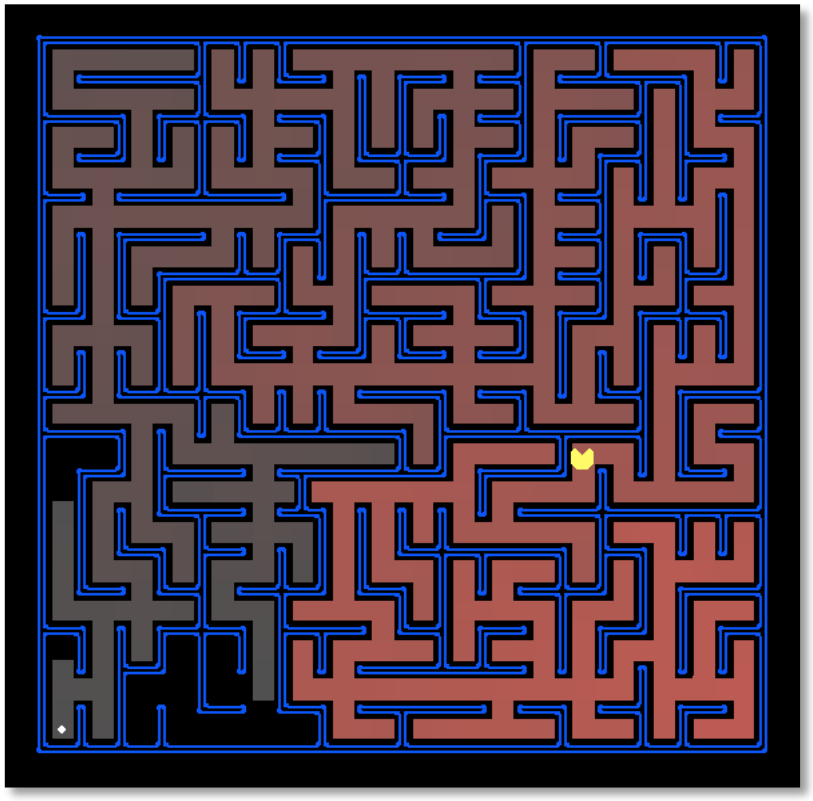
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών



Project 1 - Pacman

ARTIFICIAL INTELLIGENCE

Anna-Aikaterini Kavvada | NKUA | 30/10/2018

Table of Contents

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

In this question I have implemented the Depth First Search algorithm as given in the class presentations. To check my implementation with the autograder uncomment the lines 133-134 and comment the lines 144-145. This small change in the code changes the moment when the node is checked for being a goal state or not. In class presentation we were taught that we check the node after we pop it from the fringe, whereas the autograder does not accept that and wants the checking to take place before pushing the node in the fringe. The fringe for the implementation of this algorithm is stack, which is an implemented class in file util.py.

(Also see the inline comments of the code)

Question 2: Breadth First Search

In this question I have implemented the Breadth First Search algorithm as given in the class presentations. To check my implementation with the autograder uncomment the lines 182-183 and comment the lines 192-193. This small change in the code changes the moment when the node is checked for being a goal state or not. In class presentation we were taught that we check the node after we pop it from the fringe, whereas the autograder does not accept that and wants the checking to take place before pushing the node in the fringe. The fringe for the implementation of this algorithm is queue, which is an implemented class in file util.py.

(Also see the inline comments of the code)

Question 3: Varying the Cost Function

In this question I have implemented the Uniform-Cost Graph Search algorithm as given in the class presentations. To check my implementation with the autograder uncomment the lines 230-231 and comment the lines 241-242. This small change in the code changes the moment when the node is checked for being a goal state or not. In class presentation we were taught that we check the node after we pop it from the fringe, whereas the autograder does not accept that and wants the checking to take place before pushing the node in the fringe. The fringe for the implementation of this algorithm is priority queue, which is an implemented class in file util.py.

(Also see the inline comments of the code)

Question 4: A* search

In this question I have implemented the A* Graph Search algorithm as given in the class presentations. To check my implementation with the autograder uncomment the lines 303-304 and comment the lines 314-315. This small change in the code changes the moment when the node is checked for being a goal state or not. In class presentation we were taught that we check the node after we pop it from the fringe, whereas the autograder does not accept that and wants the checking to take place before pushing the node in the fringe. The fringe for the implementation of this algorithm is priority queue, which is an implemented class in file util.py. The priority for this queue is calculated by using a heuristic function. (I tried to use class PriorityQueueFunctionWithFunction for this implementation, but it did not seem to work properly)

(Also see the inline comments of the code)

Question 5: Finding All the Corners

There was no need to add any code in function *def* __init__(self, *startingGameState*) of the class CornersProblem(search.SearchProblem).

As a start state in function getStartState(self) of the Corners problem we return a tuple with the starting position of the pacman and a list with the corners the pacman has to visit.

As a goal state we have the state where the pacman has visited all the corners of the grid.

For the getSuccessors(self, *state*) see the code in file searchAgents.py.

(Also see the inline comments of the code)

Question 6: Corners Problem: Heuristic

This function calculates the shortest path for the pacman to reach all the corners. For that I kept track of all the corners the pacman has visited for a certain route. Everytime in the forloop we calculate the manhattan distance of the pacman from a not visited corner and store it in a list. In the end we extract the maximum distance. The function gives a better solution by choosing everytime the higher distance from the list of possible distances calculated. The maximum value we find in the list might be the greatest at this given moment, but overall the sum of all them give the best solution for the cornersHeuristic problem.

For the calculation of the Manhattan distance I have implemented the function myManhattanDistance(*position*, *goal)*.

(Also see the inline comments of the code)

Question 7: Eating All the Dots

For this problem we will need to have the "vision" of the grid quite clearly. To begin with, in order to calculate the path distances, we turn the foodGrid into a list of coordinates. We continue by calculating the aforementioned distance using the implemented function mazeDistance(*point1*, *point2*, *gameState*). We store its result in a list and after the forloop is over we extract from this list the maximum value, as we did in question 6, which is our solution.

(Also see the inline comments of the code)

Question 8: Suboptimal Search

The Goal state in this problem is whether or not a coordinate exists in the list of food coordinates in pacman's grid. If it exists, the function *def* isGoalState(self, *state*) returns True. If it is not it returns False.

For the function *def* findPathToClosestDot(self, *gameState*) I used the function for the Breadth First Algorithm that has been implemented for question 2 in file search.py. The choice of this algorithm has been easy enough since it is the fastest to find the path we need. Apart from that, A* and UCS are more like special cases of the BFS algorithm.

(Also see the inline comments of the code)