Gotta Catch Em ALL

Team members:
Khalid Jamal Abdulnasser 28-6420 T-12
Mohamed Mehany 28-4960 T-09
Mahmoud Gamal 25-1731 T-10

1. Description of our understanding of the problem:
   We do understand that the world is a rectangular grid that has empty cells, walls and pokemons. There is an agent standing at one of the grid cells (that is not a wall) with an egg that will hatch if he will walk some given number of forward steps. This agent can either walk forward, rotate left or rotate right. The agent aims to start from the initial position he is standing at, collect all the pokemons, hatch his egg and finally arrive to a destination position.

2. Our implementation of the Search Tree Node:
   The Search Tree Node is an object that has all the attributes studied in class. Which are 1) the state representing the node, 2) the parent search tree node, 3) the operator that resulted to expanding this node from its parent, 4) the depth of the node in the search tree, 5) the path cost of this node from the root to it. In addition to that we have a value priority. This value indicates the priority of this node in the search queue (The node with the least priority is popped first).

3. Our implementation of the Search Problem:
   The search problem is an object that has the following attributes: 1) the operators that an agent can use in this problem, 2) the initial state

of the problem, 3) the transition function of the problem (a function that takes as an argument a state and an operator and outputs the state resulting of performing this operator in the input state), 4) the goal test function of the problem, 5) path cost function of the problem.

We did not include the whole state space in the problem because that would be too much.

4. Our implementation of the GottaCatchEmAll problem:
   The GottaCatchEmAll problem is an instance of the generic problem class. Takes as an argument the list of operators ([left, right, forward]), an initial state, a specific transition function, a specific goal test function, and a specific path cost function.

5. The main functions:

   a) genMaze(): this function generates a rectangular grid with random dimensions N and M, randomizes every cell in it to a '#', '.' or a 'p' for wall, empty cell and a cell containing a pokemon respectively, randomizes the initial position and orientation of the agent, randomizes the destination position also and finally randomizes the time needed to hatch the egg.

   b) gottaCatchEmAllNextState(state, operator): this is the transition function of our problem, it takes a state and an operator and returns the state resulting from performing the input operator from the input state.

   c) gottaCatchEmAllGoalState(state): this is the goal state function of our problem, it takes a state as an input and checks whether the position in this state is the maze destination position, the time remaining to hatch the egg is zero and all the pokemons have been collected.

d) gottaCatchEmAllPathCostFunction(parentState, newState, operator, parentPathCost): this is the path cost function of our problem, it takes as an input a state and one if its children and the operator used to do the expansion and the path cost of the node contained the parent state. What it does is just checking if the operator was moving forward then the path cost of the node containing the child state will be one more than the path cost parent node, otherwise it will be just the same.

e) getGottaCatchEmAllInitialState(): this function generates the initial state of our problem based on the location of the initial position of the agent in the maze and the time to hatch the egg generated in the maze too.

f) generalSearch(problem, QueuingFunction): this is the general search algorithm function. It behaves as studied in class exactly. It has a queue implemented as a priority queue that has a comparator that sorts nodes on their priority value and it always initialize a hashSet called vis (this is our hashSet to store the previously visited nodes to avoid running forever).

g) expand(node, operators): this function is getting called from the general search algorithm. It takes as an input a node and a list of operators and it outputs an array of nodes containing the non-visited children of this node.

6. How the different search algorithms were implemented:
   Every algorithm has its own Queuing function that plays with the priority attribute of the nodes to make correct insertions and removals

to and from the priority queue.

a) Breadth First Search: calls the general search algorithm with the problem the breadth first search queuing function. This function takes the output of the expand function which is a list of nodes, assigns each of them a priority that is greater than any priority in the priority queue and marks the state of the node as visited and then inserts it in the priority queue. This way of assigning priority values ensures the first in first out property.

b) Depth First Search: calls the general search algorithm with the problem the depth first search queuing function. This function takes the output of the expand function which is a list of nodes, assigns each of them a priority that is less than any priority in the priority queue and marks the state of the node as visited and then inserts it in the priority queue. This way of assigning priority values ensures the last in first out property.

c) Depth Limited Search: calls the general search algorithm with the problem the depth limited search queuing function. This function takes the output of the expand function which is a list of nodes, assigns each of them a priority that is less than any priority in the priority queue and does not mark the state of the node as visited and then inserts it in the priority queue. This way of assigning priority values ensures the first in first out property. In addition to that, it looks at a global variable representing the maximum possible depth it can go and it does not add to the priority queue any node that has a depth greater than it.

d) Iterative Deepening Search: calls the depth limited search algorithm with the problem and the maximum allowed depth and also assigns the maximum allowed depth to a global variable.

e) Greedy 1 Search: calls the general search algorithm with the problem the Greedy 1 search queuing function. This function takes the output of the expand function which is a list of nodes, assigns each of them a priority that is the value of the heuristic 1 of the state of the node and marks the state of the node as visited and then inserts it in the priority queue. This way of assigning priority values ensures expanding the nodes in the greedy way of heuristic 1.

f) Greedy 2 Search: calls the general search algorithm with the problem the Greedy 2 search queuing function. This function takes the output of the expand function which is a list of nodes, assigns each of them a priority that is the value of the heuristic 2 of the state of the node and marks the state of the node as visited and then inserts it in the priority queue. This way of assigning priority values ensures expanding the nodes in the greedy way of heuristic 2.

g) Greedy 3 Search: calls the general search algorithm with the problem the Greedy 3 search queuing function. This function takes the output of the expand function which is a list of nodes, assigns each of them a priority that is the value of the heuristic 3 of the state of the node and marks the state of the node as visited and then inserts it in the priority queue. This way of assigning priority values ensures expanding the nodes in the greedy way of heuristic 3.

h) A Star 1 Search: calls the general search algorithm with the problem the A Star 1 search queuing function. This function takes the output of the expand function which is a list of nodes, assigns each of them a priority that is the value of the heuristic 1 of the state of the node added to it the value of the path cost of that node from the root and marks the state of the node as visited and then inserts it in the priority queue. This way of assigning priority values ensures expanding the nodes in the greedy way of heuristic 1 with considering the path cost from the root.

i) A Star 2 Search: calls the general search algorithm with the problem the A Star 2 search queuing function. This function takes the output of the expand function which is a list of nodes, assigns each of them a priority that is the value of the heuristic 2 of the state of the node added to it the value of the path cost of that node from the root and marks the state of the node as visited and then inserts it in the priority queue. This way of assigning priority values ensures expanding the nodes in the greedy way of heuristic 2 with considering the path cost from the root.

j) A Star 3 Search: calls the general search algorithm with the problem the A Star 3 search queuing function. This function takes the output of the expand function which is a list of nodes, assigns each of them a priority that is the value of the heuristic 3 of the state of the node added to it the value of the path cost of that node from the root and marks the state of the node as visited and then inserts it in the priority queue. This way of assigning priority values ensures expanding the nodes in the greedy way of heuristic 3 with considering the path cost from the root.

7. We have 3 heuristic functions that we use for both greedy search and A* search:

   heuristic1(): returns the remaining forward steps needed to hatch the egg from this state.
   This heuristic is admissible because a goal can never be closer to the state that this value because you at least need to hatch the egg to be in a goal state. So, this value is always less than or equal to the real distance between a state and a goal state.
   It is also monotonic (more precisely non increasing) because you can't go from state to another and increase the remaining forward

steps to hatch the egg.

heuristic2(): returns the number of remaining pokemons non-collected in the grid.
This heuristic is admissible because a goal can never be closer to the state that this value because you at least need to collect all the pokemons to be in a goal state. So, this value is always less than or equal to the real distance between a state and a goal state.
It is also monotonic (more precisely non increasing) because you can't go from state to another and increase the remaining pokemons non-collected in the grid.

heuristic3(): returns the manhattan distance between the position if the current state and the position of a goal state.
This heuristic is admissible because a goal can never be closer to the state that this value because you at least need to go to the position of the goal state to be in a goal state and the manhattan distance is the shortest distance between any two points in a grid (given that we can't move diagonally) assuming there are no walls. So, this value is always less than or equal to the real distance between a state and a goal state.
This heuristic is not a monotonic one.

8. The comparison of performance:

** Completeness:

* Depth First Search: In our implementation for our problem it is complete, because we visit each state at most once and the state space is finite. But if the state space is infinite or we don't handle previously visited state it may run forever.

* Breadth First Search: It is complete.

* Iterative Deepening: It is complete but takes much more time because it is wrong to not visit previously visited nodes. This is the wrong case, assume we are running the depth limited search on depth 3 and we have the following grapth:
1 -> 2
2 -> 3
3 -> 4
4 -> 5
1 -> 4
And assume the goal node is 5
Now if the DL search will visit 1 -> 2 -> 3 -> 4 then backtrack to 1 he will not be able to expand the 4 because it is marked as visited. So, you will not get the goal node in the least depth.
Also Iterative deepening will run forever if there is no solution for the problem

* Uniform Cost Search: It is complete for our problem since the state space is finite and all the costs are non-negative.

* The 3 greedy search algorithms are all complete because the state space is finite, and we do not visit previously visited states.

* The 3 A* search algorithms are all complete because the state space is finite, we do not visit previously visited states, and all cost are non-negative.

** Optimality:
* Depth First Search: It is not optimal as it does not consult the path cost function.

* Breadth First Search: It is not optimal as it does not consult the path cost function.

* Iterative Deepening: It is not optimal as it does not consult the path cost function. If we will mark visited nodes and do not visit them again then we can get goal nodes at depths greater than the ones we got from breadth first search. This is the wrong case, assume we are running the depth limited search on depth 3 and we have the following graph:

1 -> 2
2 -> 3
3 -> 4
4 -> 5
1 -> 4
And assume the goal node is 5
Now if the DL search will visit 1 -> 2 -> 3 -> 4 then backtrack to 1 he will not be able to expand the 4 because it is marked as visited. So, you will not get the goal node in the least depth.

* Uniform Cost Search: It is optimal because it expands the node that has the least path cost function first.

* The 3 greedy search algorithms are all not guaranteed to be complete because they do not consult the path cost function they just estimate the cost from a state to the goal state and not even using all the parameters.

* The 3 A* search algorithms are all optimal because the 3 heuristic functions are admissible and we add to their value the path cost value from the root.

** Number of expanded Nodes:

* Depth First Search: In most of the cases it is relatively low, less all other search algorithms except the greedy search.

* Breadth First Search: Usually has the third highest number after the Uniform Cost search and Iterative deepening.

* Iterative Deepening: Always have the highest number as it does multiple depth limited searches.

* Uniform Cost Search: Usually has higher number than all other searches except for Iterative deepening.

* The 3 greedy search algorithms: Usually have the least numbers, GR1 and GR2 usually have very similar numbers and GR3 has higher number in most of the cases.

* The 3 A* search algorithms: Usually have numbers higher than the greedy algorithms, the depth first search and Iterative Deepening but has lower numbers than Uniform Cost search in all cases.