|  |  |
| --- | --- |
| Group: 10  Class: INT3401 21 – Trí tuệ nhân tạo | Project Report |

*Members*

|  |  |  |  |
| --- | --- | --- | --- |
| STT | Họ tên | MSSV | Điểm đóng góp |
| 1 | Nguyễn Tuấn Anh | 17021207 | 8 |
| 2 | Nguyễn Phương Hằng | 17021241 | 9 |
| 3 | Phạm Tuấn Hiệp | 17021242 | 10 |
| 4 | Lưu Hoài Linh | 17021284 | 10 |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Question 1:** **Depth First Search**

* Idea: Depth-first search (DFS) is an algorithm for searching a graph or tree data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored.
* Pseudo code:

**Initialize** an empty stack for storage of nodes, S.

**For** each vertex u, define u.**visited** to be false.

**Push** the root (first node to be visited) into S.

**While** S is **not** empty:

**Pop** the first element in S, u.

**If** u.visited = false, then:

U.visited = **true**

**for** each **unvisited** neighbor w of u:

**Push** w into S.

**End** process when all nodes have been visited.

* Result: 3/3 tests – *successful*

**Question 2:** **Breadth First Search**

* Idea: Breadth-first search is a strategy for exploration that always selects the shallowest fringe node from the start node for expansion. It explores all the nodes at the present depth before moving on to the nodes at the next depth level.
* Pseudo code:

**BFS**(v){

{add v to queue and mark it}

**Add**(Q, v)

Mark v as visited

**while** (not IsEmpty(Q)) **do**

**begin**

w = **QueueFront**(Q)

**Remove**(Q)

{loop invariant : there is a path from vertex w to every vertex in the queue Q}

**for** each unvisited vertex u adjacent to w do

**begin**

Mark u as visited

**Add**(Q , u)

**end** { for }

**end**{ while

}

* Result: 3/3 tests – *successful*

**Question 3: Uniform-cost graph search**

* Idea: Uniform cost search (UCS) is a strategy for exploration that always selects the lowest cost fringe node from the start node for expansion.
* Pseudo code:

|  |
| --- |
| * **function** UNIFORM-COST-SEARCH(problem) **returns** a solution, or failure  **if** problem's initial state is a goal **then return** empty path to initial state  frontier ← a priority queue ordered by pathCost, with a node for the initial state  reached ← a table of {state: the best path that reached state}; initially empty  solution ← failure  **while** frontier is not empty **and** top(frontier) is cheaper than solution **do**    parent ← pop(frontier)    **for** child **in** successors(parent) **do**      s ← child.state      **if** s is not in reached **or** child is a cheaper path than reached[s] **then**        reached[s] ← child        add child to the frontier        **if** child is a goal and is cheaper than solution **then**          solution = child  **return** solution |

* Result: 3/3 tests – *successful*

**Question 4:** **A\* search**

* Idea: A\* works by making a lowest-cost path tree from the start node to the target node. What makes A\* different and better for many searches is that for each node, A\* uses a function f(n)f(n) that gives an estimate of the total cost of a path using that node.
* Pseudo code:

make an empty closed list

**while** (the destination node has not been reached):

consider the node with the lowest f score in the open list

**if** (this node is our destination node) :

we are finished

**if not**:

put the current node in the closed list and look at all of its neighbors

**for** (each neighbor of the current node):

**if** (neighbor has lower g value than current and is in the closed list) :

replace the neighbor with the new, lower, g value

current node is now the neighbor's parent

**else if** (current g value is lower and this neighbor is in the open list ) :

replace the neighbor with the new, lower, g value

change the neighbor's parent to our current node

**else if** this neighbor is not in both lists:

add it to the open list and set its g

* Result: 3/3 tests – *successful*

**Question 5:** **Finding All the Corners**

* Idea: find the shortest path through the maze (CornerMaze) that touches all four corners. In this question we have to define 3 functions: getStartState, isGoalState and getSuccessors.
* Pseudo code:

|  |
| --- |
| - getStartState:  Return the start state  - isGoalState:  If the action (path) list is empty -> return true  - getSuccessors:  Make an empty list of successors for every action (go W,E,S,N)  If next state is not a wall:  if next state is a corner:  remove that position from corner list  move to other corner append this state to successors list |

* Result: 3/3 tests – *successful*

**Question 6:** **Corners Problem: Heuristic**

* Idea: a functions that take search state and return numbers that estimate the cost to the nearest goal. Expected: functions return values closer to the actual goal cost.
* Pseudo code:

|  |
| --- |
| **Read** information about pacman ‘s location  Declare an empty list to store distances between pacman and the location of corner  **For** corner of corners:  Measure the distance by subtracting the location information of Pacman and the corners from each other between x coordinates and y coordinates, then using the absolute value.  **Then** appending all to list and selecting the max value  **If** the currentState is goal, return 0  **Else** return heristic value |

* Result: 3/3 tests – *successful*

**Question 7:** **Eating All The Dots**

* Idea: In this question we have to solve the problem: eating all the food in as few steps as possible. We’ll make the foodHeuristic function using predefined mazeDistance in searchAgent.py
* Pseudo code:

|  |
| --- |
| Make a empty list of distance from current position to food  Check if the foodGrid is a list  **If** yes, for every food in the list  **Evaluate** the distance from current position to food using mazeDistance function (bfs)  **Append** the distance to list  **Return** the max(distances) list |

* Result: 5/4 tests – *successful*

**Question 8:** **Suboptimal Search**

* Idea: an agent that always greedily eats the closest dot. Because the ClosestDotSearchAgent is predefined in searchAgents.py, so we just need to find an appropriate search function to find a path to the closest dot (We use bfs function in Q2)
* Pseudo code:

|  |
| --- |
| **While** the food on map is > 0:  **Go to** the closest dot using bfs function (define in Q2)  **Finish** |

* Result: 3/3 tests – *successful*