Artificial Intelligence

Practice 1: Pacman

Jorge de Miguel Pires
Laura de Paz Carbajo
Group 2351
Team 07

SECTION 1

1.1. Personal comment on approach

In order to implement DFS algorithm, we made use of the `GraphSearch` function studied in theory lessons using a Stack (LIFO) as the opened list.
With this function we were able to reach the target node we couldn't retrieve the path Pacman had to follow from its initial state to the goal state. That's why we created the class Node. Its attributes are the 3-element tuple we had been using (position, cost, action) and the parent of the node. The latter attribute helps us get from a given node to the root node, following the tree generated with DFS algorithm.

1.2. Functions used

We implemented the function `depthFirstSearch()` by translating the pseudocode of `GraphSearch` into Python3.x code. Once the target node is found, this function calls the `getParent()` function of every node in the path that goes from the initial position to the position where the food is. This allows the program to get the list of nodes from in the path and extract, from them, the actions to be followed.
While implementing the class Node, we had to code some extra methods such as `getTuple()` and those methods needed to get a fully functional class: `__init__()`, `__eq__()`, `__str__()`.

1.3. Code

```
def depthFirstSearch(problem):
    """ Search the deepest nodes in the search tree first """
    root = problem.getStartState()
    rootNode = Node(root, "Stop", 0, None)
    opened = util.Stack()
    opened.push(rootNode)
    closed = []
    sol = []
    # Iterate
    while not opened.isEmpty():
        # Choose from opened list a node to expand
        node = opened.pop()
        nodePos, nodeAct, nodeCost = node.getTuple()
        if problem.isGoalState(nodePos):
            # Creates and returns lists of actions
            while node.getParent() is not None:
                sol.append(node.getTuple()[1])
                node = node.getParent()
            sol.reverse()
            return sol
        if node not in closed:
```

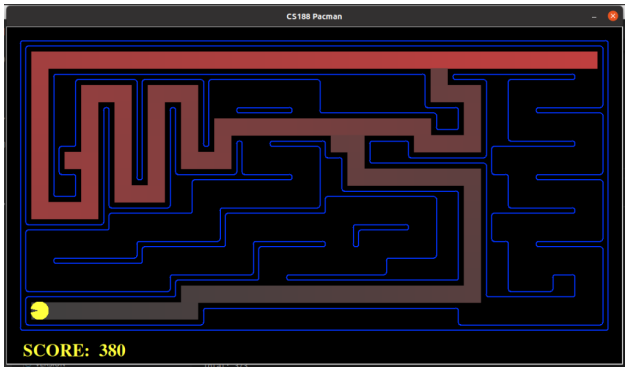
```
            closed.append(node)
            # Expand node
            for succ in problem.getSuccessors(nodePos):
                nodeSucc = Node(succ[0], succ[1], succ[2], node)
                if nodeSucc not in closed:
                    opened.push(nodeSucc)
    return False

class Node:
    def __init__(self, node, action, cost, parent):
        self.info = node
        self.action = action
        self.cost = cost
        self.parent = parent

    def __eq__(self, node):
        return self.info == node.info

    def __str__(self):
        return "Node: " + str(self.info) + ", " + self.action + ", " +
str(self.cost)

    def getTuple(self):
        return self.info, self.action, self.cost

    def getParent(self):
        return self.parent
```

1.4. Screenshots

We tried first executing the algorithm for tiniMaze and once it worked, we kept on with mediumMaze and bigMaze. These are out results:

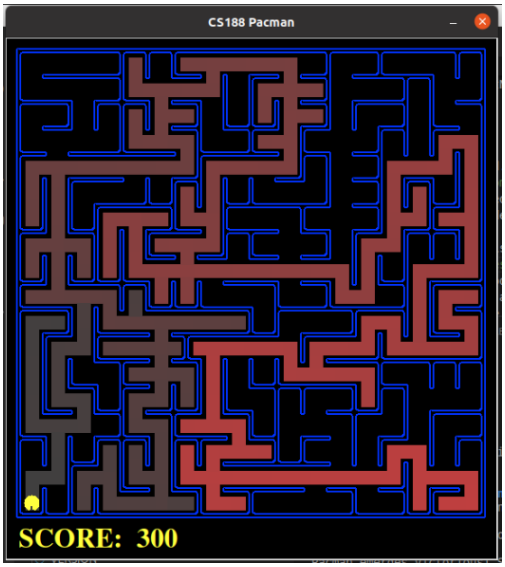

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
```

With the help of the picture we can see that every single node of the maze has been expanded before getting to the solution, making a total of 15 nodes. Besides, with just a quick glance, we can realise that the path found is not the one with the lowest cost.




```
[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
```

In the case of the medium maze not even half of the nodes have been expanded and it reaches the goal, yet the chosen path is not the shortest one. We now have proof that Depth First Search algorithm is not optimal.
Something similar as explained above happens with the big maze:




```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 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
```

If we focus on the fading red path, which represents the order of expansions of the nodes, we note that when Pacman reaches a square with more than one possible successor, it chooses one and then forgets about the rest of them. It only comes back to expand one of them if it gets to a dead end.

The autograder gave us the following results:

```
jorge@Jorge-Lenovo:~/Documentos/Uni/3_Tercero/AI/Practice1/search$ python3 autograder.py -q q1
Starting on 2-23 at 9:53:34

Question q1
===========
*** PASS: test_cases/q1/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'D', 'C']
*** PASS: test_cases/q1/graph_bfs_vs_dfs.test
***     solution:               ['2:A->D', '0:D->G']
***     expanded_states:        ['A', 'D']
*** PASS: test_cases/q1/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q1/graph_manypaths.test
***     solution:               ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases/q1/pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 130
***     nodes expanded:         146

### Question q1: 3/3 ###


Finished at 9:53:34

Provisional grades
==================
Question q1: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

1.5. Conclusions on Pacman´s behaviour

In general, DFS is a non-complete algorithm because it can explore plaths of infinite length and never find a solution but, in this case, the mazes are finite in size and the graph search function has been implemented without repeated states. Hence the algorithm is complete, in other words, Pacman will always find a path to the food.
It is not optimal because it does not expend the necessary amount of nodes to find the path with the lowest cost. DFS is efficient regarding memory usage because the space it uses is O(b*d), where b is the branching factor and d the maximum length. The nodes that it expands are fewer than those expanded in other algorithms, as detailed below.
However, its time complexity is exponential: $O(b^d)$.

1.6. Questions

1.1) Is the exploration order what you would have expected?

Yes, because as studied in theory lessons, when a node is expanded, its successors are expanded always before its sibling nodes. That's why we expected the solution path to follow straight lines in the maze before changing direction by going back to a certain node and exploring its other neighbour nodes.

1.2) Does Pacman actually go to all the explored squares on his way to the goal?

It could happen if the first path it follows leads to the food but in general this will not happen.
Pacman does not go to all the explored squares on his way to the goal because it can occur that he gets to a dead end. It has expanded all the nodes till that point but it definitely is not a node in his way to the goal square.

2) Is this a least cost solution?

Since DFS is not optimal, it does not reach the solution with the least cost. It could generate very long paths that will take Pacman nowhere and added cost would not contribute towards an optimal solution.

SECTION 2

2.1 Personal comment on approach

The main difference between BFS and DFS is the structure we used to store the opened nodes. In this case we just needed to change the Stack used in DFS for a Queue, making the structure a FIFO.

2.2. Functions used

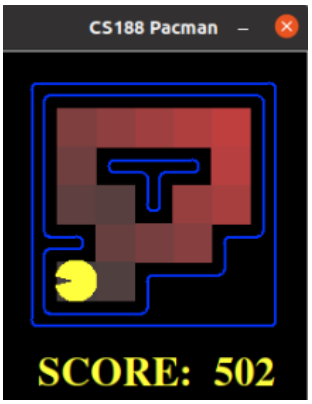
We implemented the function `breadthFirstSearch()` by copying the `depthFirstSearch()` function and modifying it as I said in the previous paragraph. We also used the `Node` class we created to keep a reference to the father of each state, just like in the previous section.

2.3. Code

```
def breadthFirstSearch(problem):
      """Search the shallowest nodes in the search tree first."""
      root = problem.getStartState()
      rootNode = Node(root, "Stop", 0, None)
      opened = util.Queue()
      opened.push(rootNode)
      closed = []
      sol = []
      # Iterate
      while not opened.isEmpty():
      # Choose from opened list a node to expand
      node = opened.pop()
      nodePos, nodeAct, nodeCost = node.getTuple()
      if problem.isGoalState(nodePos):
            # Creates and returns lists of actions
            while node.getParent() is not None:
                  sol.append(node.getTuple()[1])
                  node = node.getParent()
            sol.reverse()
            return sol
      if node not in closed:
            closed.append(node)
            # Expand node
            for succ in problem.getSuccessors(nodePos):
                  nodeSucc = Node(succ[0], succ[1], succ[2], node)
                  if nodeSucc not in closed:
                  opened.push(nodeSucc)
      return False
```

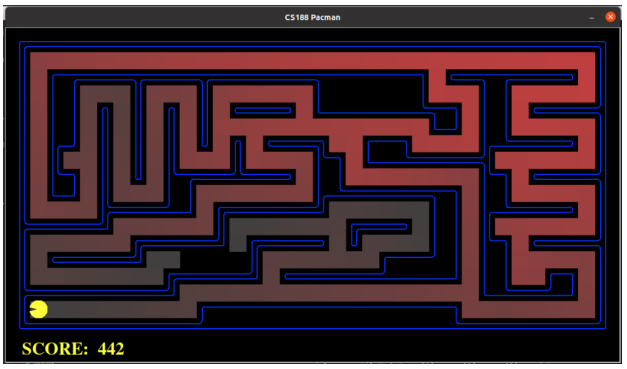
2.4. Screenshots

First of all, we tried executing the tinyMaze with BFS and we got this result:



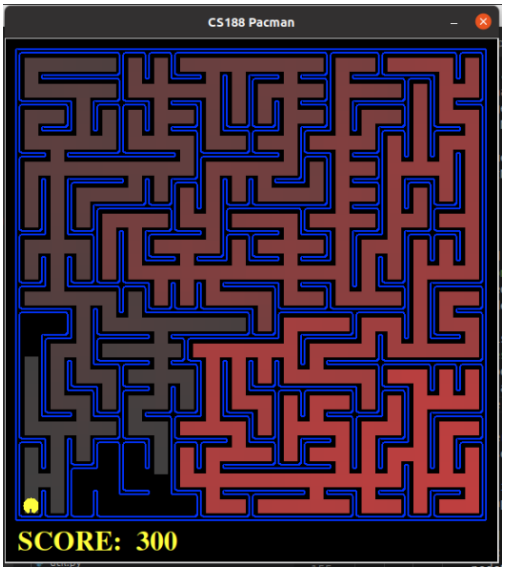

```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

If we compare the output of BFS with the one of DFS we see that BFS finds a better path than DFS although both expand the same amount of nodes. In fact, BFS finds the optimal path for this labyrinth.




```
[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
```

In the mediumMaze, BFS expands almost double the amount of nodes of DFS but the solution found has a way lower cost. Same thing happens for the bigMaze:




```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 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
```

If we focus on the mazes, we can detect a pattern by looking at the different shades of red of each square. We can see that the nodes at the same depth level are expanded first (brighter red), and once every node in that depth is expanded, then it starts all over with the nodes at the next level until it reaches the solution.

The autograder gave us the following results:

```
jorge@Jorge-Lenovo:~/Documentos/Uni/3_Tercero/AI/Practice1/search$ python3 autograder.py -q q2
Starting on 2-23 at 10:02:15

Question q2
===========
*** PASS: test_cases/q2/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q2/graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases/q2/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q2/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q2/pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###


Finished at 10:02:15

Provisional grades
==================
Question q2: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

2.5. Conclusions on Pacman´s behaviour

The BFS algorithm is complete, which means that it will always find the solution. Also, in this case, it's optimal even though we are using graph search, because the cost function increases monotonously (the cost of all the actions is constant equal to 1), so it always finds the lowest cost solution.
Focusing on the nodes expanded, the BFS algorithm is not very efficient, it has a spatial complexity of $O(b^d)$ and a time complexity of $O(b^{d+1})$ in the worst case. For that reason, using this technique for bigger problems is not recommended, because it would take a lot of time and memory to get to the lowest cost solution.

2.6. Questions

3) Does BFS find a least cost solution?
Yes, it finds the least cost solution because it is optimal.

SECTION 3

3.1. Personal comment on approach

Once again, we implemented a search algorithm following the `GraphSearch` code but with a different strategy in the implementation of the opened list. We used a priority queue, from which the node with the lowest path cost is selected and thereafter expanded.

3.2. Functions used

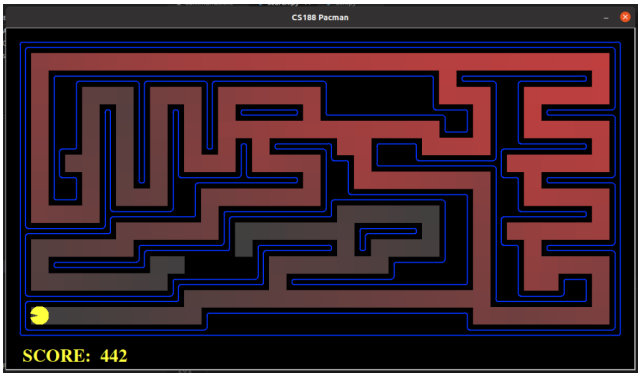
We first copied the previous `breadthFirstSearch()` function and modified it by substituting the standard queue for a priority queue to implement uniformCostSearch(). Plus, we had to change the cost associated with a node when an instance of it is created. Before every node had cost 1 but in this implementation the cost of a node is equal to the cost of the whole path from the root to that certain node. This value is what the priority queue will consider to select a node when popping.

3.3. Code

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    root = problem.getStartState()
    rootNode = Node(root, "Stop", 0, None)
    opened = util.PriorityQueue()
    opened.push(rootNode, rootNode.getTuple()[2])
    closed = []
    sol = []
    # Iterate
    while not opened.isEmpty():
        # Choose from opened list a node to expand
        node = opened.pop()
        nodePos, nodeAct, nodeCost = node.getTuple()
        if problem.isGoalState(nodePos):
            # Creates and returns lists of actions
            while node.getParent() is not None:
                sol.append(node.getTuple()[1])
                node = node.getParent()
            sol.reverse()
            return sol
        if node not in closed:
            closed.append(node)
            # Expand node
            for succ in problem.getSuccessors(nodePos):
               nodeSucc = Node(succ[0], succ[1], nodeCost + succ[2], node)
               if nodeSucc not in closed:
                    opened.push(nodeSucc, nodeSucc.getTuple()[2])
    return False
```

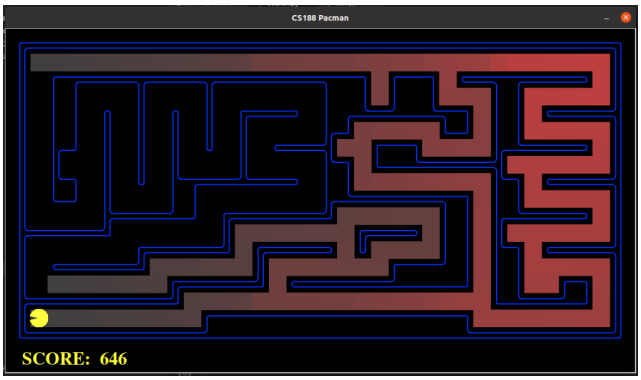
3.4. Screenshots

To check if our implementation of the Uniform Cost Search algorithm works properly we ran some test, among which are the following:



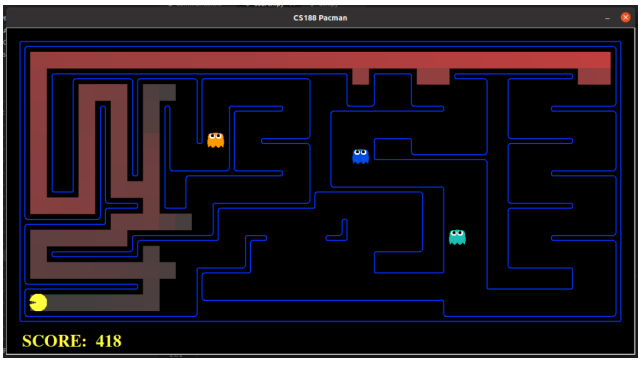

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

In this example, almost every node in the maze was expended, which is not something efficient when it comes to space complexity, but it is a way to find an optimal path.




```
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
```

In this second maze, Pacman finds the optimal path given that there was food on the roads on the right. That's why it doesn't follow the fastest path as before.




```
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
```

Now Pacman has to get to the target square while it avoids being caught by the ghosts. Consequently, when it tries to follow the lowest cost path and it runs into a ghost, he has to go change direction and that's why he follows the leftmost path.

The autograder gave us the following results:

```
jorge@Jorge-Lenovo:~/Documentos/Uni/3_Tercero/AI/Practice1/search$ python3 autograder.py -q q3
Starting on 2-23 at 10:22:48

Question q3
===========
*** PASS: test_cases/q3/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q3/graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases/q3/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q3/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q3/ucs_0_graph.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q3/ucs_1_problemC.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269
*** PASS: test_cases/q3/ucs_2_problemE.test
***     pacman layout:          mediumMaze
***     solution length: 74
***     nodes expanded:         260
*** PASS: test_cases/q3/ucs_3_problemW.test
***     pacman layout:          mediumMaze
***     solution length: 152
***     nodes expanded:         173
*** PASS: test_cases/q3/ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
*** PASS: test_cases/q3/ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###


Finished at 10:22:48

Provisional grades
==================
Question q3: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

3.5. Conclusions on Pacman´s behaviour

In these mazes there are no infinite paths and the cost of moving from one square to the one just next to it is 1, so the cost will always be finite. With all this, we can state that Uniform Cost Search is complete, it will always reach a solution.
It is optimal as well because the path cost of a node is always greater than the cost of its parent node (given that all costs are positive, in particular the cost is always 1).
UCS can expand too many nodes before getting too deep leading to an exponential space complexity, just as time complexity: $O(b^d)$.

SECTION 4

4.1. Personal comment on approach

As in the previous sections, we copied the code and changed just a line to introduce the value of the heuristic to the cost of each node. Doing that, we got ourselves the `aStarSearch()` working just fine.
But then we realized that we were just reusing code, and since that is never a good practice, we decided to implement a general function to use in the three previous sections and in this one. That's why we created the function `generalSearch()`. It takes the following arguments:

- **`problem`**: scenario in which we are working
- **`opened`**: the structure that will work as the opened list. It will change depending on the algorithm used (Stack for DFS, Queue for BFS and Priority Queue for UCS and A*)
- **`cost`**: this variable is used as a flag to tell apart between pushing into a non-priority structure (which doesn't need an argument indicating the cost of the node) and a priority structure (which needs the cost of the node to push the tuple into the structure). If nothing is specified, cost is `None` by default.
- **`heuristic`**: the heuristic function used to find the cost of a node. This parameter is only used by A*. If nothing is specified, heuristic is the `nullHeuristic` by default.

Once we finished this general function, we modified all the previous search functions to call this unified one and return directly its result.

4.2. Functions used

As I said before, we used the function `generalSearch()` and for the structure of the opened list we used a Priority Queue with its push and pop functions.

4.3. Code

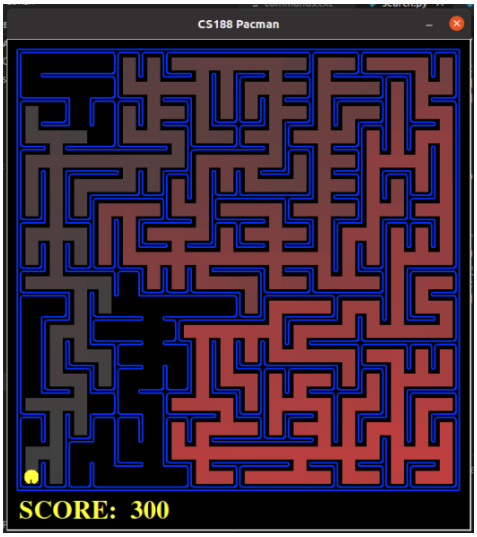
Here below we put the final version of all the search functions and the generalSearch function code:

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

def breadthFirstSearch(problem):
      return generalSearch(problem, util.Queue())

def uniformCostSearch(problem):
      return generalSearch(problem, util.PriorityQueue(), True)

def aStarSearch(problem, heuristic=nullHeuristic):
      return generalSearch(problem, util.PriorityQueue(), True, heuristic)
```

```
def generalSearch(problem, opened, cost=None, heuristic=nullHeuristic):
      root = problem.getStartState()
      rootNode = Node(root, "Stop", 0, None)

      # Depending on the structure it uses different costs
      if cost is None:
            opened.push(rootNode)
      else:
            cost = 0
            opened.push(rootNode, cost)

      closed = []
      sol = []

      # Iterate
      while not opened.isEmpty():
            # Choose from opened list a node to expand
            node = opened.pop()
            nodePos, nodeAct, nodeCost = node.getTuple()
            if problem.isGoalState(nodePos):
                  # Creates and returns lists of actions
                  while node.getParent() is not None:
                        sol.append(node.getTuple()[1])
                        node = node.getParent()
                  sol.reverse()
                  return sol

            if node not in closed:
                  closed.append(node)
                  # Expand node
                  for succ in problem.getSuccessors(nodePos):
                        nodeSucc = Node(succ[0], succ[1],
                                        nodeCost + succ[2], node)
                        if nodeSucc not in closed:
                              # Push the successor to the opened list
                              if cost is None:
                                    opened.push(nodeSucc)
                              else:
                                    heur = heuristic(succ[0], problem)
                                    cost = nodeSucc.getTuple()[2] + heur
                                    opened.push(nodeSucc, cost)

      return False
```

4.4. Screenshots

In this question we just needed to check the bigMaze with the Manhattan Heuristic and here we see the results:




```
[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
```

The path found is the optimum one, ergo the one with the lowest cost, and it expands less nodes than BFS but more than DFS. If we compare the nodes visited (red squares), just by looking at the maze we can confirm that the number of nodes expanded is obviously bigger than DFS. We can also see that the pattern is similar to the BFS one
The autograder gave us the following results:

```
jorge@Jorge-Lenovo:~/Documentos/Uni/3_Tercero/AI/Practice1/search$ python3 autograder.py -q q4
Starting on 2-23 at 10:47:17

Question q4
===========
*** PASS: test_cases/q4/astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q4/astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases/q4/astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases/q4/astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q4/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q4/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###


Finished at 10:47:17

Provisional grades
==================
Question q4: 3/3
------------------
Total: 3/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

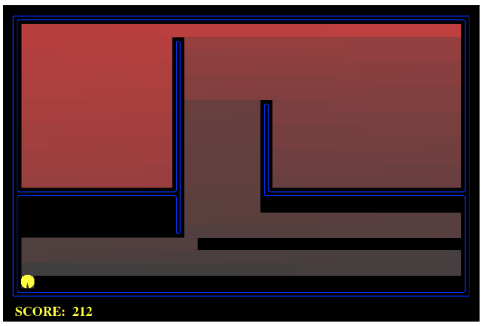
4.5. Conclusions on Pacman´s behaviour

If the heuristic is monotonic then A* using graph search is complete and optimal. Since the Manhattan Heuristic is based on the Manhattan Distance, which is a relaxation of this problem, the heuristic is admissible and, in this case, also consistent. This is why the A* algorithm using graph search always finds the optimal solution.
Although the algorithm it's not optimally efficient. This is because it is using graph search and not tree search, so it could discard a node which led to the optimum path.

4.6. Questions

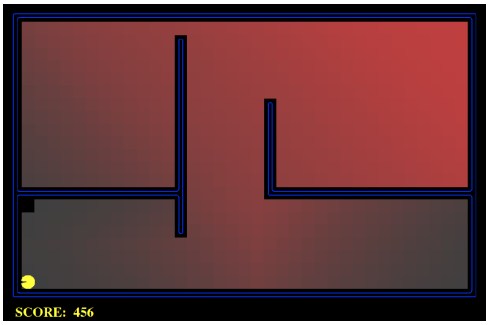
4) What happens on openMaze for the various search strategies?

- **DFS:** with DFS on openMaze it tours horizontally around all the blocks that lead to the goal state. In other words, it zig-zags horizontally till a solution is found. The solution found is not optimal (cost 298).



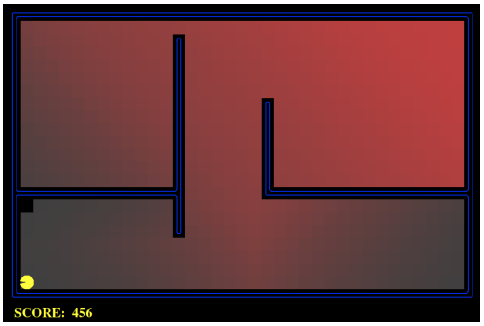

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 298 in 0.1 seconds
Search nodes expanded: 576
Pacman emerges victorious! Score: 212
Average Score: 212.0
Scores:        212.0
Win Rate:      1/1 (1.00)
Record:        Win
```

- **BFS:** with BFS it goes directly to the target tracing straight lines, first down, left, down and left again. The solution is optimal (cost 54) but it's not efficient.



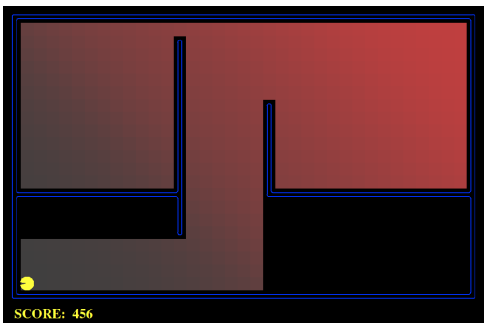

```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.3 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

- **UCS:** UCS finds the same solution as BFS (cost 54) and it expands the same amount of nodes (all of them except for 1).




```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.2 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

- **A*:** just like BFS and UCS, A* finds the optimal solution (cost 54), but in this case it expands less nodes than any of the other searches, because it is optimally efficient.




```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.2 seconds
Search nodes expanded: 535
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

SECTION 5

5.1. Personal comment on approach

In this variation of the game, Pacman has to go to every corner in order to win so finding the shortest path to a certain node as before is not enough. In this case, Pacman has to find the shortest path through the maze touching the 4 corners.
Not only the objective changed, but the states along the course of the algorithm as well. We came up with a new representation for the states of this problem which consists of the tuple (position, corners), where the position is the square where Pacman is located and corners is a list with the corners that it has not visited yet. Thus, we can check whether the algorithm has come to an end by checking the length of this list.

5.2. Functions used

In the CornersProblem class we didn't have to add anything in the initialization method but we had to implement the following methods:

- `getStartState()`: it returns the initial state consisting of the starting position/tuple and the initial list of corners which includes the four corners (none of them have been visited yet).
- `isGoalState()`: as explained in 1.1. the algorithm finishes when Pacman has been to all the four corners, so a given state is a goal if and only if the list with the corners yet to visit is empty
- `getSuccessors()`: for every direction Pacman can follow from a given state, if it does not encounter a wall, that next square automatically becomes a successor of the current position. Then, for every successor found, if it is a non-visited corner, the list of the state has to be updated.

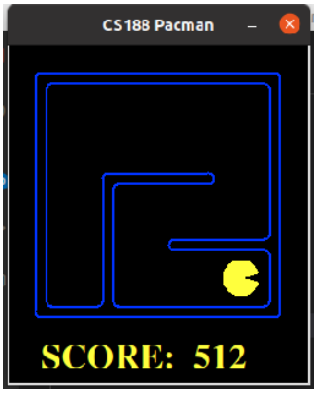
5.3. Code

```
def getStartState(self):
      return (self.startingPosition, list(self.corners))

def isGoalState(self, state):
      return len(state[1]) == 0
```

```
def getSuccessors(self, state):
    successors = [ ]
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]
        if not hitsWall:
             nextNode = (nextx, nexty)
             nextCorners = state[1].copy()
             if nextNode in nextCorners:
                  nextCorners.remove(nextNode)

             nextState = (nextNode, nextCorners)
             cost = 1
             successors.append((nextState, action, cost))
      self._expanded += 1
      return successors
```

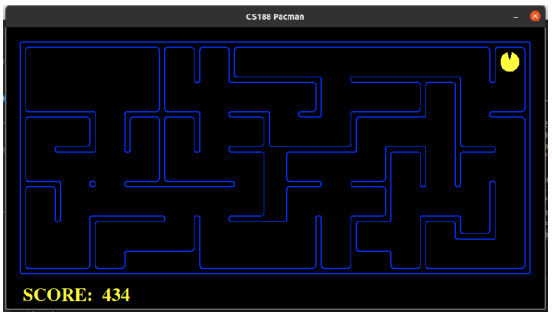
5.4. Screenshots

We executed the corners problem with tiny and medium mazes and obtained the following output:




```
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

In this test we observed that Pacman does not go to the closest corner first, but it finds a path to all the corners. Maybe it is not the path with the lowest cost because the algorithm used to find the treats is DFS and, as commented before, DFS is not optimal.




```
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

It happens the same as with the tiny maze but it still finds a path to reach every corner.

The autograder gave us the following results:

```
jorge@Jorge-Lenovo:~/Documentos/Uni/3_Tercero/AI/AI/Practice1/search$ python3 autograder.py -q q5
Note: due to dependencies, the following tests will be run: q2 q5
Starting on 3-1 at 9:24:29

Question q2
===========
*** PASS: test_cases/q2/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q2/graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases/q2/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q2/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q2/pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###


Question q5
===========
*** PASS: test_cases/q5/corner_tiny_corner.test
***     pacman layout:          tinyCorner
***     solution length:                28

### Question q5: 3/3 ###


Finished at 9:24:29

Provisional grades
==================
Question q2: 3/3
Question q5: 3/3
------------------
Total: 6/6

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

5.5. Conclusions on Pacman´s behaviour

Pacman successfully managed to go to the four corners in all the tests we put him through, but since the algorithms we used to test were oriented to find a single solution instead of the fastest way to several solutions, the results we obtained were not the best ones we could expect. For that, it is necessary to design a consistent heuristic to solve this problem, as we would explain in the following section.

SECTION 6

6.1. Personal comment on approach

For this section we started by brainstorming admissible heuristics, as the *Hint* said. First of all, we tried implementing an heuristic that calculated the distance to the closest corner, but it wasn't good enough, it expanded around 1400 nodes.
Then we thought about using a second complementary heuristic to add to this one. Our plan was to count the walls in a straight line (first going on a horizontal direction and then vertical, or vice versa) between the current state and the nearest corner. After we implemented it, we saw it was even worse than the last one.
Finally we came up with an heuristic that expanded 692 nodes for the bigMaze, just under 700. This final heuristic first calculates the Manhattan Distance to the closest non-visited corner and from that corner, it gets again the distance to the next closest non-visited corner till it visits all the corners. To sum up, it calculates the shortest path to go from the current state to every non-visited corner by minimizing the Manhattan Distance in every step of the way.

6.2. Functions used

To complete this task, we implemented the `cornersHeuristic()` function using the Node class we declared in the first section at the end of `search.py`.

6.3. Code

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

    # visited corners
    if state[0] in corners and len(state[1]) == 0:
        return 0

    minDist = 999999
    newNode = None
    for c in state[1]:
        dist = util.manhattanDistance(c, state[0])
        if minDist > dist:
            newNode = c
            minDist = dist

    newCorners = state[1].copy()
    newCorners.remove(newNode)
    newState = (newNode, newCorners)
    return minDist + cornersHeuristic(newState, problem)
```

6.4. Screenshots

After executing the command we saw how the pacman solved the maze eating the treat at each of the four corners. Here's the result we got:

```
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 692
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

As we can see, it expands 692 nodes and the cost of the solution found is 106. The autograder gave us the next output:

```
jorge@Jorge-Lenovo:~/Documentos/Uni/3_Tercero/AI/AI/Practice1/search$ python3 autograder.py -q q6
Note: due to dependencies, the following tests will be run: q4 q6
Starting on 3-1 at 9:26:31

Question q4
===========
*** PASS: test_cases/q4/astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q4/astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases/q4/astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases/q4/astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q4/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q4/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###
```

```
Question q6
===========
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West', 'North',
 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'West', 'West', 'South', '
South', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'South', 'West', 'Wes
t', 'South', 'South', 'South', 'West', 'West', 'East', 'East', 'North', 'North', 'North', 'East', 'East',
'East', 'East', 'East', 'East', 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'East',
'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'Eas
t', 'East', 'South', 'South', 'South', 'South', 'East', 'East', 'North', 'North', 'East', 'East', 'South',
 'South', 'South', 'South', 'South', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West'
, 'West', 'North', 'North', 'East', 'East', 'North', 'North']
path length: 106
*** PASS: Heuristic resulted in expansion of 692 nodes

### Question q6: 3/3 ###


Finished at 9:26:31

Provisional grades
==================
Question q4: 3/3
Question q6: 3/3
------------------
Total: 6/6

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

6.5. Conclusions on Pacman´s behaviour

Since the heuristic is consistent and we are using A*, the algorithm is optimal, complete and optimally efficient, so it always finds the lowest cost solution by expanding the lower amount of nodes possible.

6.6. Question

6) Explain the logic behind your heuristic. The logic of the heuristic is explained in paragraph 6.1.

SECTION 7

We believe this was both an interesting and mind challenging practice, and a fun and entertaining one. Implementing the different algorithms and creating our own heuristic to make the pacman find the path to the treats was fun and made us think about different ways to implement the searches and the heuristic.