Report

Assignment 1: Search

Chiara Mocetti, Ebad Malik – P10

Search algorithms (search.py)

a. **General design (0.5 points):**

   i. **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.

   ii. **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)

b. **Design:**

   i. **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.

ii. **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.

iii. **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)
```

c. **Efficiency of the search algorithm:**

   i. **How many nodes are expanded?**

   ii. **Does it reach the optimal solution?**

   iii. **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.

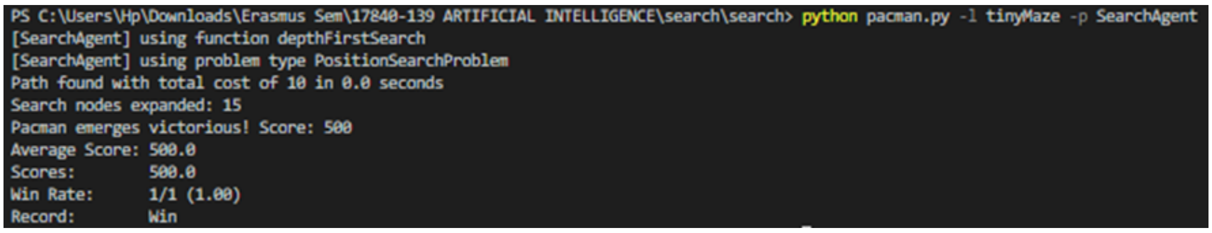
d. **Answer question 1.1 of the assignment statement.**
   The exploration order is as expected since we are using depth first search with a stack and we see in our implementation that pacman does not go to all the explored states, instead it expands the deepest node first by going to the darker red route.

e. Answer question 1.2 of the assignment statement.

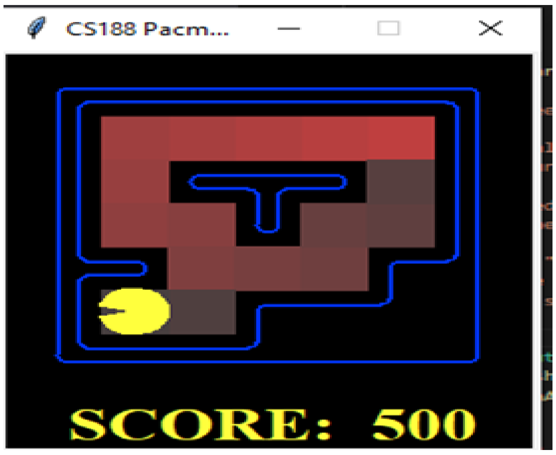
It is not a least costly solution since DFS is not optimal or complete. It requires backtracking when it reaches a dead end while it is expanding the deepest node.

f. Tests: Include the tests performed to illustrate the search results.

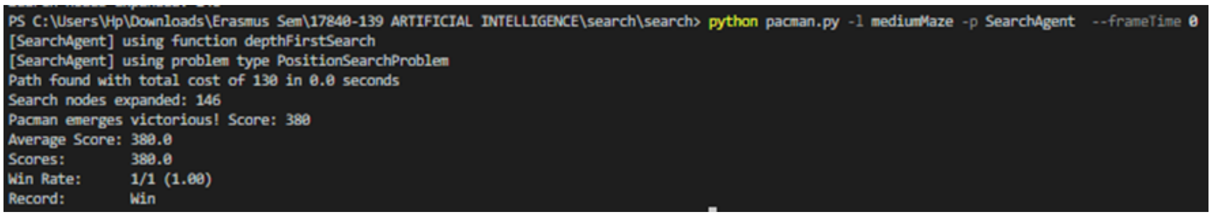
Tiny Maze:



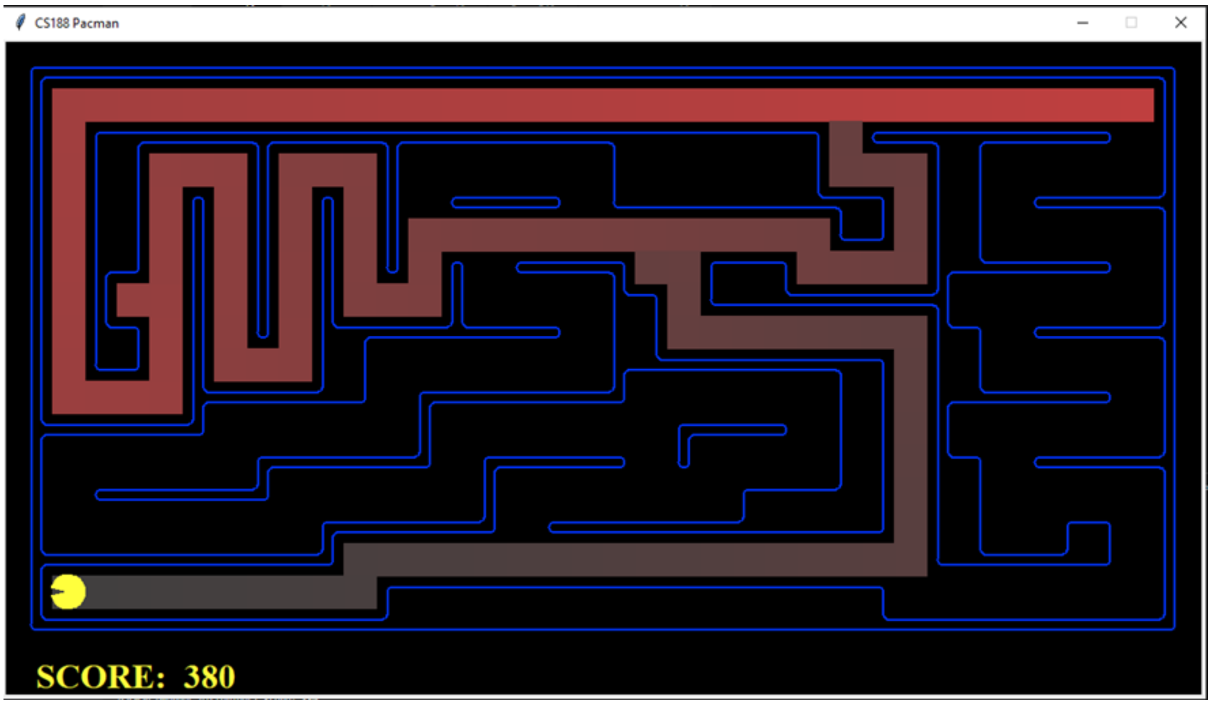
```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l tinyMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
```



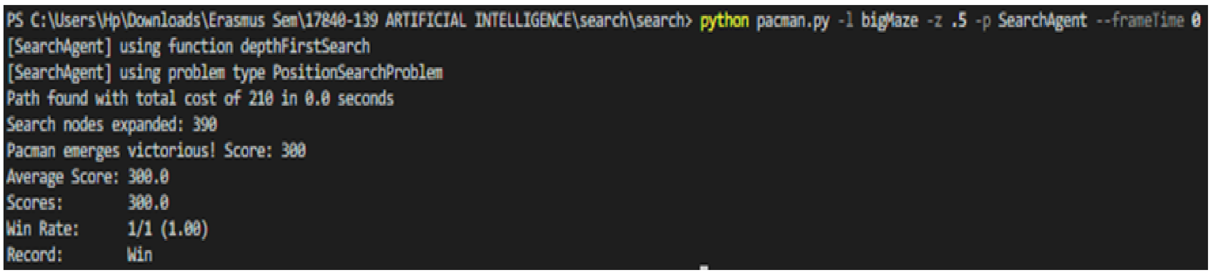
Medium Maze:



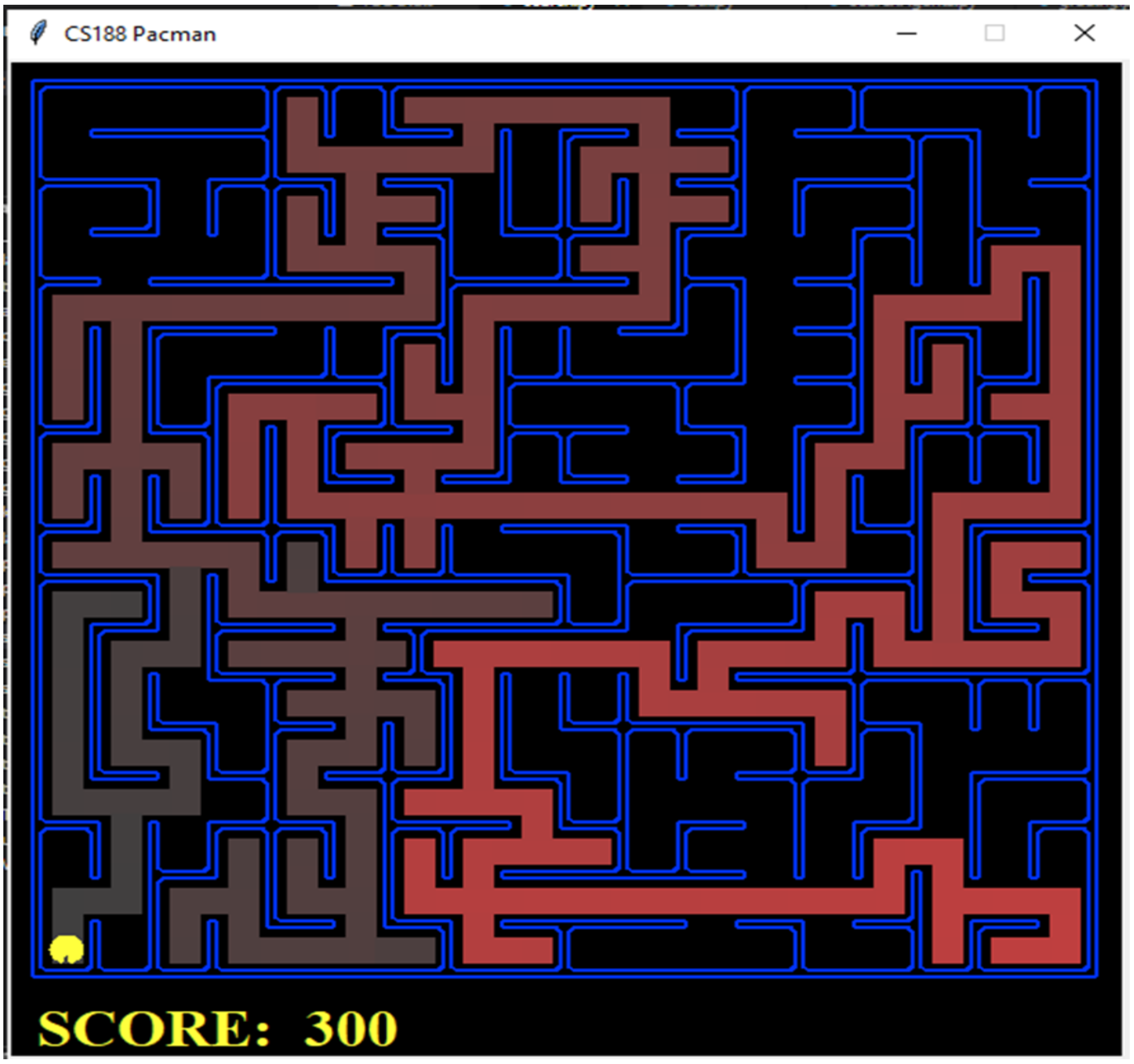
```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l mediumMaze -p SearchAgent  --frameTime 0
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Big Maze:



```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l bigMaze -z .5 -p SearchAgent --frameTime 0
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Section 2 (0.75 points)

a. **Design:**

i. **Explain the approach taken to design the search algorithm.**
Regarding the Breadth First Search we run the Search Algorithm using an open-list queue.

ii. **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.

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

```python
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)
```

b. Efficiency of the search algorithm:

i. **How many nodes are expanded?**

ii. **Does it reach the optimal solution?**

iii. **Is it optimal?**

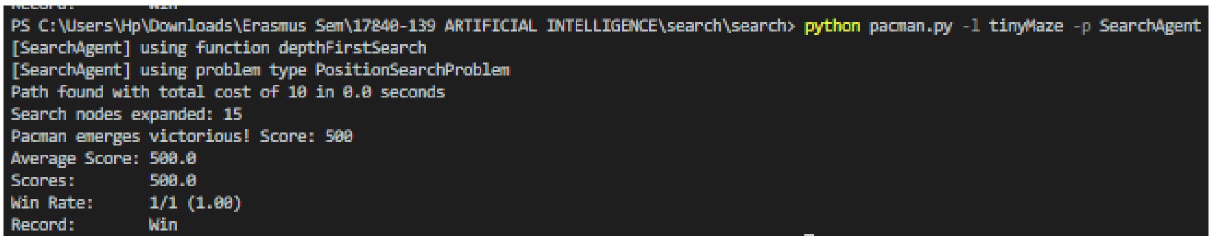
We show the run of Medium Maze with BFS:
269 nodes are expanded
Path found with total cost of 68
It reaches the goal with optimal cost. Therefore, it is optimal.

c. Answer question 2.1 of the assignment statement.

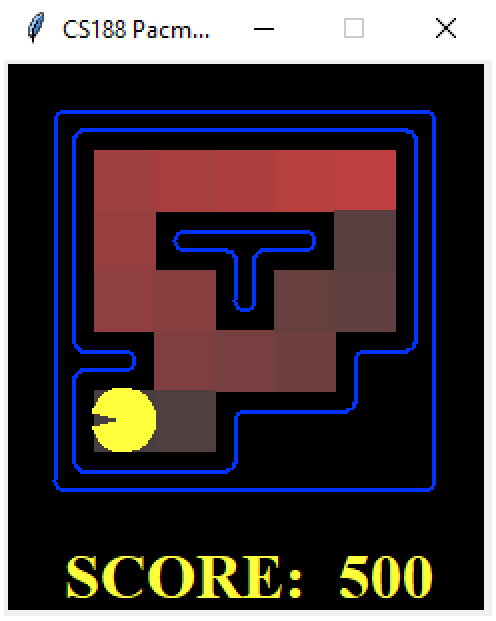
Yes, BFS finds the optimal solution.

d. Tests: Include the tests performed to illustrate the search results.

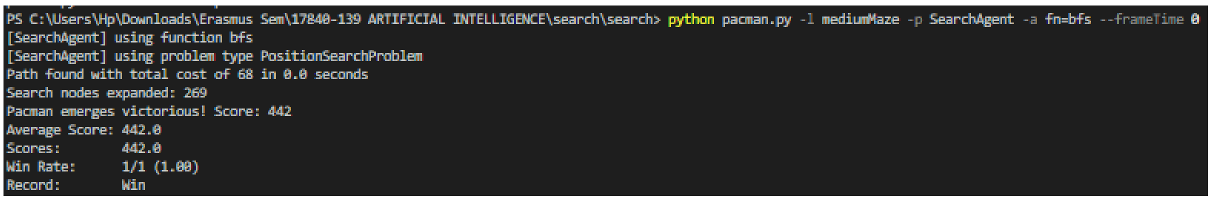
Tiny Maze:



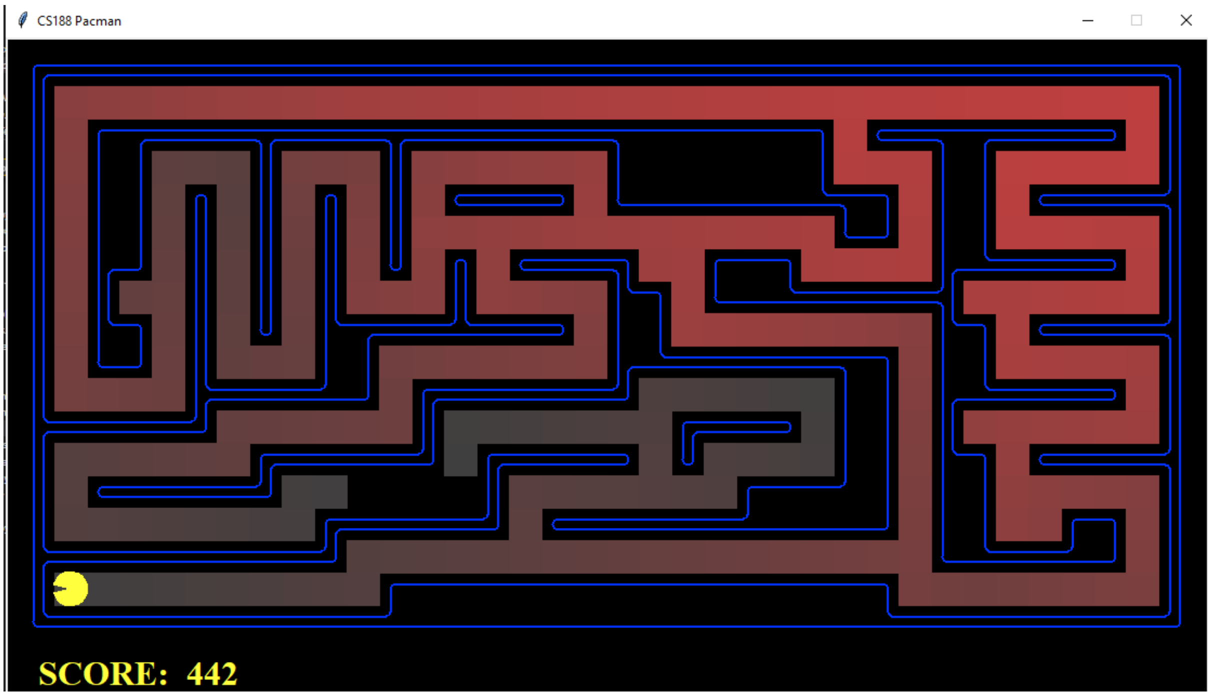
```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l tinyMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
```



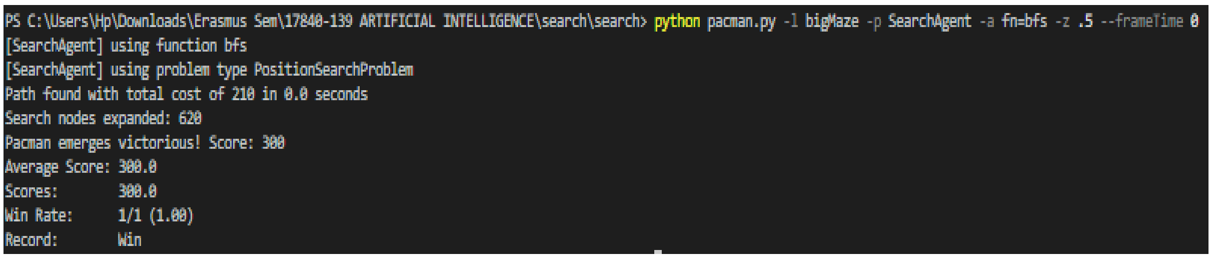
Medium Maze:



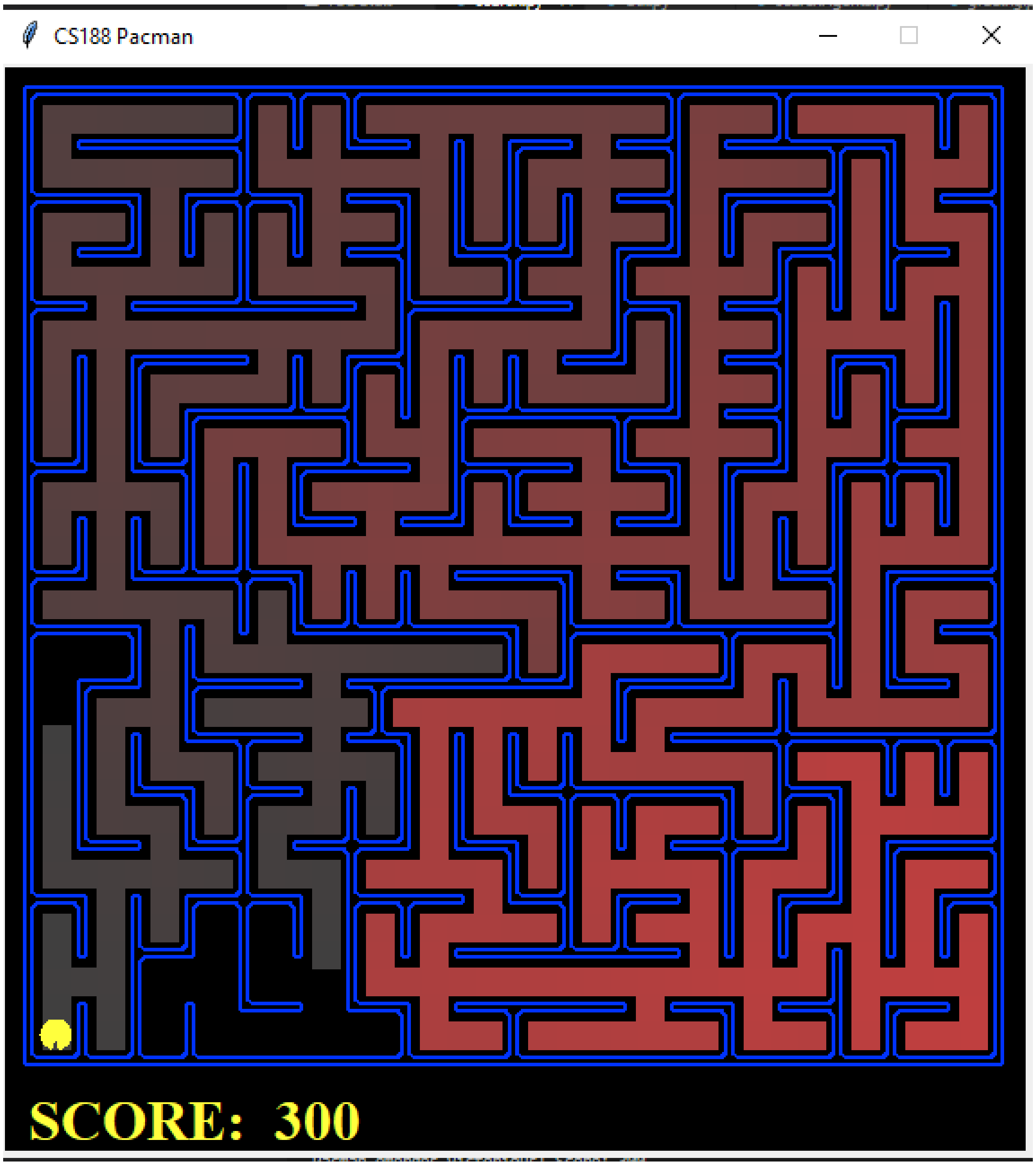
```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs --frameTime 0
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Big Maze:



```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 --frameTime 0
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Section 3 (0.75 puntos)

a. **Design:**

i. **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]))
```

ii. **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.

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

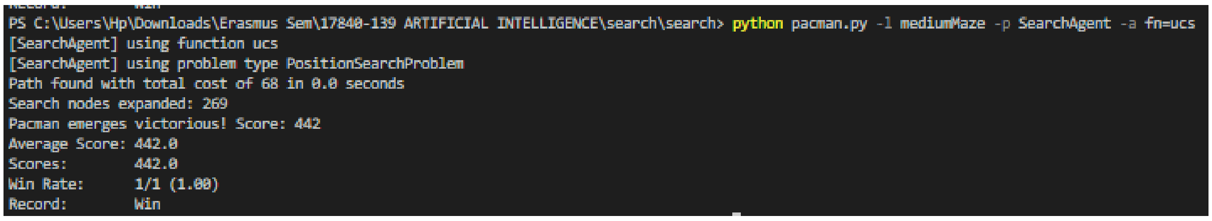
```
def uniformCostSearch(problem):
    """
    Search the node of least total cost first.
    (Using Priority Queue with Path Function)
    """

    open_list = \

util.PriorityQueueWithPathFunction(problem.getCostOfActions)
    open_list.push((problem.getStartState(), []))
    return SearchAlgorithm(problem, open_list)
```

e. **Efficiency of the search algorithm:**
   i. **How many nodes are expanded?**
   ii. **Does it reach the optimal solution?**
   iii. **Is it optimal?**

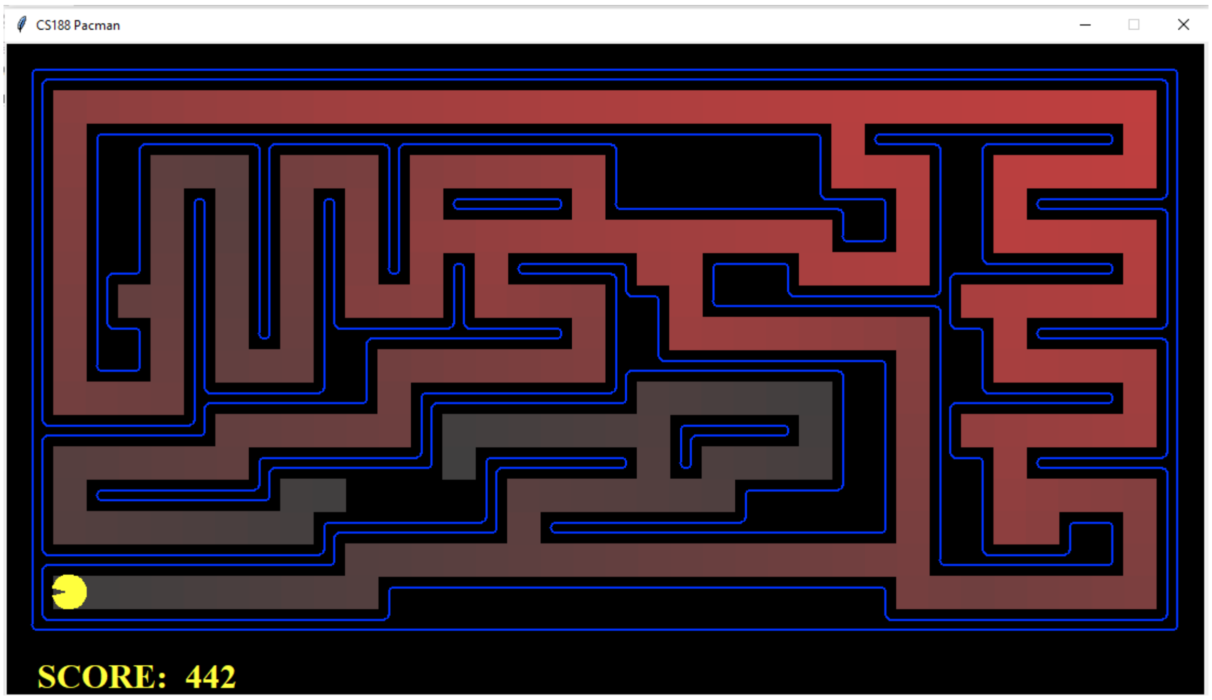
We show the run of Medium Scary Maze with UCS:
108 nodes are expanded.
It reaches the goal with optimal cost. Therefore, it is optimal. UCS is optimal.

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

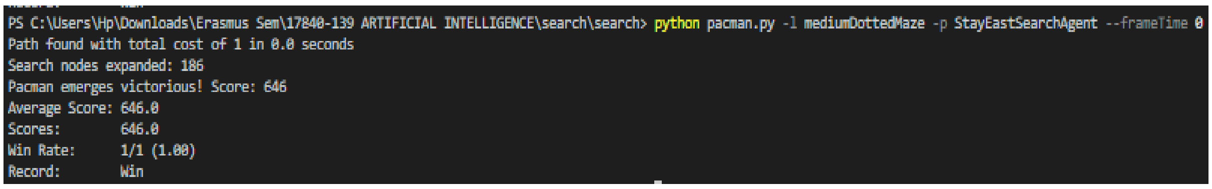
Medium Maze:



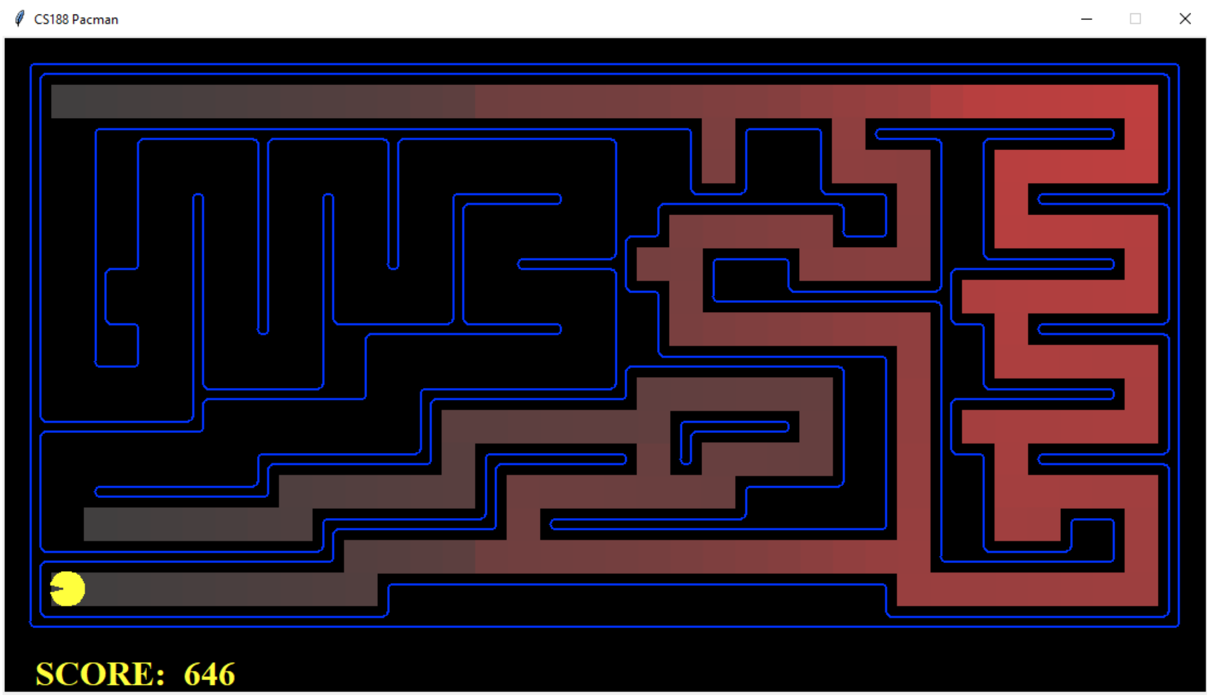
```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```



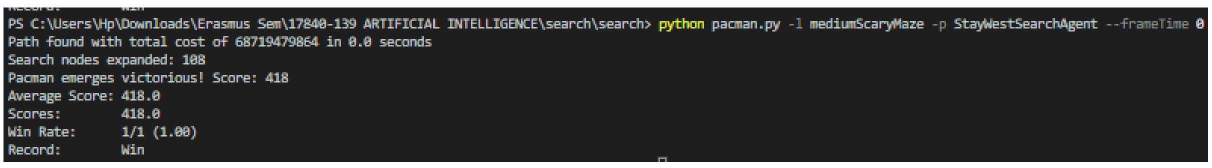
Medium Dotted Maze:



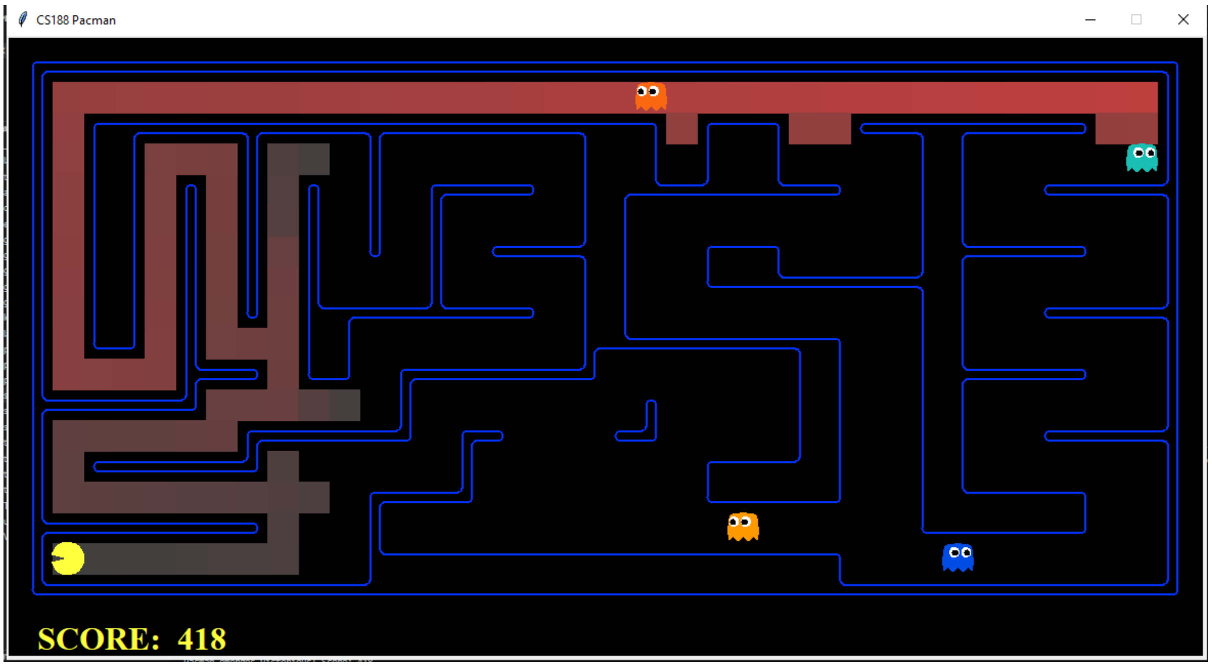
```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l mediumDottedMaze -p StayEastSearchAgent --frameTime 0
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Medium Scary Maze:



```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l mediumScaryMaze -p StayWestSearchAgent --frameTime 0
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:        418.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Section 4 (2 points)

a. **Design:**

i. **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.
```

ii. **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.

iii. **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.getCo
           stOfActions,
                                                 problem,
           heuristic)
    open_list.push((problem.getStartState(), []))
    return SearchAlgorithm(problem, open_list)
```

b. **Efficiency of the search algorithm:**

i. **How many nodes are expanded?**

ii. **Does it reach the optimal solution?**

iii. **Is it optimal?**

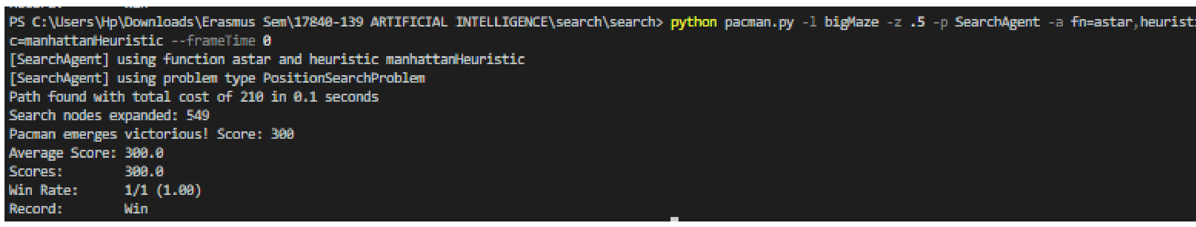
We show the run of Medium Scary Maze with UCS:
Path found with total cost of 210 in 0.1 seconds.
Search nodes expanded: 549.
It reaches the optimal solution. A* is Optimal!

c. Answer question 4.1 of the assignment statement.

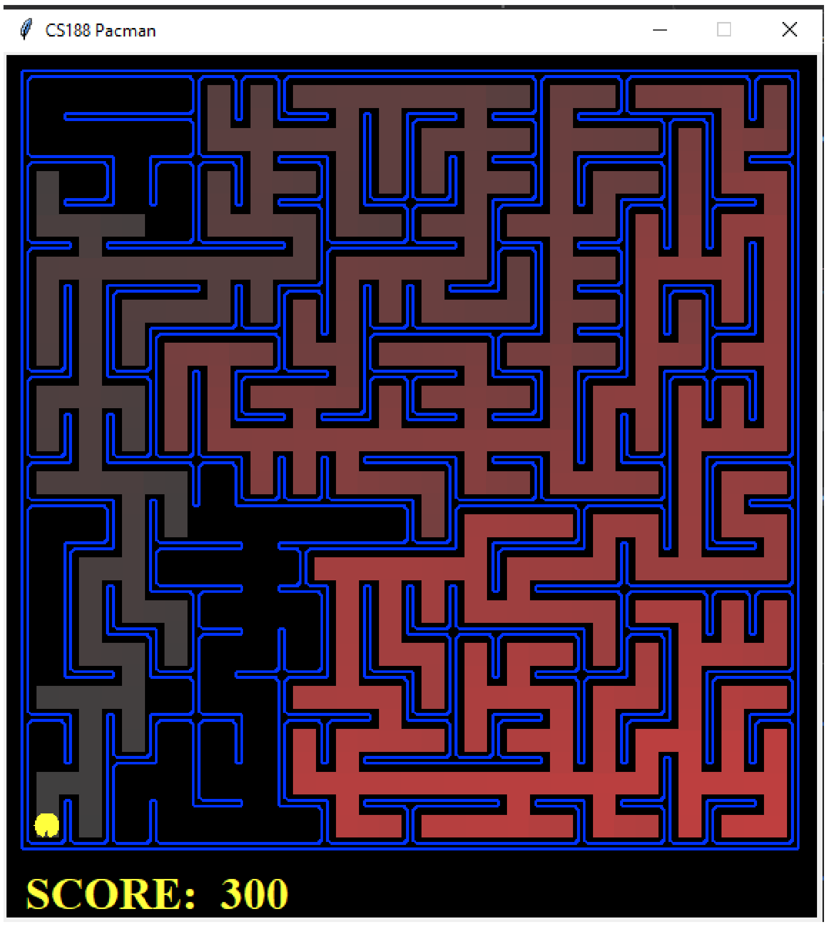
Different search strategies result in different exploration methods of the maze, As each of them follow a different strategy. We implemented DFS,BFS,UCS and A*. DFS is done with a stack, BFS with a queue, UCS and A* with a priority Queue. They all have a different strategy that can result in different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

d. Tests: Include the tests performed to illustrate the search results.

Big Maze:



```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heurist
c=manhattanHeuristic --frameTime 0
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Search-based agents (searchAgents.py)

Section 5 (2 points)

a. **System state:**

   i. **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'])
```

   ii. **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.

   iii. **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)])

b. **Implementation:**

   i. **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)

   ii. **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
```

iii. **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
```

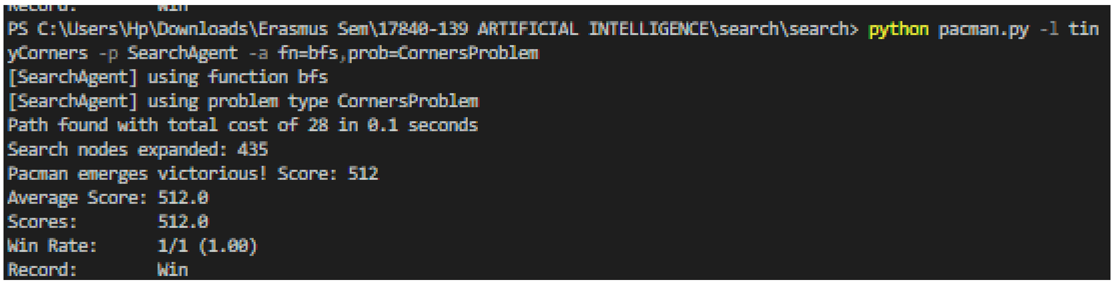
iv. **Describe the implementation of 'isGoalState' for this problem.**

```
    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
```

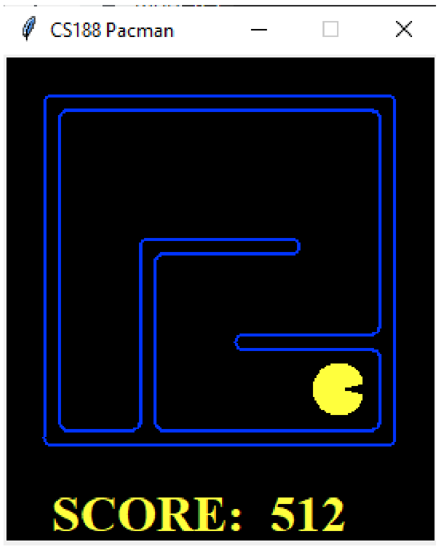
```
        if state.position in self.corners:
            if not state.position in state.visited_corners:
                state.visited_corners.append(state.position)

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

            return len(state.visited_corners) ==
len(self.corners)
        return False
```

c. Tests: Include the tests performed to illustrate the implementation.

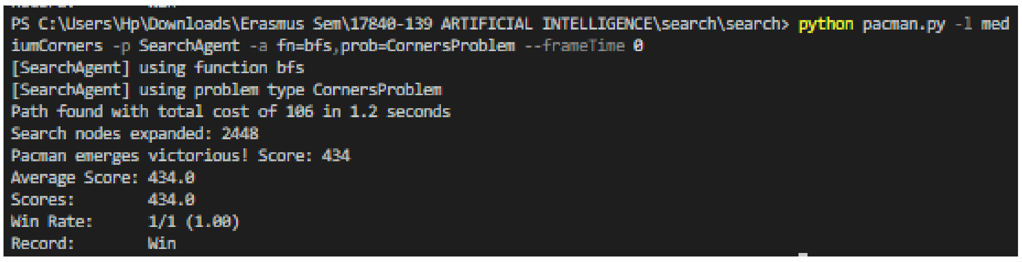
Tiny Corners



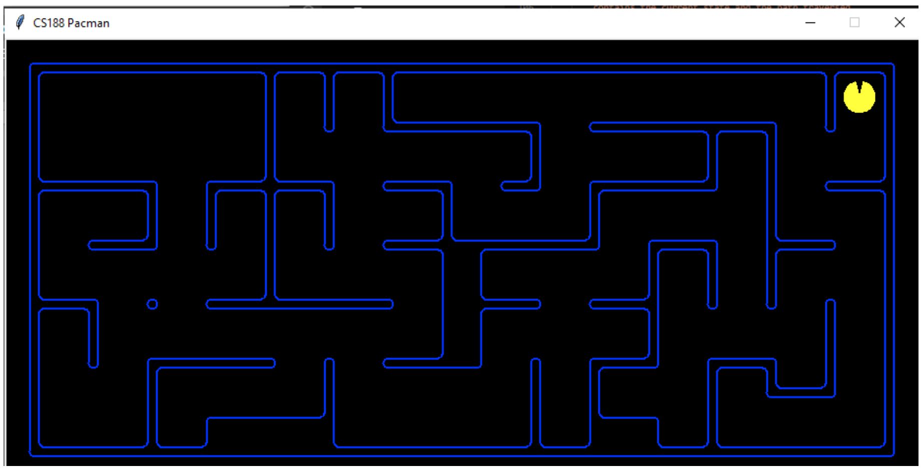
```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l tin
yCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.1 seconds
Search nodes expanded: 435
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Medium Corners



```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l med
iumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem --frameTime 0
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 1.2 seconds
Search nodes expanded: 2448
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Section 6 (2 points)

a. **Heuristic:**

   i. **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.

   ii. **Explain the heuristic logic.**

   Shown below in the code implemented

b. **Implementation:**

   i. **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.

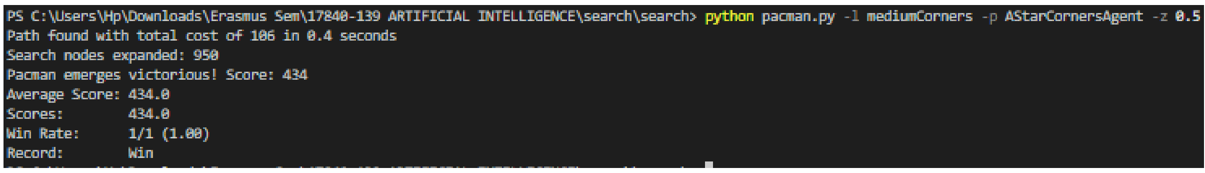
   ii. **Include and explain the code you implemented.**

```
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
```

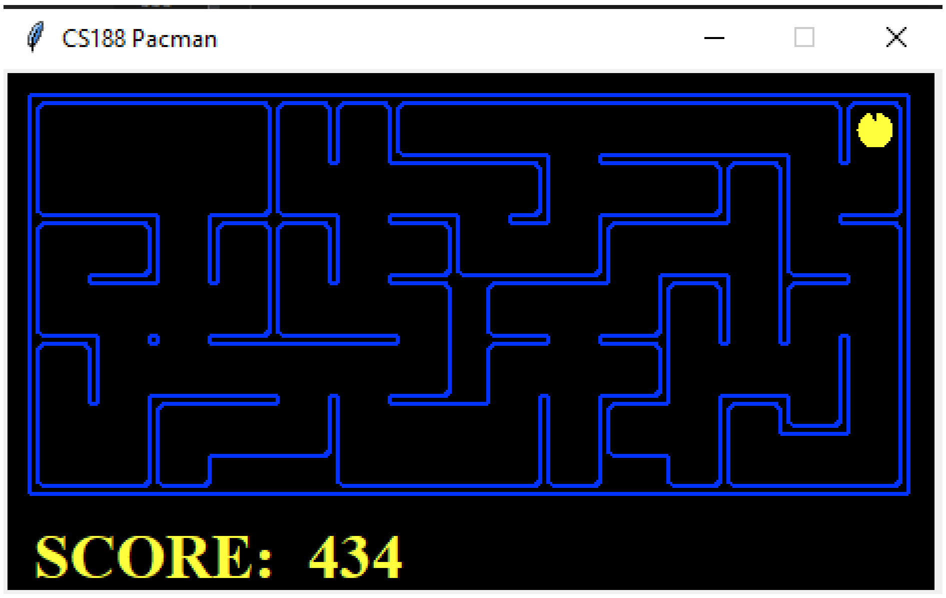
```
    walls = problem.walls  # These are the walls of the
maze, as a Grid (game.py)

    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
```

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

Medium Corners:



```
PS C:\Users\Hp\Downloads\Erasmus Sem\17840-139 ARTIFICIAL INTELLIGENCE\search\search> python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.4 seconds
Search nodes expanded: 950
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```


CS188 Pacman
SCORE: 434