Elias Asskali Assakali
Joel Lamata i Martin
Adam Dbouk Alberich

**Comments about exercise 1:**
For the Depth-First Search algorithm, as we have learnt in theory class, we have used a Stack. This stack will contain a tuple with the position and a list with the path to the current position.

**Comments about exercise 2:**
For the Breadth-First Search algorithm, we did the same as in DFS, although we have implemented a queue instead of a stack.

**Comments about exercise 3:**
For the Uniform Cost Search algorithm, we implemented a Priority queue, which has another variable which is an added cost, the priority queue as its name refers will prioritize lower added cost. We have calculated the added cost using the method “getCostOfActions”.

**Comments about exercise 4:**
As we learnt in theory classes, UCS uses a function f(n) = g(n), where g(n) means the accumulated cost from root to the current node. For A* algorithm, we did the same as in UCS but adding a heuristic function. This heuristic function for A* works as follows: f(n) = g(n) + h(n). Where h(n) is the estimated cost from the current node to the goal state.

**Comments about exercise 5:**
In the “getStartState” method, we return a tuple with the start position and a list for visited corners. This list has a False in each position at the start state and will have a True at the goal state. For that reason, in the “isGoalState” method, we will just check if there is a True in each position. Finally, in the “getSuccessors” method we add the successors if and only if they do not hit a wall. Besides, if the next position is a corner we will change its state in corners’ visited list.

**Comments about exercise 6:**
To create a heuristic which must be admissible and consistent, we have chosen a lower bound, i.e., the worst possible case.
In our case, the worst case will be the maximum of the manhattan distances between the current position and each corner.

Elias Asskali Assakali
Joel Lamata i Martin
Adam Dbouk Alberich

**Comments about exercise 7:**

Having a review of the whole code, we found a method called MazeDistance, which gives us the key to solving the hard problem.

On the one hand, using the Manhattan distance method we got an estimation of the distance between the actual position and the goal, expanding 9551 nodes.

On the other hand, using MazeDistance instead, we obtained the real distance between both points taking care of the walls. And as we said before with Maze Distance we obtain an optimal solution expanding only 4137 nodes.

**Comments about exercise 8:**

To solve this problem we completed the findPathToClosestDot method, which returns the path to the nearest food position, we used MazeDistance to get the nearest food to our actual state. After obtaining the nearest food we obtained the path calling the BFS method with a problem containing our actual state and the nearest food as the goal state.