Artificial Intelligence. Practice 1.

Pablo Cuesta Sierra, Álvaro Zamanillo Sáez
Group 2351. Pair 5.

February 27, 2022

Contents

1 Question 1: Finding a Fixed Food Dot using Depth First Search

1.1 Personal comment on the approach and decisions of the proposed solution (0.5pt)

In order to make the following questions easier, we have decided to abstract the notion of node and make a single generic search function where the type of container used to implement the *openList* determines which algorithm is performed.

Listing 1: Generic algorithm (taken from the AI theory slides)

```
function Tree-Search (problem, strategy )
    ;; returns solution or fail
    ;; opened-list contains the nodes in the fringe of the search tree
    Initialize search-tree with root-node
    Initialize opened-list with root-node
    Iterate
        If (opened-list is empty) then return fail
        Choose from opened-list, according to strategy, a node to expand.
            If (node satisfies goal-test )
                then return solution (path from root-node to current node )
            else remove node from opened-list
                expand node
                add child nodes to opened-list
```

1.1.1 List & explanation of the framework functions used

- The `Node` class, with methods:
  - `__init__(self, state, action, cost, parentNode)`
  - `path(self)`
  - `__eq__(self, __o)`
- `expandNode(problem , node , openList , closedList)`
- `search(problem, openList)`
- `depthFirstSearch(problem)`

For the generic function (`search`), we have used the implementation of 'containers' done in the *util.py* file (*Stack*, *Queue*, *PriorityQueueWithFunction*), which have the methods *pop()* and *push()*. Also, to initialize the elements of the search, we have to use the *SearchProblem* class methods, to get the start state, the successors of a state and whether a state is a goal state.

For the algorithm, we had to choose a structure for the nodes of the graph search. Each node contains a game state, the last action taken to get to that state, the cost of the path that goes from the start to this state, and a pointer to its parent node. In order to compare nodes, as we are implementing graph search and do not want to expand two nodes that have the same state, we consider that two nodes are equal if their states are equal.

The node also has a method that calculates the path taken from the root to that node, which goes up the tree using the reference to the parent to build this list of actions. Another implementation would have been to keep only state and path, for each node. However, each time we concatenate a new action to the path list of a node to create a successor, the list would have to be copied (`new_list = prev_list + [new_action]`), which means a high cost both in terms of time and space efficiency: there is a cost of $\mathcal{O}(N)$ for each operation of that kind. Our implementation has a $\mathcal{O}(N)$ cost (N being the size of the final list), but the path is only computed once in the whole search algorithm (when the solution is found).

This particular algorithm, *DFS*, requires the 'openList' to be a stack.

1.1.2 Includes code written by students

Listing 2: The Node class (*search.py*)

```
class Node:
    '''
        Node class: nodes used for the tree of the search function.
            state: state of the problem
            action: last action taken from the predecessor of the state to get to this
state
            cost: total cost from the root to this state
            parentNode: points to the parent node
            path: list of actions from the root to the node
    '''

    def __init__(self, state, action, cost, parentNode):
        self.state = state
        self.action = action
        self.cost = cost
        self.parentNode = parentNode

    @property
    def path(self):
        '''
            The path from the node to the root.
            Returns the list of actions from the beginning to the node.
        '''
        node,actions=self,[]

        while node.parentNode != None:
            actions.insert(0,node.action)
            node = node.parentNode
        return actions

    def __eq__(self, __o):
        if isinstance(__o, Node):
            return __o.state == self.state
        return False
```

Listing 3: The search function (*search.py*)

```
def expandNode(problem, node, openList, closedList):
    '''
        Expands a node for the search algorithm
            problem: problem to be solved by the search function
            node: node to be expanded
            openList: openList of the algorithm
            closedList: container of the search function
    '''
    closedList.append(node)
    for successor in problem.getSuccessors(node.state):
        openList.push(Node(
                        state=successor[0],
                        action=successor[1],
                        cost=successor[2]+node.cost,
                        parentNode=node))

def search(problem, openList):
    '''
        Search function that generalizes the different search types.
            problem: problem to solve
            openList: the empty openList to be used in the implementation.
    '''
    closedList = []
    openList.push(Node(
            state=problem.getStartState(),
            action=None,
            cost=0,
```

```
                parentNode=None))  # root node

        while not openList.isEmpty():
            currentNode = openList.pop()
            if problem.isGoalState(currentNode.state):
                return currentNode.path
            if currentNode not in closedList:
                expandNode(problem, currentNode, openList, closedList)

        return [] # No solution found
```

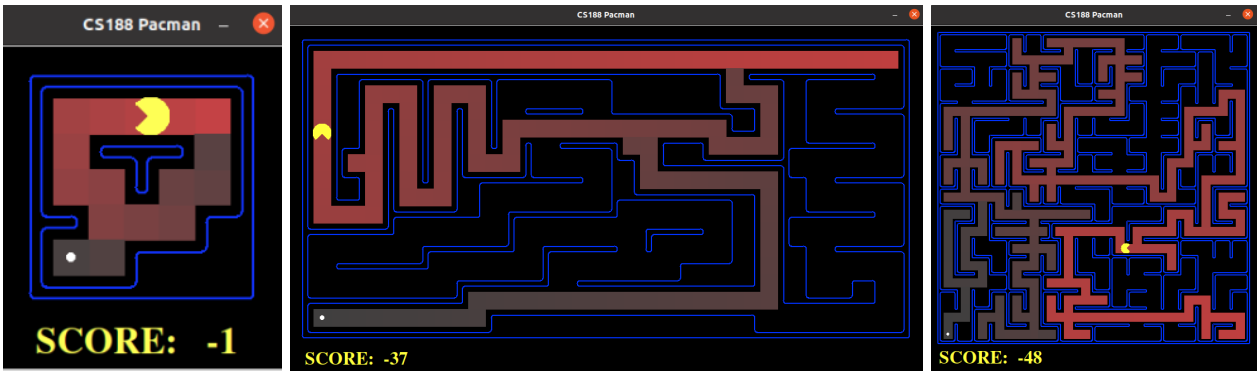
Listing 4: The *DFS* function (*search.py*)

```
    def depthFirstSearch(problem):
        """Search the deepest nodes in the search tree first."""
        return search(problem, util.Stack())
```

1.1.3 Screenshots of executions and test carried out analyzing the results

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 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
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l mediumMaze -p SearchAgent
[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
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l bigMaze -z .5 -p SearchAgent
[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
```

Figure 1: Results of the three tests using DFS



(a) (b) (c)

Figure 2: Exploration for mazes with *DFS*: *tinyMaze*, *mediumMaze* and *bigMaze*.

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 autograder.py -q q1
Starting on 2-12 at 11:50:32

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_imp.test
***     solution:               []
***     expanded_states:        ['A', 'C', 'B']
*** 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
*** PASS: test_cases/q1/pacman_imp.test
***     pacman layout:          trapped
***     solution length: 0
***     nodes expanded:         6

### Question q1: 7/7 ###


Finished at 11:50:32

Provisional grades
==================
Question q1: 7/7
------------------
Total: 7/7

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

Figure 3: Results of the autograder tests

1.2 Conclusions on the behavior of Pacman, it is optimal (y/n), reaches the solution (y/n), nodes that it expands, etc (0.5pt)

1.2.1 Answer to question 1.1: Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

First of all, we notice (by looking at the code of the function `getSuccessors` in *searchAgents.py*) that the order in which the children are generated is `(NORTH,SOUTH,EAST,WEST)`. Therefore, when using *DFS* we expect that the algorithm explores all the way north until it reaches a wall, then all the way east (if it goes south, it will be a duplicate state) and so on.

On the other hand, tiles explored by the algorithm but belonging to a branch that does not resolute, are not visited by Pacman.

This two behaviours are clearly shown in the screenshots of the tests (figure 2).

1.2.2 Answer to question 1.2: Is this a least cost solution? If not, think about what depth-first search is doing wrong.

DFS is not optimal. The problem lies on the fact that the solution provided by DFS is the first branch visited which reachs the goal node; however, it is likely to happen that there exists another branch which is sorter and also leads to the solution.

2 Question 2: Breadth First Search

2.1 Personal comment on the approach and decisions of the proposed solution (0.5pt)

Having done in the previous question the generic search algorithm, we just have to use the same algorithm, in this case with the 'openList' being a queue (LIFO).

2.1.1 List & explanation of the framework functions used

- `breadthFirstSearch(problem)`

We used the `search()` function explained in the previous section (1.1), along with the *util.Queue* class for the open list.

2.1.2 Includes code written by students

Listing 5: The *BFS* function (*search.py*)

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    return search(problem, util.Queue())
```

2.1.3 Screenshots of executions and test carried out analyzing the results

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 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
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
[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
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
[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
```

Figure 4: Results of the three tests using BFS

```
A random puzzle:
-------------
| 3 | 2 | 5 |
-------------
| 6 | 1 | 8 |
-------------
| 7 |   | 4 |
-------------
BFS found a path of 9 moves: ['right', 'up', 'up', 'left', 'down', 'down', 'left', 'up', 'up']
```

(a)

```
After 1 move: right
-------------
| 3 | 2 | 5 |
-------------
| 6 | 1 | 8 |
-------------
| 7 | 4 |   |
-------------
Press return for the next state...
After 2 moves: up
-------------
| 3 | 2 | 5 |
-------------
| 6 | 1 |   |
-------------
| 7 | 4 | 8 |
-------------
Press return for the next state...
After 3 moves: up
-------------
| 3 | 2 |   |
-------------
| 6 | 1 | 5 |
-------------
| 7 | 4 | 8 |
-------------
```

(b)

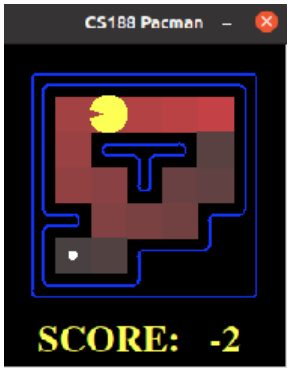
```
After 4 moves: left
-------------
| 3 |   | 2 |
-------------
| 6 | 1 | 5 |
-------------
| 7 | 4 | 8 |
-------------
Press return for the next state...
After 5 moves: down
-------------
| 3 | 1 | 2 |
-------------
| 6 |   | 5 |
-------------
| 7 | 4 | 8 |
-------------
Press return for the next state...
After 6 moves: down
-------------
| 3 | 1 | 2 |
-------------
| 6 | 4 | 5 |
-------------
| 7 |   | 8 |
-------------
```

(c)

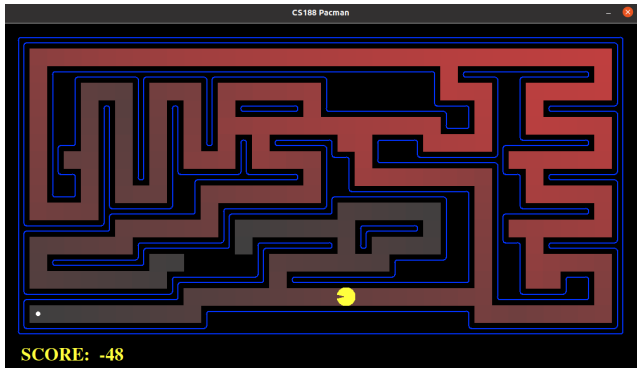
```
After 7 moves: left
-------------
| 3 | 1 | 2 |
-------------
| 6 | 4 | 5 |
-------------
|   | 7 | 8 |
-------------
Press return for the next state...
After 8 moves: up
-------------
| 3 | 1 | 2 |
-------------
|   | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
After 9 moves: up
-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
```

(d)

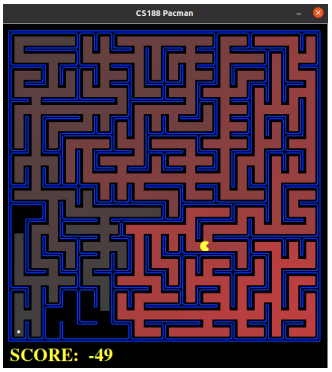
Figure 5: Execution of the eightpuzzle



(a)



(b)



(c)

Figure 6: Exploration for mazes with *BFS*: *tinyMaze*, *mediumMaze* and *bigMaze*.

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 autograder.py -q q2
Starting on 2-12 at 12:16:14

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_imp.test
***     solution:               []
***     expanded_states:        ['A', 'B', 'C']
*** 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
*** PASS: test_cases/q2/pacman_imp.test
***     pacman layout:          trapped
***     solution length: 0
***     nodes expanded:         6

### Question q2: 7/7 ###


Finished at 12:16:14

Provisional grades
==================
Question q2: 7/7
------------------
Total: 7/7

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

Figure 7: Results of the autograder tests

2.2 Conclusions on the behavior of Pacman, it is optimal (y/n), reaches the solution (y/n), nodes that it expands, etc (0.5pt)

2.2.1 Answer to question 3: Does BFS find a least cost solution? If not, check your implementation.

BFS algorithm does not expand to height $h+1$ until all nodes of height h have been explored. Thus, the solution obtained has to be the one with minimum height; in other words, the shortest path.

3 Question 3:

3.1 Personal comment on the approach and decisions of the proposed solution (0.5pt)

We have implement the *UCS* search as an $A*$ search where the heuristic is the trivial one ($h \equiv 0$). Therefore we only take into account the total cost to reach a node from the root.

3.1.1 List & explanation of the framework functions used

- `uniformCostSearch(problem)`

As mentioned before, we have used the function for $A*$ search which, once again, uses the generic function `search()`. In the next section, we will cover with more detail the function `aStarSearch()`. The 'null heuristic' is a function already given.

3.1.2 Includes code written by students

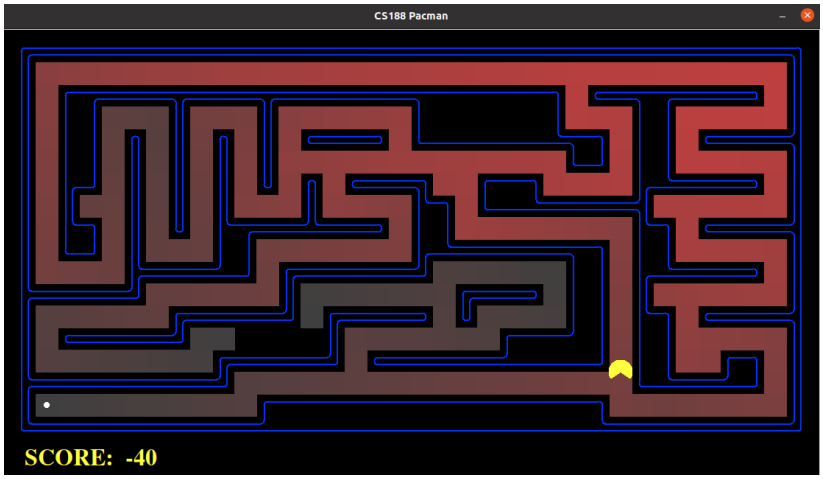
Listing 6: The *UCS* function (*search.py*)

```
    def uniformCostSearch(problem):
        """Search the node of least total cost first."""
        return aStarSearch(problem, nullHeuristic)
```

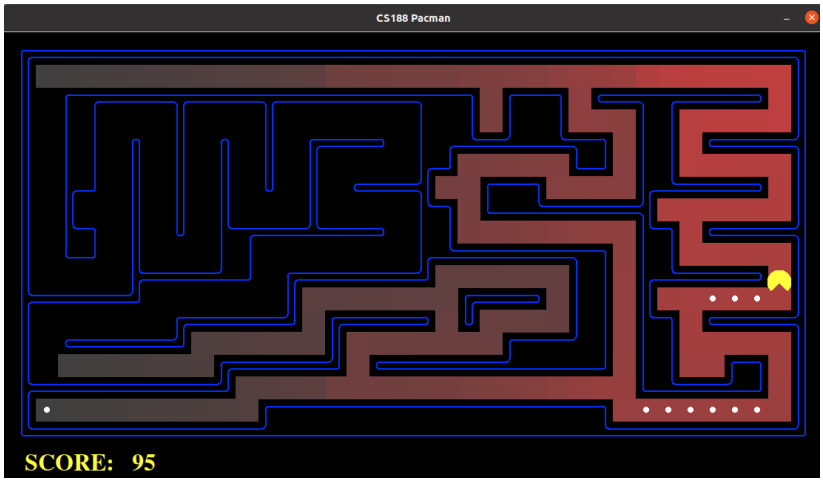
3.1.3 Screenshots of executions and test carried out analyzing the results

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 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
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l mediumDottedMaze -p StayEastSearchAgent
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
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l mediumScaryMaze -p StayWestSearchAgent
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
```

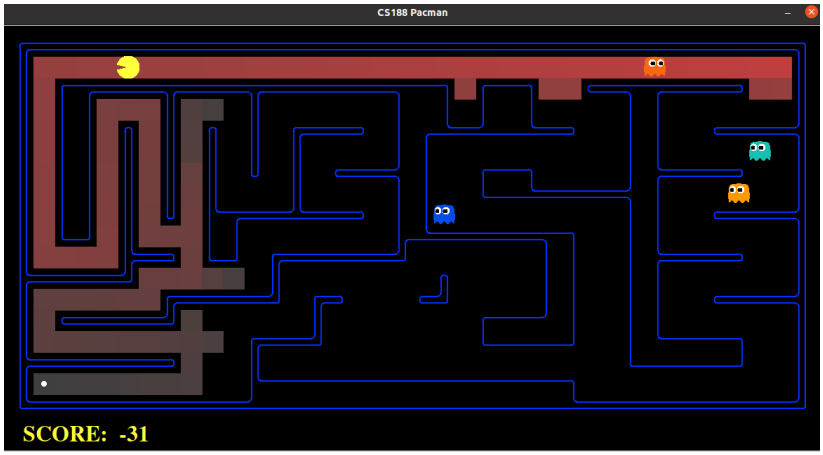
Figure 8: Results of the three tests using UCS



(a)



(b)



(c)

Figure 9: Exploration for mazes: (a) *mediumMaze* with *UCS*, (b) *mediumDottedMaze* with *StayEastSearchAgent*, (c) *mediumScaryMaze* with *StayWestSearchAgent*

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 autograder.py -q q3
Starting on 2-12 at 12:33:57

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_imp.test
***     solution:               []
***     expanded_states:        ['A', 'B', 'C']
*** 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/pacman_imp.test
***     pacman layout:          trapped
***     solution length: 0
***     nodes expanded:         6
*** 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: 7/7 ###


Finished at 12:33:57

Provisional grades
==================
Question q3: 7/7
------------------
Total: 7/7
```

Figure 10: Results of the autograder tests

3.2 Conclusions on the behavior of Pacman, it is optimal (y/n), reaches the solution (y/n), nodes that it expands, etc (0.5pt)

Uniform cost search is always optimal in terms of the cost defined. Thus, executing *UCS* assigning a cost of 1 to all the edges, results on a solution of minimum length. However, if the agents `StayWestSearchAgent()` or `StayEastSearchAgent()` are used, the solution is not the shortest because this time, the cost function does not represent length of the path to a node but longitude of its position.

4 Question 4:

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

The algorithm for *A** is just the same as for the other search types. The only difference, as expected, is the way we extract nodes from the *openList*. Thus, we have reused our generic search function once again.

4.1.1 List & explanation of the framework functions used

- `aStarSearch(problem)`

The container used for this search is a priority queue (*PriorityQueueWithFunction*) where the priority is defined by the evaluation function which is the sum between the cost to reach the node and the heuristic value of it. This function is defined on the fly using a lambda expression.

4.1.2 Includes code written by students

Listing 7: The *A** function (*search.py*)

```
    def aStarSearch(problem, heuristic=nullHeuristic):
        """Search the node that has the lowest combined cost and heuristic first."""
        return search(problem,
                      util.PriorityQueueWithFunction(
                        lambda node: node.cost + heuristic(node.state, problem)))
```

4.1.3 Screenshots of executions and test carried out analyzing the results

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[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
```

Figure 11: Results of the test using *A**

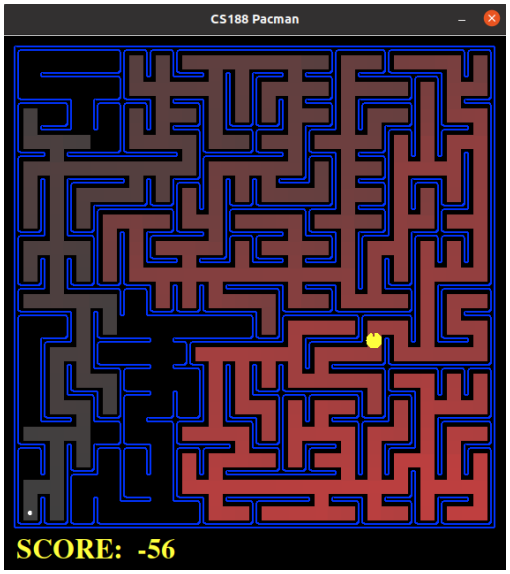


Figure 12: Exploration of the maze from previous execution

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 autograder.py -q q4
Starting on 2-12 at 12:53:48

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 12:53:48

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.
```

Figure 13: Results of the autograder tests

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

As we know from theory, $A*$ with elimination of repeated states is optimal if the heuristic used is admissible and consistent. For the position problem, the *ManhattanHeuristic* satisfies both conditions. If we consider a relaxation of the problem, a maze with no walls, the optimal path has exactly the length given by the Manhattan metric. It is also consistent because the cost of going from a node to one of its successor is always equal to 1, and at most, the successor will be one tile closer to the solution, (it may be 1 tile farther too). In other words, the following inequality holds:

$$\forall n, n' : n' \text{ succesor of } n,\ h(n) \leq \text{cost}(n \to n') + h(n') \tag{1}$$

Indeed, as this heuristic is consistent, it immediately follows that it is also admissible.

4.2.1 Answer to question 4: What happens on openMaze for the various search strategies?

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l openMaze -p SearchAgent -a fn=dfs --frameTime 0
[SearchAgent] using function dfs
[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
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l openMaze -p SearchAgent -a fn=bfs --frameTime 0
[SearchAgent] using function bfs
[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
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l openMaze -p SearchAgent -a fn=ucs --frameTime 0
[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
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l openMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic --frameTime 0
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.1 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
```

Figure 14: Execution for the four strategies in the *openMaze*.

The layout *openMaze* has almost no walls so the perfomance of *DFS* (which is not an optimal algorithm) will be determined by the order of how the successors are generated. Just for instance, with the order `(NORTH,SOUTH,EAST,WEST)`, the solution has a length of 298 and 576 are expanded. However if we change the order to `(EAST,SOUTH,NORTH,WEST)`, the solution has a cost of 54 and the expanded nodes are 247.

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l openMaze -p SearchAgent -a fn=dfs --frameTime 0
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 247
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figure 15: *BFS* with the order of successor selection: `(EAST,SOUTH,NORTH,WEST)` in *openMaze*.

On the other hand, we know that *BFS*, *UCS* and *A** (with a consisten heuristic), will provide an optimal solution. Nonetheless, the number of nodes expanded vary. *BFS* and *UCS* do not take into account how close a position is from the solution, therefore nodes are expanded only taking into account the cost of reaching them. However, if we introduce the Manhattan heuristic into the calculation, we can ensure that nodes which are further from the solution will only be explored when it is the only available way. So, to sum up, using a consistent heuristic will result in reaching an optimal solution and generally with a lower number of nodes expanded. It is worth mentioning that less nodes expanded may not always result in less computing time.

5 Question 5: Finding All the Corners

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

For this problem (*CornersProblem*) we have chosen the following state representation:

$$s = (s_p, s_c), \tag{2}$$

where s_p is the 2-tuple that denotes the position of Pacman (coordinates as defined in the *Grid* class –which must be a legal position) in the corners problem; and s_c is the set of corners (their coordinates) that have yet not been visited by Pacman. We have chosen to keep track of the unvisited corners instead of the visited ones because it makes easier the computations of the heuristic for the next section.

For the problem class itself, it is quite similar to the *PositionProblem* class but we have added two new atributes (*shortSide* and *longSide*) which hold the size of the maze (the distances between adjacent corners). These attributes are used in our heuristic of the next section.

5.1.1 List & explanation of the framework functions used

- The `CornersProblemState` class, with methods:
  - `__init__(self, position, remainingTargets)`
  - `__eq__(self, __o)`
- The `CornersProblem` class, with methods:
  - `__init__self, startingGameState)`
  - `getStartState(self)`
  - `isGoalState(self, state)`
  - `getStartState(self)`
  - `getSuccessors(self, state)`
- `expandNode(problem , node , openList , closedList)`
- `search(problem, openList)`
- `depthFirstSearch(problem)`

For this section we had to complete the *CornersProblem* class, so that this problem had appropriate states and the successors were calculated according to the defined problem.

5.1.2 Includes code written by students

Listing 8: The state and problem definition of the corners problem (*searchAgents.py*)

```
CORNERS_PROB_ACTION_COST = 1

class CornersProblemState():
    '''
        The state of the corners problem.
            position: (x,y) - position in the game
            remainingTargets: (list of) remaining targets (positions)
            to solve the game - (it is not possible to have the position
            of a state inside its own remainingTargets, this initialization
            takes care of that)
    '''
    def __init__(self, position, remainingTargets):
        self.position = position
        self.remainingTargets = remainingTargets.copy()
```

```
            if position in remainingTargets:
                self.remainingTargets.remove(position)

        def __eq__(self, __o):
            if isinstance(__o, CornersProblemState):
                return (__o.position == self.position) \
                        and (set(__o.remainingTargets) == set(self.remainingTargets))
            return False

    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

            self.startState = CornersProblemState(self.startingPosition, list(self.corners))
            self.shortSide = min(self.walls.height,self.walls.width)-3
            self.longSide = max(self.walls.height,self.walls.width)-3

        def getStartState(self):
            """
            Returns the start state (in your state space, not the full Pacman state
            space)
            """
            return self.startState

        def isGoalState(self, state):
            """
            Returns whether this search state is a goal state of the problem.
            """
            return not state.remainingTargets

        def getSuccessors(self, state):
            """
            Returns successor states, the actions they require, and a cost of 1.

            As noted in search.py:
                For a given state, this should return a list of triples, (successor,
                action, stepCost), where 'successor' is a successor to the current
                state, 'action' is the action required to get there, and 'stepCost'
                is the incremental cost of expanding to that successor
            """

            successors = []
            for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.
    WEST]:
                "*** YOUR CODE HERE ***"
                x,y = state.position
                dx, dy = Actions.directionToVector(action)
                nextx, nexty = int(x + dx), int(y + dy)
                if not self.walls[nextx][nexty]:
                    nextState = CornersProblemState((nextx,nexty),state.remainingTargets)
```

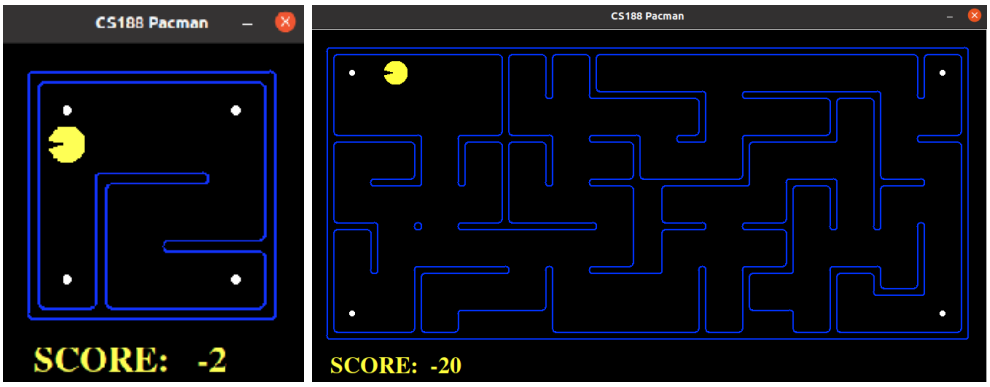
```
                    cost = CORNERS_PROB_ACTION_COST
                    successors.append((nextState, action, cost))

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

Note: the `getCostOfActions` method from the *CornersProblem* class has not been included, because we did not have to modify it.

5.1.3 Screenshots of executions and test carried out analyzing the results

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=Corne
rsProblem
[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
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=Cor
nersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 1.3 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figure 16: Results of the two corner tests



(a) (b)

Figure 17: Execution of the (a) *tinyCorners* maze and the (b) *mediumCorners* maze.)

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 autograder.py -q q5
Note: due to dependencies, the following tests will be run: q2 q5
Starting on 2-13 at 11:56:12

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_imp.test
***     solution:               []
***     expanded_states:        ['A', 'B', 'C']
*** 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
*** PASS: test_cases/q2/pacman_imp.test
***     pacman layout:          trapped
***     solution length: 0
***     nodes expanded:         6

### Question q2: 7/7 ###


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

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


Finished at 11:56:12

Provisional grades
==================
Question q2: 7/7
Question q5: 3/3
------------------
Total: 10/10

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

Figure 18: Results of the autograder tests

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

Pacman reaches the solution in all tests. The fact that this is a different problem does not change the way the algorithms find solutions; therefore, the comments made in previous sections about each algorithm still hold in this problem. *BFS* is optimal.

6 Question 6: Corners Problem: Heuristic

6.1 Personal comment on the approach and decisions of the proposed solution (1.5pt)

6.1.1 List & explanation of the framework functions used

- `minCornerTravel(numCorners, remainingCorners, shortSide, longSide)`
- `cornersHeuristic(state, problem)`

In this section, we only had to complete the `cornersHeuristic` function.
We also used the `util.manhattanDistance()` function to calculate some distances.

6.1.2 Includes code written by students

Listing 9: The heuristic definition (*searchAgents.py*)

```
    def minCornerTravel(numCorners, remainingCorners, shortSide,longSide):
        '''
        Returns the distance that Pacman would travel (starting from one of the
        remaining corners and ending in the last remaining corner) in the corners
        problem if there were no internal walls, and starting form the corner that
        makes this distance the minimum.
            numCorners: the length of remaining corners
            remainingCorners: the remaining corners to visit
            shortSide: the length of the short side of the grid
            longSide: the length of the long side of the grid
        '''
        if numCorners == 1:
            return 0
        if numCorners == 2:
            return util.manhattanDistance(remainingCorners[0],remainingCorners[1])
        if numCorners == 3:
            return shortSide + longSide
        return 2*shortSide + longSide

    def cornersHeuristic(state, problem):
        """
        A heuristic for the CornersProblem that you defined.

        state:   The current search state
                 (a data structure you chose in your search problem)

        problem: The CornersProblem instance for this layout.

        This function should always return a number that is a lower bound on the
        shortest path from the state to a goal of the problem; i.e.  it should be
        admissible (as well as consistent).
        """
        if not state.remainingTargets:
            return 0

        nearestCornerDistance = min(
            [util.manhattanDistance(state.position, c)
                for c in state.remainingTargets]
        )
        minDistanceBetweenCorners = minCornerTravel(
            len(state.remainingTargets),
            state.remainingTargets,
            problem.shortSide,
            problem.longSide
        )
        return nearestCornerDistance + minDistanceBetweenCorners
```

6.1.3 Screenshots of executions and test carried out analyzing the results

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 pacman.py -l mediumCorners -p AStarCornersAgent
Path found with total cost of 106 in 0.2 seconds
Search nodes expanded: 774
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figure 19: Results of the test using $A*$ for the corners problem.

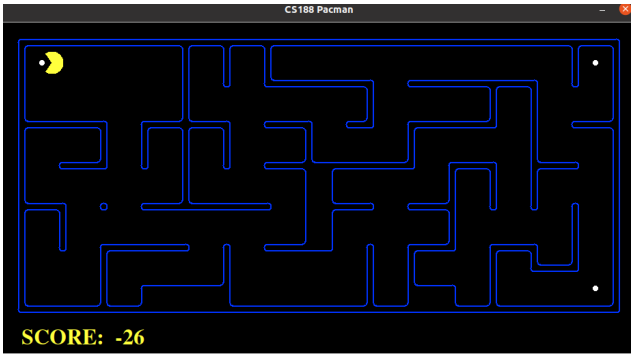


Figure 20: Exploration of the maze from previous execution

```
pcuestas@ptp:~/rep/IA/p1/scripts$ python3 autograder.py -q q6
Note: due to dependencies, the following tests will be run: q4 q6
Starting on 2-13 at 13:11:32

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
*** 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', 'West', 'South', 'South', 'South', 'West', 'West', 'East', 'East', 'North', 'No
rth', 'North', 'East', 'East', 'East', 'East', 'East', 'East', 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'E
ast', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'East', '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 774 nodes
path: ['East', 'South', 'South', 'West', 'West', 'South', 'West', 'South', 'South', 'East', 'East', 'East', 'West', 'West', 'West',
 'West', 'West', 'North', 'North', 'North', 'North', 'North']
path length: 22
*** PASS: Heuristic resulted in expansion of 24 nodes

### Question q6: 6/6 ###


Finished at 13:11:32

Provisional grades
==================
Question q4: 3/3
Question q6: 6/6
------------------
Total: 9/9
```

Figure 21: Results of the autograder tests

6.2 Conclusions on the behavior of Pacman, it is optimal (y/n), reaches the solution (y/n), nodes that it expands, etc (1.5pt)

This implementation is optimal, because it uses $A*$, with elimintation of repeated states, and a consistent heuristic (proof in the next subsection). It also reaches the solution, and expands 774 nodes in the *mediumCorners* maze.

6.2.1 Answer to question 5: Explain the logic behind your heuristic

For the heuristic, we have solved the cost of the problem with a relaxation: there are no internal walls. In the case of 3 remaining corners, there is also another relaxation: we assume that Pacman is closest to one of the 2 corners that are opposite in the grid. This second relaxation could be solved by complicating the heuristic a little, but we have observed that it is not worth the computational cost it adds, taking into account that the improvement in terms of explorated nodes is not significant.

Notation: we will use d to denote the Manhattan Distance (in this case, in $(\mathbb{N} \cup \{0\})^2 \times (\mathbb{N} \cup \{0\})^2$).

$$d((x_1, x_2), (y_1, y_2)) = |y_1 - x_1| + |y_2 - x_2| \tag{3}$$

The heuristic chosen (h) has been defined as follows: given a state s, as defined in 2, we define:

$$h_1(s) = \left\{ \begin{array}{ll} 0 & \text{if} \quad |s_c| = 0 \\ \min\{d(s_p, c) : c \in s_c\} & \text{if} \quad |s_c| > 0 \end{array} \right\} \tag{4}$$

This h_1 function calculates the minimum distance from Pacman's position to a corner that has not been visited.

$$h_2(s) = \left\{ \begin{array}{ll} 0 & \text{if} \quad |s_c| \leq 1 \\ d(c_1, c_2), \text{ where } \{c_1, c_2\} = s_c & \text{if} \quad |s_c| = 2 \\ m + M & \text{if} \quad |s_c| = 3 \\ 2m + M & \text{if} \quad |s_c| = 4 \end{array} \right\}, \tag{5}$$

where $m = \min S$ and $M = \max S$, $S = \{\text{distances between any two adjacent corners in the grid}\}$.
Finally,

$$h(s) = h_1(s) + h_2(s) \tag{6}$$

We will now prove that h is consistent, i.e. that for each node s and for each successor s' of s, $h(s) \leq \text{cost}(s, s') + h(s') = 1 + h(s')$. (The cost is always 1, from any node to any of its successors).

Proof. The proof will be separated into two cases.

1. Case $s_c = s'_c$, this happens if s'_p is not a corner. By definition of h_2, in this case, $h_2(s) = h_2(s')$. To be proven: $h_1(s) \leq h_1(s') + 1$:

   (a) If $|s_c| = 0$, the s is a goal state and has no succesors.

   (b) If $|s_c| > 0$; $h_1(s) = d(s_p, c_0)$ and $h_1(s') = d(s'_p, c_1)$, for $c_0, c_1 \in s_c = s'_c$. Also, $d(s_p, s'_p) = 1$. So (triangle inequality):

$$\begin{gathered} d(s_p, c_1) \leq 1 + d(s'_p, c_1) \\ d(s'_p, c_1) \leq 1 + d(s_p, c_1) \\ \Rightarrow -1 \leq d(s_p, c_1) - d(s'_p, c_1) \leq 1 \\ \Rightarrow \exists k \in \{-1, 0, 1\} : d(s_p, c_1) = d(s'_p, c_1) + k \\ \Rightarrow h_1(s) = d(s_p, c_0) \leq d(s_p, c_1) = d(s'_p, c_1) + k \leq d(s'_p, c_1) + 1 = h_1(s') + 1 \end{gathered}$$

2. Case $s_c \neq s'_c$, this happens if s'_p is a corner. In this case, $h_1(s) = 1$.

   (a) If $|s_c| = 0$ (again, this does not make much sense, because the algorithm would stop before evaluating successors of s if s is a goal state, and we have defined goal states as those whose remaining targets-set is empty).

   (b) If $|s_c| = 1$, then $|s'_c| = 0$, so $h(s') = 0$, and $h(s) = 1 \leq 1 + h(s') = 1$

   (c) If $|s_c| = 2$, then $|s'_c| = 1$. $s_c = \{s'_p, c_1\}$, $s'_c = \{c_1\}$. $h(s) = 1 + d(s'_p, c_1) = 1 + h(s')$

   (d) If $|s_c| = 3$, then $|s'_c| = 2$. $s_c = \{s'_p, c_0, c_1\}$, $s'_c = \{c_0, c_1\}$.
   So $h(s) = 1 + m + M$. And $h(s') = \min\{d(s'_p, c_0), c(s'_p, c_1)\} + d(c_0, c_1)$. It is easy to see that if $\{s'_p, c_0, c_1\}$ are three different corners, $\min\{d(s'_p, c_0), c(s'_p, c_1)\} + d(c_0, c_1) \geq m + M$. Therefore, $h(s) \leq 1 + m + M \leq 1 + h(s')$.

   (e) If $|s_c| = 4$, then $|s'_c| = 3$. For a corner $c_0 \in s'_c$, $h_1(s') = d(s'_p, c_0) \geq m$.
   Then $h(s') = d(s'_p, c_0) + m + M \geq 2m + M = h(s) - 1$

□

7 Question 7: Personal comments on the development of this practice.

During these exercises, we have tried to develop code that not only solved the Pacman game but any other defined-state (single player) game by generalizing the functions and by abstraction. One example of this can be found in *search.py* where the different search methods are performed with the same algorithm, changing only the data structure where the discovered nodes are added and later on, popped to be expanded. Moreover, the class *Node* is an example of our intention to make an abstraction of the problem. While the code provided used a 3-tuple to encode succesors, we have avoided these kind of tuples for legitibility sake.

On the other hand, the only part of the assigment where we actually have had to consider the features of the Pacman was when coding the heuristic for the corners problem. Indeed, the only assumption needed was the maze being a rectangle (or square). Talking about the heuristic, our approach was to consider the four different scenarios of how many corners could remain unvisited. The advantage of breaking the heuristic into cases (divide & conquer) is that any modification to make a more complex heuristic (like considering if Pacman's way out is blocked when only one corner is left) would have been easier to implement. From the four scenarios, the case of having three remaining corners was the one that presented more possibilities of implementation. As mentioned in the previous section, our approach (that considers the most favorable case) gave very similar results to other implementations we tried while making less computations. So at the end, it resembled to many other situations where the decisions are down to a cost vs perfomance trade-off.