**Question 1 Technique & Analysis**

Depth-first search together with graph search algorithm is implemented for question 1. DFS is usually implemented with a Last In, First Out queue. For question 1, util.Stack() has been used to implement DFS. It stores the frontier node and the route to that particular frontier node as one entity in util.Stack(). DFS is implemented with GSA in order to avoid exploring redundant paths, especially paths that it has visited before, by storing already explored states.

|  |  |
| --- | --- |
| Complete | DFS fails in infinite-depth spaces or spaces with loops. Nevertheless, in this case, since it is implemented with GSA and the maze is finite spaces, it is complete. |
| Time | O(bm) |
| Space | O(bm) |
| Optimal | DFS does not necessarily return optimal solution since it only returns the first solution it finds. |
| *\* b: maximum branching factor of the search tree*  *\*\*m: maximum length of any path in the state space* | |

**Question 2 Technique & Analysis**

Breadth-first search together with graph search algorithm is implemented for question 2. BFS is usually implemented with a First In, First Out queue. For question 2, util.Queue() has been used to implement DFS. It stores the frontier node and the route to that particular frontier node as one entity in util.Queue(). BFS is implemented with GSA in order to avoid exploring redundant paths, especially paths that it has visited before, by storing already explored states.

|  |  |
| --- | --- |
| Complete | Provided that the branching factor is finite in the case, it is complete. |
| Time | O(bd+1) |
| Space | O(bd+1) |
| Optimal | If only consider each step as cost, it will return a optimal solution. |
| *\* b: maximum branching factor of the search tree*  *\*\*d: depth of the shallowest solution* | |

**Question 3 Technique & Analysis**

Uniform-cost search together with graph search algorithm is implemented for question 3. UCS is usually implemented with a priority queue that pops data with higher priority first. For question 3, util.PriorityQueu() has been used to implement UCS. It stores the frontier node, the route to and the total path cost to travel to that particular frontier node as one entity in util.Queue(). UCS is implemented with GSA in order to avoid exploring redundant paths, especially paths that it has visited before, by storing already explored states.

|  |  |
| --- | --- |
| Complete | It is complete in the case given that every step exceed some small positive constant and the branching factor is finite. |
| Time | O(b1+[C\*/e]) |
| Space | O(b1+[C\*/e]) |
| Optimal | It returns optimal solution since nodes are expanded in increasing order of path cost. |
| *\*\*C\*: cost of the optimal solution*  *\*\*\*e: small positive constant* | |

**Question 4 Technique & Analysis**

A\* search together with graph search algorithm is implemented for question 4. A\* is usually implemented with a priority queue that pops data with higher priority first. For question 4, util.PriorityQueu() has been used to implement A\*. It stores the frontier node, the route to to that particular frontier node and the sum of path cost and estimate cost from frontier node to goal as one entity in util.Queue(). A\* is implemented with GSA in order to avoid exploring redundant paths, especially paths that it has visited before, by storing already explored states. In question 4, Manhattan distance is used as the estimated cost from frontier node to goal. This heuristic function is considered admissible since the true path cost to goal should not be less than Manhattan distance.

|  |  |
| --- | --- |
| Complete | It is complete in the case given that every step exceed some small positive constant and the branching factor is finite. |
| Time | O(b1+[C\*/e]) |
| Space | O(b1+[C\*/e]) |
| Optimal | It return the optimal solution in this case given that Manhattan distance is an admissible heuristic |
| *\*\*C\*: cost of the optimal solution*  *\*\*\*e: small positive constant* | |

**Question 5 Technique & Analysis**

In question 5, it is required to implement CornersProblem. One additional field has been added for state – visitedCorners. isGoalState check whether visitedCorners has contained all four corners. If so it will return true, otherwise return false. getSuccessors will check whether the nextState is one of the four corners. If so, it will append this corner into the visitedCorners so that the program remember the corners it has been to.

**Question 6 Technique & Analysis**

In question 6, cornerHeuristic is implemented with self-designed heuristic function. The whole function is to return the sum of the distance from the current position to the nearest corners and the distance between corners. The distance is measured in Manhattan distance format.

This heuristic is admissible since the Manhattan distance should never be more than the true path cost given that there are wall in the maze. It should be non-trivial since its estimation should be fairly close to the true path cost given that it considers not only the distance between current position and one of the corners but also the distance between the rest of the corners. This heuristic is consistent as well since the heuristic value of successor generate by any action (presumably having only 1 unit of cost) should be equal to the sum of the heuristic value of original state and the 1-unit step cost in this case.

**Question 7 Technique & Analysis**

In question 7, foodHeuristic is implemented with self-designed heuristic function. The whole function is to return the sum of the distance of minimum spanning tree of all the food point including the current position. The distance is measured in Euclidean distance instead of Manhattan distance. The program first initializes every distance within food points by Euclidean distance. Then it follows Kruskal’s algorithm to find the minimum spanning tree to connect every foot point in the graph.

The MST heuristic function is admissible and non-trivial since it is able to solve a relaxed sub-problem – to connect all food point with minimal path cost assuming pacman can travel diagonally and through walls. It is always shorter than the true path cost which have to consider walls and travel back and forth food points. This heuristic function is also consistent since the current position of pacman is also considered. When pacman move towards its nearest food point, the heuristic value for the successor should be decrease accordingly.

**Question 8 Technique & Analysis**

Steepest Hill-Climbing Search is implemented in question 8. This is an algorithm to find the next step with the best improvement in heuristic cost. It first try move the Queen along the column to see the resulting number of attack. If that move generate the lowest number of attack, it will return the board as the step.

The search is not complete since it cannot find the solution when it is forced to make each move with improvement. If it is allowed to make move with no improvement (i.e. allow travel in the “shoulder”), the success rate should be much higher.