## Artificial Intelligence Course

## Project 1: Search in Pacman

|  |  |  |
| --- | --- | --- |
| First name | Last name | Student number |
| Oskari | Kotajärvi | 2462701 |
| Matias | Lang | 2478533 |
|  |  |  |

### Comments about the assignment (if you have)

|  |
| --- |
|  |

### Question 1: Finding a Fixed Food Dot using Depth First Search (3 points)

### Explain the data structure that you used and also the way you keep the trace from the starting state to the end state …

### The data structure used is Stack, that we call *stack* in our code. Stack works with first in last out principle. First we get the startState and store it in two different variables and initialize other variables as well; explored (list of explored nodes), and the stack. We push the startState into the stack as it is the first node to be explored. After this we enter a loop where we stay as long as the stack is not empty and we haven’t reached our goal. We pop the current node from the stack and see if it is the winning node, if it is not we get the successors of that node and check the child nodes if they have been explored. If the coordinate is not explored we push it in to the stack and the loop continues. After the loop we return the actions and directions.

### Question 2: Breadth First Search (3 points)

### Explain the data structure that you used and also the way you keep the trace from the starting state to the end state …

### This one works with queue data structure. The queue works with first in first out principle (fifo). First we do almost the same initialization as we did with the solution above. We push the initial state to the queue. We enter a loop where we stay as long as the queue is not empty or the winning node is found. We pop the first one in queue to and store in action and state. We then get the successor of the said state and check if the coordinate is explored. If it is not we push it to the queue and add it to explored. The loop continues. After the loop is finished we return the actions.

### Question 3: Varying the Cost Function (3 points)

### Explain the data structure that you used and also the way you keep the trace from the starting state to the end state …

### In this algorithm we used a priority queue. The priority queue works in the same way as a normal queue, but the objects in the queue are assigned with a priority value, and the object with highest priority is at the one end of the queue and the object with the lowest priority value is at the other end of the queue.

### First we do the same as in the explanations above: we create an empty data structure and in this case a priority queue. After which, we get the starting state and push it to the queue with a priority of 0. After this we declare an empty list of explored nodes. While the queue is not empty we stay in the loop and pop the state and actions from the queue. After this we check if the node we just popped is the goal node (winning node). If it is we return actions. If it is not we add the node to the explored list. We then enter a for loop where we check our surroundings and if there is a coordinate not explored yet, we push it to the queue with priority of problem.getCostOfActions(actions + [direction]) –> cost of the action.

### We loop this as long as the queue is not empty or the winning node is found.

### The goal of this function is to find the most cost efficient way to reach the goal.

### Question 4: A\* search (3 points)

### Explain the data structure that you used and also the way you keep the trace from the starting state to the end state …

### This works almost the same way as the algorithm in q3. Only difference being the last few lines. Instead of pushing only the costofActions as a priority value, we push the costofActions + the value from heuristic function. Heuristic value being the estimation of the cost to reach from the coordinates to the goal (which is the food).

### **Question 5: Finding all the corners (3 points)**

### Explain the new state representation…

### GetStartState(): in getStartState we store visited corners in to an array and give the state as a tuple of the current state and the visited corners.

### IsGoalState(): Corners are all visited if there are 4 elements in the corners array and we return this as len(state[1]) == 4

### GetSuccessors(): For every direction from the current position we check to see if moving will hit a wall. If it doesn’t, we see if making the move would lead to a corner. If it does, we give that move as a possible successor and update the visited corners to reflect that the corner is visited if the move is made. If not we just update the position without changing the visited corners.

### **Question 6: Corners problem: Heuristic (3 points)**

### Explain your Heuristic function…

### First we find all the corners that we haven’t visited yet, and append them to a list so we can go through them easily.

### After this we find the sum of the shortest distances between the unvisited corners and use this as the heuristic. The heuristic will return 0 at a goal state since the minimum distance to a corner when in a corner is 0, and will never return negative since manhattanDistance can never be negative.

### **Question 7: Eating all the dots (3 points)**

### Explain your new state representation and the heuristic function…

### First we check if there is any food, if there is not, there isn’t a problem to solve and we return 0.

### Heuristic return 0 at goal. We use the given mazeDistance function to determine the heuristic which takes quite a while, but is worthwhile. Heuristic never returns negative since distances to the food cant be negative (lowest value 0 at the last food dot where the pacman is on).

### **Question 8: Suboptimal search (3 points)**

### Explain how you did the subotpimal search…

### The function performs a Breadth first search (explained above on question 2) to find the closest dot. The algorithm doesn’t always find the shortest path through the maze as the closest food might not be in the path of the shortest path through the maze. If there are 2+ dots to choose from the algorithm chooses one arbitratily., which can cause the other food as the only food left in that area of the maze. Then, pacman will eat everythin else before returning to eat that food.