## Artificial Intelligence Course

## Project 1: Search in Pacman

|  |  |  |
| --- | --- | --- |
| First name | Last name | Student number |
| Tapio | Kursula | 2402868 |
| Toni | Kuosmanen | 1986123 |
|  |  |  |

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

|  |
| --- |
|  |

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

All the search methods use the TreeNode data structure for storing the search tree nodes in the fringe. TreeNode contains the state of the node, the node’s parent node, the action that was applied to the parent node in order to move to current node, and the cost of that action. When a solution is found, the path can be constructed by moving from node to parent node and saving each action to a list until the root of the search tree has been reached. All the search methods also kept track of visited nodes so that no node was expanded multiple times.

For the depth first search we used the Stack data structure for storing the fringe nodes. Nodes are popped from the stack on the LIFO-principle. This ensures that the search proceeds depth first since if a node has any children, they are added straight to the top of the stack after expanding the node.

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

For the breadth first search, we used the Queue data structure. Queue operates under the FIFO principle which ensures that the nodes added to the fringe are expanded in the order they were added. This means the search proceeds breadth first since the children of any expanded node are searched only after all the previously added nodes have been searched.

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

For the uniform cost search, we used the PriorityQueue data structure. PriorityQueue is similar to the Queue data structure but the nodes are sorted in the queue by their priority values. This means that the node with the highest priority value gets popped first even if it was not the first node added to the queue. The priority value we assigned to the nodes was the total path cost to reach the node’s state. The total cost was calculated as the sum of the current node’s and all it’s parent nodes costs.

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

Our implementation of the A\* search is very similar to the UCS. The only difference is the priority value used when adding nodes to the queue. In A\* search, instead of the total cost, we used the combined cost of the heuristic value and the total path cost.

### Question 5: Finding All the Corners (3 points)

In addition to the current position, the state keeps track of all the visited corners. Goal state is reached when all of the corners are visited.

The starting state contains only the position and an empty tuple. When successors are chosen, the agent looks if the current position is a corner and if it is, adds it to a list of visited corners. This list is then put inside the tuple in the state. Goal state is reached when all of the corners are found in the list containing visited corners.

### Question 6: Corners Problem: Heuristic (3 points)

We first tried implementing this heuristic with creating a manhattan path to the nearest unvisited corner, and seeing if the path goes through any walls. If it does, it weighs the path more with every colliding point. This, however was not a very efficient heuristic as it needed about 1750 nodes to find the path.

Current implementation returns the total weight of a a minimum spanning tree containing the current position and all the unvisited corners. The tree was constructed using Prim’s algorithm, where the cost of each edge in the tree was calculated as the euclidean distance between the two vertices. Additionally, the heuristic calculated the maximum manhattan distance to unvisited corners, i.e, the distance to furthest corner. Finally, it returns the maximum of the two calculated values as the estimated cost.

### Question 7: Eating All The Dots (4 points)

We came by our food search heuristic by trying to solve a relaxed version of the search problem. Our first heuristic attempt calculated the shortest path that included all the food positions while ignoring the walls. However, we could not find an algorithm that could consistently calculate the shortest path correctly so this approach was abandoned.

Our next attempt calculated the total weight of the minimum spanning tree that includes the current position and all the remaining positions with food. The minimum spanning tree was constructed using the same Prim’s algorithm used in Q6. This heuristic was found to be consistent and also gave reasonably good results. For the final heuristic we also calculated the manhattan distance to the furthest food position. The heuristic then returned the larger of the two calculated costs.

### Question 8: Suboptimal Search (3 points)

The suboptimal food search proceeds towards the closest available food until all the food is eaten. The solution utilizes the AnyFoodSearchProblem and A\* search to find the path to the closest food dot. The implementation was easy since basically all we had to do was to fill in the isGoalState method in AnyFoodSearchProblem class. The goal state of AnyFoodSearchProblem is reached if the current position happens to be any of the food positions stored in the problem.