UCS potential. If the shortest path had a smaller cost, we could see that UCS wouldn't expand many nodes.

Section 4

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

In this section we needed to create a function that added the accumulated cost and the value of the heuristic in that state, we created a function that did it and we sent it to the priority queue with function class.

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

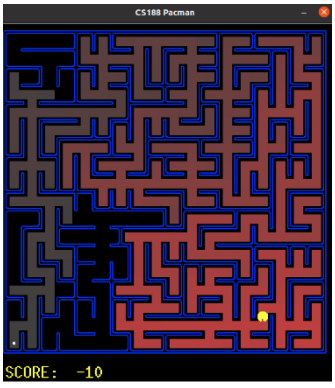
As in *section 3, 2* and *1*, we used the generic code, the difference is that in this case the priority function needed to be different, the priority function had the addition we mentioned in 4.1. We used the methods **push**, **pop** and **isEmpty**.

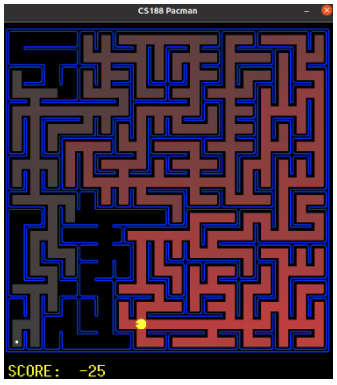
4.3. Includes code written by students (0.25 pts)

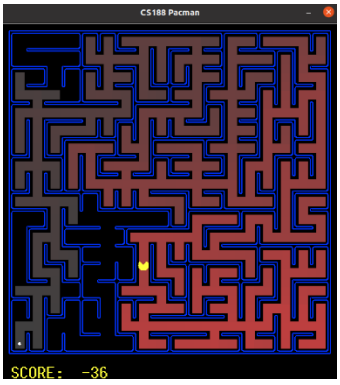
```
def aStarSearch(problem, heuristic=nullHeuristic):
    # priorityFunction = lambda state: state[2] + heuristic(state[0],
problem)
    def priorityFunction(state):
        return state[2] + heuristic(state[0], problem)
    openlist = util.PriorityQueueWithFunction(priorityFunction)
    return solveSimpleSearch(problem, openlist)
```

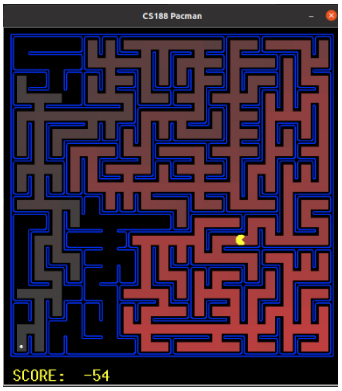
4.4. Screenshots of executions and test carried out analyzing the results (1pt)

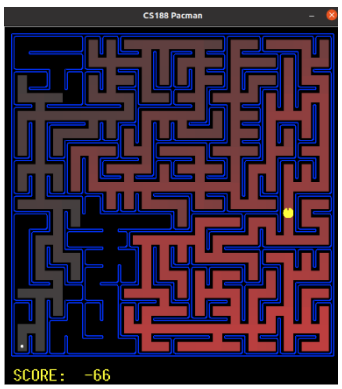
```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
```

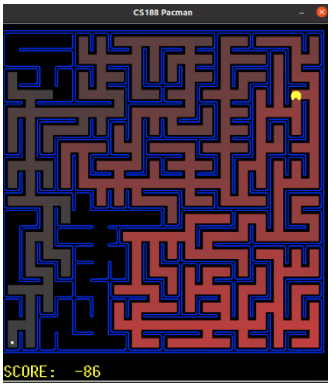


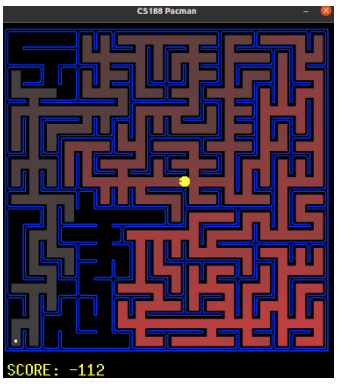


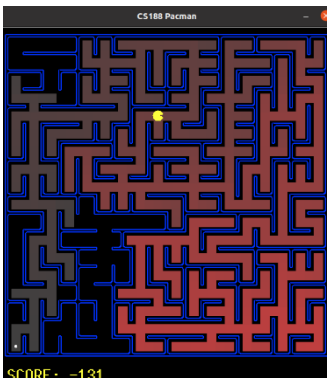


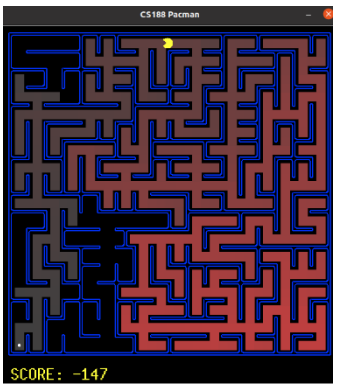


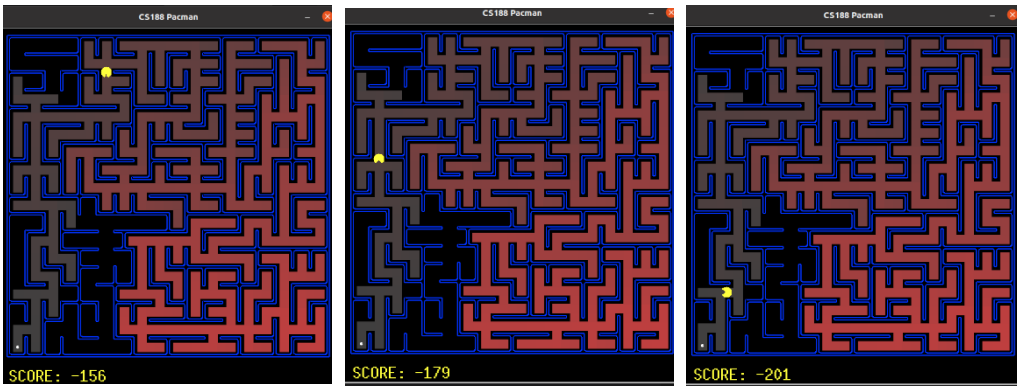












This is by far the best algorithm. We can see that in this maze, the explored states are less than BFS, this is also because we are using the Manhattan distance heuristic, this heuristic allows us to give a value to each state depending on how far the state is. The issue with this heuristic is that pacman goes back and forward a lot of times, it doesn't go to the goal state directly, so the heuristic is not consistent, because at some moments the heuristic of the next state would be bigger than the heuristic of the actual one.

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

If the heuristic is admissible then A* it is optimal. The null heuristic is admissible but trivial because it returns 0, so we could say that in that case A* is optimal, but the Manhattan distance heuristic it is not consistent.

- In the bigMaze **549 nodes** were expanded with a **final cost of 210**.

4.6. Answer to question 4 (1pt) *What happens on openMaze for the various search strategies?*

A*, UCS and BFS find the shortest solution, A* expands 535 nodes having a final cost of 54, UCS and BFS expand 682 nodes having a final cost of 54, and DFS is the worst one, because it has the longest solution with a cost of 212 having 576 nodes expanded. DFS is always exploring the first successor so this makes that the path found it is not optimal. BFS and UCS are similar but if the maze has a smaller cost in the shortest path, UCS will have a better performance than BFS. If we have a maze with smaller cost in the shortest path and we use a good heuristic, A* has the best performance.

Section 5

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

In this section we had several issues and problems in order to solve it:

1. Pacman was not going back after eating one food. When we were testing our first implementation, we realised that the path that the algorithm was finding was ending in the first food it found. This was because our "state" (when we refer to state, we mean the position of the pacman) included the position of the pacman, so once the closed-list had this position, it cannot be expanded again.
2. Pacman did not find the most efficient route. In order to solve the previous problem, we **deleted** the data in the closed list after one food was found. We

realised that this was not a good practice because the algorithm was modified and because the route was not efficient.

3. We were expanding too many nodes. Before we continued, we were thinking about a way to solve the first 2 problems, we decided that each state would be unique by assigning it a number, later in the code we were getting the position of each index by functions. The problem is that each position was visited plenty of times.

So, how did we fix this? We realised that a position may have been visited more than once, but each time the position is visited something must have been changed. Then we also realised that we were focusing on modifying the node (when we refer to node, we refer to the complete information about a state which includes, position, last action, and cost) and we did not need to modify this! We needed to modify the state so every time a node is revisited it is because a food has been found before. So, we modified the state, so it contained: the position and a list that included the corners that have not been visited.

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

In the getSuccessors function we used the method **directionToVector** from the **Actions** class, this method allowed us to get the "movement" of an action, for example if we go to the south, the movement would be to decrease the 'y' value by one.

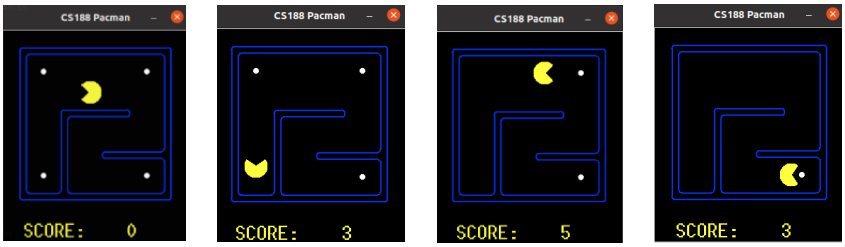
5.3. Includes code written by students (0.25 pts)

```
def __init__(self, startingGameState):
...
# We define a starting state which includes the starting positions
self.starting_state = (self.startingPosition, starting_position_corners)

def getStartState(self):
        return self.starting_state

def isGoalState(self, state):
        # Checking the lenght of the corners in the state
        # If list containing the corners is empty, return true
        return len(state[1]) == 0

def getSuccessors(self, state):
        successors = []

        # Getting the position in the state
        (x, y) = state[0]

        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                # Building the structure of the child state
                child_state = ((nextx, nexty), state[1].copy())

                # If the postion is in a corner, we remove that
                # position from the not visited corners
                if child_state[0] in child_state[1]:
```

```
                child_state[1].remove(child_state[0])

            # Building the structure of the child node
            child_node = (child_state, action, 1)

            successors.append(child_node)

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

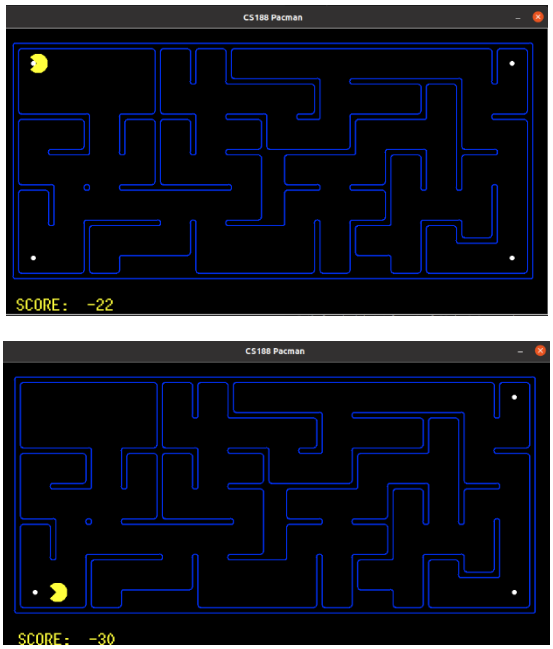
5.4. Screenshots of executions and test carried out analyzing the results (1pt)

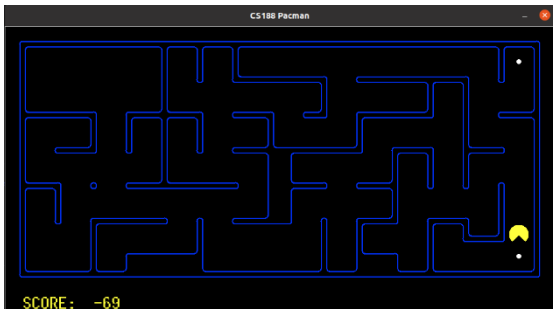
```
python pacman.py -l tinyCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
```

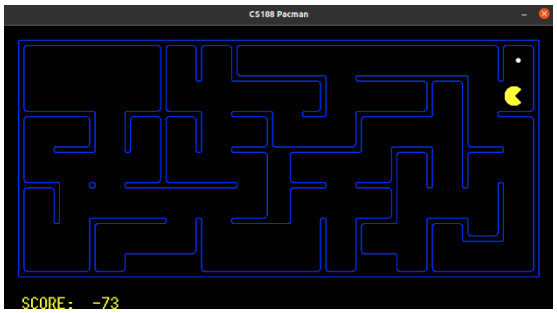


We can see that pacman goes through the shortest path, this path takes first the down-left food then the up-left, then the up-right and finally the down-right.

```
python pacman.py -l mediumCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
```







We can see how Pacman eats the food in the following order: up-left, down-left, down-right and up-right. This order seems to be the shortest one.

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

Yes, it is optimal and **yes**, it reaches the solution.

- In the tinyCorners **252 nodes** were expanded with a **final cost of 28**.
- In the mediumCorners **1966 nodes** were expanded with a **final cost of 106.**
- In the bigCorners **7949 nodes** were expanded with a **final cost of 162**.

Section 6

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

First of all, we would like to say that our solution of this part is wrong. In one of the tests of the autograder the heuristic is inconsistent, and we are going to explain why. Making an admissible heuristic is not very difficult, we just need that the value returned by the heuristic is not bigger than the total cost, the problem is making it consistent in the Pacman game.

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

We used the already defined variable **walls**, this variable contains the walls in the entire problem but in order to use it, we needed to get the coordinates of the walls and this was done by using the method **asList**.

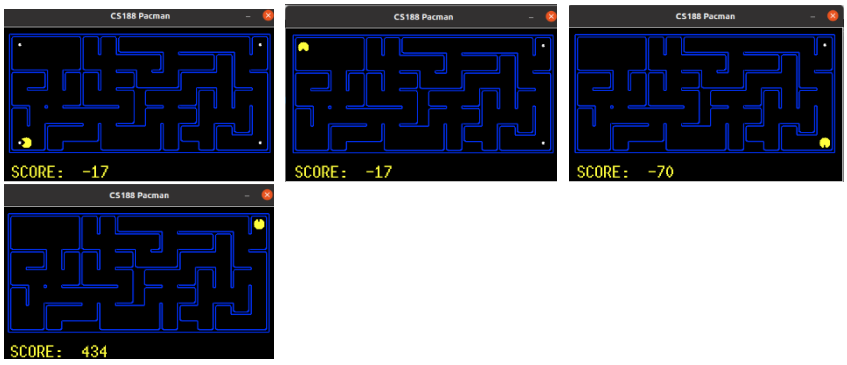
We also used the state we defined in the previous section; it was useful because we were able to see which corners have been visited.

6.3. Includes code written by students (0.25 pts)

```
def cornersHeuristic(state, problem):
    corners = problem.corners  # These are the corner coordinates
    # These are the walls of the maze, as a Grid (game.py)
    walls = problem.walls

    if len(state[1]) == 0:
        return 0
    value = (walls_between(state, walls.asList()))**len(state[1])
    return value
```

```
def closest_corner(state):
    position = state[0]
    distance = 0
    minimum_distance = 99999
    closest_corner = None
    for corner in state[1]:
        distance = abs(position[0] - corner[0]) + abs(position[1] - corner[1])
        if minimum_distance > distance:
            minimum_distance = distance
            closest_corner = corner
    return closest_corner


def closest_corner_distance(state):
    position = state[0]
    distance = 0
    minimum_distance = 99999
    for corner in state[1]:
        distance = abs(position[0] - corner[0]) + abs(position[1] - corner[1])
        if minimum_distance > distance:
            minimum_distance = distance
    return minimum_distance


def walls_between(state, walls):
    x, y = state[0]
    corner = closest_corner(state)
    if corner is None:
        return 0
    else:
        cor_x, cor_y = corner
    num_walls = 1

    # Moving in X and counting the walls between
    while x != cor_x:
        if x < cor_x:
            x += 1
        else:
            x -= 1
        if (x,y) in walls:
            num_walls += 1

    # Moving in Y and counting the walls between
    while y != cor_y:
        if y < cor_y:
            y += 1
        else:
            y -= 1
        if (x,y) in walls:
            num_walls += 1
    return num_walls
```

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



We can see that the pacman follows the correct order taking the food.

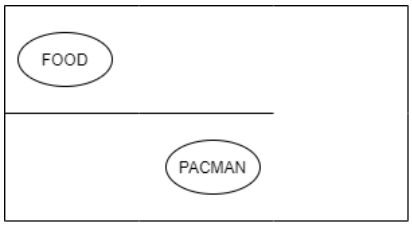
6.5. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

Yes, it is optimal, and **yes** it reaches the solution.

- **263 nodes** were expanded with a **final cost of 106**. The heuristic is not consistent.

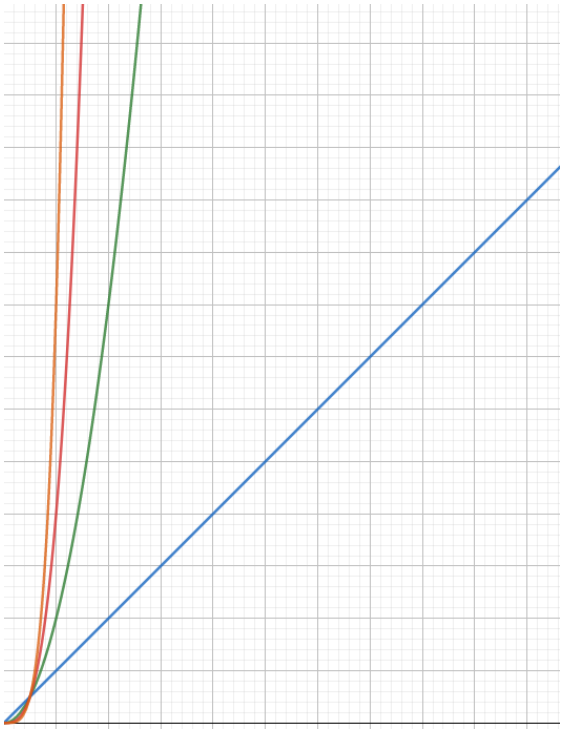
6.6. **Answer to question 5: heuristics (1pt)**

Our idea was that the more corners we have left, the bigger the value of the heuristic. The logic behind our heuristic is by looking at the walls between Pacman and the closest food.



As we can see here there is a clear path to the food, but our heuristic would return 1 because there is one wall between the food and Pacman. But why is there a wall when there is a path? This is due to the way we calculate the number of walls; we first move to the X coordinate of the food and then we move to the Y coordinate of the food. We know is not the most intelligent way to calculate the number of walls between the elements.

So, once we calculated that, we wanted to do "stairs" in the value which the heuristic would return.



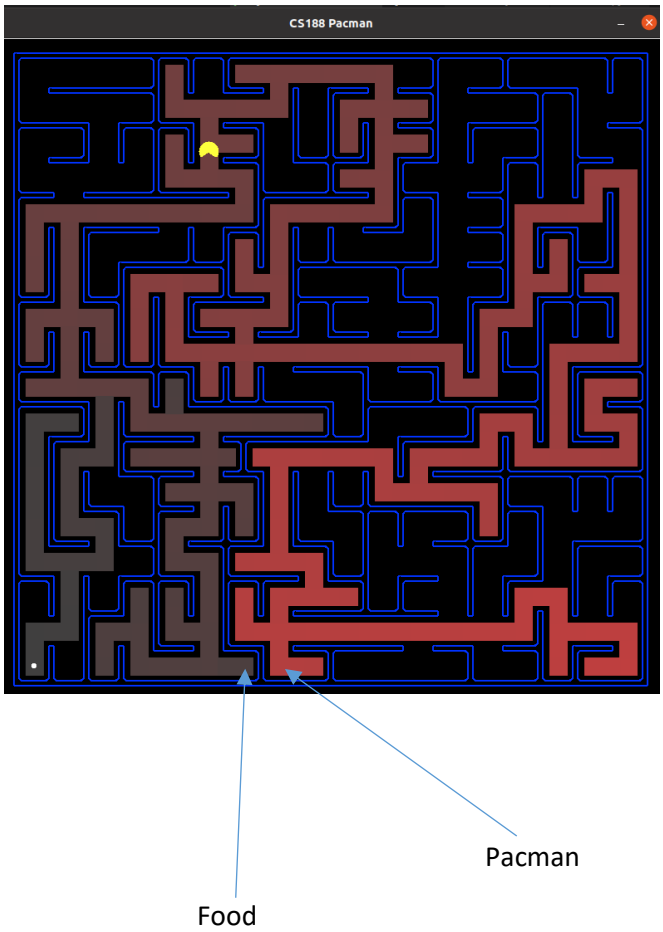
The orange line corresponds to x^4.

The red line corresponds to x^3.

The green line corresponds to x^2.

The blue line corresponds to x^1.

We can see that in almost all the X values, the bigger the exponent the bigger the value, but here we have the first problem, "in almost all X values". If we check the smaller values of X, we can see that this condition is not fulfilled, so in order to fix this we could give a linear value to the heuristic when X is in that range, right? No, because X in our case is not something static, every time the Pacman moves the X value changes. This is the second problem, the X value depends in the number of walls between the packman and the closest food (by the way, the exponent in our problem is the number foods left). To see why taking into account the number of walls is a bad idea, let's see one example:



Let's say that the Pacman and the food are in the marked positions, how many walls are between each other? **At this moment 1**, but if we continue the execution, we will have a value bigger than 1, this is making the heuristic not being consistent because for each step in the algorithm, the value does not decrease.

Another thing is that the closest food may not be the one which Pacman is going to take.

The heuristic gives a good performance, but it is not consistent.

Section 7

The assignment gave us 3 main problems:

1. Making generic function.
2. Changing the state for the corner problems.
3. Creating a consistent heuristic.

The first 2 problems were fixed but we got stuck in the 3rd one, the idea we had for a heuristic was not good because it is very "volatile", the value changes a lot and it's not decrementing.

The assignment helped us understanding what is a search problem and the ways to solve these ones.

Memory grade (40% of practice)

Total points (X / 31.5)