Assignment 1 report

Submitted By: Moaaz Maamoon 28-10898, Omar Adel 28-10999, Hossam Hazem 28-6205

Description of the problem:

The problem is about implementing a search agent to satisfy the goal which is represented by collecting all pokemons around the maze and finding the ending point of the maze. The problem can be seen by the agent as a states, each state represents his position, number of pokemons to be collected and steps left to hatch. And the agent has to decide where to go next, to minimize the steps taken to generate series of steps needed for the player to hatch the egg, collect all pokemons and reach the end point.

A discussion of your implementation of the search-tree node ADT:

The search-tree node consist of the essential elements as the State class that holds the current state of the node, Parent which holds the parent node, operator which contains the operators that has been done to produce this state and node, depth which represents the depth of the node in the search tree, pathCost which is the total cost from the root of the tree to this node, and lastly predictedCost which holds the heuristic cost as the predicted cost to solve the problem

A discussion of your implementation of the search problem ADT.

The search problem Consists of an array of operators which holds the allowed operators(actions) in the problem to be applied on the state, also it has an instance of the initial state of the problem, also holds the heuristic function, this class is the one responsible of expanding the nodes, it has a function expand which takes a search node and applies the operators on the state of this search node to generate new search nodes and apply the heuristic function on these nodes.

Discussion of the implementation of the Gotta Catch'em All problem:

The implementation of the problem is an extension to the general structure of a search problem. Initially we created an abstract structure for any search problem where it contains the main characteristics of a search problem. Then, the we created a special instance of the main search problem by adding additional field and information that is useful during the search process. The following attributes were added or customized to suit the new problem:

1- State: we created a customized stated called mazeState which extends the main state class but it has additional attributes to have more specific information about the status with respect to the maze. Those attributes are

a. Pokemons Left: which is used by the goal test function to determine whether there are any left pokemons or not.
b. Steps Left: this attribute is used by the goal test to find out if the egg is hatched or not.
c. Current Position & direction: both are used to get the position of the player and to determine the following action upon that location.

2- Operators: The set of operators of the new search problem are almost the same as the generic one. The set off attributes used in our implementation are move forward, move backward, move left, and move right.

3- Search Problem: the maze Search Problem which extends the main search problem implementation has defined the set of attributes of the problem(e.g. Operators, states, etc) in terms of the generated maze. The extended search solution used with the maze contains the number of nodes expanded during the search process and also defines the path of the search as a set of maze states to suit the maze generated.

4- Solution: the solution of the Gotta catch'em all problem extends the solution of the generic problem too so that it can express the result of the search in more detailed

format. The solution of the generic search problem contains information whether the search succeeded to find a goal or not and if yes it contains data about the path followed and the pathcost to reach the goal.

Description of the main functions' implementation:

The maze generator is an algorithm that generates a maze in a totally random manner. The recursive backtracking algorithm was used to generate it. The algorithm as its name suggests, generates the maze recursively.

In the beginning, a whole grid of cells is generated by random number of cells which varies between 5 ~ 30 cells for each dimension if the number exceeds these limits most probably the Java Virtual Machine will have a stack overflow due to much functions calls. After that, each cell is assigned a boolean value indicating whether it has a passage through specific direction or not, north, south, east and west. The algorithm picks a start point in the maze , mark it as visited, and then check if it has a neighbour that is not visited and create a passage between both of them, then go to this neighbour, and recursively call the function again with the new numbers of the cell. If the cell happens to be having neighbours that all of them are visited, then this means no passages to be created further and it is popped permanently from the stack.

After that if the algorithm encounters a cell that is visited but still has neighbours, it mark it as not visited and explore it again. After finding a dead end, the algorithm backtracks to the last cell that was not visited and do it over and over again.

The algorithm ensures there is not cell exists in a closed loop or a path that is not accessible from anywhere. In the process of creating a maze, pokemons are added randomly in the cells, indicating each pokemon by its initial. And finally, the player and end point are placed randomly in the maze.

Many functions were implemented to create the maze, only the main functions not the helpers will be mentioned.

The whole program of the maze starts with a *MainEngine* that is responsible of creating the maze and handling the whole process flow. Within creating the MainEngine,

an object Maze is created, calling the function *Maze.init()* which initializes the maze and generates it, and then the pokemons, player and end point are placed. After that, the MainEngine offers an interface to access the Maze data and return needed data to create the initial maze state needed for the search algorithms. After finishing the search algorithms, the function *Visualize*() is called with an array list of maze states to draw the steps the player has taken to reach his goal.

Some helpers needed for the search algorithms were implemented, for example, *getPossibleActions(mazeState state)* this is needed for returning what actions can be done within specific cell. And *nerestPokemon(position, pokemonsState)* which returns where exists the nearest pokemon to this specific location, this function was used for the heuristic function that depends on where is the nearest pokemon from this location.

A discussion of how you implemented the various search algorithms.

A Qing Fuction interface was implemented to take care of the search queue input and output it contains two different methods one for the initialization of the queue with the initialization of the comparable that take cares of the priority when cost is included in the dequeuing, there are four different *QingFuns*, *EnqueueAtEnd* that is a FIFO queue, *EnqueueAtFront* that is a LIFO queue and lastly *OrderedInsert* that is a priority queue. Different search algorithm apply their needed queue, BFS applies *EnqueueAtEnd,* DFS and ID apply *EnqueueAtFront*, while UC,GR and AS implements *OrderedInsert* where they supply the *QingFun* with their needed comparable for the dequeuing, UC only considers path cost of the nodes in the comparable class while GRD only considers predicted cost of the nodes in the comparable class while AS considers path cost and predicted cost in making the comparable class each of the searching algorithm chooses their own QingFuns and then begins the search.

A discussion of the heuristic functions.

We came up with three heuristic functions.

1- The first heuristic function depends on the distance between the end point and each of the states resulting from applying the operators on the current state. The distance is an approximate as the walls are not being considered. This function is admissible with A* algorithm because the calculated distance is in all cases less than or equal to the real distance between the source and destination cell because the walls are not being put into consideration.

2- The second heuristic function depends on the hatching time and the distance between the current cell state and the end point. The reason why this function is admissible is that if the hatching time is greater than the distance to the end point, the output of the heuristic is the hatching time. Otherwise, the output is the distance to the endpoint as in heuristic function number 1. This means that in all cases the estimated output of the heuristic function is always less than or equal to the read distance.

3- The third and last heuristic function is based on the distance to nearest pokemon. This function is admissible because the distance to the nearest pokemon is forsure less that the distance to the nearest pokemon plus the distance from the cell of the distance pokemon to the end point. This is because the endpoint will not pass the goal test without having all the pokemons. In case all pokemons are collected, the heuristic function acts like heuristic number 1.

A comparison of the performance of the different algorithms implemented in terms of completeness, optimality, and the number of expanded nodes.

The results of different algorithms:

Search results on a Maze 6*3 with 8 pokemons

Algorithm	Success?	Cost	Expanded	Time

			nodes	
BF	Yes	36	381	9
DF	Yes	36	381	5
UC	Yes	36	354	3
ID	Yes	64	5378	23
GR1	Yes	92	326	14
GR2	Yes	76	283	6
GR3	Yes	58	305	2
GR4	Yes	36	30	2
AS1	Yes	36	309	1
AS2	Yes	36	351	2
AS3	Yes	36	321	1
AS4	Yes	36	329	1

Search results on a maze 7*4 with 12 pokemons

Algorithm	Success?	Cost	Expanded Nodes	Time
BF	Yes	76	2750	63
DF	Yes	76	2750	46
UC	Yes	76	2722	47
ID	Yes	152	105746	708
GR1	Yes	94	2480	30
GR2	Yes	170	232	2
GR3	Yes	92	2111	20
GR4	Yes	92	93	1
AS1	Yes	76	2697	31

AS2	Yes	76	2722	32
AS3	Yes	76	2683	32
AS4	Yes	76	2717	32

Citation:

http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtrackin