Project 1 - Pacman

Gkika Violetta
1115201600222

Search.py:
All four functions are implemented in a similar way based on the pseudocode from the slides of the lecure. All four functions use utility functions from util.py to create some data structure to store the states of the fringe.

- Q1: Depth First Search

  Uses Stack() from util.py to store the states of the fringe. Each state stores a tuple (position in the maze, list of actions followed from the starting position to reach current position).

- Q2: Breadth First Search

  Uses `Queue()` from util.py to store the states of the fringe. Each state stores a tuple (position in the maze, list of actions followed from the starting position to reach current position).

- Q3: Uniform Cost Search

  Uses PriorityQueue from util.py. Each state stores a tuple(current_node, path) where current node is the position on the maze and path a list of actions followed from the starting position to reach the current position and also a priority which is the total cost of actions.

  Uses update function implemented in util.py inside PriorityQueue class and, as described in comments inside the file it works well in our case, but based in the way we choose to store the elements in the PriorityQueue where a state is represented by the tuple as described above, the given update function will never find the item in the heap (cause we give as parameter a tuple containing a position and a path and we are searching for a tuple with same position but different path). The update function will always add the state as a new item in the priority queue.

- Q4: A* Search

  Uses PriorityQueue from util.py. Each state stores a tuple(current_node, path) where current node is the position on the maze and path a list of actions followed from the starting position to reach the current position and also a priority which is the total cost of actions.

  In Q4 we have the same problem with the update function as the one in Q3 but here I implemented a new update function as described previously in Q3. Anyway, the code works in the same way no matter which update function we choose to run.

searchAgents.py:

- Q5: Corners Problem: Representation

  __init__() : I added a list of tuples. Each tuple represent a corners and stores (corners position in the maze , information if the corner is visited or not)

  False means the corner is not visited yet, True means corens is already visited.

  getStartState(): returns a tuple(current position in the maze, a list of corners state when the pacman has reached that position) .

  isGoalState(): by traversing the list of the corners state given as a parameter in the function decides if the goal state is reached or not. Return False even if one of the corners in the list of corners state has FALSE value.

  getSuccessors(): This function takes a state as parameter and by considering each possible action pacman can make from that position, finds all the possible positions pacman can reach. Next, check if that position is a corner and if it is, checks if it is an unvisited corner. If so changes the state of corners in the list of tuples. Creates a new state for the possible position, which is a successor of the current position and returns it.

- Q6: Corners Problem: Heuristic

  We need to return an estimated cost for pacman to reach all four corners for each position that it may be in the current state. The idea is that we need a lower bound of the real distance. So, If pacman is in a x position to reach all four

corners from there it has to go from x to the closest unvisited corner then from that corner to the closest unvisited corner and so on. This distance can not be lower from the distance that pacman need to cross to reach from x position to the most distant corner(let's say y), since h(x) ≤ c(x,action, an unvisited corner) + h(unvisited corner), where h(x) is the distance pacman need to cross to reach from x position on y position, c the distance pacman need to cross from x position to reach the closest unvisited corner and h(corner) the distance pacman need to cross to reach from that unvisited closest corner of x t y position. So, what we really need to calculate in this function is the distance pacman need to cross from from the current state to reach the most distant unvisited corner.

To calculate the distance we use the given manhattanDistance() function as the maze is

- Q7: Eating All The Dots: Heuristic

  The idea is the same as the one described in the previous question, but here instead of a maximum of four positions to reach (four corners) we have a larger number of positions to reach. The implementation follows the same logic as Q6 but uses as metric function mazeDistance().

- Q8: Suboptimal Search

  findPathToClosesDot(): modified so as to use the BFS function we implemented in search.py to find the shortest path to reach a goal position from a given position on the maze.

  isGoalState(): modified to check if the given state is a goal state, that means if all there is no more food in the maze when pacman has reached the state given as parameter in the function.